

Object Oriented Programming (CSC161)

Sulav Nepal

Email: nep.sulav@live.com

Contact No.: 9849194892

Master's in Computer Information System (MCIS) – Pokhara University
Bachelor of Science. Computer Science & Information Technology (B.Sc. CSIT) – Tribhuwan University
Microsoft Technology Associate (MTA): Windows Server Administration Fundamentals
Microsoft Certified Technology Specialist (MCTS): Windows Server 2008 R2, Server Virtualization
Microsoft Specialist (MS): Programming in HTML5 with JavaScript and CSS3
Microsoft Students Partner (MSP) 2012 for Nepal

Syllabus

- ***UNIT 1: Introduction to Object Oriented Programming***
 - *Overview of structured programming approach*
 - *Problems with structured programming*
 - *Object oriented programming approach*
 - *Characteristics of object oriented languages*

UNIT 1

***INTRODUCTION TO OBJECT ORIENTED
PROGRAMMING***

Language Paradigms

- *Imperatives – Procedural Programming*
 - C, Pascal, COBOL, FORTRAN, etc
- *Applicative – Functional Programming*
 - LISP, ML
- *Rule-based – Logic Programming*
 - PROLOG
- *Object-Oriented Programming*
 - C++, JAVA, SMALLTALK

Procedural Programming Language

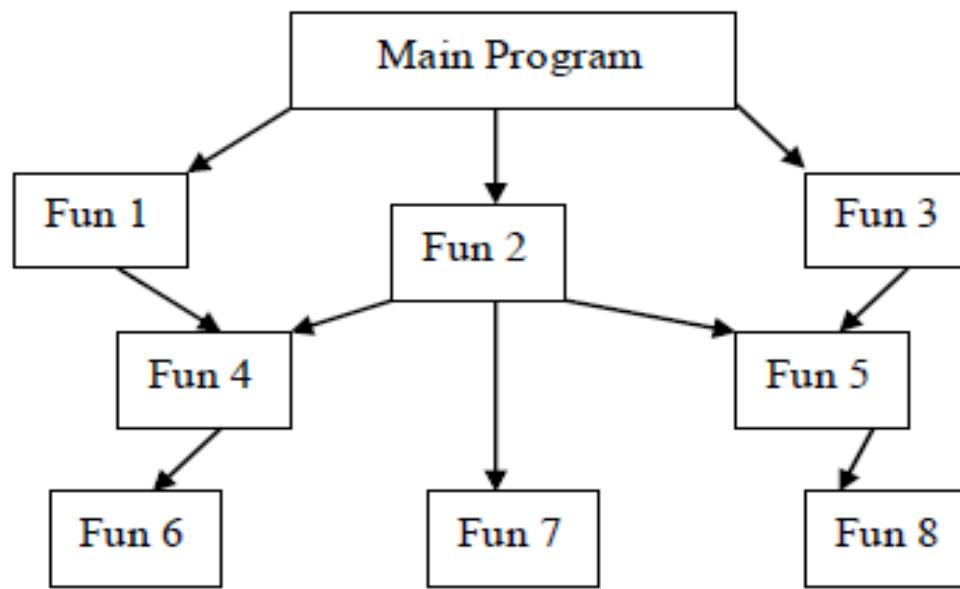
- In procedural programming, programs are organized in the form of subroutines.
- The subroutines do not let code duplication.
- This technique is only suitable for medium sized software applications.
- Conventional programming, using HLL e.g. COBOL, FORTRAN, C etc. is commonly known as procedural programming.

Procedural Programming Language

- A program in Procedural Language is a list of instructions each statement in the language tells the computer to do something involving – reading, calculating, writing output.
- A number of functions are written to accomplish such tasks.
- Program become larger, it is broken into smaller units – functions.
- The primary focus of procedural oriented programming is on functions rather than data.
- Procedure oriented programming basically consists of writing a list of instructions for computer to follow, and organize these instructions into groups known as functions.

