# B.C.A. I YEAR
# Programming Methodology

SRGPGPI

**OBJECTIVE:-**

The prime purpose of C++ programming was to add object orientation to the C Programming language, which is in itself one of the most powerful programming languages.The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object. Data abstraction referto, providing only essential information to the outside world and hiding their background Details, i.e to represent the needed information in program without presenting the details.For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

**1 Principles of OOPS:-**

- Software crisis
- Software Evaluation
- Basic concepts of OOP
- Benefits ofOOP
- Object Oriented Language

**1.1 Software Crisis**

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession. This has forced the software engineers and industry to continuously look for new approaches to software design and development, and they are becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. These rapid advances appear to have created a situation of crisis within the industry. The following issued need to be addressed to face the crisis:

- How to represent real-life entities of problems in systemdesign?

- How to design system with openinterfaces?
- How to ensure reusability and extensibility ofmodules?
- How to develop modules that are tolerant of any changes infuture?
- How to improve software productivity and decrease softwarecost?
- How to improve the quality ofsoftware?
- How to manage timeschedules?

### 1.2 Software Evaluation

The software evolution has had distinct phases "layers" of growth. These layers were building up one by one over the last five decades with each layer representing an improvement over the previous one.

*Object Oriented Programming* (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do

With any particular language. However, not all languages are suitable to implement the OOP concepts easily.

### 1.3 Procedure-OrientedProgramming

In the procedure oriented approach, the problem is viewed as the sequence of things to be done such as reading, calculating and printing such as cobol, fortran and c. The primary focus is on functions. A typical structure for procedural programming. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving aproblem.

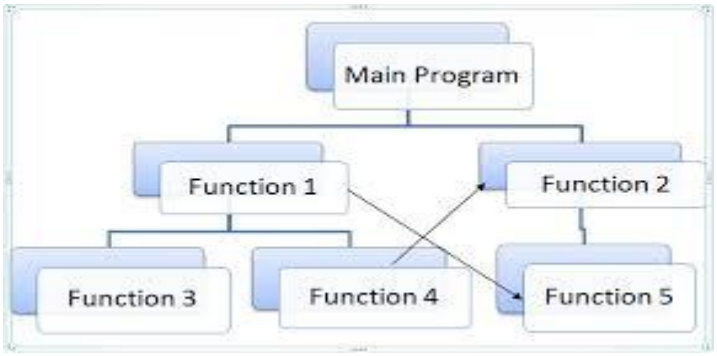



**Figure: Typical structure of procedural oriented programs**

Procedure oriented programming basically consists of writing a list of instructions for

the computer to follow, and organizing these instructions into groups known as *functions.* We normally use flowcharts to organize these actions and represent the flow of control from one action to another.

Some Characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things(algorithms).
- Large programs are divided into smaller programs known asfunctions.
- Most of the functions share global data.
- Data move openly around the system from function tofunction.
- Functions transform data from one form toanother.
- Employs top-down approach in programdesign.

### 1.4 Object OrientedParadigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.

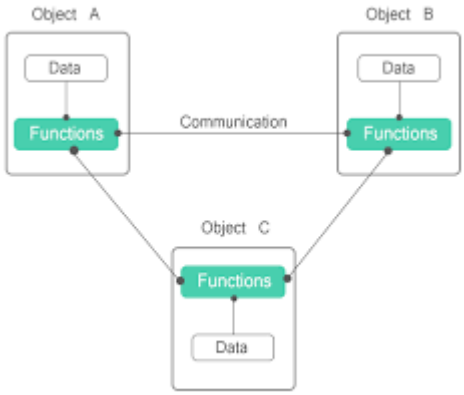



**Figure: Object Oriented Paradigm**

**Some of the features of object oriented programming are:**

- Emphasis is on data rather thanprocedure.
- Programs are divided into what are known asobjects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in thedata structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other throughfunction.

- New data and functions can be easily added whenevernecessary.
- Follows bottom up approach in programdesign.
- Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

### 1.5 Basic Concepts of Object OrientedProgramming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamicbinding
- Message passing

It shall discuss these concepts in some detail in this section.

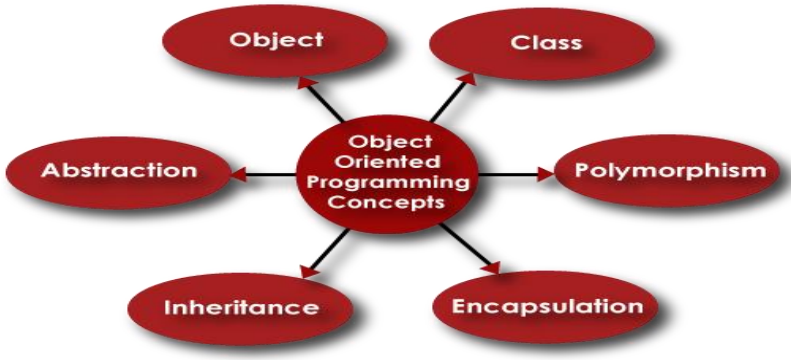



**Figure:basic concept of oops**

#### Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists.

OBJECTS: STUDENT

DATA

Name

Date-of-birth Marks

FUNCTIONS

Total Average Display

……

**Class**

Objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class.

Fruit Mango:

create an object **mango** belonging to the class **fruit.**

**Data Abstraction and Encapsulation**

**Encapsulation**

- The wrapping up of data and function into a single unit (called class) is known as *encapsulation*
- Data and encapsulation is the most striking feature of a class..
- This insulation of the data from direct access by the program is called *data hiding or information hiding*.

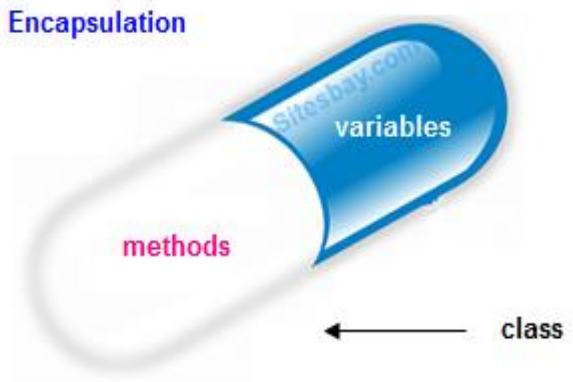
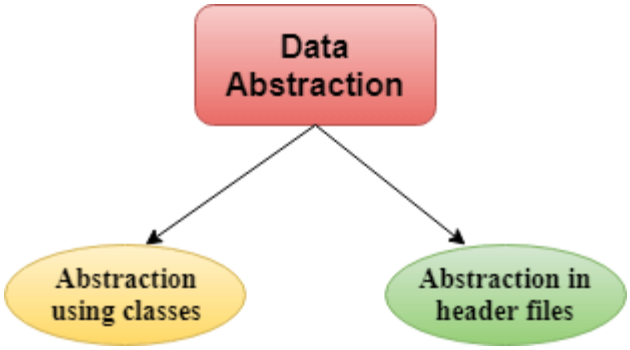



**Figure: Encapsulation**

**Abstraction**

- Abstraction refers to the act of representing essential features without including the background details or explanation.
- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes.
- They encapsulate all the essential properties of the object that are to be created.




**Figuure:Abstraction**

**Inheritance**

- Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of hierarchical classification.
- In OOP, the concept of inheritance provides the idea of *reusability.*
- This means that we can add additional features to an existing class without modifying it.
- This is possible by deriving a new class from the existing one.
- The new class will have the combined feature of both the classes.
- The real appeal and power of the inheritance mechanism is that it

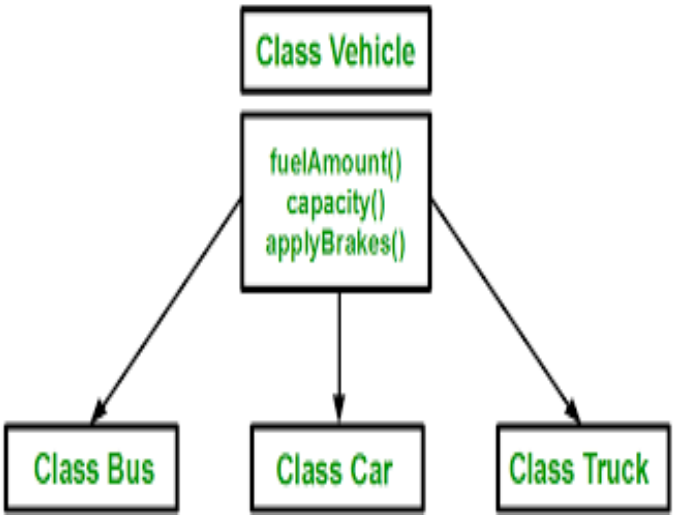



**Figure: Property inheritance**

**Polymorphism**

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

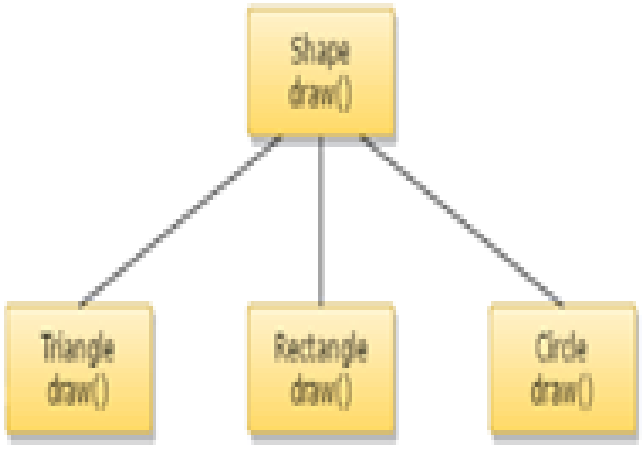



**Figure:Polymorphism**

**Dynamic Binding**

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a

given procedure call is not known until the time of the call at run time.

**MessagePassing**

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

- Creating classes that define object and their behavior,
- Creating objects from class definitions,and
- Establishing communication amongobjects

A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. *Message passing* involves specifying the name of object, the name of the function (message) and the information to be sent.

**Example:**Employee. Salary (name);


**Figure: Example of Message Passing Mechanism**

### 1.6 Benefits ofOOP

OOP offers several benefits to both the program designer and the user. Object-Orientation contributes to the solution of many problems associated with the development and quality of software products.

**Adavantages :**

- Through inheritance, we can eliminate redundant code extend the use ofexisting Classes.
- The principle of data hiding helps the programmer to build secure program that can not be invaded by code in other parts of aprograms.
- It is possible to have multiple instances of an object to co-exist

without any interference.

- It is possible to map object in the problem domain to those in theprogram.
- Object-oriented system can be easily upgraded from small to largesystem.
- Message passing techniques for communication between objects makes to interface descriptions with external systems muchsimpler.
- Software complexity can be easilymanaged.

### 1.7 Object OrientedLanguage

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

- Object-based programming languages,and
- Object-oriented programminglanguages.

**Object-based programming**

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

- Data encapsulation
- Data hiding and accessmechanisms
- Automatic initialization and clear-up ofobjects
- Operator overloading

Languages that support programming with objects are said to the objects-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

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

Object-based features + inheritance + dynamic binding

### 1.8 Applications ofOOP

- Object-oriented technology is certainly going to change the way the software engineers think, analyze, design and implement future system.
- The promising areas of application of OOP include:
- Real-timesystem
- Simulation andmodeling
- Object-oriented data bases
- Hypertext, Hypermedia, andexpertext
- AI and expertsystems

**SUMMARY:-**

- The most popular phase till recently was procedure- oriented programming(pop)
- POP employs top-down programming approach where a problem is viewed as a sequence of tasks to be performed.
- A number of functions are written to implement these tasks.
- In OOP , a problem is considered as a collection of a number of entities called objects are instance are of class.
- Insulation of data from direct access by the program is called data hiding.
- Inheritance is the process by which objects of one class acquire properties of objects of another class.

**Review Questions:-**

1) What is procedure- oriented programming? What are the main characteristics?

2) Discuss an approach to the development of procedure- oriented programs.

3) Describe how data are shared by functions in a procedure oriented program.

4) What is object Oriented programming?how is different from the procedure oriented programming?

5) How are data and functions organized in an object Oriented program?

6) What are unique advantages of an object Oriented programming paradigm?

7) Distinguish between the following terms:

(a)Objects and classes

(b)Data abstraction and data encapsulation

(c)Inheritance and polymorphism

## 2.Begining With C++

### 2.1 Introduction of C++

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. The most important facilities that C++ adds on to C care classes, inheritance, function overloading and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

### 2.2 APLLICATION OF C++

C++ is a versatile language for handling very large programs;it is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life applications systems.

- C++ allow us to create hierarchy related objects, we can build special object-oriented libraries which can be used later by manyprogrammers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get closed to the machine-leveldetails.

- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of anobject.
- It is expected that C++ will replace C as a general-purpose language in the near future.

**2.3 Simple C++Program**

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

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

This simple program demonstrates several C++ features.

**Program feature**

Like C, the C++ program is a collection of function.

The above example contain only one function **main().** As usual execution begins at main(). Every C++ program must have a **main()**.

C++ is a free form language.

With a few exception, the compiler ignore carriage return and white spaces.

Like C, the C++ statements terminate with semicolons.

**Comments**

C++ introduces a new comment symbol // (double slash).

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

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

The C comment symbols /*,*/ are still valid and are more suitable for

multiline comments. The following comment is allowed:

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

**Output operator**

**The statement**

- This statement introduces two new C++ features, cout and <<.
- The identifier cout(pronounced as C out) is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen.

- It is also possible to redirect the output to other output devices.

- The operator << is called the insertion or put to operator.

**The iostreamFile**

- We have used the following #include directive in the program:
- #include <iostream>
- The header file **iostream.h** should be included at the beginning of all programs that use input/outputstatements.

**Namespace**

- Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the **namespace** scope we must include the using directive, like
- Using namespace std;
- All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope.
- **Using** and **namespace** are the new keyword of C++.

**Return Type of main()**

```
main ()
{
   ……..
   ……
}
```

**Figure:More C++Statements**

In C++, main () returns an integer value to the operating system. Therefore, every main () in C++ should end with a return (0) statement; otherwise a warning an error might occur.

Assume that we should like to read two numbers from the keyboard and display their average on the screen.

```
#include<iostream.h> // include header

file Using namespace std;

Int main()

{

        Float number1, number2,sum,
        average; Cin >> number1;  // Read
        Numbers Cin >> number2; // from
        keyboard Sum = number1
        +number2;
        Average = sum/2;
        Cout << ”Sum = “ << sum << “\n”;
        Cout << “Average = “ << average << “\n”;

        Return 0;

}       //end of
```

***The output would be:***

```
Enter two numbers:
6.57.5
Sum = 14
```

**Average of Two Numbers**

**Variables**

The program uses four variables number1, number2, sum and average.

They are declared as type float by the statement.

Float  number1, number2, sum, average;

All variable must be declared before they are used in the program.

**InputOperator**

The statement

cin >> number1;

An input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier cin (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as extraction or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right fig 1.8. This corresponds to a familiar scanf() operation. Like <<, the operator >> can also be overloaded

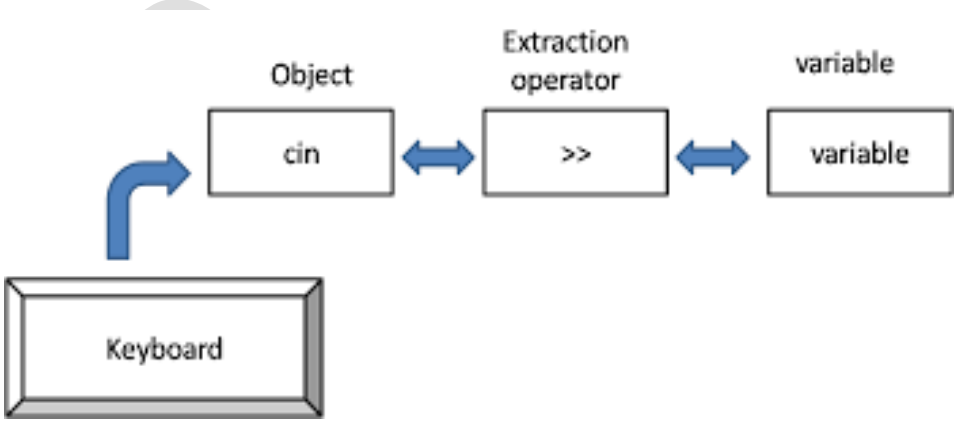



**Figure: Input using extraction operator**

**Cascading of I/OOperators**

It have used the insertion operator << repeatedly in the last two statements for printing results.

The statement

Cout << “Sum = “ << sum << “\n”;

First sends the string “Sum = “ to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined asfollows:

Cout << “Sum = “ << sum << “\n”

<< “Average = “ << average << “\n”;

This is one statement but provides two line of output. If it want only one line of output, the statement will be:

Cout << “Sum = “ << sum << “,”

<< “Average = “ << average << “\n”;

***The output will be*:**

Sum = 14, average = 7

We can also cascade input iperator >> as shown below:

Cin >> number1 >> number2;

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to munber1 and 20 tonumber2.

## 2.6 Structure of C++Program

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

| Include Files |
|---|

| |
|---|
| Class declaration |
| Member functions definitions |
| Main function program |

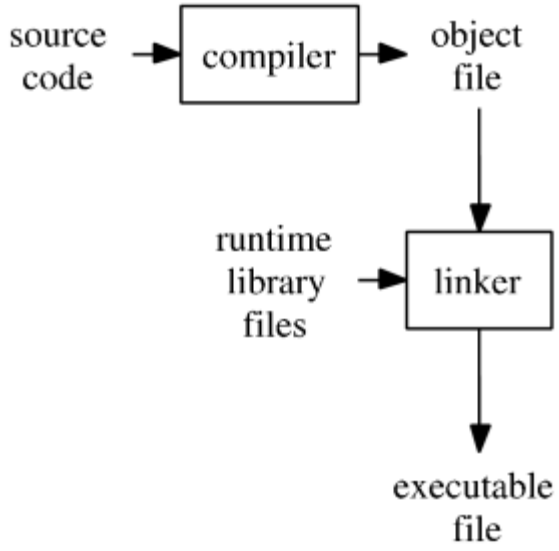
**Figure: Structure of a C++ program**

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

**Reading the SourceFile**

- Like C programs can be created using any text editor. Foe example, on the UNIX, we can use vi or ed text editor for creating using any text editor for creating and editing the source code.
- Some systems such as Turboc C++ provide an integrated environment for developing and editing programs file name should have a proper file extension to indicate that it is a C++ implementations use extensions such as .c,.C, .cc, .cpp and .cxx. Turboc C++ and Borland C++ use .c for C programs and .cpp(C plus plus) for C++ programs.
- Zortech C++ system use .cxx while UNIX AT&T version uses .C (capital C) and .cc.
- The operating system manuals should be consulted to determine the proper file name exCompiling and Linking
- The process of compiling and linking again depends upon the operating system.
- A few popular systems are discussed in this section.
- Unix AT&T C++

**Example**


source
code
compiler
object
file
runtime
library
files
linker
executable
file

SRGPG

### *2.5 CC example*

- At the UNIX prompt would compile the C++ program source code contained in the file **example.C**.
- The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a. out**.

A program spread over multiple files can be compiled as follows:

***C file1.C file2.o***

- The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file. This is useful when only one of the files needs to be modified.
- The files that are not modified need not be compiled again.
- **Turbo C++ and Borland C++**
- Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar includes options such as File, Edit, Compile and Run.
- We can create and save the source files under the **File option,** and edit them under the **Edit option.**
- We can then compile the program under the **compile** option and execute it under the **Run option.** The **Run option** can be used without compiling the source code.

**SUMMARY:-**

- C++ is a superset language
- C++ adds a number of object Oriented features such as objects inheritance Functions overloading and operator Overloading to c.
- C++ can be used to build a variety of systems such as editors,compilers,datatypes,and many more complex real life application systems
- All ANSI c++ programs must include using namespace STD directive.
- A typical c++ program would contain four basic sections,namely ,include files section, class declaration section, member function and section and main Function section

**Review Questions:-**

1) Why do we need the preprocessor directive #include<iostream.h>?

2) How does a main() function in c++differ from main() in c?

3) Object Oriented programming with++

4) What do you think is the main advantage of the comment// in c++ as compared to the old c type comment?

5) Describe the major parts of a c++program

**Debugging Exercises:-**

1)Identify the error in the following program

```
# include <iostream.h>
     Void main( )
    {
   int i=0;
   i = i+ 1;
 cout <<I<<" ";
 /*comment \*//i =I+1;
Cout <<I;
}
```

2) Identify the error in the following program

```
Include<upstream.h>
{
   Short i= 2500, j = 3000;
   Cout >>" I+j = >>-(I+j);
}
```

3) What will happen when you run the following program?

```
# include <upstream.h>
Void main( )
{
Inti = 10;j=5;
Int mod result =0;
Int div result =0;
```

```
Mod result =i%j;
Cout<<modresult <<" ";
Divresult =I/modresult;
Cout<< divResult;
}
```

4) Identify the error in the following program.

```
# include <iostream.h>
Void main ()
{
Short i= 2000, j= 3000;
Cout >>"I + j +=>> -(i+j);
}
```

5) ) Identify the error in the following program

```
# include <iostream.h>
Void main ()
{
Short i= 2000, j= 3000;
Cout >>"I + j +=>> -(i+j);
Div result =i/mod result;
Cout<,<div result ;
}
}
```

**Programming Exercise:-**

1.Write a program to display the following output using a single cout statement

Maths = 90

Physics = 77

Chemistry = 69

2.Write a program to read numbers from the keyboard and display the larger value on the screen.

3.Write a program to read that inputs from keyboard and displays its corresponding ASSCII value on the screen

4.Write a program to read of value a, b and display the value of x where

X=a/b-c

5. ldenticy the error in the following program.

```
#Include <fostream.h>
void main(} (
}
inti=O; 1 • 1 + l;
cout << i << " " . • /•crttrrr.ent\•//I • I + l;
cout << i;
```

6.Identify the error in the following program.

```
linclude <iostream.h>
void main()
{
short 1•2500, J•JOOO;
cout >> "i + j • • >> -(i+j);
}
```

## 3.TOKENS, EXPRESSIONS AND CONTROL STRUCTURES :

### 3 TOKENS :

C++ has the fo11owing tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators
- Datatypes

A C++ program is written using the setokens,whitespaces,and the syntax of the 1anguage.

### 3.4 IDENTIFIERS AND CONSTANTS :

Identifiers refer to the names of variab1es, functions, arrays, c1asses, etc. created by the programmer. They are the fundamenta1 requirement of any 1anguage.Each 1anguage has its ownru1esfornamingtheseidentifiers.

The fo11owing ru1es are common to both C and C++:

- Only alphabetic characters, digits and underscores arepermitted.
- The name cannot start with adigit.
- Uppercase and lowercase letters aredistinct.
- A declared keyword cannot be used as a variablename.

### 3.3 KEYWORDS :

- Theyareexp1icit1yreservedidentifiers and cannot be used as names for the program variab1es or other user-defined program e1ements.

| C and C++ Common Keywords | | | |
|---|---|---|---|
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |

**Table : C & C++ Keywords**

**CONSTANTS :**

Constants refer to fixed va1ues that do not change during the execution of a program.

Theyinc1udeintegers,characters,

f1oatingpointnumbersandstrings.

Litera1constantdonothavememory1ocations.

```
l23             // decima1integer
l2.34           // f1oating pointinteger
037             // octa1integer
0X2             // hexadecima1integer
"C++"           // stringconstant
'A'             // characterconstant
L'ab'           // wide-characterconstant
```

**STRING**

String class stores the characters as a sequence of bytes with a functionality of allowing access to single byte character.

- std:: string vs Character Array.
- Operations on strings.
- Input Functions.
    - getline() :- This function is used to store a stream of characters as entered by the user in the object memory.

C++ string class internally uses char array to store character but all memory management, allocation and null termination is handled by string class itself that is why it is easy to use. The length of c++ string can be changed at runtime because of dynamic allocation of memory similar to vectors. As string class is a container class, we can iterate over all its characters using an iterator similar to other containers like vector, set and maps, but generally we use a simple for loop for iterating over the characters and index them using operator.

C++ string class has a lot of functions to handle string easily. Most useful of them are demonstrated in below code.

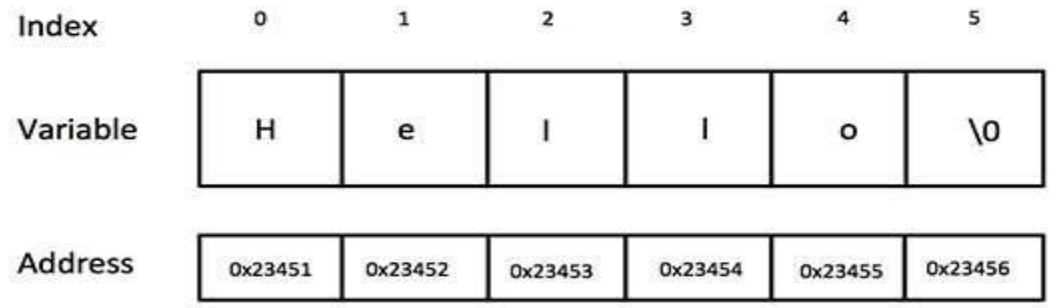
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

It follow the rule of array initialization, then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C++




Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string

**Operator**

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

**Arithmetic Operators**

There are following arithmetic operators supported by C++ language.Assume variable A holds 10 and variable B holds 20, then,

**Show Examples**

| Operator | Description | Example |
|---|---|---|
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| * | Multiplies both operands | A * B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | **Increment operator**, increases integer value by one | A++ will give 11 |
| -- | **Decrement operator**, decreases integer value by one | A-- will give 9 |

**Relational Operators**

There are following relational operators supported by C++ language.Assume variable A holds 10 and variable B holds 20, then,

**Show Examples**

| Operator | Description | Example |
|---|---|---|
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |

| | | |
|---|---|---|
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

**Logical Operators**

There are following logical operators supported by C++ language.Assume variable A holds 1 and variable B holds 0, then,

**Show Examples**

| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. If both the operands are non-zero, then condition becomes true. | (A && B) is false. |
| \|\| | Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true. | (A \|\| B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false. | !(A && B) is true. |

**Bitwise Operators**

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows,

| P | Q | p & q | p \| q | p ^ q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

---

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

**Show Examples**

| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right | A << 2 will give 240 which is 1111 0000 |

| | | |
|---|---|---|
| | operand. | |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |

**Assignment Operators**

There are following assignment operators supported by C++ language.

**Show Examples**

| Operator | Description | Example |
|---|---|---|
| = | Simple assignment operator, Assigns values from right side operands to left side operand. | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand. | C -= A is equivalent to C = C - A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand. | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator, It | C /= A is equivalent to C = C / A |

| | | |
|---|---|---|
| | divides left operand with the right operand and assign the result to left operand. | |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| \|= | Bitwise inclusive OR and assignment operator. | C \|= 2 is same as C = C \| 2 |

**Misc Operators**

The following table lists some other operators that C++ supports.

| Sr.No | Operator & Description |
|---|---|
| 1 | **Sizeof** returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4. |
| 2 | **Condition ? X : Y =Conditional operator (?)**. If Condition is true then it returns value of X otherwise returns value of Y. |
| 3 | **, Comma operator** causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list. |

| | |
|---|---|
| 4 | **. (dot) and -> (arrow) Member operators** are used to reference individual members of classes, structures, and unions. |
| 5 | **Cast Casting operators** convert one data type to another. For example, int(2.2000) would return 2. |
| 6 | **&Pointer operator &** returns the address of a variable. For example &a; will give actual address of the variable. |
| 7 | **Pointer operator *** is pointer to a variable. For example *var; will pointer to a variable var. |

### 3.4 Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

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

**Show Examples**

| Category | Operator | Associativity |
|---|---|---|
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)* & sizeof | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |

| | | |
|---|---|---|
| Shift | <<>> | Left to right |
| Relational | <<= >>= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Comma | , | Left to right |

**3.5 BASIC DATA TYPES :**

C++ compi1ers support a11 the bui1t-indata types. With the exception of void, the asic data types may have severa1 modifier preceding the to serve the needs of various situations.Datatype representation is machine specific in C++. Datatypes in C++ can be classified under various categories as shown in,

**Primitive data types available in C++ are:**

- Integer.
- Character.
- Boolean.
- Floating Point.
- Double Floating Point.
- Valueless or Void.
- Wide Character.

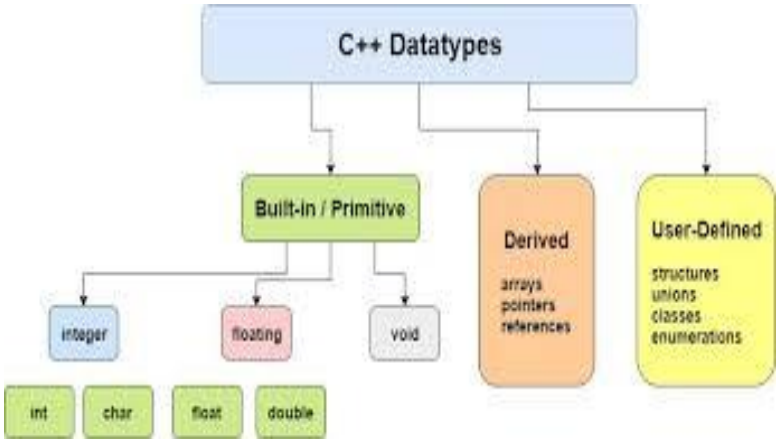



**Figure: Datatypes in C++**

**Table: Size and range of C++ basic datatypes**

| *Type* | *Bytes* | *Range* |
|---|---|---|
| Char | 1 | –128 to 127 |
| unsigned char | 1 | 0 to 255 |
| signed char | 1 | – 128 to 127 |
| Int | 2 | – 32768 to 32767 |
| unsigned int | 2 | 0 to 65535 |
| signed int | 2 | – 31768 to 32767 |
| short int | 2 | – 31768 to 32767 |
| unsigned short int | 2 | 0 to 65535 |
| signed short int | 2 | –32768 to 32767 |
| long int | 4 | –2147483648 to 2147483647 |
| signed long int | 4 | –2147483648 to 2147483647 |
| unsigned long int | 4 | 0 to 4294967295 |
| Float | 4 | 3.4E–38 to 3.4E+38 |
| Double | 8 | 1.7E–308 to 1.7E+308 |
| lomg double | 10 | 3.4E–4932 to 1.1E+4932 |

### 3.6 USER DEFINED DATA TYPES :

**Structures and Classes**

- Used user-defined data types such as strugt and uniom in C.
- C++ a1so permits us to define another user-defined data type known as g1ass which can be used, just 1ike any other basic data type, to dec1are variab1es.
- The c1ass variab1es are known as objects, which are the centra1 focus of object-oriented programming.

**Enumerated Data Type**

- An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibi1ity of the code.
- enum shape{circ1e, square, triang1e}; enum co1our{red, b1ue, green, ye11ow}; enum position{off, on};
- co1our background=b1ue; //a11owed
- co1our background=7; // Error in C++
  co1our ackground =(co1our)7; //0K

### 3.8 Derived Data Types:

**Arrays**

- The only exception is the way character arrays are initia1ized.When initia1izing.
- Character array in ANCIC, the compi1er wi11 a11ow us to dec1are the array size as the exact 1ength of the string constant.For instance,

```
char string[3] = "xyz";
```

**Functions**

Modifications and improvements were driven by the requirements of the object-oriented conceptofC++.TheG++programmorere1iab1eand readab1e.

**Pointers**

Pointers are dec1ared and initia1ized as

```
int*ip;        // intpointer
ip=&x;         // address of x assigned toip
*ip=l0;        // l0 assigned to x throughindirection
```

C++ adds the concept of constant pointer and pointer to a constant.

char * const ptrl="900D"; // constantpointer

It cannot modify the address that ptr1 is initia1ized to.

int const * ptr2 = &m; // pointer to a constant

ptr2 is dec1ared as pointer to a constant.

It can point to any variab1e of correcttype,butthe contents of what it points to cannot bechanged.

**Example program for pointer**

```
#include<iostream>
usingnamespace std;
int main()
{
int*pc, c;
      c =5;
   cout <<"Address of c (&c): "<<&c << endl;
   cout <<"Value of c (c): "<< c << endl << endl;
   pc =&c;// Pointer pc holds the memory address of variable c
   cout <<"Address that pointer pc holds (pc): "<< pc << endl;
   cout <<"Content of the address pointer pc holds (*pc): "<<*pc << endl << endl;
  c =11;// The content inside memory address &c is changed from 5 to 11.
   cout <<"Address pointer pc holds (pc): "<< pc << endl;
   cout <<"Content of the address pointer pc holds (*pc): "<<*pc << endl << endl;
*pc =2;
   cout <<"Address of c (&c): "<<&c << endl;
   cout <<"Value of c (c): "<< c << endl << endl;
return0;
  }
```

**Output :**

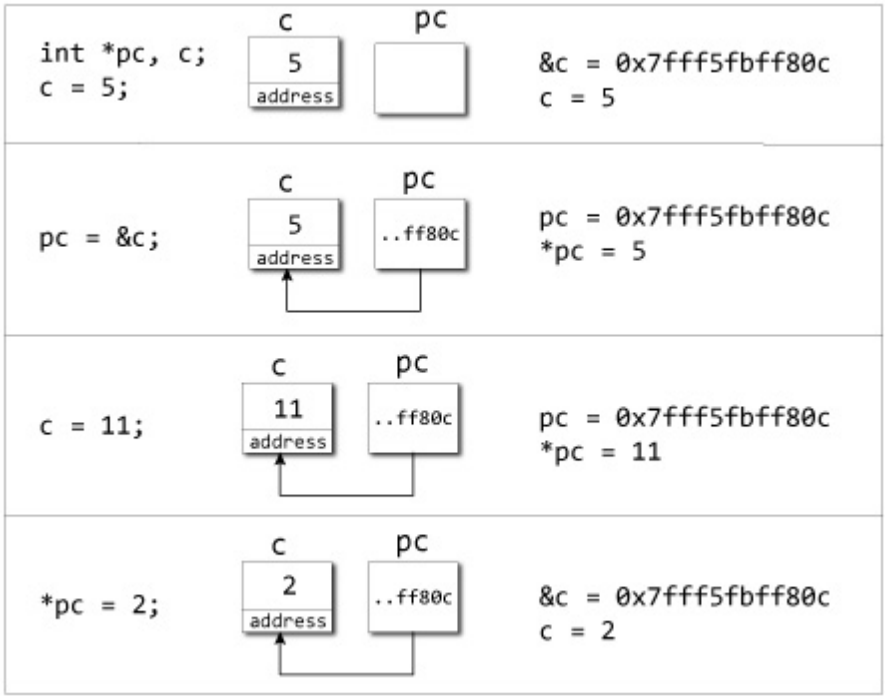
```
Address of c (&c): 0x7fff5fbff80c
Value of c (c): 5
Address that pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 5
Address pointer pc holds (pc): 0x7fff5fbff80c
```

Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7fff5fbff80c

Value of c (c): 2

**Explanation in Diagrammatical View:**




**Symbolic Constant**

There are two ways of creating symbo1ic constants in G++:

- Using the qualifier **const**,and
- Defining a set of integer constantsusing **enum** keyword.

**Syntax:**

constant expression, such as

const int size = l0; char

name[size];

**Example Program for Symbolic constant**

#include <vector>

#include <algorithm>

```
#include <iostream>
int main() {
// a container
   std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// version 1: Range-based for
for (int i : vec) {
       std::cout << i <<' ';
   }
   std::cout <<'\n';
// version 2: std::for_each
   std::for_each(vec.begin(), vec.end(), [](int i){ std::cout << i <<' '; });
return 0;
}
```

### 3.20 Expressions

"**Expression in C++** is form when we combine operands (variables and constant) and **C++** OPERATORS." **Expression** can also be defined as: "**Expression** in C++ is a combination of Operands and Operators." OPERANDS IN **C++** PROGRAM are those values on which we want to perform perform operation.

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

**Constant expressions:**

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

**Integral expressions:**

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

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

**Float expressions:**

The expressions that produce floating-point value as output after performing all types of conversions are called **float expressions.** For example, 9.25, x-y and 9+ float (7) are float expressions. Here, x 'and yare variables of type float.

**Relational or Boolean expressions:**

The expressions that produce a bool type value, that is, either true or false are called **relational or Boolean expressions.** For example, x + y<100, m + n==a-b and a>=b + c .are relational expressions.

**Logical expressions:**

The expressions that produce a bool type value after combining two or more relational expressions are called **logical expressions.** For example, x==5 &&m==5 and y>x I I m<=n are logical expressions.

**Bitwise expressions:**

The expressions which manipulate data at bit level are called **bitwise expressions.** For example, a >> 4 and b<< 2 are bitwise expressions.

**Pointer expressions:**

The expressions that give address values as output are called **pointerexpressions.** For example, &x, ptr and -ptr are pointer expressions. Here, x is a variable of any type and ptr is a pointer.

**Special assignment expressions:**

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

**Chained assignment:**

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

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

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

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

**Embedded assignment:**

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

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

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

**Compound Assignment:**

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

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

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

**Type Conversion**

An expression· may involve variables and constants either of same data type or of different data types. However, when an expression consists of mixed data types then they are converted to the same type while evaluation, to avoid compatibility issues. This is accomplished by type conversion, which is defined as the process of converting one predefined data type into another. Type conversions are of two types, namely, *implicitconversions* and *explicit conversions* also known as *typecasting.*

**Implicit Conversions**

Implicit conversion, also known as automatic type conversion refers to the type conversion that is automatically performed by the compiler. Whenever compiler confronts a mixed type expression, first of all char and short int values are converted to int. This conversion is known as integral promotion. After applying this conversion, all the other operands are converted to the type of the largest operand and the result is of the type of the largest operand. Table illustrates the implicit conversion of data type starting from the smallest to largest data type.

For example, in expression 5 + 4.25, the compiler converts the int into float as float is larger than int and then performs the addition.

**Typecasting**

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

**data_type (expression) //expression in parentheses**

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

where,

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

To understand typecasting, consider this example.

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

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

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

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

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

## 3.25 Control Structure

A program is usually not limited to a **linear** sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

**Conditional structure: *if* and *else***

It is used to execute an instruction or block of instructions only if a condition is fulfilled. Its form is:

**if (***condition***)** *statement*

where *condition* is the expression that is being evaluated. If this condition is **true**, ***statement*** is executed. If it is false, *statement* is ignored (not executed) and the program continues on

the next instruction after the conditional structure.

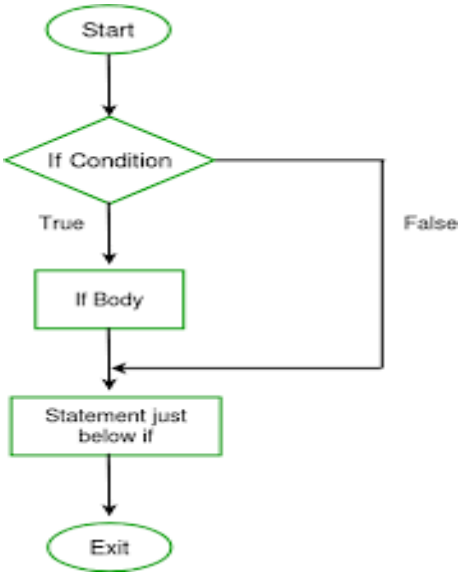



**Figure : Conditional structure: *if* and *else***

For example, the following code fragment prints out **x is 100** only if the value stored in variable **x** is indeed 100:

```
if (x == 100)   cout << "x is 100";
```

If we want more than a single instruction to be executed in case that *condition* is **true** we can specify a *block of instructions* using curly brackets { }:

```
if (x == 100)
 {
 cout << "x is ";
 cout << x;
 }
```

We can additionally specify what we want that happens if the condition is not fulfilled by using the keyword *else*.

Its form used in conjunction with **if** is:

**if (***condition***)** *statement1* **else** *statement2*

For example:

```
if (x == 100)  cout << "x is 100";
else
```

```
cout << "x is not 100";
```

prints out on the screen **x is 100** if indeed x is worth 100, but if it is not -and only if not- it prints out **x is not 100**.

The *if* + *else* structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the present value stored in **x** is positive, negative or none of the previous, that is to say, equal to zero.

```
if (x > 0)
 cout << "x is positive";
else
 if (x < 0)
 cout << "x is negative";
else
cout << "x is 0";
```

Remember that in case we want more than a single instruction to be executed, we must group them in a *block of instructions* by using curly brackets **{ }**.

**Repetitive structures or loops**

*Loops* have as objective to repeat a *statement* a certain number of times or while a condition is fulfilled.

**The *while* loop.**

Its format is:

**while (***expression***)** *statement*

and its function is simply to repeat *statement* while *expression* is true.

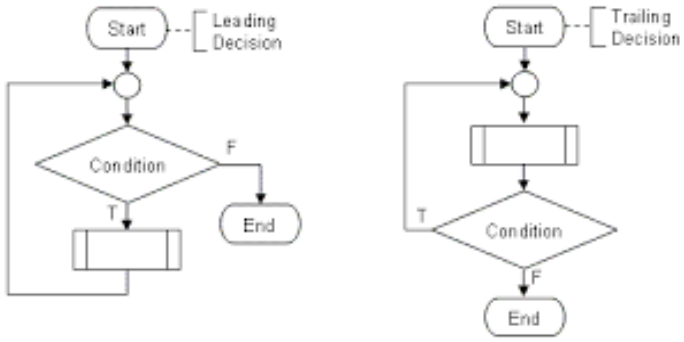



**Figure :Repetitive structures or loops**

For example, they are going to make a program to count down using a *while* loop:

```
// custom countdown using while
#include <iostream.h>
int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "FIRE!";
  return 0;
}
```

**Output**

**Enter the starting number** >8

**8, 7, 6, 5, 4, 3, 2, 1, FIRE!**

When the program starts the user is prompted to insert a starting number for the countdown. Then the *while* loop begins, if the value entered by the user fulfills the condition **n>0** (that **n** be greater than **0** ), the block of instructions that follows will execute an indefinite number of times while the condition **(n>0)** remains true.

All the process in the program above can be interpreted according to the following script: beginning in **main**:

- **1.** User assigns a value to **n**.
- **2.** The while instruction checks if **(n>0)**. At this point there are two possibilities:
  - **true:** execute *statement* (step **3,**)
  - **false:** jump *statement*. The program follows in step **5.**.
- **3.** Execute *statement*:
- cout << n << ", ";
- --n;(prints out **n** on screen and decreases **n** by 1).
- **4.** End of block. Return Automatically to step **2.**
- **5.** Continue the program after the block: print out **FIRE!** and end of program.

We must consider that the loop has to end at some point, therefore, within the block of instructions (loop's *statement*) we must provide some method that forces *condition* to become false at some moment, otherwise the loop will continue looping forever. In this case we have included **--n;** that causes the *condition* to become **false** after some loop repetitions: when **n** becomes **0**, that is where our countdown ends.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without practical delay between numbers.

**The *do-while* loop.**

Format:

**do** *statement* **while (***condition***);**

Its functionality is exactly the same as the *while* loop except that *condition* in the *do-while* is evaluated after the execution of *statement* instead of before, granting at least one execution of *statement* even if *condition* is never fulfilled.

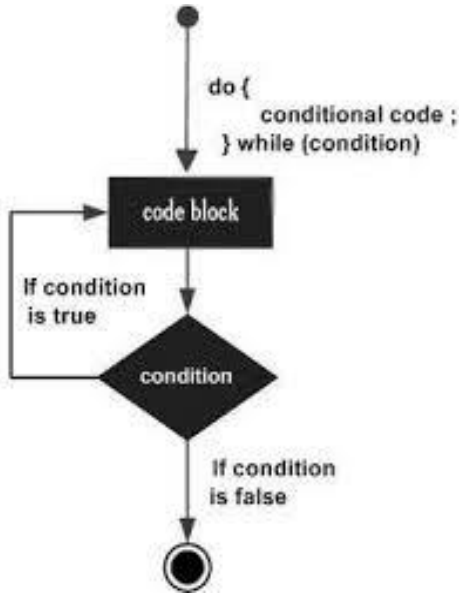



**Figure : do-While loop**

For example, the following program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream.h>
int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

**Output**

**Enter number (0 to end):** 12345
**You entered: 12345**
**Enter number (0 to end):** 160277
**You entered: 160277**
**Enter number (0 to end):** 0
**You entered: 0**

**The *for* loop.**

Its format is:

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

and its main function is to repeat *statement* while *condition* remains true, like the *while* loop. But in addition, **for** provides places to specify an *initialization* instruction and an *increase* instruction. So this loop is specially designed to perform a repetitive action with a counter.

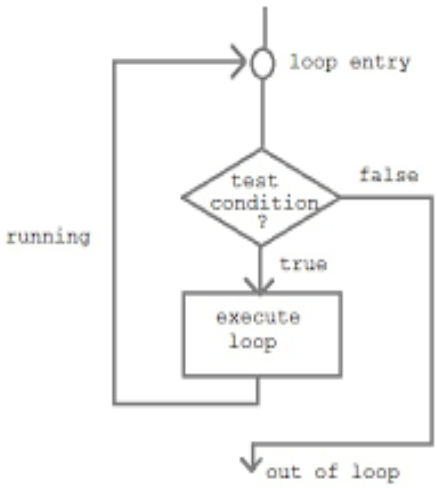



**Figure: For loop**

It works the following way:

1, *initialization* is executed. Generally it is an initial value setting for a counter varible. This is executed only once.

2, *condition* is checked, if it is **true** the loop continues, otherwise the loop finishes and *statement* is skipped.

3, *statement* is executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets **{ }**.

4, finally, whatever is specified in the *increase* field is executed and the loop gets back to step 2.

Here is an example of countdown using a *for* loop.

```
// countdown using a for loop
#include <iostream.h>
int main ()
```

```
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!";
  return 0;
}
```

**Output**

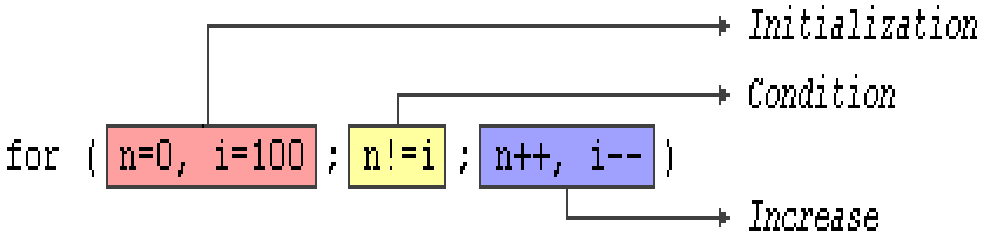
**10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE**

The *initialization* and *increase* fields are optional. They can be avoided but not the semicolon signs among them. For example we could write: **for (;n<10;)** if we want to specify no *initialization* and no *increase*; or **for (;n<10;n++)** if we want to include an *increase* field but not an *initialization*.