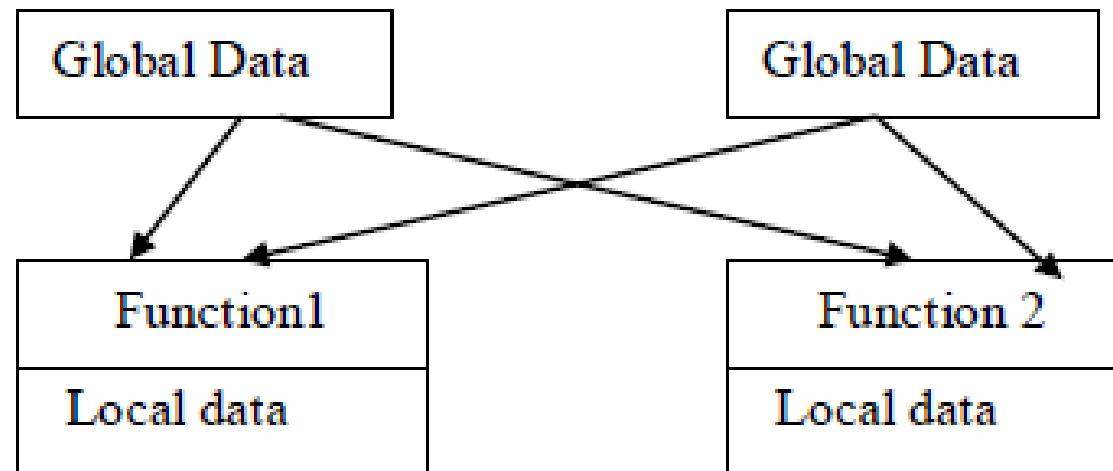
Procedural Programming Language

- In procedural approach,
 - A program is a list of instruction.
 - When program in PL become larger, they are divided into functions(subroutines, sub-programs, procedures).
 - Functions are grouped into modules.



Procedural Programming Language

- In a multi-function program, two types of data are used: local and global.
 - Data items are placed as global so that they can be accessed by all the functions freely.
 - Each function may have their own data also called as local data.



Characteristics of Procedural Approach

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs – function.
- Most of functions share global data.
- Data more openly around system from function to function.
- Function transform data from one form to another.
- Employs top-down approach for program design.

Limitations of Procedural Language

- *In large program, it is difficult to identify which data is used for which function.*
- *Global variable overcome the local variable.*
- *To revise an external data structure, all functions that access the data should also be revised.*
- *Maintaining and enhancing program code is still difficult because of global data.*
- *Focus on functions rather than data.*
- *It does not model real world problem very well. Since functions are action oriented and do not really correspond to the elements of problem.*