Optionally, using the comma operator (**,**) we can specify more than one instruction in any of the fields included in a **for** loop, like in *initialization*, for example. The comma operator (**,**) is an instruction separator, it serves to separate more than one instruction where only one instruction is generally expected. For example, suppose that we wanted to intialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
// whatever here...
}
```

This loop will execute 50 times if neither **n** nor **i** are modified within the loop:



**n** starts with **0** and **i** with **100**, the condition is **(n!=i)** (that **n** be not equal to **i**). Beacuse **n** is increased by one and **i** decreased by one, the loop's condition will become false after the 50th loop, when both **n** and **i** will be equal to 50.

**Bifurcation of control and jumps.**

**The *break* instruction.**

Using *break* we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.

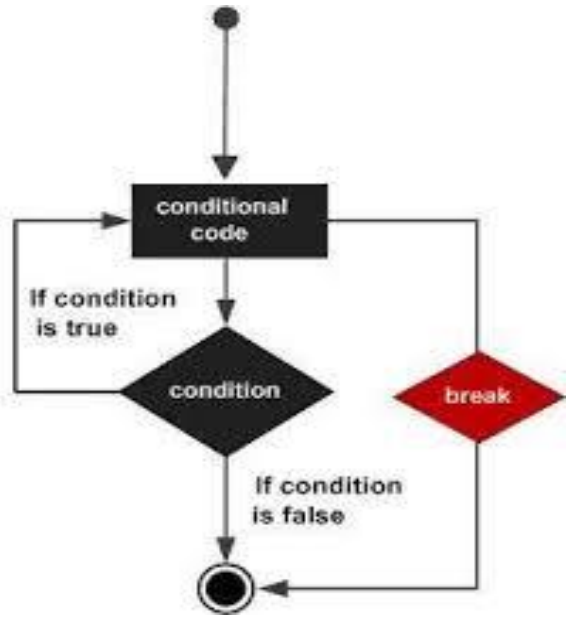



**Figure: Bifurcation of control and jumps**

For example, we are going to stop the count down before it naturally finishes (an engine failure maybe):

```
// break loop example
#include <iostream.h>
int main ()
{
  int n;
  for (n=10; n>0; n--) {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
```

```
    break;
   }
  }
  return 0;
}
```

**Output**

**10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!**

**The *continue* instruction.**

The *continue* instruction causes the program to skip the rest of the loop in the present iteration as if the end of the *statement* block would have been reached, causing it to jump to the following iteration.

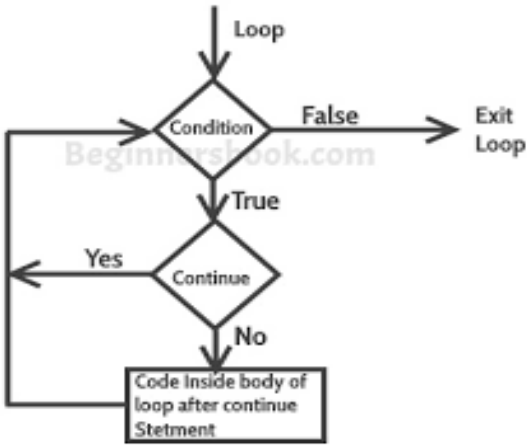



**Figure: Continue instruction**

For example, we are going to skip the number 5 in our countdown:

```
// break loop example
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!";
```

```
  return 0;
}
```

**Output**

**10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!**

**The *goto* instruction.**

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation.

The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (**:**).

This instruction does not have a concrete utility in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using **goto**:

```
// goto loop example
#include <iostream.h>
int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!";
  return 0;
}
```

**Output**

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

**The *exit* function.**

*exit* is a function defined in **cstdlib** (stdlib.h) library. The purpose of *exit* is to terminate the running program with an specific exit code. Its prototype is:

**void exit (int** *exit code***);**

The *exit code* is used by some operating systems and may be used by calling programs. By convention, an *exit code* of 0 means that the program finished normally and any other value means an error happened.The selective Structure: *switch*.

The syntax of the *switch* instruction is a bit peculiar.

Its objective is to check several possible constant values for an expression, something similar to what we did at the beginning of this section with the linking of several *if* and *else if* sentences.

Its form is the following:\

**switch (***expression***) {**
**case** *constant1***:**
*block of instructions 1*
**break;**
**case** *constant2***:**
*block of instructions 2*
**break;**
.
.
.
**default:**
*default block of instructions*
**}**

It works in the following way: **switch** evaluates *expression* and checks if it is equivalent to *constant1*, if it is, it executes *block of instructions 1* until it finds the **break** keyword, then the program will jump to the end of the *switch* selective structure.

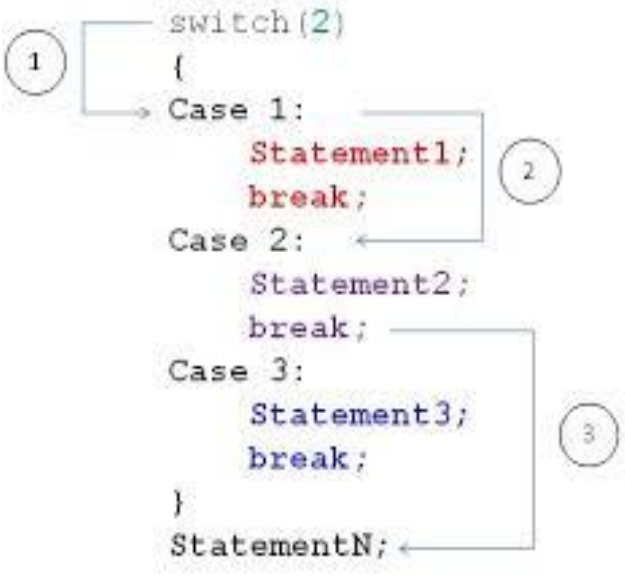



**Figure: Switch statement**

If *expression* was not equal to *constant1* it will check if *expression* is equivalent to *constant2*. If it is, it will execute *block of instructions 2* until it finds the **break** keyword. Finally, if the value of *expression* has not matched any of the previously specified constants (you may specify as many **case** sentences as values you want to check), the program will execute the instructions included in the **default:** section, if this one exists, since it is optional. Both of the following code fragments are equivalent:

***switch* example**

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

***if-else* equivalent**

```
if (x == 1) {
  cout << "x is 1";
  }
else if (x == 2) {
  cout << "x is 2";
  }
else {
  cout << "value of x unknown";
  }
```

This is necessary because if, for example, we did not include it after *block of instructions 1* the program would not jump to the end of the switch selective block (**}**) and it would continue executing the rest of the blocks of instructions until the first appearance of the **break** instruction or the end of the switch selective block. This makes it unnecessary to include curly brackets **{ }** in each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression evaluated.

**For example:**

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

**Summary:-**

- C++ provides various types of tokens that include keywords, identifiers, constants, strings and operators.
- Identifiers refer to the names of variables, Functions, arrays, classes ,etc…
- C++ provides an Additional use of void, for declaration of generic pointers.
- In c++, the size of character array should Pointers are widely used in c++ for memory management and to achieve polymorphism
- C++ provides a qualifier called const to declare named constants which are just like variables except that their values can not be changed. A const modifier defaults to an int.
- C++ is very strict regarding type checking of variables.
- C++ allows us to declare a variable anywhere I the program as also it's initialization at runtime, using the expressions at the place of declaration.
- C++ also provides manipulators to format the data display.
- The most commonly buses manipulators are endl and setw

**Review Question:-**

1)what is a reference variable?what is it's major use ?

2) List at least four few operators added by c++ which aid oop

3) what is the application of the scope resolution operator::in c++?

4) what are the advantage s of using new operator as compared to the function malloc()?

5) which manipulate is used to control the precision of floating point numbers?

6) lllustrate with an example,how the setw manipulator works.

7) how do the following statement s differ?

(a)char*const p;

(b)char constant*p;

8) Describe the different styles of writing prototypes.

9) Find errors,if any ,in the following function prototypes.

(a) float average(x,y)

(b)int mul (int a,b);

(C)int display (….);

10) What is the main advantage of passing arguments by reference?

What is most. Significant advantage that you see in using reference s instead of pointers?

## 4 FUNCTION IN C++:

### 4.1 Introduction:-

- The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program's execution.
- Functions help to reduce the program size when same set of instructions are to be executed again and again. A general function consists of three parts, namely, function declaration (or prototype), function definition and function call.

### 4.2 Function declaration — prototype:

- A function has to be declared before using it, in a manner similar to variables and constants. A function declaration tells the compiler about a function's name, return type, and parameters and how to call the function. The general form of a C++ function declaration is as follows:
- return_type function_name( parameter list );

**Function definition**

The function definition is the actual body of the function. The function definition consists of two parts namely, function header and function body.

The general form of a C++ function definition is as follows:

return_type function_name( parameter list )

{ body of the function }

Here,

**Return Type**: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

**Function Name**: This is the actual name of the function.

**Parameters**: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

**Function Body**: The function body contains a collection of statements that define what the function does.

**Calling a Function**

To use a function, you will have to call or invoke that function. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

A c++ program calculating factorial of a number using functions

```
#include<iostream.h>
#include<conio.h>
intfactorial(intn);
//function declaration
 int main()
{
int no, f;
cout<<"enter the positive
number:-"; cin>>no;
  f=factorial(no);                    //function
  call cout<<"\nThe factorial of a
  number"<<no<<"is"<<f; return0;
  }
  intfactorial(intn)          //functiondefinition
  {          int  i  ,
  fact=1;
  for(i=1;i<=n;i+
  +){
  fact=fact*i;
  }
  return fact;
  }
```

**Inline Functions**

An inline function is a function that is expanded inline at the point at which it is invoked, instead of actually being called.

The reason that inline functions are an important addition to C++ is that they allow you to create very efficient code.

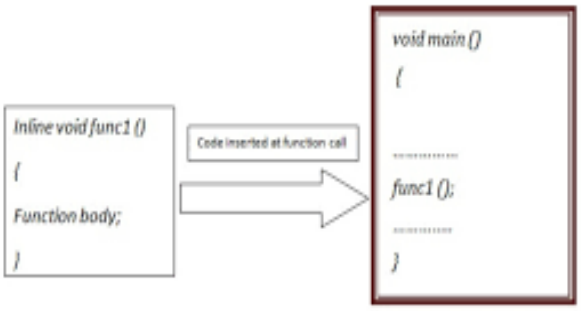



**Figure: Inline Function**

Each time a normal function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns.

A function can be defined as an inline function by prefixing the keyword inline to the function header as given below:

inline function
header
{
function body
}
// A program illustrating inline
function #include<iostream.h>
#include<conio.h>
inline int max(int x,
int y){ if(x>y)
retur
n x;
else

```
}
int
main( )
{
int a,b;
cout<<”enter two
numbers”;
cin>>a>>b;
cout << "The max is: " <<max(a,,b)
<< endl; return 0;
}
```

**Macros Vs inline functions**

- Inline functions follow all the protocols of type safety enforced on normalfunctions.
- Inline functions are specified using the same syntax as any other function except that they include the inline keyword in the functiondeclaration.
- Expressions passed as arguments to inline functions are evaluatedonce.
- In some cases, expressions passed as arguments to macros can be evaluated more thanonce.
- Macros are expanded at pre-compile time, you cannot use them for debugging, but you can use inlinefunctions.

**Reference variable**

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. To declare a reference variable or parameter, precede the variable's name with the &.

The syntax for declaring a reference variable is:

data type &Ref = variable name;

**Example program for Reference Variable**

```
/*C++ program to demonstrate use of reference variable.*/
#include <iostream>
usingnamespace std;
int main()
```

```
{
int a=10;
    /*reference variable is alias of other variable,  It does not take space in memory*/
int&b = a;
cout << endl << "Value of a: " << a;
cout << endl << "Value of b: " << b << endl;
return 0;
}
```

**Output**

Value of a=10

Value of b=10

**Call by reference**

Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function.Call-by-reference passes the address of the argument to the function.By default, C++ uses call-by-value.

**Example**

```
#includeeostrea#include<conio.h>
void swap(int &x, int &y); // function
declaration int main (){
int a = 10, b=20;
cout << "Before swapping”<<endl;
cout<< “value of a :" << a <<” value of b :" <<
b << endl; swap(a,b);  //calling a function to
swap the values. cout <<
"Afterswapping”<<endl;
cout<<” value of a :" << a<< “value of b :" <<
b << endl; return0;
}
void swap(int &x, int&y){        //function definition to swap
thevalues. inttemp;
```

```
temp =
x; x =
y;
y = temp;
}
```

**Output:**

Beforeswapping value of a:10 value of b:20 Afterswapping value of a:20 value ofb:10

### 4.7 Default arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function.

The default value is specified in a manner syntactically similar to a variable initialization.

All default parameters must be to the right of any parameters that don't have defaults.

**Example**

```
#include<iostream.h <iostream.h>#include<conio.h>
int sum(int a, int
b=20){ return( a
+ b);
}
int main (){
int a = 100, b=200, result;
result =sum(a, b);          //here  a=100
, b=200 cout << "Total value is :" <<
result <<endl;
result=sum(a);              //here  a=100  ,  b=20(using
defaultvalue) cout << "Total value is :" << result
<<endl;
return 0;
}
```

Summary:-

- It is possible to reduce the size of program by calling and using Functions at different places.
- Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.

- When a Function is declared inline the complier replaces the Function call with the respective Function code.
  - Reference variables In c++ permit us to pass parameters to the Functions by reference.
  - A function can also return a reference to a variable.
  - The complier may ignore the inline the complier replaces the function call with the respective complicated and hence complier the Functions as a normalfunction.
  - In c++ , arguments to a Function can be declared as const, indicating that the Function should not modify the arguments.
  - C++ supports two new types of Functions, namely friend Functions and virtual Functions.

**Review Question:-**

1) write a c++ program that will ask for a temperature in Fahrenheit and display it in Celsius.
2) Redo excecise 24 using a class called temp and member functions
3) Enumerate the rules of naming variables in c++ .how do they differ from ANSIC rules?
4) An unsigned int can be twice as large as the signed int.Explain how?
5) why does c++ have type modifiers?

## UNIT –III

**OBJECTIVE:-**

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated memory is allocated.A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.Default constructor is the constructor which doesn't take any argument.

## 5.1 INTRODUCTION:-

The most important feature of c++ is the “class “ . Its significance is highlighted by the fact that stroustrup intialy gave the name “ c with class “ th this new language. A class is an extension of they idea of structure used in c. It is a new way of creating and implementaing a user defined data type.

## 5.2 C STRUCTURE REVISITED

To known that one of the unique feature of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user define data type with a template that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declaration that are similar to the built in type declaration.

For example, consider the following declaration:

```
struct student
{
charname[20];
introll_number;
floattotal_marks;
};
```

The keyword struct declare student as a new data type that can hold three fields of different data types. These field are known as structure member or elements. The identifier student, which is referred to as structure name or structure tag, can be used to create variables of type student.

Example:

```
struct  student A; // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variable can be accessed using the dot or period operator as fallows:

```
strcpy(A.name, "John");
```

A.roll_number = 999;

A.total_marks = 595.5;

final_total = A.total_marks + 5;

Structure can have arrays, pointer or structure as members.

**5.3 SPECIFYING THE CLASS:-**

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, to are creating a new abstract data type that can be treated like any other build-in data type.

Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type scope of its members. The class function definitions describe how the class functions are implemented.

**The general form of a class declaration is:**

```
class class_name
{
private:
variable declaration;
function declaration;
public:
variable declaration;
function declaration;
};
```

**5.4 Definning a member function:-**

Member functions of a class can be defined either outside the class definition or inside the class definition. **In** both the cases, the function body remains the same, however, the function header is different.

The definition of member function outside the class differs from normal function

definition, as the function name in the function header is preceded by the class name and the scope resolution operator (: :). The scope resolution operator informs the compiler what class the member belongs to. The syntax for defining a member function outside the class is

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

To understand the concept of defining a member function outside a class, consider this example.

Example : Definition of member function outside the class

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

Note that the member functions of the class can access all the data members and other member functions of the same class (private, public or protected) directly by using their names. In addition, different classes can use the same function name.

Inside the Class: A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope

resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

To understand the concept of defining a member function inside a class, consider this example.

Example : Definition of a member function inside a class

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

In this example, the member function putdata() is defined inside the class book. Hence, putdata()is by default an inline function.

Note that the functions defined outside the class can be explicitly made inline by prefixing the keyword inline before the return type of the function in the function header. For example, consider the definition of the function getdata().

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

**5.3 Specifying a class**

- A class is way to bind the data and its associated functions together.
- It allows the data to be hidden, if necessary,from external use.

- When defining a class ,we are creating a new class abstract data type that can be treated light any other built in data type.
- Generally, a class Specification has two parts:
    1. Class declaration
    2. Class function definitions

**The general of a class declaration is:**

```
Class Class_ name
{
Private:
   Variable declarations;
    Functions declarations;
Public
     Variable declarations;
    Functions declarations;
};
```

A Simple class example:-

```
 class Test
{
   private:
      int data1;
      float data2;
   public:
      void function1()
      {   data1 = 2;  }
      float function2()
      {
         data2 = 3.5;
         return data2;
      }
  };
```

RGPI

### 5.4 Definning member functions:-

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

**Outside the class definitions:-**

- Member functions that are declared inside a class have to be defined separately outside the class.
- Their definitions are very much like the normal functions.
- They should have a function header and a function body.
- Since C++ does not support the old version of function definition, the ANSI
- Prototype form must be used for defining the function header.

**The general form of a member function definition is:**

```
Return – type Class – name::
{
  Function body
}
```

**Inside the class definition**

Another method of difinning a member function is to replace the function declaration by the actual function definition the class .For example , we could the item class as follows:

```
    Class item;
  {
       Int number;
      Float cost;
    Public:
       Void getdata( int a, float b);   // declaration
```

```
        // inline function
        Void putdata (void)          // definition inside the class
    Void putdata (void)
  {
    Cout << number <<"\n";
    Cout<< cost << '\n";
}
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restricitions and limitations that apply to an inline function are also applicate here. Normally, only small functions are defined inside the class definition.

**CLASS AND OBJECT :**

**Classes :**

- A class is a user defined data type.
- A class is a logical abstraction.
- It is a template that defines the form of an object.
- A class specifies both code and data. It is not until an object of that class has been created that a physical representation of that class exists in memory.

**SYNTAX:**

```
class class-name
{
access-specifier:
data and functions
access-specifier:
data and functions
// ...
```

```
access-specifier:
data and functions
}
```

**Example:**

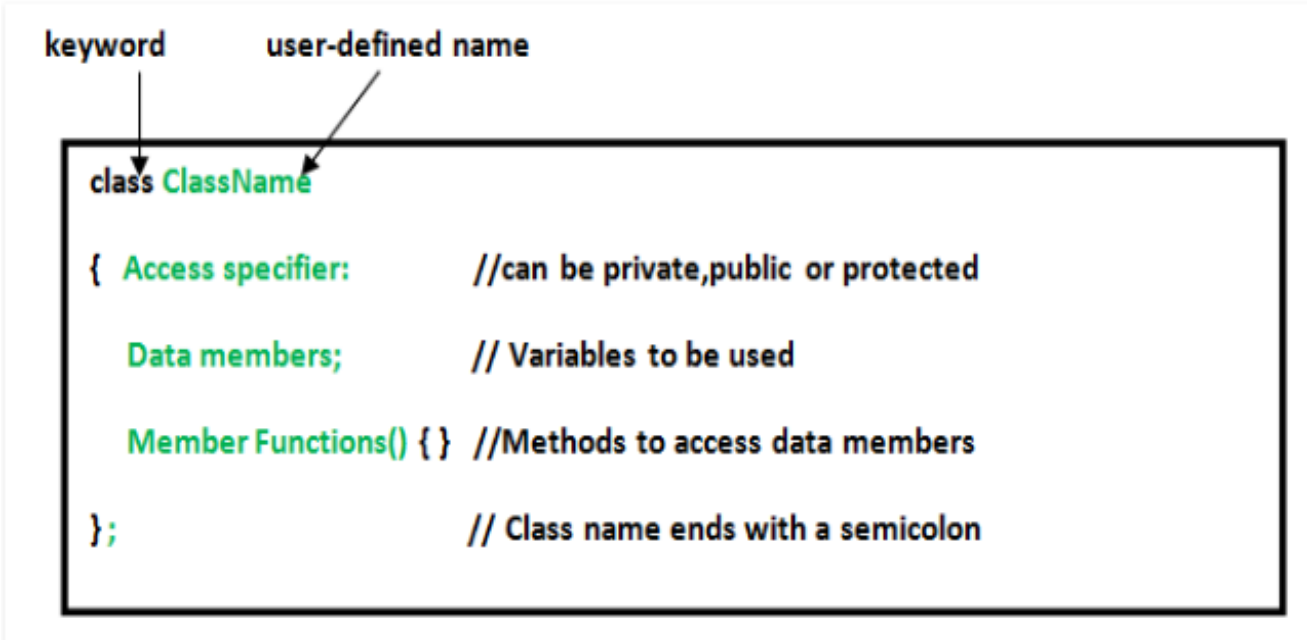
```
classTest
{
private:
int data1;
float data2;
public:
void function1()
{   data1 =2;}
float function2()
{
        data2 =3.5;
return data2;
}
};
```

**OBJECT**

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

**Defining Class and Declaring Object**

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.




**Declaring Objects:** When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

**Syntax:**

Classname Objectname;

**Accessing data members and member functions**: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

**Accessing Data Members**

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++.

There are three access modifiers :

- **Public**

- **Private**
- **protected**.

**Example program for Object**

```
#include <bits/stdc++.h>
using namespace std;
class Geeks
{
   // Access specifier
   public:

  // Data Members
   string geekname;
   // Member Functions()
   void printname()
   {
      cout << "Geekname is: " << geekname;
   }
};
int main() {
   // Declare an object of class geeks
   Geeks obj1;
   // accessing data member
   obj1.geekname = "Abhi";
 // accessing member function
   obj1.printname();
   return 0;
}
```

**Output**

Geekname is : Abhi

**SUMMARY:-**

- A class is an extension to the structure data type.
- A class can have both variables and Functions as members.
- By default, members of the class are private where as that of structure are public.
- In c++ , the class variables are called objects.
- A static member variable must be defined outside the class
- C++ allows us to have arrays of objects.
- A Function can also return an object
- It is also possible to define and use a class inside Function.

**Review Question:-**

1) How do structures in c and c++ differ?

2) What is class?how does it accomplish data hiding?

3) How does a c++structure differ from a c++class?

4)What are objects?How are they created?

5)How is a member function of a class defined?

6)Can we use the same function name for a member function of a class and an outside function in the same program file?

7) How does a C++ structure differ from a C++ class?

8) What are objects? How are they created?

# 6 CONSTRUCTORS AND DESTRUCTORS

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class**.**

constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

## 6.1 Introduction:-

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

## 6.2 Constructor

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.

**How constructors are different from a normal member function?**

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

**Types of Constructors**

1.**Default Constructors:** Default constructor is the constructor which doesn't take any argument.

**Example:-**

```
#include <iostream>
using namespace std;
class construct
 {
public:
    int a, b;
    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};
int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
<< "b: " << c.b;
    return 1;
}
```

Output:

a: 10

b: 20

**6.3 .Parameterized Constructors**: **It is possible to pass arguments to constructors.**

Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other

function. When you define the constructor's body, use the parameters to initialize the object.

**Example Program:-**

```
#include <iostream>
using namespace std;
class Point {
private:
   int x, y;
public:
   // Parameterized Constructor
   Point(int x1, int y1)
   {
      x = x1;
      y = y1;
   }
   int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
};
int main()
{
   // Constructor called
   Point p1(10, 15);
   // Access values assigned by constructor
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
   return 0;
}
```

**Output:**

p1.x = 10, p1.y = 15

Copy Constructor:-

**When an object is declared in actor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.

**Example program:-**

```
#include "iostream"
using namespace std;
class point {
private:
  double x, y;
public:
  // Non-default Constructor & default Constructor
  point (double px, double py) {
    x = px, y = py;
  }
};
int main(void) {
  // Define an array of size 10 & of type point
  // This line will cause error
  point a[10];
  // Remove above line and program will compile without error
  point b = point(5, 6);
}
```

**Output:**

Error: point (double px, double py): expects 2 arguments, 0 provided

**Destructor:-**

A destructor destroys an object after it is no longer in use. The destructor, like constructor, is a member function with the same name as the class name. But it will be

preceded by the character Tilde (~).A destructor takes no arguments and has no return value. Each class has exactly one destructor. .

**Example program for Destructor**

```
#include<iostream>
usingnamespace std;
class Marks
{
public:
int maths;
int science;  //constructor
  Marks() {
 cout << "Inside Constructor"<<endl;
 cout << "C++ Object created"<<endl;
} //Destructor
~Marks() {
 cout << "Inside Destructor"<<endl;
  cout << "C++ Object destructed"<<endl;
  }};
int main( )
{
 Marks m1;
 Marks m2;
return 0;
}
```

**Output**

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

Inside Destructor

C++ Object destructed

Inside Destructor

C++ Object destructed

**SUMMARY:-**

- C++ provides a special member Function called the constructor which enables on object.
- A constructor has same name as that of class.
- A constructors are normally used to initialized variables and to allocated memory.
- Similar to normal Functions , constructor may be overloaded.
- When an object is created and initialized at the same time, a copy constructor gets called.

**Review Questions:-**

1) What is the need of constructor and destructor in C++?

2)What is constructor in C++ and its types with examples?

3)What are constructors in C++?

4)What is a destructor C++?

5)What is difference between constructor and destructor in C++

6)What is operator overloading

7) Why is the necessary to overload an operator?

8) Name the Operator Than cannot be overload in c+=

9) What is an operator function

## 7 OPERATOR OVERLOADING AND TYPE CONVERSION

### 7.1 Introduction:-

The new operator denotes a request for memory allocation on the Heap.
If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- **Syntax to use new operator**: To allocate memory of any data type, the syntax is:
- pointer-variable = **new** data-type;

**Example:**

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;
        OR
// Combine declaration of pointer
// and their assignment
int *p = new int;
```

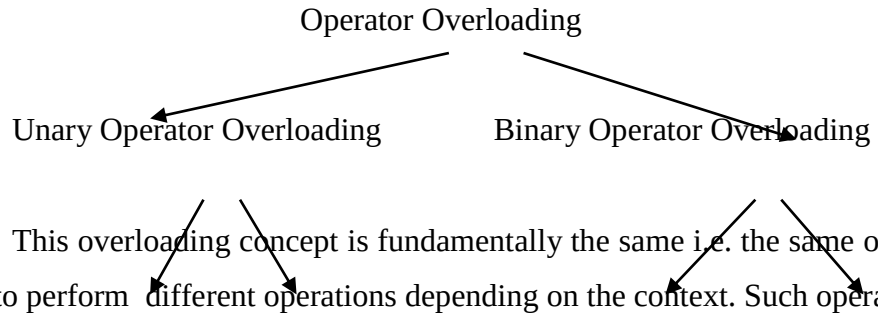
- **Initialize memory:** We can also initialize the memory using new operator:
  pointer-variable = **new** data-type(value);
  **Example:**
  int *p = new int(25);
  float *q = new float(75.25);
- **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type *data-type*.
  pointer-variable = **new** data-type[size];
  where size(a variable) specifies the number of elements in an array.
  Example:
  int *p = new int[10]

### 7.2 Operator overloading

There is another useful methodology in C++ called operator overloading. The language allows not only functions to be overloaded, but also most of the operators, such as +, -, *, /, etc. As the name suggests, here the conventional operators can be

programmed to carry out more complex operations.

The Operator Overloading structure are;




This overloading concept is fundamentally the same i.e. the same operators can be made to perform different operations depending on the context. Such operators have to be specifically defined and appropriate function programmed. When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is defined. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack.

An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword operator.

The general form of an operator function is

type classname::operator#(arg-list)

{ // operations

}

Here, the operator that you are overloading is substituted for the #, and type is the type of value returned by the specified operation. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are often friend functions of the class.

These operators cannot be overloaded:- ., : :,

.*,?

**Example program for Unary Operator Overloading**

#include<iostream>

usingnamespace std;

classTest

{

```
private:
int count;
public:
Test(): count(5){}
voidoperator++()
{
 count = count+1;
}
voidDisplay(){ cout<<"Count: "<<count;}
};
int main()
{
Test t;
// this calls "function void operator ++()" function
++t;
   t.Display();
return0;
}
```

**7.4 Overloading a Binary operator**

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

**Example program for Binary operator overloading**

```
#include <iostream>
using namespace std;
 class Box {
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box
    public:
```

```
   double getVolume(void) {
     return length * breadth * height;
   }
   void setLength( double len ) {
     length = len;
   }
   void setBreadth( double bre ) {
     breadth = bre;
   }
   void setHeight( double hei ) {
     height = hei;
   }
   // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
     Box box;
     box.length = this->length + b.length;
     box.breadth = this->breadth + b.breadth;
     box.height = this->height + b.height;
     return box;
   }
};

// Main function for the program
int main() {
   Box Box1;          // Declare Box1 of type Box
   Box Box2;          // Declare Box2 of type Box
   Box Box3;          // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
```

SRGPGPI

```
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);
   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
    // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   // Add two object as follows:
   Box3 = Box1 + Box2;
   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;
   return 0;
}
```

**Output:**

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

### 7.9 TYPE CONVERSION :

The process of converting one predefined type into another is called as type conversion. When constants and variables of different types are mixed in an expression, they are converted to the same type. When variables of one type are mixed with variables of another type, a type conversion will occur.

C++ facilitates the type conversion into the following two forms :

- Implicit Type Conversion
- Explicit Type Conversion

**Implicit Type Conversion**

An implicit conversion is applied generally whenever differing data types are intermixed in an expression (mixed mode expression), so as not to lose information.

The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable).

Therefore, the types of right side and left side of an assignment should be compatible so that type conversion can take place. The compatible data types are mathematical data types i.e., char, int ,float, double. For example, the following statement :

ch = x ;  (where ch is char and x is int)

The C++ compiler converts all operands upto the type of the largest operand, which is called type promotion. This is done operation by operation, as described in the following type conversion algorithm :

- If either operand is of type long double, the other is converted to long double.
- Otherwise, if either operand is of type double, the other is converted to double.
- Otherwise, if either operand is float, the other is converted to float.
- Otherwise, the integral promotions are performed on both operands. The process of integral promotion is described below :
  A char, a short int, enumerator or an int (in both their signed and unsigned varieties) may be used as an integer type. If an int can represent all the values of the original type, the value is converted to int ; otherwise it is converted to unsigned int. This process is called integral promotion.
- Then, if either operand is unsigned long, the other is converted to unsigned long.
- Otherwise, if one operand is a long int and the other unsigned int, then if a long int can represent all the values of an unsigned int, the unsigned int is converted to long int; otherwise both operands are converted to unsigned long int.
- Otherwise, if either operand is long, the other is converted to long.

When converting from integers to characters and long integers to integers, the appropriate amount of high-orders bits (depending upon target type's size) will be removed. In many environments, this means that 8 bits will be lost when going from an integer to a character and 16 bits will be lost when going from a long integer to an integer.

The following table summarizes the assignment type conversions :

| Common Type Conversion (assuming a 16-bit word) | | |
|---|---|---|
| **Target Type** | **Expression Type** | **Possible Info Loss** |
| signed char | Char | if value > 127, target is negative |
| char | short int | High-order 8 bits |
| char | Int | High-order 8 bits |
| char | long int | High-order 8 bits |
| int | long int | High-order 16 bits |
| int | Float | Fractional part and possibly more |
| float | double | Precision, result rounded |
| double | long double | Precision, result rounded |

**Explicit Type Conversion :**

The explicit conversion of an operand to a specific type is called type casting. An explicit type conversion is user-defined that forces an expression to be of specific type.
This is the general form to perform type casting in C++

(type) expression

where type is a valid C++ data type to which the conversion is to be done. For example, to make sure that expression (x + y/2) evaluates to type float, write it as :

(float) (x + y/2)

**SUMMARY:-**

- Operator Overloading vis one of the important features of c++ language that enhances it's exhaustibility.
- Using overloading feature we can add two user defined data types such as objects, with the same, syntax just as basic data types
- We can overload almost all the c++ operators except the following:
    - Class member access operators (., .*)
    - Scope Resolution operator(::)
    - Size operator ( size of)
    - Conditional operator (?:

**Review Questions:-**

1) describe the syntax of an operator function?

2) How many arguments are request in the definition of an overload unary operator?

3) List Overloading in C++

4)Types of overloading in C++

5)Why is operator overloading used?

6)Implementing Operator Overloading in C++

7)Types of overloading approaches

8) Overloading Unary Operators

9) Overloading Binary Operatorof C++ Operator Overloading programs

10) describe the differences in the implementation of enum data type in ANSIC and c++

11) describe with, example s,the uses the uses of enumeration data types.

12)why is an array called a dervird data type ?

## 8. INHERITANCE

### 8.1 Introduction:-

This also provides an opportunity to reuse the code functionality and fast implementation time.When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

### 8.2 Definning a derived class

Any class type (whether declared with class-key class or struct) may be declared as derived from one or more base classes which, in turn, may be derived from their own base classes, forming an inheritance hierarchy.

The list of base classes is provided in the base-clause of the class declaration syntax. The base-clause consists ofthe character : followed by a comma-separated list of one or more base-specifiers.

If access-specifier is omitted, it defaults to public for classes declared with class-key struct and to private for classes declared with class-key class.

```
struct Base {
   int a, b, c;
};
// every object of type Derived includes Base as a subobject
struct Derived : Base {
   int b;
};
// every object of type Derived2 includes Derived and Base as subobjects
struct Derived2 : Derived {
   int c;
};
```

The classes listed in the base-clause are direct base classes. Their bases are indirect base classes. The same class cannot be specified as a direct base class more than once, but the same class can be both direct and indirect base class.

Each direct and indirect base class is present, as base class subobject, within the object representation of the derived class at implementation-defined offset. Empty base classes usually do not increase the size of the derived object due to empty base optimization. The constructors of base class subobjects are called by the constructor of the derived class: arguments may be provided to those constructors in the member initializer list.

**Virtual base classes**

For each distinct base class that is specified virtual, the most derived object contains only one base class subobject of that type, even if the class appears many times in the inheritance hierarchy (as long as it is inherited virtual every time).

```
struct B { int n; };
class X : public virtual B {};
class Y : virtual public B {};
class Z : public B {};
// every object of type AA has one X, one Y, one Z, and two B's:
// one that is the base of Z and one that is shared by X and Y
struct AA : X, Y, Z {
   AA() {

      X::n = 1; // modifies the virtual B subobject's member
      Y::n = 2; // modifies the same virtual B subobject's member
      Z::n = 3; // modifies the non-virtual B subobject's member
      std::cout << X::n << Y::n << Z::n << '\n'; // prints 223
   }
};
```

An example of an inheritance hierarchy with virtual base classes is the iostreams hierarchy of the standard library: std::istream and std::ostream are derived from std::ios using virtual inheritance. std::iostream is derived from both std::istream and std::ostream, so every instance of std::iostream contains a std::ostream subobject, a std::istream subobject, and just one std::ios subobject (and, consequently, one std::ios_base).

All virtual base subobjects are initialized before any non-virtual base subobject, so only the most derived class calls the constructors of the virtual bases in its member initializer list:

```
struct B {
    int n;
    B(int x) : n(x) {}
};
struct X : virtual B { X() : B(1) {} };
struct Y : virtual B { Y() : B(2) {} };
struct AA : X, Y     { AA() : B(3), X(), Y() {} };

// the default constructor of AA calls the default constructors of X and Y
// but those constructors do not call the constructor of B because B is a virtual base
AA a; // a.n == 3
// the default constructor of X calls the constructor of B
X x; // x.n == 1
```

There are special rules for unqualified name lookup for class members when virtual inheritance is involved (sometimes referred to as the rules of dominance), see unqualified_lookup#Member_function_definition.

**Public inheritance**

When a class uses public member access specifier to derive from a base, all public members of the base class are accessible as public members of the derived class and all protected members of the base class are accessible as protected members of the derived class (private members of the base are never accessible unless friended)

Public inheritance models the subtyping relationship of object-oriented programming: the derived class object IS-A base class object. References and pointers to a derived object are expected to be usable by any code that expects references or pointers to any of its public bases (see LSP) or, in DbC terms, a derived class should maintain class invariants of its public bases, should not strengthen any precondition or weaken any postcondition of a member function it overrides.

**Protected inheritance**

Protected inheritance may be used for "controlled polymorphism": within the members of Derived, as well as within the members of all further-derived classes, the derived class IS-A base: references and pointers to Derived may be used where references and pointers to Base are expected.

**Private inheritance**

When a class uses private member access specifier to derive from a base, all public and protected members of the base class are accessible as private members of the derived class (private members of the base are never accessible unless friended).
Private inheritance is commonly used in policy-based design, since policies are usually empty classes, and using them as bases both enables static polymorphism and leverages empty-base ptimization

Private inheritance can also be used to implement the composition relationship (the base class subobject is an implementation detail of the derived class object). Using a member offers better encapsulation and is generally preferred unless the derived class requires access to protected members (including constructors) of the base, needs to override a virtual member of the base, needs the base to be constructed before and destructed after some other base subobject, needs to share a virtual base or needs to control the construction of a virtual base.

Use of members to implement composition is also not applicable in the case of multiple inheritance from a parameter pack or when the identities of the base classes are determined at compile time through template metaprogramming.

Similar to protected inheritance, private inheritance may also be used for controlled polymorphism: within the members of the derived (but not within further-derived classes), derived IS-A base.

```
template<typename Transport>
class service : Transport  // private inheritance from the Transport policy
{
public:
   void transmit() {
      this->send(...);  // send using whatever transport was supplied
   }
};
```

```
// TCP transport policy
class tcp {
public:
    void send(...);
};
// UDP transport policy
class udp {
public:
  void send(...);
};
service<tcp> service(host, port);
service.transmit(...);
```

**Syntax for Inheritance:**

```
class derived-class-name : access base-class-name
{
// ...
}
```

Here access is one of the **three keywords**:

i)public,

ii)private, and

iii)protected.

When the access specifier for the inherited base class is **public**, all public members of the base class become public members of the derived class. If the access specifier is **private**, all public members of the base class become private members of the derived class. In either case, any private members of the base class remain private to it and are inaccessible by the derived class.

It is important to understand that if the access specifier is **private**, public members of the base become private members of the derived class. If the access specifier is not present, it is private by default.

The **protected** access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class

derived from that base.

**Example Program for Inheritance:Extending Class :**

```
#include <stdio.h>
using namespace std;
class Parent //Base class
{
  public:
  int id_p;
};
class Child : public Parent     // Sub class inheriting from Base Class(Parent)
{
  public:
  int id_c;
};
  int main()          //main function
 {
Child obj1;
obj1.id_c = 7;
 obj1.id_p = 91;
 cout << "Child id is " <<  obj1.id_c << endl;
 cout << "Parent id is " <<  obj1.id_p << endl;
  return 0;
   }
```

**Output:**

Child id is 7

Parent id is 91

**Types of Inheritances**

**8.3 Single Inheritance**

The process in which a derived class inherits traits from only one base class, is called single inheritance. In single inheritance, there is only one base class and one derived class.

SRGPGPI

The derived class inherits the behavior and attributes of the base class.

**Syntax :**

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

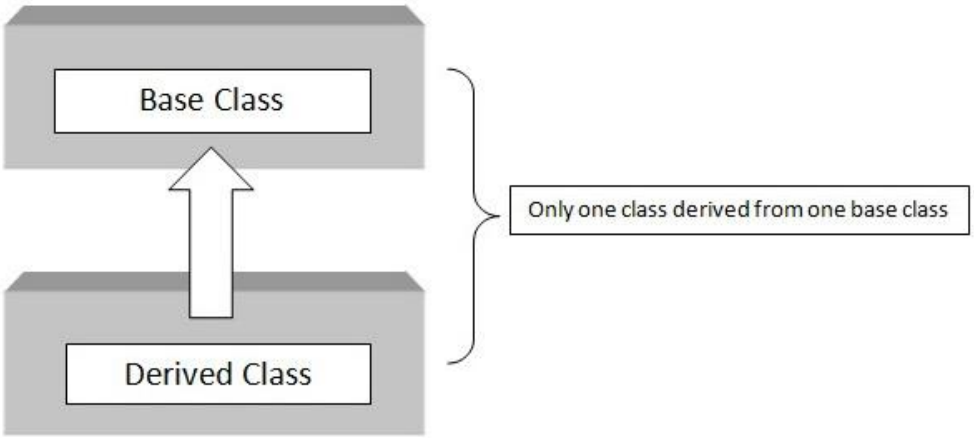



**FIgure : Single Inheritance**

**Example program:-**

```
#include <iostream>
using namespace std;
class base    //single base class
{
  public:
    int x;
  void getdata()
```

```
  {
    cout << "Enter the value of x = "; cin >> x;
  }
};
class derive : public base    //single derived class
{
  private:
   int y;
  public:
  void readdata()
  {
    cout << "Enter the value of y = "; cin >> y;
  }
  void product()
  {
    cout << "Product = " << x * y;
  }
};

int main()
{
   derive a;    //object of derived class
   a.getdata();
   a.readdata();
   a.product();
   return 0;
}
```

Output:-

Enter the value of x = 3

Enter the value of y = 4

Product = 12

**8.6 Multiple Inheritance**

The process in which a derived class inherits traits from several base classes,



SRGPGPI

is called multiple inheritance. In Multiple inheritance, there is only one derived class and several base classes.

**Syntax :**

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

Diagram:-

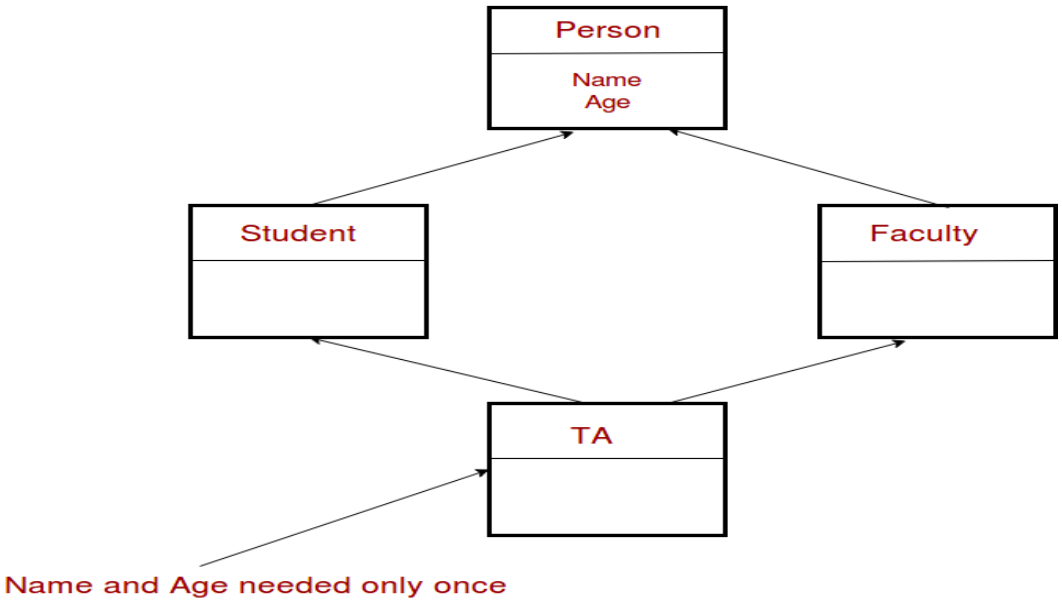



**Figure : Multiple Inheritance**

**Example Program :**

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

```
class B :public A
{
};
class C :public B
{
};
int main()
{
 C obj;
 obj.display();
return0;
}
```

**Output**

Base class content.

Multilevel Inheritance:-

In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

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

Here, class B is derived from the base class A and the class C is derived from the derived class B.

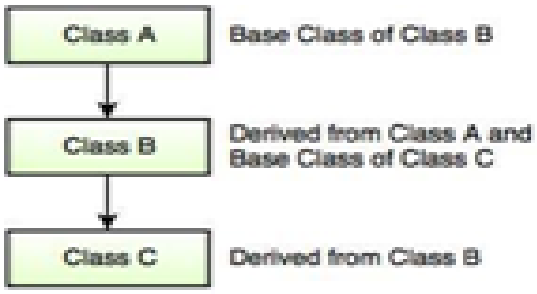



**Figure :Multilevel Inheritance**

Example

```
#include <iostream>
using namespace std;
class A
{
   public:
     void display()
     {
        cout<<"Base class content.";
     }
};
class B : public A
{
};
class C : public B
{

};
int main()
{
   C obj;
   obj.display();
   return 0;
}
```

Output

Base class content.

### 8.7 )Hierarchical Inheritance

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

Synatx:-

```
class base_class {
    ... .. ...
}
class first_derived_class: public base_class {
    ... .. ...
}
class second_derived_class: public base_class {
    ... .. ...
}
class third_derived_class: public base_class {
   ... .. ...
}
```

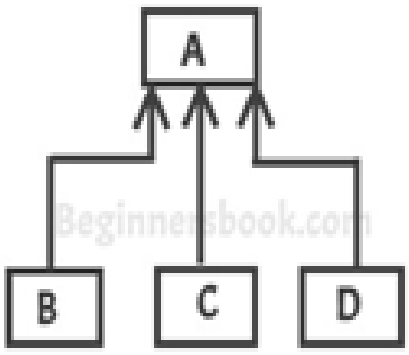



**Figure : Hierarchical Inheritance**

### 8.8 )Hybrid Inheritance

SRGPGPI

The inheritance hierarchy that reflects any legal combination of other types of inheritance is known ashybrid Inheritance.

Syntax:-

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

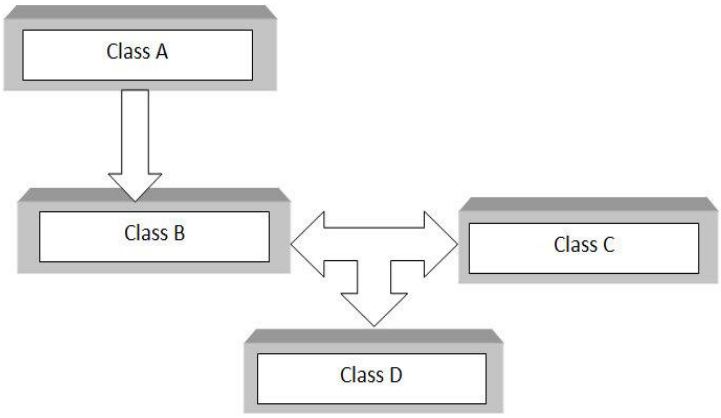



**Figure : Hybrid Inheritance**

```
#include <iostream>
```

```
namespace std;
class A
{
        public:
        int x;
};
class B : public A
{
        public:
        B()     //constructor to initialize x in base class A
        {
          x = 10;
        }
};
class C
 {
        public:
        int y;
        C()   //constructor to initialize y
        {
           y = 4;
      }
};
class D : public B, public C   //D is derived from class B and class C
{
        public:
        void sum()
        {
           cout << "Sum= " << x + y;
        }
};

int main()
{
```

SRGPGPI

```
    D obj1;         //object of derived class D
      obj1.sum();
      return 0;
}                   //end of program
```

Output:-

Sum= 14

**8.10 ABSTRACT CLASS:-**

An abstract class is, conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions.

A pure virtual function is one which must be overridden by any concrete (i.e., non-abstract) derived class. This is indicated in the declaration with the syntax " = 0" in the member function's declaration.

**Example:-**

```
class AbstractClass
 {
public:
  virtual void AbstractMemberFunction() = 0; // Pure virtual function makes
                                  // this class Abstract class.
  virtual void NonAbstractMemberFunction1(); // Virtual function.
  void NonAbstractMemberFunction2();
};
```

In general an abstract class is used to define an implementation and is intended to be inherited from by concrete classes. It's a way of forcing a contract between the class designer and the users of that class. If we wish to create a concrete class (a class that can be instantiated) from an abstract class we must declare and define a matching member function for each abstract member function of the base class. Otherwise, if any member function of the base class is left undefined, we will create a new abstract class (this could be useful



SRGPGPI

sometimes).

Sometimes we use the phrase "pure abstract class," meaning a class that exclusively has pure virtual functions (and no data). The concept of interface is mapped to pure abstract classes in C++, as there is no "interface" construct in C++ the same way that there is in Java.

**Example:-**

```
class Vehicle {
public:
   explicit
   Vehicle( int topSpeed )
   : m_topSpeed( topSpeed )
   {}
   int TopSpeed() const {
      return m_topSpeed;
   }
   virtual void Save( std::ostream& ) const = 0;
private:
   int m_topSpeed;
};

class WheeledLandVehicle : public Vehicle {
public:
   WheeledLandVehicle( int topSpeed, int numberOfWheels )
   : Vehicle( topSpeed ), m_numberOfWheels( numberOfWheels )
   {}
   int NumberOfWheels() const {
     return m_numberOfWheels;
   }
   void Save( std::ostream& ) const; // is implicitly virtual
private:
   int m_numberOfWheels;
};
```

SRGPGPI

```
class TrackedLandVehicle : public Vehicle {
public:
  TrackedLandVehicle ( int topSpeed, int numberOfTracks )
  : Vehicle( topSpeed ), m_numberOfTracks ( numberOfTracks )
  {}
  int NumberOfTracks() const {
    return m_numberOfTracks;
  }
  void Save( std::ostream& ) const; // is implicitly virtual

private:
  int m_numberOfTracks;
};
```

In this example the Vehicle is an abstract base class as it has an abstract member function.The class WheeledLandVehicle is derived from the base class. It also holds data which is common to all wheeled land vehicles, namely the number of wheels. The class TrackedLandVehicle is another variation of the Vehicle class.

This is something of a contrived example but it does show how that you can share implementation details among a hierarchy of classes. Each class further refines a concept. This is not always the best way to implement an interface but in some cases it works very well. As a guideline, for ease of maintenance and understanding you should try to limit the inheritance to no more than 3 levels. Often the best set of classes to use is a pure virtual abstract base class to define a common interface. Then use an abstract class to further refine an implementation for a set of concrete classes and lastly define the set of concrete classes.

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

**The following is an example of an abstract class:**

```
class AB {
public:
  virtual void f() = 0;
```



SRGPGPI

```
};
```

Function AB::f is a pure virtual function. A function declaration cannot have both a pure specifier and a definition.

Abstract class cannot be used as a parameter type, a function return type, or the type of an explicit conversion, and not to declare an object of an abstract class. It can be used to declare pointers and references to an abstract class.

**Pure Abstract Classes**

An abstract class is one in which there is a declaration but no definition for a member function. The way this concept is expressed in C++ is to have the member function declaration assigned to zero.

**Example**

```
class PureAbstractClass
{
public:
  virtual void AbstractMemberFunction() = 0;
};
```

A pure Abstract class has only abstract member functions and no data or concrete member functions. In general, a pure abstract class is used to define an interface and is intended to be inherited by concrete classes. It's a way of forcing a contract between the class designer and the users of that class. The users of this class must declare a matching member function for the class to compile.

**Example of usage for a pure Abstract Class**

```
 class DrawableObject
 {
  public:
    virtual void Draw(GraphicalDrawingBoard&) const = 0; //draw to GraphicalDrawingBoard
 };
 class Triangle : public DrawableObject
```



SRGPGPI

```
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a triangle
};
class Rectangle : public DrawableObject
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a rectangle
};
class Circle : public DrawableObject
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a circle
};
typedef std::list<DrawableObject*> DrawableList;
DrawableList drawableList;
GraphicalDrawingBoard drawingBoard;
drawableList.pushback(new Triangle());
drawableList.pushback(new Rectangle());
drawableList.pushback(new Circle());
for(DrawableList::const_iterator iter = drawableList.begin(),
   endIter = drawableList.end();
   iter != endIter;
   ++iter)
{
  DrawableObject *object = *iter;
  object->Draw(drawingBoard);
}
```

**SUMMARY:-**

- The c++ classes can be reused using inheritance.
- The derived class inherits some or all of the properties of the base class.
- .A derived class with only one base class is called single inheritance.



SRGPGPI

- A class can inherit properties for more than class which is known as multiple inheritance.
- A class can be derived from another class which is known as multilevel inheritance.
- A private member of class cannot be inherited either in public mode or in private mode.
- In multiple inheritance , the base clasdes are constructed in the order in which they appear in declaration of the derived class.
- In multilevel inheritance, the Constructors are executed in the order of inheritance.

**Reviwe Questions:-**

1) What is inheritance?
2) How to implement inheritance ?
3) What is Base class ?
4) What is Subclass?
5) What is the difference between public and private access specifier ?
6) What are the advantages of inheritance ?
7) What are the types of inheritance ?
8) How to implement data and functionality as a single entity ?
9) What is the difference between inheritance and polymorphism?
10) Is inheritance possible in C++ ?
11)Describe the syntax of the single inheritance in C++
12) How do the properties of the following two derived classes differ?
13) When do we the protected visibility specifier to a class member?
14) What is a virtual base class?
15) When do we make a class virtual?
16) what is an abstract class?
17) In what order are the class consructors called when a derived class object is created?
18)what doe8 inheritance mean i11 C++?
19) What are the different forms of inheritance? Gi~ an example for each.
20) Describe. the syntax of the. sing!€ inherita.nce in C++.
21)we knew that a private member of a base. class is not inheritable.
22)it anyway possible [or the objecl8 of a deri.ved class to access the private m.4!-mbe,rs of the bo.se class? If yes, how? Remember, the base class co1t1Wt be m«lified.

SRGPGPI

23) How do the properties of the following two derived classes di/ft.rt (a) cla•• DI: private 8(11 .. J; (bl class D2: public B(l l .. J;

24) \Vhen do we use the protected vi8ibility S{>«i{ier to a clas8 member?

25) Describe the S>•nta.x of multiple inheritance. When do we use such an inheritance'!

## 9 POINTERS , VIRTUAL FUNCTIONS AND POLYMORPHISM

### 9.1 Introduction:-

We have been accessing members of an object by using the dot operator. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator (->) rather than the dot operator is employed. We can declare an object pointer just as a pointer to any other type of variable is declared. Specify its class name, and then precede the variable name with an asterisk**.**

### 9.2 POINTER;-

A pointer is a variable that contains a memory address. Very often this address is the location of another object, such as a variable.

type *var-name;

Here, type is the pointer's base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable.

To use pointer:

- We define a pointervariable.
- Assign the address of a variable to apointer.
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

**Example :**

```
#include<iostream>
usingnamespace std;
```

SRGPGPI

```
int main()
{
int var1 =3;
int var2 =24;
int var3 =17;
 cout <<&var1 << endl;
 cout <<&var2 << endl;
 cout <<&var3 << endl;
}
```

**Output**

```
0x7fff5fbff8ac
0x7fff5fbff8a8
0x7fff5fbff8a4
```

### 9.3 OBJECT POINTERS

We have been accessing members of an object by using the dot operator. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator (->) rather than the dot operator is employed. We can declare an object pointer just as a pointer to any other type of variable is declared. Specify its class name, and then precede the variable name with an asterisk.

### 9.5 Pointers to Derived Types

Pointers to base classes and derived classes are related in ways that other types of pointers are not.

In general, a pointer of one type cannot point to an object of another type. However, base class pointers and derived objects are the exceptions to this rule. In C++, a base class pointer can also be used to point to an object of any class derived from that base.

```
B*p;          //pointer p to object of typeB
BB_ob;        //object of typeB
DD_ob;        //object of typeD
```

both of the following statements are perfectly valid:

```
p = &B_ob;    //p points to objectB
```

p= &D_ob;  //p points to object D, which is an object derived fromB

**File pointers**

C++ also supports file pointers. A file pointer points to a data element such as character in the file.The pointers are helpful in lower level operations in files. There are two types of

- Get pointer
- Putpointer

The **get pointer** is also called input pointer. When we open a file for reading, we can use the get pointer.

The **put pointer** is also called output pointer. When we open a file for writing, we can use put pointer.

**File pointer functions**

There are essentially four functions, which help us to navigate the file as given below Functions Function Purpose

| | |
|---|---|
| tellg() | Returns the current position of the getpointer |
| seekg() | Moves the get pointer to the specifiedlocation |
| tellp() | Returns the current position of the putpointer |
| seekp() | Moves the put pointer to the specifiedlocation |

**9.6 Virtual Function in C++:**

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

**Rules of Virtual Function**

RGPGPI

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

**Pure Virtual Function**

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.

**C++ Polymorphism**

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

**Runtime Polymorphism Example Program :**

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

class base
{
public:
 virtual void print ()
 {
```

```
cout<< "print base class" <<endl;
}
 void show ()
 {
cout<< "show base class" <<endl;
 }
};
class derived:public base
{
public:
void show ()
{
cout<< "show derived class" <<endl; }
};
int main()        //main function
{
 base *bptr;
 derived d;
bptr = &d;
 bptr->print();
 bptr->show();
 return 0; }
```

Output:

```
print derived class
show base class
```

**There are two types of polymorphism in C++:**

- **Compile time polymorphism**:

  The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function

SRGPGPI

overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

- **Run time polymorphism**:

  Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

| Compile time polymorphism | Run time polymorphism |
|---|---|
| The function to be invoked is known at the compile time. | The function to be invoked is known at the run time. |
| It is also known as overloading, early binding and static binding. | It is also known as overriding, Dynamic binding and late binding. |
| Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters. | Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters. |
| It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and pointers. |
| It provides fast execution as it is known at the compile time. | It provides slow execution as it is known at the run time. |

**SUMMARY:-**

- ✓ The mechanism or deriving a new class from an old class is called inheritance.
- ✓ Inheritance provides the concept ofreusability.
- ✓ The C++ classes can be reused using inheritance.
- ✓ The derived class inherit& some or all of the properties of the base class.
- ✓ A derived class with only one base class is called single inheritance.
- ✓ A class can inherit properties from more than one class which is known as multiple inheritance.
- ✓ A class can be derived from another derived class which is known as multilevel inheritance.
- ✓ \Vhen the properties of one class are inherited by more than ooe class, it is called

hierarchical inheritance.

- ✓ A private member of a class cannot be inherited either in public mode or in private mode.
- ✓ A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in the derived class.
- ✓ A public member inhe:rit.ed in public mode become;; public, whereas inherited in private mode becomas private in the derived class.

Review Questions:-

1) what is containership? How does differ from inheritance?
2) Describe how an object of a class that contains objects of other classes created
1. 3)What are the different forms of inheritance? Give an example for each.
3) what does polymorphism mean in c++ languages?
4) How is polymorphism achieved at compile time and run time?
5) What does this pointer point to?

**Debugging Exercises:-**

1) Identify the error in the following program.

```
#include<iostream.h>
Class Student
{
  Char* name;
  Int roll Number;
Private:
  Student()
  {
    Name = “Alankay”;
   Roll number = 1025;
  }
 Void set Number(int no)
{
 Roll number=no;
 }
Int get roll number(0)
```

SRGPGPI

```
    Return roll number;
  }};
```

2) Implementation of Inheritance in C++ Programming

```
  class Person
  {
    ... .. ...
  };
  class MathsTeacher : public Person
  {
    ... .. ...
  };
  class Footballer : public Person
  {
    .... .. ...
  };
```

2)Identify the error in the following program.

```
  #include <iostream>
  using namespace std;
 class base
{
public:
  virtual void print()
  {
     cout << "print base class" << endl;
  }
 void show()
  cout << "show base class" << endl;
  }
};
 class derived : public base {
public:
  void print()
```

SRGPGPI

```
    {
        cout << "print derived class" << endl;
    }
   void show()
    {
        cout << "show derived class" << endl;
    }
};
  int main()
{
    base* bptr;
    derived d;
    bptr = &d;
   // virtual function, binded at runtime
    bptr->print();
  // Non-virtual function, binded at compile time
    bptr->show();
}
```

3) identify the error in the following program.

```
class base  {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};
  class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};
  int main()
```

SRGPGPI

```
{
    base* p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1() is non-virtual
    // in base
    p->fun_1();
// Late binding (RTP)
    p->fun_2();
  // Late binding (RTP)
    p->fun_3();
    // Late binding (RTP)
    p->fun_4();
    // Early binding but this function call is
    // illegal(produces error) becasue pointer
    // is of base type and function is of
    // derived class
    // p->fun_4(5);
}
```

15)Identify the error in the following program

```
class Loader
{
   public:
     void loadFeatures(Weapon *weapon)
     {       weapon->features();
     }
};
```

4) Correct the errors in the following program

```
    this->Weapon::features();
#include <iostream>
using namespace std;
class Weapon
{
```

SRGPGPI

```
  public:
     virtual void features()
       { cout << "Loading weapon features.\n"; }
};
class Bomb : public Weapon
{
   public:

         cout << "Loading gun features.\n";
       }};
class Loader
{ void features()
       { this->Weapon::features();
         cout << "Loading bomb features.\n";
       }};
class Gun : public Weapon
{
   public:
void features()
         {
       l->loadFeatures(w);
       return 0;   public:
        void loadFeatures(Weapon *weapon)
        {
          weapon->features();
        } };
    int main()
    { Loader *l = new Loader;
       Weapon *w;
       Bomb b;
       Gun g;
       w = &b;
       l->loadFeatures(w);
       w = &g;
```

SRGPGPI

}

## 10 MANAGING CONSOLE WITH I/O FILES

**OBJECTIVE:-**

In the above figure, ios is the base class. The iostream class is derived from istream and ostream classes. The ifstream and ofstream are derived from istream and ostream, respectively. These classes handles input and output with the disk files.The fstream.h header file contains a declaration of ifstream, ofstream and fstream classes.

The iostream.h file contains istream, ostream and iostream classes and included in the program while doing disk I/O operations.The filebuf class contains input and output operations with files.

The streambuf class does not organize streams for input and output operations, only derived classes of streambuf performs I/O operations. These derived classes arranges a space for keeping input data and for sending output data.The istream and ostream invokes the filebuf functions to perform the insertion or extraction on the streams.

### 10.1 Introduction

In general, each program takes some information as input and tries to produce processed information as output with the help of input-process-output cycle. C++ language has introduced a wide variety of features which can be used for displaying the output with the help of wide variety of functions. C++ makes use of data manipulators directly to display the output using outputstream.

### 10.2 C++Streams

In C++ there are number of stream classes for defining various streams related with files an for doing input-output operations. All these classes are defined in the file iostream.h. Figure given below shows the hierarchy of these classes. ios class is topmost class in the stream classes hierarchy.

It is the base class for istream, ostream, and streambuf class.istream and ostream serves the base classes for iostream class. The class istream is used for input and ostream for the output.

Class ios is indirectly inherited to iostream class using istream and ostream. To avoid the duplicity of data and member functions of ios class, it is declared as virtual base class

SRGPGPI

wheni nheriting in istream and ostream as

```
class istream: virtual public ios
{
};
class ostream: virtual public ios
{
};
```

The _withassign classes are provided with extra functionality for the assignment operations that's why _withassign classes.

**Header files available in C++ for Input – Output operation are:**

- **iostream**: iostream stands for standard input output stream. This header file contains definitions to objects like cin, cout, cerr etc.
- **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.
- **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

### 10.5 Formatted Console I/OOperations

C++ supports a wide variety of features which can be used for designing the output.
The features consist of:

- ios class functions andflags
- Manipulators
- User-definedManipulators
-

**i)ios class functions and flags**

The ios class consists of a variety of member functions that helps to format the output in many ways. The most commonly used ios functions that are used for formatting are:

| Function Name | Task |
|---|---|
| **width()** | To indicate the essential field size for displaying an output value |
| **precision()** | To indicate the digits that are to be displayed after the decimal point of a float value |
| **fill()** | To denote a character that is used to fill up the unused portion of afield |
| **setf()** | To denote the format flags that can organize the form of output display |
| **unsetf()** | To clear the flag that is being specified |

**a)width()**

It is mainly used to describe the width of a field that is required for the output of an item. Because, it is a member function, it is must to invoke it using an object. The width( ) function be able to give the field width for only one item, i.e., the item that followinstantly.**yntax:**

cout.width(w)

Where w is the field width (number of columns).

**b)precision()**

It is mainly used to indicate the number of digits that are to be displayed after the decimal point while printing the numbers that are of floating-point type.

The default precision for floating numbers is six digits after the decimal point while printing.

The precision ( ) function tries to preserve the situation in result until itis reorganized.

**Syntax:**

cout.precision(d);

Where d is the number of digits to the right of the decimal point.

**c)fill()**

By default, while printing the values that are having larger field width than required by the values, the idle positions of the field are filled with white spaces. This problem

SRGPGPI

can be solved by using the fill( ) function, because it fill up the unused positions by any preferred character.

**Syntax:**

cout.fill(ch);

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

**d)setf()**

When the width() function is used, it tries to print the value with right justification by using the width that is created. But, to print the text with left justification, it is must to use the setf( ) function.

**Syntax:**

cout.setf(arg1, arg2)

**Syntax Explanation :**

- arg1 is the value of formatting flags that are defined in ios class, it denote the formataction that is required for theoutput.
- arg2 is the bit field in constant in the ios class and it denotes the collection to which the formatting flagbelongs.

The flags and bit fields that can be used for setf() function

| **Format required** | **Flag (arg1)** | **Flag (arg2)** |
|---|---|---|
| Left-justified output | ios::left | ios::adjustfield |
| Right-justified output | ios::right | ios::adjustfield |
| Padding after sign or base | ios::internal | ios::adjustfield |
| Indicator | | |
| Scientific notation | ios::scientific | ios::floatfield |
| Fixed point notation | ios::fixed | ios::floatfield |
| Decimal base | ios::dec | ios::basefield |
| Octal base | ios::oct | ios::basefield |
| Hexadecimal base | ios::hex | ios::basefield |

SRGPGPT

**ii)Floatfield Bit FormatFlag**

Based on the requirement, sometimes a floating point value may have to be exhibit in scientific notation instead of fixed point notation.

**a)Scientific**

When the flag scientific is set, then the floating point values are inserted using scientific notation. There will be only one digit before the decimal point followed by the particular number of precision digits which in turn is followed by an

uppercase letter "E" or a lower case letter "e" depending on the situation of uppercase and the exponentvalue.

Syntax:

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

**b)Fixed**

When the flag fixed is set, and then the value is inserted using decimal notation with the particular number of precision digits following the decimal point.

**Syntax:**

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

**iii)Basefield FormatFlag:**

The basefield format flag is mainly used to show the integers in the appropriate base.

At a time, one of the base field, i.e., dec, oct or hex be able to be set at any time. The basefield format flags manage the base in which the numbers are displayed.

**10.6 Managing output withManipulators**

The manipulators are the unique and special stream functions that can be included in the Input/Output statements to alter certain characteristics of the input and output. To use the manipulators, it is must to include the header file <iomanip.h> or



SRGPGPT

<iomanip> in the program.

The predefined manipulators and their equivalent ios functions are:

| Manipulator | Equivalent ios function |
|---|---|
| **setw()** | **width()** |
| **setprecision()** | **precision()** |
| **setfill()** | **fill()** |
| **setiosflags()** | **setf()** |
| **resetiosflags()** | **unsetf()** |

**Example Program :**

```
#include<iostream>
usingnamespace std;
int main()
{
char c=cin.get();
cout.put(c); //Here it prints the value of variable c;
cout.put('c'); //Here it prints the character 'c';
return 0;
}
```

**Output**

```
I
Ic
```

**10.4 UNFORMATTED OPERATIONS**:-

The printed data with default setting by the I/O function of the language is known as unformatted data.

It is the basic form of input/output and transfers the internal binary representation of the data directly between memory and the file.

For example, in cin statement it asks for a number while executing. If the user enters a decimal number, the entered number is displayed using cout statement. There is no need to apply any external setting, by default the I/O function represents the number in decimal format.

**Formatted data**

If the user needs to display a number in hexadecimal format, the data is represented with the manipulators are known as formatted data.

It converts the internal binary representation of the data to ASCII characters which are written to the output file.

It reads characters from the input file and coverts them to internal form.

For example, cout<<hex<<13; converts decimal 13 to hexadecimal d. Formatting is a representation of data with different settings (like number format, field width, decimal points etc.) as per the requirement of the user.

**Input/Output Streams**

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

The cin and cout are pre-defined streams for input and output data.

**Syntax:**

cin>>variable_name;

cout<<variable_name;

The cin object uses extraction operator (>>) before a variable name while the cout object uses insertion operator (<<) before a variable name.

The cin object is used to read the data through the input device like keyboard etc. while the cout object is used to perform console write operation.

Example: Program demonstrating cin and cout statements

```
#include<iostream>
using namespace std;
int main()
{
```

```
 char sname[15];
      cout<<"Enter Employee Name : "<<endl;
      cin>>sname;
      cout<<"Employee Name is : "<<sname;
      return 0;
}
```

**Output:**

Enter Employee Name : Prajakta

Employee Name is : Prajakta

In the above example, cout<<"Employee Name is : "<<sname displays the contents of character array sname (student name). The cout statement is like printf statement as used in C language.

The cin statement cin>>sname reads the string through keyboard and stores in the array sname[15]. The cin statement is like scanf statement as used in C language. The endl is a manipulator that breaks a line.

Typecasting

Typecasting is a conversion of data in one basic type to another by applying external use of data type keywords.

**SUMMARY :**

- In c++, the I/O systems is designed to work with different I/O devices.
- A stream is a sequence of bytes and serves as a source or destination for an I/O data.
- The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream.
- The c ++, and I/O system contains a hierarchy of stream classes used for input and output operations
- These classes are declared in the standard output devices.

- Cin represents the input stream connected to the standard input device and cout represents the output stream connected to the standard input device
- The istream and ostream classes define two member Functions get() to handle the single character I/O stream
- We can read and write a line of text more efficiently using the line oriented I/O functions getline() and write () respectively.
- We can also design our own manipulators for certain special purposes.

**Review Questions:-**

1) What type of objects are used in C++ to handle standard input and output operations?
2) What is cin and cout?
3) What is console IO?
4) What is formatted and unformatted data?
5) What is stream?
6) Describe briefly the features of i/o system supported by c++
7) How do the I/o facilities in c++differ from that in c?
8) How is cout able to display various types of data without any special instructions?
9) why is it necessary to include the flr iostream in all our programs?
10) Discuss the various forms of get() function supported by the input stream. How are they used?
11) How do the following two statements differ in operation?

Cin>>c:

Cin.get(c);

**Debugging Exercise:-**

1.identify the error in the following program

```
int main()
{
   istringstream str("          Programmer");
   string line;
   // Ignore all the whitespace in string
   // str before the first word.
```

```
    getline(str >> std::ws, line);
    // you can also write str>>ws
    // After printing the output it will automatically
    // write a new line in the output stream.
    cout << line << endl;
    // without flush, output will be the same.
    cout << "only a test" << flush;
    // Use of ends Manipulator
    cout << "\na";
    // NULL character will be added in the Output
    cout << "b" << ends;
    cout << "c" << endl;
    return 0;
```

2)identify the error in the following program

```
int main()
{
    double A = 100;
    double B = 2001.5251;
    double C = 201455.2646;
    // We can use setbase(16) here instead of hex
    // formatting
    cout << hex << left << showbase << nouppercase;
    // actual printed part
    cout << (long long)A << endl;
    // We can use dec here instead of setbase(10)
    // formatting
    cout << setbase(10) << right << setw(15)
<< setfill('_') << showpos
<< fixed << setprecision(2);
    // actual printed part
    cout << B << endl;
    // formatting
    cout << scientific << uppercase
```

```
<< noshowpos << setprecision(9);

    // actual printed part
    cout << C << endl;
}
```

## 11 WORKING WITH FILES :

### 11.1 Introduction

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.

A stream is an abstraction that represents a device on which operations of input and output are performed. A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

In C++ we have a set of file handling methods. These include ifstream, ofstream, and fstream. These classes are derived from fstrembase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include fstream and therefore we must include this file in any program that uses files.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream.ofstream: This Stream class signifies the output file stream and is applied to create files for writing information to files

ifstream: This Stream class signifies the input file stream and is applied for reading information from files
fstream: This Stream class can be used for both read and write from/to files.

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

C++ provides us with the following operations in File Handling:

Creating a file: open()
Reading data: read()
Writing new data: write()
Closing a file: close()

## 11.2 CLASSES OF FILE OPERATIONS:-

Whether it is the programming world or not, files are vital as they store data. This article discuss working of file handling in C++. Following pointers will be covered in the article,

- Opening a File
- Writing to a File
- Reading from a File
- Close a File

## 11.3 Opening a File

Generally, the first operation performed on an object of one of these classes is to associate it to a real file. This procedure is known to open a file.

We can open a file using any one of the following methods:

1. First is bypassing the file name in constructor at the time of object creation.
2. Second is using the open() function.

To open a file use

```
1    open() function
```

**Syntax**

```
1    void open(const char* file_name,ios::openmode mode);
```

Here, the first argument of the open function defines the name and format of the file with the address of the file.

The second argument represents the mode in which the file has to be opened. The following modes are useWriting to a File

**Example:**

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
```

SPGPGPI

```
fstream new_file;
new_file.open("new_file_write.txt",ios::out);
if(!new_file)
{
cout<<"File creation failed";
}
else
{
cout<<"New file created";
new_file<<"Learning File handling";    //Writing to file
new_file.close();
}
return 0;
}
```

Output:

Output-File Handling in C++- Edureka

**Working With Text Files :**

- A text file is also called *file with format*
- It only contains printable characters
- A printable character is a character with an ASCII code greater or equal to 32.
- What is ASCII code? It's a code which maps each character to a number (computers only store numbers).
- Text file examples: C++ source code, a webpage (HTML), a makefile, . . .

**Declaration of file variables :**

To use them, add

#include <fstream>: ifstream fich_read; (read only)

ofstream fich_write;(write only)

fstream fich_read_write;(unusual in text files)

SRGPGPI

**OPENING FILES :**

Opening modes: read, write, read/write, append Files can be opened in C++ using "open".:

```
const char nombre[]="mifichero.txt";
fichero.open(nombre,ios::in);
```

When the filename is a string, it must be converted into an array of characters using the function c_str().

**C++ opening modes:**

| | | | |
|---|---|---|---|
| read | ios::in | | |
| write | ios::out | | |
| read/write | ios::in\|ios::out | (fstream) append | ios::out \|ios::app |

**Binary files :**

- It's also called *file without format*
- In a binary file, data are stored as they are in memory, without being converted into characters.
- Usually, each element to be stored is written using a structure (struct)
- When elements are stored using structures, it's possibletodirectlyaccessthe*n*-thelementwithout readingthe$n-1$previousdata.
- Text files have sequential access, whereas binary files have direct (random) access.

**Declaring , opening & closing binary files:**

Variable declaration like in text files:

```
ifstream fbl;   // file forreading
ofstream fbe;   // file forwriting
```

File opening: The "ios::binary" flag must be added

Other opening modes:

**read/write ios::in | ios::out |ios::binary**

**append ios::out | ios::app |ios::binary**

SRGPGPI

**Writing binary files :**

The method seekpmust be used for positioning the writing pointer instead of seekg(which is only for reading).

```
if (fbe.is_open())
{
  // positioning to write the third element
  fbe.seekp ( (3-1)*sizeof(ciudad), ios::beg);
  fbe.write( (const char *)&ciudad, sizeof(ciudad)
  );
  ...
```

### 11.9 C++ Error Handling Functions

There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.

Following table lists these error handling functions and their meaning :

| Function | Meaning |
|---|---|
| int bad() | Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations. |
| int eof() | Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value). |
| int fail() | Returns non-zero (true) when an input or output operation has failed. |
| int good() | Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out. |
| clear() | Resets the error state so that further operations can be attempted. |

The above functions can be summarized as eof() returns true if eofbit is set; bad() returns true if badbit is set. The fail() function returns true if failbit is set; the good() returns true there are no errors. Otherwise, they return false.

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby take the necessary corrective measures. For example :

```
:
ifstream fin;
fin.open("master", ios::in);
while(!fin.fail())
{
        :       // process the file
}
if(fin.eof())
{
        :       // terminate the program
}
else if(fin.bad())
{
        :       // report fatal error
}
else
{
        fin.clear();    // clear error-state flags
        :
}
:
```

**C++ Error Handling Example**

Here is an example program, illustrating error handling during file operations in a C++ program:

```
/* C++ Error Handling During File Operations
 * This program demonstrates the concept of
 * handling the errors during file operations
 * in a C++ program */
```

```
#include<iostream.h>
#include<fstream.h>
#include<process.h>
#include<conio.h>
void main()
{
	clrscr();
	char fname[20];
	cout<<"Enter file name: ";
	cin.getline(fname, 20);
	ifstream fin(fname, ios::in);
	if(!fin)
	{
		cout<<"Error in opening the file\n";
		cout<<"Press a key to exit...\n";
		getch();
		exit(1);
	}
	int val1, val2;
	int res=0;
	char op;
	fin>>val1>>val2>>op;
	switch(op)
	{
		case '+':
			res = val1 + val2;
			cout<<"\n"<<val1<<" + "<<val2<<" = "<<res;
			break;
		case '-':
			res = val1 - val2;
			cout<<"\n"<<val1<<" - "<<val2<<" = "<<res;
			break;
		case '*':
			res = val1 * val2;
```

```
                cout<<"\n"<<val1<<" * "<<val2<<" = "<<res;
                break;
        case '/':
                if(val2==0)
                {
                        cout<<"\nDivide by Zero Error..!!\n";
                        cout<<"\nPress any key to exit...\n";
                        getch();
                        exit(2);
                }
                res = val1 / val2;
                cout<<"\n"<<val1<<" / "<<val2<<" = "<<res;
                break;

    }
    fin.close();
    cout<<"\n\nPress any key to exit...\n";
    getch();
}
```

Let's suppose we have four files with the following names and data, shown in this table:

| File Name | Data |
|---|---|
| myfile1.txt | 10<br>5<br>/ |
| myfile2.txt | 10<br>0<br>/ |
| myfile3.txt | 10<br>5<br>+ |
| myfile4.txt | 10<br>0<br>+ |

Now we are going to show the sample run of the above C++ program, on processing the files listed in the above table. Here are the four sample runs of the above C++ program, processing all the four files listed in the above table. Here is the sample out

**11.10 Command line arguments**:-

C++ it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

The full declaration of main looks like this:

1

```
int main ( int argc, char *argv[] )
```

The integer, argc is the ARGument Count (hence argc). It is the number of arguments passed into the program from the command line, including the name of the program.

The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument. You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

**Example:**

```
#include <fstream>
#include <iostream>
using namespace std;
int main ( int argc, char *argv[] )
{
  if ( argc != 2 ) // argc should be 2 for correct execution
    // We print argv[0] assuming it is the program name
    cout<<"usage: "<< argv[0] <<" <filename>\n";
```

SRGP

```
 else {
  // We assume argv[1] is a filename to open
  ifstream the_file ( argv[1] );
  // Always check to see if file opening succeeded
  if ( !the_file.is_open() )
    cout<<"Could not open file\n";
  else {
    char x;
    // the_file.get ( x ) returns false if the end of the file
    //  is reached or an error occurs
    while ( the_file.get ( x ) )
      cout<< x;
  }
  // the_file is closed implicitly here
 }
}
```

This program is fairly simple. It incorporates the full version of main. Then it first checks to ensure the user added the second argument, theoretically a file name. The program then checks to see if the file is valid by trying to open it. This is a standard operation that is effective and easy. If the file is valid, it gets opened in the process. The code is self-explanatory, but is littered with comments, you should have no trouble understanding its operation this far into the tutorial. :-)

**SUMMARY:-**

- The c++I/O system contains classes such as ifstream,ifstreamand fstream to deal with file handling.
- These classes are derived from fstreambase class and are declared in a header file iostream.
- A file can be opened in two ways by using the Constructors Function of the class and using the member Function open() of the class
- While opening the file using Constructors, we need to pass the desired filename as a parameter to the Constructors.
- The open() Function can be used to open multiple files that use the same stream object.

SRGPGPI

- The second arugument of the open() Function is called file mode,specifiesthe purpose for which the file is opened.
- To open an existing file for updating without losing its original contents, we need to open it in an append mode.
- Each file has associated two file pointers, one is called Input or get pointer.

**Review Questions:-**

1) How do the following two statements differ in operation?

Cin>>c:

Cin.get(c);

2) Both cin and getlne() function can be used for reading a string.comment.

3) discuss the implications of size parameter in the following statement.

Cout.write(line,size);

4) What does the following statement do?

5) what roles does the iomanip file play?

6) what is the role of file() function?when do we use this function?

7) discuss the syntax of set() function

8) what is the basic different between manipulators and ios member functions in implementation? Geve examples

**Debugging exercise:-**

1) what will be the result of the following programs segment?

```
#include <iostream>
using namespace std;
 int main()
{
   int age;
 cout << "Enter your age:";
   cin >> age;
   cout << "\nYour age is: " << age;
 return 0;
}
```

SRGPGPI

2)Will the statement cout.setf(ios::right)work or not?

```
#include <iostream>
 using namespace std;
 int main()
{
   clog << "An error occured";
  return 0; }
```

3)identify the error in the following program.

```
#include<iostream>
using namespace std;
int main()
{
        char c=cin.get();
 cout.put(c); //Here it prints the value of variable c;
 cout.put('c'); //Here it prints the character 'c';
 return 0;
}
```

SRGPGPI

# 12 TEMPLATE

## 12.1 Introduction:-

- Template is simple and yet very powerful tool in C++.
- The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types.

C++ adds two new keywords to support templates:

***a)Template***

***b)Type Name***

***Template Working Function :***

- Templates are expanded at compiler time. This is like macros.
- The difference is, compiler does type checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

## 12.4 Function Templates:

A generic function that can be used for different data types.

**Examples of function templates :**

sort(), max(), min(), printArray()

## 12.2 Class Templates:

class templates are useful when a class defines something that is independent of data type.

a)LinkedList

b)BinaryTree

c)Stack

d)Queue

e) Array, etc.

**Nontype Parameters to Template :**

Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of template.

SRGPCP

The important thing to note about non-type parameters is, they must be const. Compiler must know the value of non-type parameters at compile time.Because compiler needs to create functions/classes for a specified non-type value at compile time.

```
fbl.open("mifichero.dat", ios::in | ios::binary);
fbe.open("mifichero.dat", ios::out | ios::binary);
```

**12.8.Function templates and static variables :**

Each instantiation of function template has its own copy of local static variables.

For example, in the following program there are two instances:

a)*void fun(int )* and

b)*void fun(double ).*

**12.3.Class templates and static variables:**

The rule for class templates is same as function templates Each instantiation of class template has its own copy of member static variables.

**Example Program to Template :**

```
#include<iostream>
usingnamespace std;
// template function
template<class T>
T Large(T n1, T n2)
{
return(n1 > n2)? n1 : n2;
}
int main()
{
int i1, i2;
float f1, f2;
char c1, c2;
cout <<"Enter two integers:\n";
```

```
cin >> i1 >> i2;
cout <<Large(i1, i2)<<" is larger."<< endl;
cout <<"\nEnter two floating-point numbers:\n";
cin >> f1 >> f2;
cout <<Large(f1, f2)<<" is larger."<< endl;
cout <<"\nEnter two characters:\n";
cin >> c1 >> c2;
cout <<Large(c1, c2)<<" has larger ASCII value.";
return0;
}
```

**Output**

```
Enter two integers:
5
10
10 is larger.

Enter two floating-point numbers:
12.4
10.2
12.4 is larger.

Enter two characters:
z
Z
z has larger ASCII value.
```

**12.7 MEMBER FUNCTION TEMPLATES**:-

**Member templates**

Template declarations (class, function, and variables (since C++14)) can appear inside a member specification of any class, struct, or union that aren't local classes.

Run this code

```
#include <iostream>
#include <vector>
#include <algorithm>
struct Printer { // generic functor
   std::ostream& os;
   Printer(std::ostream& os) : os(os) {}
   template<typename T>
   void operator()(const T& obj) { os << obj << ' '; } // member template
};
int main()
{
   std::vector<int> v = {1,2,3};
   std::for_each(v.begin(), v.end(), Printer(std::cout));
   std::string s = "abc";
   std::for_each(s.begin(), s.end(), Printer(std::cout));
}
```

**Output:**

1 2 3 a b c

Partial specializations of member template may appear both at class scope and at enclosing namespace scope, but explicit specializations may only appear at enclosing namespace scope.

```
struct A {
   template<class T> struct B;         // primary member template
   template<class T> struct B<T*> { }; // OK: partial specialization
//  template<> struct B<int*> { };      // Error: full specialization
};
template<> struct A::B<int*> { };       // OK
template<class T> struct A::B<T&> { };  // OK
```

If the enclosing class declaration is, in turn, a class template, when a member template is defined outside of the class body, it takes two sets of template parameters: one for the enclosing class, and another one for itself:

```
template<typename T1>
struct string {
   // member template function
   template<typename T2>
   int compare(const T2&);
   // constructors can be templates too
   template<typename T2>
   string(const std::basic_string<T2>& s) { /*...*/ }
};
// out of class definition of string<T1>::compare<T2>
template<typename T1> // for the enclosing class template
template<typename T2> // for the member template
int string<T1>::compare(const T2& s) { /* ... */ }
```

Member function templates

Destructors and copy constructors cannot be templates. If a template constructor is declared which could be instantiated with the type signature of a copy constructor, the implicitly-declared copy constructor is used instead.

A member function template cannot be virtual, and a member function template in a derived class cannot override a virtual member function from the base class.

```
class Base {
   virtual void f(int);
};
struct Derived : Base {
   // this member template does not override Base::f
   template <class T> void f(T);

   // non-template member override can call the template:
   void f(int i) override {
       f<>(i);
   }
};
```

A non-template member function and a template member function with the same name may be declared. In case of conflict (when some template specialization matches the non-template function signature exactly), use of that name and type refers to the non-template member unless an explicit template argument list is supplied.

```
template<typename T>
struct A {
   void f(int); // non-template member

   template<typename T2>
   void f(T2); // member template
};

//template member definition
template<typename T>
template<typename T2>
void A<T>::f(T2)
{
   // some code
}

int main()
{
   A<char> ac;
   ac.f('c'); // calls template function A<char>::f<char>(int)
   ac.f(1);   // calls non-template function A<char>::f(int)
   ac.f<>(1); // calls template function A<char>::f<int>(int)
}
```

An out-of-class definition of a member function template must be equivalent to the declaration inside the class (see function template overloading for the definition of equivalency), otherwise it is considered to be an overload.

```
struct X {
   template<class T> T good(T n);
   template<class T> T bad(T n);
};

template<class T> struct identity { using type = T; };

// OK: equivalent declaration
template<class V>
V X::good(V n) { return n; }

// Error: not equivalent to any of the declarations inside X
template<class T>
T X::bad(typename identity<T>::type n) { return n; }
```

Conversion function templates

A user-defined conversion function can be a template.

```
struct A {
   template<typename T>
   operator T*(); // conversion to pointer to any type
};

// out-of-class definition
template<typename T>
A::operator T*() {return nullptr;}

// explicit specialization for char*
template<>
A::operator char*() {return nullptr;}

// explicit instantiation
template A::operator void*();

int main() {
```

```
    A a;
    int* ip = a.operator int*(); // explicit call to A::operator int*()
}
```

During overload resolution, specializations of conversion function templates are not found by name lookup. Instead, all visible conversion function templates are considered, and every specialization produced by template argument deduction (which has special rules for conversion function templates) is used as if found by name lookup.

Using-declarations in derived classes cannot refer to specializations of template conversion functions from base classes.

**12.8 NON- TYPE TEMPLATE ARUGUMENTS**:-

A non-type template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions, addresses of functions or objects with external linkage, or addresses of static class members. Non-type template arguments are normally used to initialize a class or to specify the sizes of class members.

For non-type integral arguments, the instance argument matches the corresponding template parameter as long as the instance argument has a value and sign appropriate to the parameter type.

For non-type address arguments, the type of the instance argument must be of the form identifier or &identifier, and the type of the instance argument must match the template parameter exactly, except that a function name is changed to a pointer to function type before matching.

The resulting values of non-type template arguments within a template argument list form part of the template class type. If two template class names have the same template name and if their arguments have identical values, they are the same class.

In the following example, a class template is defined that requires a non-type template int argument as well as the type argument:

```
template<class T, int size> class Myfilebuf
{
    T* filepos;
    static int array[size];
public:
    Myfilebuf() { /* ... */ }
    ~Myfilebuf();
    advance(); // function defined elsewhere in program
};Copy
```

In this example, the template argument size becomes a part of the template class name. An object of such a template class is created with both the type argument T of the class and the value of the non-type template argument size.

An object x, and its corresponding template class with arguments double and size=200, can be created from this template with a value as its second template argument:

```
Myfilebuf<double,200> x;Copy
```

x can also be created using an arithmetic expression:

```
Myfilebuf<double,10*20> x;Copy
```

The objects created by these expressio

If the template arguments do not evaluate identically, the objects created are of different types:

```
Myfilebuf<double,200> x;          // create object x of class
                                  // Myfilebuf<double,200>
Myfilebuf<double,200.0> y;        // error, 200.0 is a double,
                                  // not an intCopy
```

The instantiation of y fails because the value 200.0 is of type double, and the template argument is of type int.

The following two objects:

```
Myfilebuf<double, 128> x
```

Myfilebuf<double, 512> yCopy
are objects of separate template specializations. Referring either of these objects later with Myfilebuf<double> is an error.

A class template does not need to have a type argument if it has non-type arguments. For example, the following template is a valid class template:

```
template<int i> class C
{
public:
  int k;
  C() { k = i; }
};Copy
```

This class template can be instantiated by declarations such as:

```
class C<100>;
class C<200>;Copy
```

**SUMMARY:-**

1) Templates allows us to generate a family of classes or a family of Functions to handle differently data types.
2) Templates classes and Functions eliminate code duplication for different types and thus make the program development easier and more manageable.
3) We can use multiple parameters in both the class templates and Function templates .
4) Similarly a C++ support s a mechanism known as template to implement the concept of generic programming.specific Function created from a Function template is called a template Function.
5) Like other functions, template functions can be overloaded.
6) Member Functions of a class template must be defined as Function templates using the parameters of the class template
7) We many also use nontype parameters such basic or derived data types as arguments templates.

SRG

**Review Questions:-**

1)Write a function template for finding the minimum value contained in array

2) Writte a class template to represent a generic vector. Include member functions to perform the following tasks:

(a) to create the vector

(b) to modifiy the value of givrn element.

(c0 to multiply by a scalar value

3) write a function template to preform linear search in an array

4) Define template

5) writte a short program in Template

6)what is an excepton?

7)How is an exception handled in c++?

8) What are the advantages of using exception handling mechanism in a program?

9) When should a program throw an exception?

10) When is a catch(…)handled used?

11) Can we throw a class types as exceptions?

12) What is an exception specification? When is used?

13)what should be placed inside a try block?

14) what should be placed inside a catch block?

15)when do we use multiple catch handlers?

**DEBUGGING EXERCISE:-**

1)Identify the error in the following program.

```
#include<iostream>
using namespace std;
int main()
{
try
        {
                throw(55); //Here we input integer value;
        }
        catch(int e) //This catch takes int value;
        {
```

```
		cout<<"Input is Integer value "<<e<<endl;
	}
	catch(double e) //This catch takes double value;
	{
		cout<<"Input is Double value "<<e<<endl;
	}
	catch(char e) //This catch takes character value;
	{
		cout<<"Input is Character "<<e<<endl;
	}
	return 0;
}
```

2) Identify the error in the following program.

```
#include<iostream>
using namespace std;
int main()
{
try
	{
		throw(55.5); //Here we input double value;
	}
	catch(int e) //This catch takes int value;
	{
		cout<<"Input is Integer value "<<e<<endl;
	}
	catch(double e) //This catch takes double value;
	{
		cout<<"Input is Double value "<<e<<endl;
	}
	catch(char e) //This catch takes character value;
	{
		cout<<"Input is Character "<<e<<endl;
	}
```

```
        return 0;
}
```

3) Identify the error in the following program.

```
#include<iostream>
using namespace std;
int main()
{
        try
        {
                throw('I'); //Here we input character;
        }
        catch(int e) //This catch takes int value;
        {
                cout<<"Input is Integer value "<<e<<endl;
        }
        catch(double e) //This catch takes double value;
        {
                cout<<"Input is Double value "<<e<<endl;
        }
        catch(char e) //This catch takes character input ;
        {
                cout<<"Input is Character "<<e<<endl;
        }

        return 0;
};
```

## 13 EXCEPTION HANDLING

### 13.1 Introduction:-

Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions:

a)Synchronous

b)Asynchronous

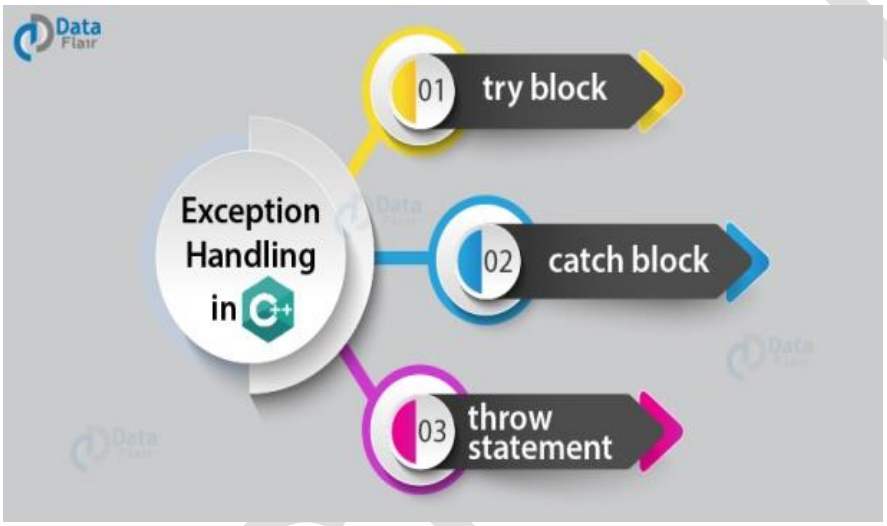



**Figure : Exception Handling**

***13.4 try*:**

represents a block of code that can throw an exception.

Synatyx:

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

SRGT

***13.5 catch*:**

represents a block of code that is executed when a particular exception is thrown.

Syntax:-

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

***13.3 throw*:**

Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Syntax:-

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

**Example Program :**

```
#include <iostream>
using namespace std;
 int main()
{
 int x = -1;
 cout << "Before try \n";
 try
{
 cout << "Inside try \n";
  if (x < 0)
 {
```

RGPGPI

```
 throw x;
 cout << "After throw (Never executed) \n";
 }
 }
 catch (int x )
{
 cout << "Exception Caught \n";
 }
cout << "After catch (Will be executed) \n";
 return 0;
}
```

**Output:**

Before try
Inside try
Exception Caught
After catch (Will be executed)

**Specifying exceptions:-**

Earlier versions of C++ allowed you to use the throw specifier on a function in three ways: firstly, you can provide a comma separated list of the types of the exceptions that may be thrown by code in the function; secondly, you can provide an ellipsis (...) which means that the function may throw any exception; and thirdly, you can provide an empty pair of parentheses, which means the function will not throw exceptions. The syntax looks like this:

```
   int calculate(int param) throw(overflow_error)     {  // do ...
```

**ADVANTAGES OF EXCEPTION HANDLING :**

***1)*** **Separation of Error Handling code from Normal Code:**

- In traditional error handling codes, there are always if else conditions to handle errors.
- These conditions and the code to handle errors get mixed up with the normal flow.
- This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

***2)*** **Functions/Methods can handle any exceptions they choose:**

- A function can throw many exceptions, but may choose to handle some of them.
- The other exceptions which are thrown, but not caught can be handled by caller.
- If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

***3)*** **Grouping of Error Types:**

In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

**C++ Standard Exceptions**

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy.

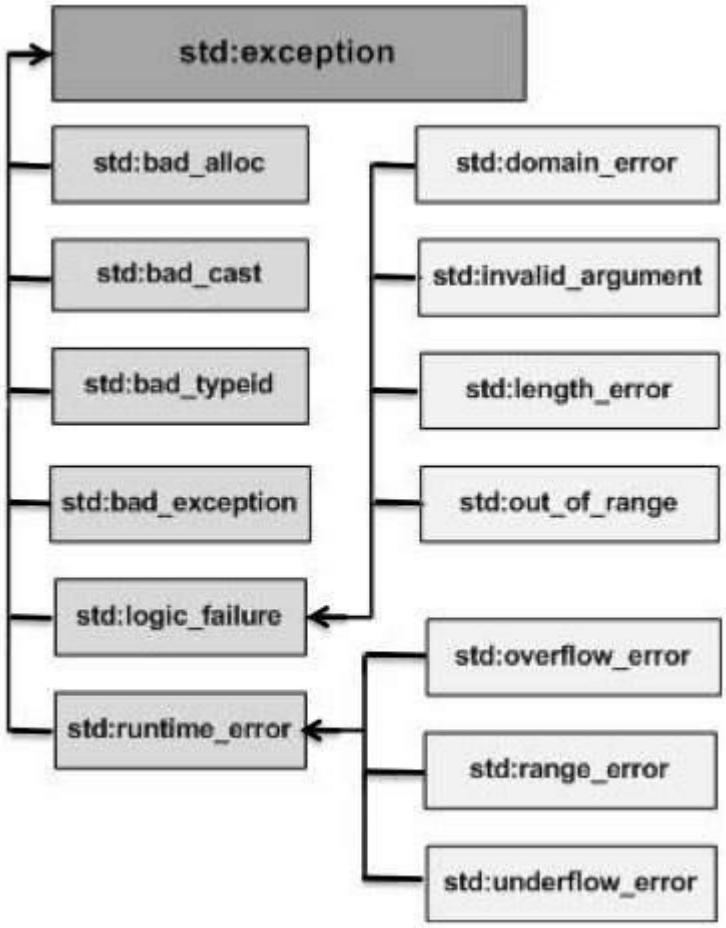



**Figure : Standard Exception**

**Multiple catch exception**

Multiple catch exception statements are used when a user wants to handle different exceptions differently. For this, a user must include catch statements with different declaration.

**Syntax**

```
try{
   body of try block
}

catch (type1 argument1)
{
   statements;
   ... ... ...
}

catch (type2 argument2)
{
   statements;
   ... ... ...
}
... ... ...
... ... ...
catch (typeN argumentN)
{
   statements;
   ... ... ...
}
```

**Catch all exceptions**

Sometimes, it may not be possible to design a separate catch block for each kind of exception. In such cases, we can use a single catch statement that catches all kinds of exceptions.

**Syntax**

```
catch (...)
{
   statements;
   ... ... ...
}
```

**13.8 Exception in constructors and destructors**:-

Exceptions can also be thrown in constructors and destructors. These are cases particularly hard to handle.

If an exception is thrown inside a constructor, the object in statu nascendi will not be completed and its destructor will not be invoked (what is logical, because formally the object has not even been created). However, it may happen that the object has object-type members which have already been fully constructed.

These object will be destroyed and their destructors will be executed. However, the object can contain pointer-type members pointing to already allocated objects: their destructors will not be called. The same holds for other resources, like open files or data base connections — they are usually released and closed in the destructor, which will not be called. All this can lead to serious memory leakages and/or problems with open files, data base connections and the like.

We can often avoid such problems by wrapping members referring to resources acquired so far by a constructor in member objects equipped with appropriate destructors: when the constructor fails, these destructors, being destructors of already complete member objects, will be executed. Let us look at an example

**Example : Releasing resources after exceptions**

```
1. #include <iostream>
2. #include <cstring>
3. #include <cstdio>  // FILE, fopen, fclose
4. using namespace std;
5.
6. class A {
7.     struct nam {
8.         char* n;
```

```
 9.         nam(const char* n)
10.             : n(strcpy(new char[strlen(n)+1],n))
11.         { }
12.         ~nam() {
13.             cerr << "dtor nam: " << n << endl;
14.             delete [] n;
15.         }
16.     };
17.
18.     nam Name;
19.     FILE*    file;
20.  public:
21.     A(const char* n, const char* p)
22.         : Name(n)
23.     {
24.         file = fopen(p,"r");
25.         // ...
26.  //     throw 1;
27.         // ...
28.     }
29.
30.     // other fields and methods
31.
32.     ~A() {
33.         cerr << "dtor A" << endl;
34.         if (file) fclose(file);
35.     }
36.  };
37.
38.  int main() {
39.     try {
40.         A a("Carrington","afile.cpp");
41.     } catch(...) {
42.         cerr << "object instantiation failed\n";
```

43. }
44. }

Class A has a member corresponding to a name represented as a C-string. The string is dynamically allocated , but the pointer to it is not a member of class directly: instead, there is an auxiliary structure nam, and it is an object of this class which is a member of A. The pointer to the C-string with the name is a member of this object. The structure nam has a destructor, which therefore will be called if constructor of A fails when the member object Name has already been cretaed. The destructor takes care of the memory allocated for the name.

Class A has additional pointer as the field: this is a pointer to an object of structure FILE (this is a standard type describing files in C).

What will happen if the line 26 (throw 1) is commented out? The object is created properly; no exception is thrown. When the flow of control leaves the try block, the object a of class A, being a local object in the block, is removed and its destructor is invoked which, in turn, closes the file. Then member objects are removed and their destructors called; in our case it is the member object Name which is destroyed — its destructor will deallocate memory occupied by the C-string with the name. The printout

```
dtor A
dtor nam: Carrington
```

shows that object a has been removed correctly.

Let us now activate the line 26, which causes an exception to be thrown during the process of constructing the object a. At that time, member Name has already been created, so its destructor is called and the memory for the C-string is released. However, as the printout shows

```
dtor nam: Carrington
object instantiation failed
```

In the example above, exception thrown in the constructor was not handled in situ, before leaving the constructor; it escapes from the constructor and is handled by a catch block in the calling function (main, in our case). This could be dangerous in the case of a destructor. The problem here is that destructors are called automatically in the process of

rewinding the stack while handling an exception. An additional exception in a destructor would then lead to a situation when there are two exceptions being handled simultaneously. This is not possible in C++. If this happens, the program is unavoidably terminated by calling the function terminate. Therefore, if an exception can occur in a destructor, it should be handled by an appropriate catch block inside the destructor, thus preventing it from escaping to the caller.

**SUMMARY :-**

- Exceptions are peculiar problems that a program may encounter at run time.
- Exceptions are of two types: synchronous and asynchronous, c++ provides mechanism for handling synchronous exceptions.
- The catch statement defines a block of statements to handle the exception appropriately.
- When an exception is not caught the program is abroted.
- We can place two or more catch blocks together to catch and handle multiple types of exceptions thrown by a try block.
- It is also possible to make a catch statement to catch all types of exceptions by adding a throw specification clause to the Function definition.

**Review Questions:-**

*1) what are the steps involved in using a file in a c++ program?*

*2) Describe the various classes available for file operations*

3) Explain how while(fin) statement detects the end of a file that connected to fin stream.

4)What is a file mode?

5)Describe the various file mode options available

6) Write a statement that will create an object called fob for writing and associate it with a file name DATA

7)What are the Advantages of saving data in binary form?

8) describe how would you determine number of objects in a file. When do you need such information?

9) Describe the various approaches by which we can detect the end- *of-file condition successfully.*

**DEBUGGING EXERCISE:-**

*1) Identify the error in the following program.*

```
#include <iostream>
#include <fstream.h>
void main () {
  ofstream file;
  file.open ("egone.txt");
  file << "Writing to a file in C++....";
  file.close();
  getch();
}
```

*2) Identify the error in the following program*

```
#include <iostream>
#include<fstream.h>
void main()
{
  char c,fn[10];
  cout<<"Enter the file name....:";
```

```
 cin>>fn;
 ifstream in(fn);
 if(!in)
 {
   cout<<"Error! File Does not Exist";
   getch();
   return;
 }
 cout<<endl<<endl;
 while(in.eof()==0)
 {
   in.get(c);
   cout<<c;
 }
 getch();
}
```

3) Identify the error in the following program.

```
#include <iostream>
#include<fstream.h>
#include<math.h>
void main()
{
 ofstream fileo("Filethree");
 fileo<<"Hello GS";
 fileo.close();
 ifstream fin("Filethree");
 char ch;
 while(fin)
 {
  fin.get(ch);
  cout<<ch;
 }
 fin.close();
 getch();}
```

# 16 OBJECT ORIENTED SYSTEM DEVELOPMENT

## 16.1 INTRODUCTION:-

- Object oriented systems development is a way to develop software by building self – contained modules or objects that can be easily replaced, modified andreused.
- In an object–oriented environment, software is a collection of discrete objects that encapsulate their data as well as the functionality of model real–world events "*objects*" and emphasizes its cooperative philosophy by allocating tasks among the objects of the applications.
- An object orientation produces systems that are easier to evolve, more flexible,more robust, and more reusable than a top – down approach.
  An object orientation
- ***Allows higher level of abstraction***: Object oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) & functions (methods), they work at a higher level of abstraction. This makes designing, coding, testing & maintaining the system much simpler.

### *16.3 Object Oriented System Development : -*

- Object oriented systems development is a way to develop software by building self – contained modules or objects that can be easily replaced, modified andreused.
- In an object–oriented environment, software is a collection of discrete objects that encapsulate their data as well as the functionality of model real–world events "*objects*" and emphasizes its cooperative philosophy by allocating tasks among the objects of theapplications.
- A *class* is an object oriented system carefully delineates between its interface (specifications of what the class can do) and the implementation of that interface (how the class does what itdoes).

SRGPGPI

## 16.2 Procedure oriented program:-

High level languages such as COBOL, FORTRAN and C, is commonly known as procedure oriented programming(POP). In the procedure oriented programming, program is divided into sub programs or modules and then assembled to form a complete program. These modules are called functions.

In a multi-function program, many important data items are placed as global so that they may be accessed by all functions. Each function may have its own local data. If a function made any changes to global data, these changes will reflect in other functions. Global data are more unsafe to an accidental change by a function. In a large program it is very difficult to identify what data is used by which function.

### Characteristics of Procedure Oriented Programming

- C++ is an Object Oriented Programming Language (OOPL).
- C++ have huge Function Library.
- C++ is highly Flexible language with Versatility.
- C++ can be used for developing System Software viz., operating systems, compilers, editors and data bases.
- C++ is suitable for Development of Reusable Software. , thus reduces cost of software development.
- C++ is a Machine Independent Language.

### Object oriented paradigm

### A Brief History

- The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.

- The first object–oriented language was Simula (Simulation of real systems)

SRGPGPI

that was developed in 1960 by researchers at the Norwegian Computing Center.

- In 1970, Alan Kay and his research group at Xerox PARK created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL) - Smalltalk, for programming the Dynabook.

- In the 1980s, Grady Booch published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object–oriented design method.

- In the 1990s, Coad incorporated behavioral ideas to object-oriented methods.

- The other significant innovations were Object Modelling Techniques (OMT) by James Rumbaugh and Object-Oriented Software Engineering (OOSE) by Ivar Jacobson.

## 17.2 Object-Oriented Programming Paradigm

- The major motivating factor in the invention of object-oriented approch is to remove some of the flaws encountered in the procedural approch.
- OOP treats data as a critical element in the program development and does not allow it to flow freely around the systems.
- It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.
- OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.
- The data of an object can be accessed only by the function associated with that object.
- However, functions of one object can access the the functions of other objects.

RGPGPI

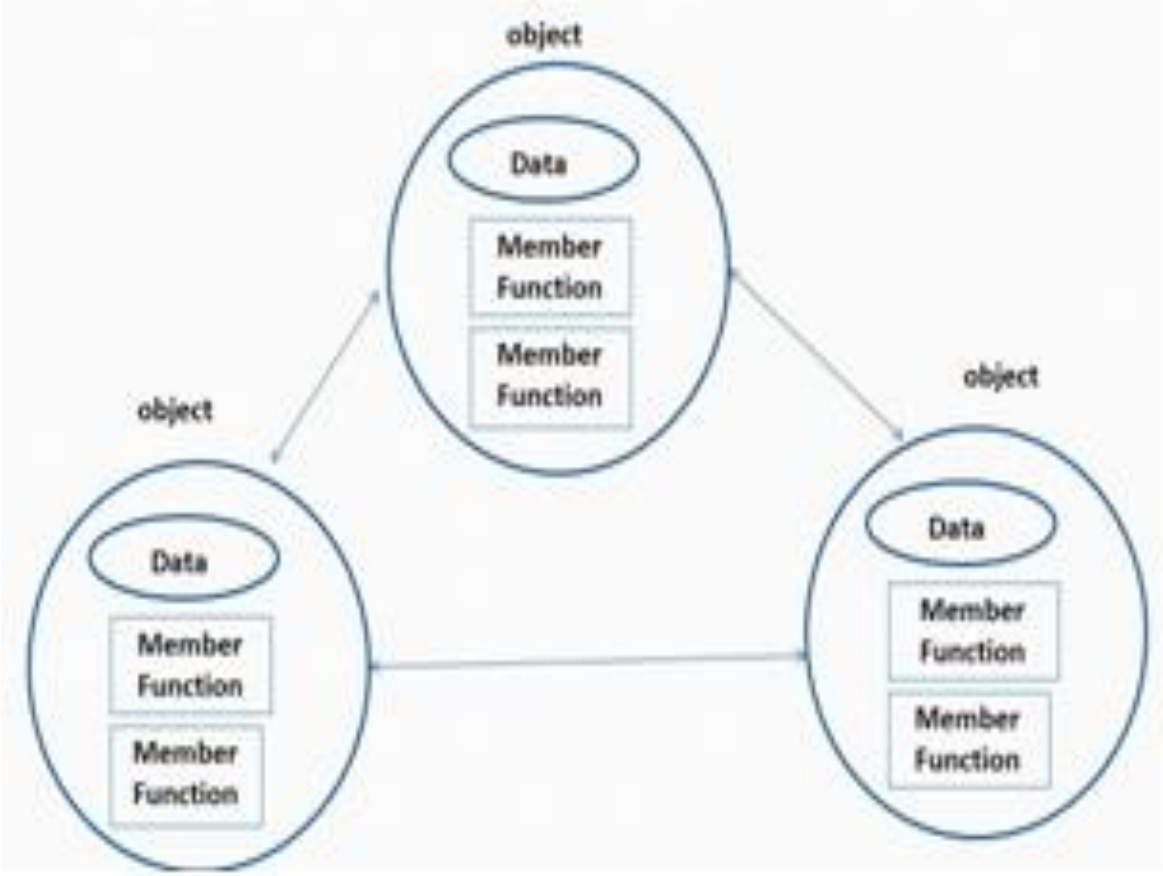



**Figure : Object Oriented Paradigm**

### 16.6 Object-Oriented Analysis

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

SRGI

Grady Booch has defined OOA as, "Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain".

The primary tasks in object-oriented analysis (OOA) are –

**Identifying objects**

Organizing the objects by creating object model diagram

Defining the internals of the objects, or object attributes

Defining the behavior of the objects, i.e., object actions

Describing how the objects interact

The common models used in OOA are use cases and object models.

### 16.7 Object-Oriented Design

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include −

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.
- Grady Booch has defined object-oriented design as "a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

SRGPGPI

**Process of Object Oriented Design**:

Understanding the process of any type of software related activity simplifies its development for the software developer, programmer and tester. Whether you are executing functional testing, or making a test report, each and every action has a process that needs to be followed by the members of the team. Similarly, Object Oriented Design (OOD) too has a defined process, which if not followed rigorously, can affect the performance as well as the quality of the software. Therefore, to assist the team of software developers and programmers, here is the process of Object Oriented Design (OOD):

1. To design classes and their attributes, methods, associations, structure, and even protocol, design axiom is applied.
    - The static UML class diagram is redefined and completed by adding details.
    - Attributes are refined.
    - Protocols and methods are designed by utilizing a UML activity diagram to represent the methods algorithm.
    - If required, redefine associations between classes, and refine class hierarchy and design with inheritance.
    - Iterate and refine again.
2. Design the access layer.
    - Create mirror classes i.e., for every business class identified and created, create one access class.
3. Identify access layer class relationship.
4. Simplify classes and their relationships. The main objective here is to eliminate redundant classes and structures.
    - **Redundant Classes:** Programmers should remember to not put two classes that perform similar translate requests and translate results activities. They should simply select one and eliminate the other.
    - **Method Classes:** Revisit the classes that consist of only one or two methods, to see if they can be eliminated or combined with the existing classes.
5. Iterate and refine again.
6. Design the view layer classes.
    - Design the macro level user interface, while identifying the view layer objects.
    - Design the micro level user interface.
    - Test usability and user satisfaction.

- Iterate and refine.

7. At the end of the process, iterate the whole design. Re-apply the design axioms, and if required repeat the preceding steps again.

**16.8 Concepts of Object Oriented Design**:

In Object Oriented Design (OOD), the technology independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and the interfaces are designed, which results in a model for the solution domain. In short, a detailed description is constructed to specify how the system is to be built on concrete technologies. Moreover, Object Oriented Design (OOD) follows some concepts to achieve these goals, each of which has a specific role and carries a lot of importance. These concepts are defined in detail below:

1. **Encapsulation:** This is a tight coupling or association of data structure with the methods or functions that act on the data. This is basically known as a class, or object (object is often the implementation of a class).
2. **Data Protection:** The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as private or protected to the owning class.
3. **Inheritance:** This is the ability of a class to extend or override the functionality of another class. This so called child class has a whole section that is the parent class and then it has its own set of functions and data.
4. **Interface:** A definition of functions or methods, and their signature that are available for use as well as to manipulate a given instance of an object.
5. **Polymorphism:** This is the ability to define different functions or classes as having the same name, but taking different data typ

**Advantages of Object Oriented Design:**

The discussion above has elaborated on several advantages of Object Oriented Design (OOD). From enabling the implementation of a software based on the concepts of objects and deleting the shared data areas to distributing and executing the object sequentially or in parallel, the benefits of this approach of software design are numerous. Hence, provided here some of the other advantages of using Object Oriented Design (OOD).

- Easier to maintain objects.
- Objects may be understood as stand-alone entities.
- Objects are appropriate reusable components.
- For some systems, there may be an obvious mapping from real entities to system objects.

**Object-Oriented Programming**

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are −

- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes
- Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Grady Booch has defined object–oriented programming as "a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships".

**Building High–quality Software (BHS) :**

Ultimate goal of BHS is user satisfaction. Blum describes a means of system evaluation in terms of four quality measures: correspondence, correctness, verification & validation.

SRGPGPI

- **Correspondence** measures how well the delivered system matches the needs of the operational environment as described in the original requirements statement. **Correctness** measures consistency of product requirements with respect to the design specification.
- **Verification** is the exercise of determining correctness.
- **Validation** is the task of predicting correspondence. True correspondence cannot be determined until the system is in place.

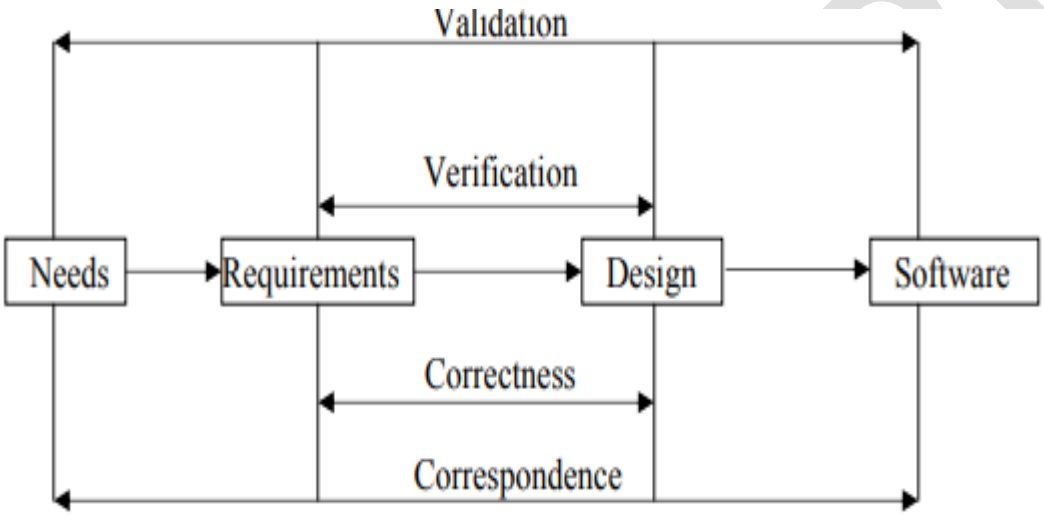



**Figure : High Quality Software**

***Need of Object Orientation***

An object orientation produces systems that are easier to evolve, more flexible,more robust, and more reusable than a top – down approach. An objectorientation

- ***Allows higher level of abstraction***: Object oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) & functions (methods), they work at a higher level of abstraction. This makes designing, coding, testing & maintaining the system much simpler.

- ***Provides Seamless transition among different phases of software development***: Object oriented approach, essentially uses the same language to talk about analysis, design, programming and database design. This seamless approach reduces the level of complexity and redundancy and makes for clearer, more

robust systemdevelopment.

- ***Encourage good development techniques***: In a properly designed system, the routines and attributes within a class are held together tightly, the classes will be grouped into subsystems but remain independently and therefore, changing one class has no impact on other classes and so, the impact isminimized.
- ***Promotes of reusability***: Objects are reusable because they are modeled directly out of a real – world problem domain. Here classes are designed, with reuse as a constant background goal. All the previous functionality remains and can be reused withoutchanged.

***Object Basics: -***

- ***Objects are Grouped in Classes*** **:**

  Class is a set of objects that share a common structure and a common behavior, a single object is simply an instance of a class. A class is a specification of structure (instance variables), behavior (methods) and inheritance for objects. A method or behavior of an object is defined by itsclass.

- ***Attributes*****:**

  Object state and properties. Properties represent the state of an object. Often, we refer to the description of these properties rather than how they are represented in a particular programminglanguage.

- ***Object Behavior and Methods*****:**

  Behavior denotes the collection of methods that abstractly describes where an object is capable of doing. Methods encapsulate the behavior of the object, provide interfaces to an object and hide any of the internal structures and states maintained by theobject.

- ***Objects respond to Messages*****:**

  Message is the instruction and method is the implementation. An object understands a message when it can match message to a method that has same name as ofit.

SRGPGPI

- ***Encapsulation and Information Binding*:**

  Information hiding is principle of concealing the internal data & procedures of an object and providing an interface to each object in such a way as to reveal as little as possible about its inner workings. Encapsulation protection deals with protection of an object capsule from other object by means of per–class protection and per–objectprotection.

- ***Class Hierarchy***:

  At the top of the class hierarchy are the most general classes & at the bottom are themostspecific.Asubclassinheritsalloftheproperties andmethodsdefinedinitssuperclass.

- ***Object Relationships*:**

  Association represents the relationships between objects and classes. Associations are bidirectional with different annotations. Cardinality specifies how many instances of one class may relate to a single instance of an associated class.


- ***Aggregation and Object Containment***: As each object has an identity, one object can refer to other objects and is known as aggregation, where an attribute can be an objectitself.

***Object – Oriented System Development Life Cycle :-***

- The basic software development life cycle consists of *analysis, design, implementation, testing and refinement*. Its main aim is to transform users' needs into a softwaresolution.
- The development is a process of change, refinement, transformation or addition to the existing product. The software development process can be viewed as a series

SRGPGPI

of transformations, where the output of one transformation becomes input of the subsequenttransformation.

**16.9 Prototyping :**

A prototype is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes.

It enables to fully understand how easy or difficult it will be to implement some of the features of the system.

Prototyping can further define the use cases, and it actually makes use–case modeling much easier. The main idea is to build a prototype (use–case modeling) to design systems that users like & need. o It provides the developer a means to test & refine user interface & increase usability of system.

The following categories are some of accepted prototypes each having its own strength.

⇒ Horizontal Prototype: It is a simulation of interface but contain no functionality. Advantages: Very quick to implement, good overall of system, user to evaluate interface on normal base

⇒ Vertical Prototype: It is a subset of system features with complete functionality. Advantage: few implemented functions can be tested in great depth ⇒ Analysis Prototype: It is an aid for exploring problem domain. It used to inform user & demonstrate proof of a concept

⇒ Domain Prototype: It is an aid for incremental development of ultimate software development o Prototyping should involve representation from all user groups that will be affected by the project, especially the end users and management members to ascertain that the general structure of the prototype meets the requirements established for the over all design.

The purpose of the review is three fold are

1. To demonstrate that the prototype has been developed according to the specification and that the final specification is appropriate.

2. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.

3. To give management and everyone connected with the project the first glimpse of what the technology can provide. o Prototyping is a useful exercise at almost any stage of the

SRGPGPI

development. It should be done in parallel with the preparation of the functional specification.

**Object Oriented Methodologies :**

Numbers of methodology are based on modeling the business problem and implementing the application in an object oriented fashion.

The major differences between different methodologies lie primarily in the documentation of information, and modeling notations and language.

The **three major methodologies** and their modeling notations developed by Rumbaugh et al.,

- Booch and Jacobson which are the origins of the Unified Modeling Language.

Each method has its strengths.

- Rambaugh method is well-suited for describing the object model or the static structure of the system.
- The Jacobson et al. method is good for producing user – driven analysis models.

**Patterns:**

A pattern is instructive information that captures essential structure, insight of a successful family of proven solutions to a recurring problem that arises within a certain context & system of forces.

A pattern involves a general description of a solution to a recurring problem bundle with various goals and constraints.

**A good pattern will do the following:**

- It solves a problem. Patterns capture solutions, not just abstract principles or strategies

• It is a proven concept. Patterns capture solutions with a tract record, not theories or speculation

• The solution is not obvious. Best patterns generate a solution to problem indirectly in design • It describes a relationship. Patterns describe deeper system structures and mechanisms

• The pattern has a significant human component. All s/w serves human comfort or quality of life; best patterns explicitly appeal to aesthetics and utility.

The following essential components be clearly mentioned:



DGPI

⇒ Name: A meaningful name must be given along with nickname if exists

⇒ Problem: A statement describing goals, objectives to reach within given context & forces

⇒ Context: Preconditions of problem recurring & desirable solution stating applicability

⇒ Forces: Reveal Motivation factors for s/w necessity & encapsulate all forces impact on it

⇒ Solution: It should describe static structure & dynamic behavior

⇒ Examples: It may be supplemented by sample implementation to show one way to get solution

⇒ Resulting Context: Has post-conditions, side effects of pattern said as resolution of forces

⇒ Rationale: Provides insight into deep structures & key mechanisms going deep into system ⇒ Related Patterns: Must have static & dynamic relationships between types of similar patterns

⇒ Known uses: Occurrences of pattern , its application and proven solution of recurring problem o A good pattern often begins with an abstract that provides a short summary or overview.

This gives readers a clear picture of pattern and quickly informs them of its relevance to any problems they may solve sometimes called a thumbnail sketch of pattern or pattern thumbnail o Antipatterns:

It represents worst practice or a lesson learned come in two varieties those describing

1. a bad solution to a problem that resulted in a bad situation

2. how to get out of a bad situation & how to proceed from there to good ones o Capturing Patterns: Process of looking for patterns to document is pattern mining or reverse architecting and guidelines and certain guidelines to get pattern of such type are

⇒ Focus on practicability: Patterns should describe proven solutions to recurring problems

⇒ Aggressive disregard of originality: Patterns writers need to be original inventor of document ⇒ Non-anonymous review: Pattern submitted to shepherd & they discuss with authors

⇒ Writers' workshops instead of presentations: Patterns are discussed in workshops to improve

⇒ Careful editing: Authors have opportunity to improve in shepherd comments & insights

**SUMMARY:-**

**Review Questions:-**

1)How does a string type string differ from a c-type string?

2) the following statements are available to read strings from the keyboard

(a) cin>>s1;

(b)getline(cin,s1)

3)consider the following segment of a program

String s1(“man”),s2,s3:

S2.assign (s1);

S3= s1;

String s4(“wo” +s1);

S1[0] =’v’;

4) we can access string elements using

(a) at()function

(b) subscribt operator[]

Comparetheir behaviour.

5) what does each of the following statements do?

(a) s.replace(n,1”/”);

(b)s.erase(10);

(c) s1.insert(10,s2);

6) List five most important in feature ,that a software developer should keeping mind while designing a system?

7) Describe why the testing software is important

8)list out the key difference between procedure oriented an object oriented program

9) what do you mean by maintenance of software?

10)”software development process is an interative process” discuss?

**DEBUGGING EXERCISE:-**

1) Identify the error in the following program.

```
#include <string.h>
 /* strcat */
 char *(strcat)(char *restrict s1, const char *restrict s2)
 {
    char *s = s1;
    /* Move s so that it points to the end of s1.  */
    while (*s != '\0')
       s++;
    /* Copy the contents of s2 into the space at the end of s1.  */
    strcpy(s, s2);
    return s1;
 }
```

2 ) Identify the error in the following program

```
#include <string.h>
 /* strchr */
 char *(strchr)(const char *s, int c)
 {
    char ch = c;
    /* Scan s for the character.  When this loop is finished,
       s will either point to the end of the string or the
       character we were looking for.  */
    while (*s != '\0' && *s != ch)
       s++;
    return (*s == ch) ? (char *) s : NULL;
 }
```

3) Identify the error in the following program

```
#include <string.h>
 /* strcmp */
 int (strcmp)(const char *s1, const char *s2)
 {
    unsigned char uc1, uc2;
    /* Move s1 and s2 to the first differing characters
       in each string, or the ends of the strings if they
       are identical.  */
    while (*s1 != '\0' && *s1 == *s2) {
       s1++;
       s2++;
    }
    /* Compare the characters as unsigned char and
       return the difference.  */
    uc1 = (*(unsigned char *) s1);
    uc2 = (*(unsigned char *) s2);
    return ((uc1 < uc2) ? -1 : (uc1 > uc2)
}
```

## PRACTICAL II- C++ PROGRAMMING LAB

**OBJECTIVE:**

To impart practical in C++ programming language.

1.CLASSES:

Write a program using a class to represent a bank account with data members –name of depositor, account number, type of account and balance and member functions-deposit amount-withdrawal amount show name and balance check the program with own data

2.CONSTRUCTOR & DESTRUCTOR:

Write a program to read an integer and find the sum of all the digits until it reduces to a single digit using constructor, destructor and default constructor

3.DEFAULT & REFERENCE ARGUMENT:

Write a program using function overloading to read two matrices of different data types such as integers and floating point numbers. Find out the sum of the above matrices separately and display the total sum of these arrays individually

4.OPERATOR OVERLOADING:

a. addition of two complex numbers

b. matrix multiplication

5.INHERITANCE

Prepare pay roll of an employee using inheritance

6.POINTERS:

a. write a program to find the number of vowels in a given text

b. write a program to check for palindrome

7.FILES:

Prepare students mark list in a file with student number, mark in four subjects and mark total. Write a program to arrange these records in the ascending order of mark total and write them in the same file overwriting the earlier records

SRGPGPI

8.EXCEPTION HANDLING:

Prepare electricity bill for customers generating and handling any two exceptions.

SRGPGPI

## EX .NO:1 BANK TRANSACTION

**AIM:**

To write a c++ program using class to represent a bank transaction

**ALGORITHM:**

Step 1: start the program

Step 2: real account no,dep,name,type acc,balance value

Step 3: check if [ral<500] if the condition is true check the draw all account

Step 4: if balance wd yes with draw all account is sanction

Step 5: calculate balance -1=d balance=-d

Step 6: print dep,name,acc no,balance

Step 7: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
int capital;
char acctype[50],accname[50];
class account
{
public:
account()
{
cout<<"enter the customername,acctype,capital amt"<<endl;
cin>>accname;
cin>>acctype;
cin>>capital;
}
};
class transcation
{
public:
int widamt,depamt,balance;
void withdraw()
{
cout<<"enter the withdraw amount:";
cin>>widamt;
if(capital>500&&widamt<capital)
{
capital=capital-widamt;
cout<<"The customer name is:"<<accname<<endl;
cout<<"The account type is:"<<acctype<<endl;
cout<<"after withdraw remaining balance:"<<capital<<endl;
}
else
```

SRGPGPI

```
{
cout<<"insuficient fund";
}
}
void deposit()
{
cout<<"enter the deposit amount"<<"\n";
cin>>depamt;
if(capital>=depamt)
{
capital=capital+depamt;
cout<<"The customer nameis:"<<accname<<endl;
cout<<"The account type is:"<<acctype<<endl;
cout<<"after deposit remaining balance:"<<capital<<endl;
}
}
void bal()
{
cout<<"The customer name is:"<<accname<<endl;
cout<<"The account type is:"<<acctype<<endl;
cout<<"your current balance is:"<<capital<<endl;
}
};
void main()
{
clrscr();
account a;
cout<<" Enter 1.withdraw 2.deposit 3. current balance"<<endl;
int option;
cin>>option;
if(option==1)
{
transcation t;
t.withdraw ();
```

SRGPGPI

```
}
else if(option==2)
{
transcation t;
t.deposit ();
}
else if(option==3)
{
transcation t;
t.bal ();
}
getch();
}
```

SRGPGPI

**OUTPUT:**

enter the customer name, acctype, capital amt
janani
savings
2000

Enter 1.withdraw 2.deposit 3.current balance
1
Enter the withdraw amount : 500
The customer name is: janani
The account type is: savings
After withdraw remaining balance: 1500

**RESULT:**

Thus, the output is verified using a c++ program

SRGPGPI

## EX NO:2 SUM OF DIGITS

**AIM:**

To write a c++ program to read on integers and find the sum of all the digits unit it reduced to a single digits

**ALGORITHM:**

Step 1: start the program

Step 2: declare a sum as long integer variable

Step 3: read the value of n

Step 4: call the function sum digits(n)

Step 5: declare d, ras integer variable and assing r=0

Step 6: using while loop check n!=0

Step 7: if the condition is true ,then calculate the following steps

i)d=n%10

ii)r=r+d

iii)n=n/10

Step 8: continue while loop until the condition is true

Step 9: return the value of n stored in sum

Step 10: print the value of sum

Step 11: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
class digits
{
int sum,i,n,a[100];
public:
digits(void);
~digits(void);
};
digits::digits()
{
cout<<"enter the number of digits"<<endl;
cin>>n;
for(i=0;i<=n;i++)
{
cout<<"enter the number"<<i+1<<endl;
cin>>a[i];
}
cout<<"the given digits are"<<endl;
for(i=0;i<n;i++)
{
cout<<"\t"<<a[i]<<"\t"<<endl;
}
sum=0;
for(i=0;i<n;i++)
{
sum=sum+a[i];
}
cout<<"the sum of all the digits are"<<sum<<endl;
}
digits::~digits()
```

SRGPGPI

```
{
cout<<"destructor function call object destroyed"<<endl;
}
void main()
{
clrscr();
{
digits s;
}
getch();
}
```

SRGPGPI

**OUTPUT:**

```
enter the number of digits
3
enter the number1
2
enter the number2
3
enter the number3
2
enter the number4
3
the given digits are
        2
        3
        2
the sum of all the digits are 7
destructor function call object destroyed
```

**RESULT:**

Thus, the output is verified using c++ program

SRGPGPI

## EX. NO:3 FUNCTION OVERLOADING

**AIM:**

To write a c++ program using function overloading to read matrics of different data types such as floating point numbers and integers

**ALGORITHM:**

Step 1: start the program

Step 2: declare the variable a,b,c,d result as integer data type

Step 3: read the value of a and b

Step 4: call the function sum(a,b)

Step 5: add the value of a and b return to the function calling

Step 6: return value stored in results

Step 7: print the value of result

Step 8: read the values

Step 9: call the function sum(a,b,c)

Step 10: add the value of a,b,c and return to function calling

Step 11: return value is stored in result

Step 12: print the value of a,b,c and d

Step 13: call the function sum(a,b)

Step 14: add the values of a,b,c and return to the calling function

Step 15: return values stored in result

Step 16: print the value of result

Step 17: declare the variable and number[50] as integer data type

Step 18: using the for loop i=0 do the following

Step 19: read the value of number [i]

Step 20: continue for loop until the condition is true

Step 21: call the function sum(num,n)

Step 22: declare sum as integer value and assign sum=0

Step 23: using for loop i=0 to n do the following

Step 24: calculate sum=sum+a[i] continue for loop until the condition is true

Step 25: return the value of sum

Step 27: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
class matrix
{
int a[3][3],b[3][3],c[3][3],i,j;
float d[3][3],e[3][3];
public:
void addition(void);
void addition(float f[3][3]);
};
void matrix::addition(void)
{
cout<<"enter 9 elements for A matrix :";
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
{
cin>>a[i][j];
}
}
cout<<"enter 9 elements for B matrix:";
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
{
cin>>b[i][j];
}
}
cout<<"the added integer value:"<<endl;
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
```

SRGPGPI

```
{
c[i][j]=a[i][j]+b[i][j];
cout<<c[i][j]<<"\t";
}
cout<<"\n";
}
}
void matrix::addition(float f[3][3])
{
cout<<"enter 9 elements for D matrix: ";
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
{
cin>>d[i][j];
}
}
cout<<"enter 9 elements for E matrix:";
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
{
cin>>e[i][j];
}
}
cout<<"the added float vlaue :"<<endl;
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
{
f[i][j]=d[i][j]+e[i][j];
cout<<f[i][j]<<"\t";
}
cout<<"\n";
```

SRGPGPI

```
}
}
void main()
{
char op;
float f[3][3];
clrscr();
matrix m;
cout<<"enter your option here \n\t a.integer addition \n\t b.float addition" <<endl;
cin>>op;
if (op=='a')
{
m.addition();
}
else if(op=='b')
{
m.addition(f);
}
else
{
cout<<"invalid option"<<endl;
}
getch();
}
```

SRGPGPI

**OUTPUT:**

enter your option here

a.integer addition

b.float addition

a

enter 9 elements for A matrix :

2 2 2

2 2 2

2 2 2

enter 9 elements for B matrix :

4 4 4

4 4 4

4 4 4

the added integer value :

6 6 6

6 6 6

6 6 6

**RESULT:**

Thus ,the output is verified using a c++ program

SRGPGPI

## EX.NO:4(a) ADDITION OF TWO COMPLEX NUMBERS

**AIM:**

To write a c++ program to add two complex number using operator overloading

**ALGORITHM:**

Step 1: start the program

Step 2: declare read and imaginary as integer type

Step 3: calculate if (image >0) read image else read<image

Step 4: if operator nx real=read+c1,read x,image+c2 image return x

Step 5: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
class complex
{
float x,y;
public:
complex()
{
}
complex(float real,float imag)
{
x=real;
y=imag;
}
void disp()
{
cout<<"the complex number is:"<<x<<"+"<<y<<endl;
}
complex operator+(complex&c);
};
complex complex::operator+(complex&c)
{
complex t;
t.x=x+c.x;
t.y=y+c.y;
return t;
}
void main()
{
```

SRGPGPI

```
float a,b,c,d;
clrscr ();
cout<<"Enter FOUR float number to add:"<<endl;
cin>>a>>b>>c>>d;
complex c1(a,b),c2(c,d),c3;
c3=c1+c2;
c1.disp ();
c2.disp ();
c3.disp ();
getch ();
};
```

SRGPGPI

**OUTPUT:**

Enter FOUR float number to add:

2.4

3.5

4.5

2.5

The complex number is: 2.4+3.5

The complex number is: 4.5+2.5

The complex number is: 6.9+6

**RESULT:**

Thus, the output is verified using a c++ program

SRGPGPI

## EX.NO:4(b) MATRIX MULTIPLICATION

**AIM:**

To devolope c++ program for multiplying two matrix using operator overloading

**ALGORITHM:**

Step 1: start the program

Step 2: to creat a class matrix with the necessary variable and member function

Step 3: declare type as function overloading using operator and return types as class name

Step 4: creat three object for the class matrix

Step 5: read the matrix value using the two object

Step 6: using the above two object and store the value in oject

Step 7: display the matrix value using the objects

Step 8: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
class multi
{
int a[3][3],b[3][3],c[3][3],i,j,k;
public:
void get();
void disp(void);
void  operator*();
};
void multi::get()
{
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
cout<<"enter the integer value: a["<<i<<"]["<<j<<"]:" ;
cin>>a[i][j];
cout<<"enter the integer value: b["<<i<<"]["<<j<<"]:";
cin>>b[i][j];
}
}
cout<<"the integer value of 3*3 matrix a,b is :"<<endl;
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
cout<<" "<<a[i][j]<<"\t";
}
cout<<endl<<endl;
}
```

SRGPGPI

```
cout<<endl<<endl;
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
cout<<" "<<b[i][j]<<"\t";
}
cout<<endl<<endl;
}
}
void multi::disp()
{
cout<<"the output is:"<<endl<<endl;
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
cout<<" "<< c[i][j]<<"\t";
}
cout<<endl<<endl;
}
}
void multi::operator*()
{
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
c[i][j]=0;
for(k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}
}
```

SRGPGPI

```
}
}
void main()
{
clrscr();
multi m;
m.get();
*m;
m.disp();
getch();
}
```

SRGPGPI

**OUTPUT:**

enter the integer value :a[0][0]:1 1
enter the integer value :b[0][0]:enter the integer value :a[0][1]:1 2
enter the integer value :b[0][1]:enter the integer value :a[0][2]:1 3
enter the integer value :b[0][2]:enter the integer value :a[1][0]:2 1
enter the integer value :b[1][0]:enter the integer value :a[1][1]:2 2
enter the integer value :b[1][1]:enter the integer value :a[1][2]:2 3
enter the integer value :b[1][2]:enter the integer value :a[2][0]:3 1
enter the integer value :b[2][0]:enter the integer value :a[2][1]:3 2
enter the integer value :b[2][1]:enter the integer value :a[2][2]:3 3
enter the integer value :b[2][2]:the integer value of 3*3 matrix a,b is:
1 1 1
2 2 2
3 3 3

1 2 3
1 2 3
1 2 3

the output is

3 6 9
6 12 18

**RESULT:**

Thus, the output is verified using c++ program

SRGPGPI

## EX.NO:5 INHERITANCE

**AIM:**

To write the c++ program to prepare pay roll

**ALGORITHM:**

Step 1: start the program

Step 2: declare the nessary variable

Step 3: read the declared variables

Step 4: calculate the hra,da,pf,lic,gn net basic

Step 5: print the result in arranged format

Step 6: stop the program

SRGPGPI

**PROGRAM**

```
#include<iostream.h>
#include<conio.h>
class emp
{
public:
char empname[50],empaddress[50];
long empnumber;
int basicsalary;
void get()
{
cout<<"enter the employee name"<<endl;
cin>>empname;
cout<<"enter the employee number"<<endl;
cin>>empnumber;
cout<<"enter the employee address"<<endl;
cin>>empaddress;
cout<<"enter the basic salary"<<endl;
cin>>basicsalary;
}
};
class detail:public emp
{
public:
int hra,ta,pf,lic,net,gross,ded;
void calculation()
{
hra=basicsalary*0.03;
ta=basicsalary*0.02;
lic=basicsalary*0.04;
pf=basicsalary*0.03;
```

SRGPGPI

```
ded=lic+pf;
gross=basicsalary+hra+ta;
net=gross-ded;
}
};
class result:public detail
{
public:
void display()
{
cout<<"employee name:"<<empname<<endl;
cout<<"employee number:"<<empnumber<<endl;
cout<<"the basic salary:"<<basicsalary<<endl;
cout<<" the hra amount:"<<hra<<endl;
cout<<"the travelling allowance amount:"<<ta<<endl;
cout<<"lic amount:"<<lic<<endl;
cout<<"the pf amount:"<<pf<<endl;
cout<<"the deductiohn amount:"<<ded<<endl;
cout<<"the gross pay:"<<gross<<endl;
cout<<"the net amount:"<<net<<endl;
}
};
void main()
{
clrscr ();
result res;
res.get();
res.calculation();
res.display();
getch ();
}
```

SRGPGPI

**OUTPUT:**

enter the employee name
hathoon
enter the employee number
13
enter the employee address
thanjavur
enter the basic salary
20000
employee name:hathoon
employee number:13
the basic salary:20000
the hra amount:599
the travelling allowance amount:400
lic amount:800
the pf amount:599
the deduction amount:1399
the gross pay:20999
the net amount:19600

**RESULT:**

Thus, the output is verified using a c++ program

SRGPGPI

## EX.NO:6(a) VOWELS USING POINTER

**AIM:**

To write a c++ program using pointer to find the number of vowels in a given table

**ALGORITHM:**

Step 1: start the program

Step 2: declare the necessary variables and read as a strlen(string lenth)

Step 3: find the lenth of s and k assing to the pointer variable (ie) i=strlen(s)

Step 4: using if certain condition compare vowles (a,e,i,o,u) and the given string

Step 5: print the value of s and k

Step 6: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
int main()
{
int*i,j,k,z;
char s[15];
clrscr();
cout<<"\n enter the string in lower case:";
cin>>s;
i=&z;
*i=strlen(s);
k=0;
for(j=0;j<*i;j++)
{
if(s[j]=='a'||s[j]=='e'||s[j]=='i'||s[j]=='o'||s[j]=='u')
k++;
}
cout<<"\n number of vowels in given string:"<<" "<<"is"<<k<<" ";
getch();
return 0;
}
```

SRGPGPI

**OUTPUT:**

enter the string in lower case:hathoon
number of vowels in given string:3

**RESULT:**

Thus, the result in output is verified using a c++ program

SRGPGPI

## EX.NO: 6(b) PALINDROME USING POINTER

**AIM:**

To write a c++ program using pointer to cheack for palindrome

**ALGORITHM:**

Step 1: start the program

Step 2: declare the variable s,[25],s2[25],c1;

Step 3: read the variable s1

Step 4: copy the string s1tos2 ie string copy (s1,s2)

Step 5: reverse the string s2 (ie) string roll(s2) x=string amp (s1,s2)

Step 6: check the condition (* x==0) if true the string is not palindrome

Step 7: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
int main()
{
char s1[25],s2[25];
int*x,a;
x=&a;
clrscr();
cout<<"\n enter the string:";
cin>>s1;
strcpy(s2,s1);
strrev(s2);
*x=strcmp(s1,s2);
if(*x==0)
cout<<"\n This string is palindrome";
else
cout<<"\n This string is not a palindrome";
getch ();
return 0 ;
}
```

SRGPGPI

**OUTPUT:**

enter the string: madam

This string is palindrome

enter the string: moderm

This string is not a palindrome

**RESULT:**

Thus, the output is verified using a c++ program

SRGPGPI

## EX NO:7 FILE HANDLING

**AIM:**

To write a c++ program to prepare students marklist in a file with student no marks in file subject and total

**ALGORITHM:**

Step 1: start the program

Step 2: declare the variable roll no,name,m1,m2,m3,m4 and total

Step 3: declare the file object s[10] temp

Step 4: open the output file and record type using the loop

Step 5: reading records are write into out files and then close the file

Step 6: read the value for rollno, name, m1,m2,m3,m4,total

Step 7: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
class stud
{
int m0,m1,m2,m3,m4,tot,i;
float avg;
char name [20];
char g;
public:
void get(void);
void cal(void);
void disp();
};
void stud::get(void)
{
cout<<"Enter the student m0,name m1,m2,m3,m4:"<<endl;
cin>>m0>>name>>m1>>m2>>m3>>m4;
}
void stud::cal(void)
{
tot=m1+m2+m3+m4;
avg=tot/4;
if(avg>=85)
{
g='s';
}
else if(avg>=75)
{
g='A';
}
```

SRGPGPI

```
else if(avg>=65)
{
g='B';
}
else if(avg>=55)
{
g='C';
}


else if(avg>=35)
{
g='D';
}
else if(avg>=20)
{
g='F';
}}
void stud::disp()
{
cout<<"name:"<<name<<endl<<"ID
number:"<<m0<<endl<<"MARK1:"<<m1<<endl<<"MARK2:"<<m2<<endl<<"MARK3:"<
<m3<<endl<<"MARK4:"<<m4<<endl;
if((m1>=35)&&(m2>=35)&&(m3>=35)&&(m4>=35))
{
cout<<"PASS"<<endl;
}
else
{
cout<<"FAIL"<<endl;
}
cout<<"TOTAL:"<<tot<<endl<<"average:"<<avg<<endl<<"Grade:"<<g<<endl;
}
void main()
```

SRGPGPI

```
{
stud s[20];
int n;
clrscr();
fstream f;
f.open("stu",ios::in/ios::out);
cout<<"Enter the number of student:"<<endl;
cin>>n;
for(int i=1;i<=n;i++)
{
s[i].get();
s[i].cal();
f.write((char*)&s[i],sizeof(s[i]));
}
f.close();
for(i=1;i<=n;i++)
{
f.read((char*)&s[i],sizeof(s[i]));
cout<<"The detail of the student is:"<<i<<endl;
s[i].disp();
}
getch(); }
```

SRGPGPI

**OUTPUT:**

```
Enter the number of student :
1
Enter the student m0,name,m1,m2,m3,m4:
18
Janani
70
80
90
100
The detail of the student is:1
name:janani
ID number:18
Mark1:70
Mark2:80
Mark3:90
Mark4:100
PASS
TOTAL:340
average:85
```

**RESULT:**

Thus the output is verified using a c++ program

SRGPGPI

## EX.NO:8 EXCEPTION HANDLING

**AIM:**

To write a c++ program to prepare electricity bils for customber generating and handling any two exception

**ALGORITHM:**

Step 1: start the program
Step 2: declare the necessary variables
Step 3: calculate the amount as per the recording
Step 4: if the assumed is zero, exception is raised
Step 5: print the result in array format
Step 6: stop the program

SRGPGPI

**PROGRAM:**

```
#include<iostream.h>
#include<conio.h>
class ebill
{
int cno;
char cname[20];
int units;
double bill;
public:
void get()
{
cout<<"enter customer no, name and no.of units"<<endl;
cout<<"enter customerno:";
cin>>cno;
cout<<"\nenter customer name:";
cin>>cname;
cout<<"\nenter no.of units used:";
cin>>units;
}
void put()
{
cout<<"\ncustomer no is:"<<cno;
cout<<"\ncustomer name is:"<<cname;
cout<<"\nnumber of units consumed:"<<units;
cout<<"\nbill of customer:"<<bill;
}
void calc()
{
if(units==100)
bill=units*1.20;
```

SRGPGPI

```
else if(units==300)
bill=100*1.20+(units-100)*2;
else
bill=100*1.20+200*2+(units-300)*3;
}
};
void main()
{
clrscr();
ebill p1;
try
{
p1.get();
p1.calc();
p1.put();
}
catch(char*excep)
{
cout<<"exception caught";
}
catch(...)
{
cout<<"default exception";
}
getch ();
}
```

SRGPGPI

**OUT PUT:**

Enter customer no, name and no. of units
Enter customer no: 102
Enter customer name: janani
Enter no. of units used: 300
Customer no is: 102
Customer name is: janani
Number of units consumed: 300
Bill of customer: 520

**RESULT:**

Thus, the output is verified using a c++ program.

SRGPGPI