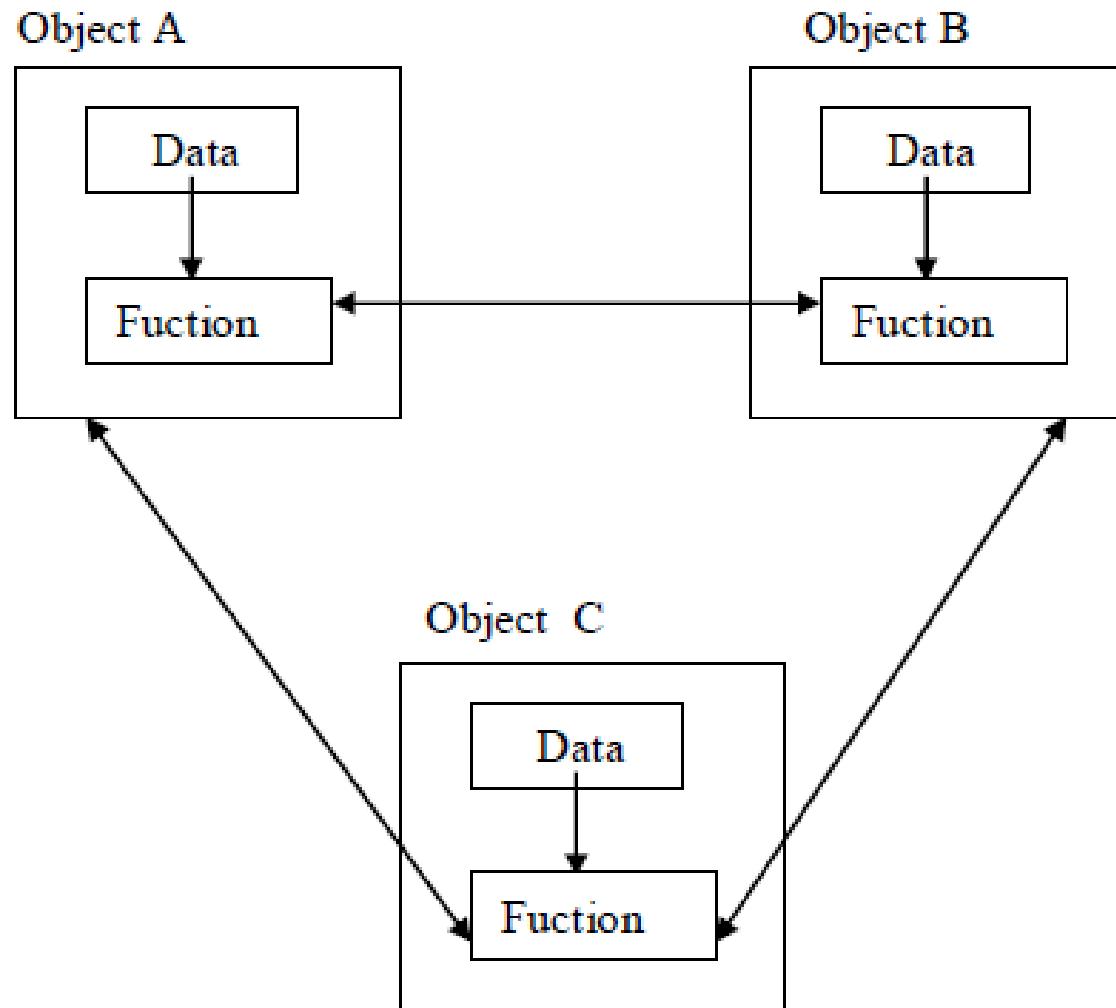
The Object Oriented Approach

- *Object-Oriented programming is a programming methodology that associates data structures with a set of operators which act upon it.*
- *In OOP, an instance of such an entity is known as object.*
- *In other words, OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.*

The Object Oriented Approach

- In Object Oriented programming
 - Emphasis is on data rather than procedures.
 - Programs are divided into objects.
 - Data structures are designed such that they characterize the objects.
 - Functions & data are tied together in the data structures so that data abstraction is introduced in addition to procedural abstraction.
 - Data is hidden & can't be accessed by external functions.
 - Object can communicate with each other through function.
 - New data & functions can be easily added.
 - Follows Bottom up approach.

The Object Oriented Approach



Features of Object Oriented Language

- *Objects*
- *Classes*
- *Encapsulation and Data Abstraction*
- *Inheritance*
- *Polymorphism*
- *Message Passing*

Features of Object Oriented Language

#1 Objects

- *Objects are the entities in an object oriented system through which we perceive the world around us.*
- *We naturally see our environment as being composed of things which have recognizable identities & behavior.*
- *The entities are then represented as objects in the program.*
- *They may represent a person, a place, a bank account, or any item that the program must handle.*
- *For example Automobiles are objects as they have size, weight, color etc. as attributes (i.e. data) and starting, pressing the brake, turning the wheel, pressing accelerator pedal etc. as operation (that is functions).*

Features of Object Oriented Language

#1 Objects

| Object: Student | |
|-----------------|--|
| Data | |
| Name | |
| Date of birth | |
| Marks | |
| ----- | |
| ----- | |
| Functions | |
| Total() | |
| Average() | |
| Display() | |
| ----- | |

| Object: Account | |
|-----------------|--|
| Data | |
| Account number | |
| Account Type | |
| Name | |
| balance | |
| ----- | |
| Functions | |
| deposit() | |
| withdraw() | |
| enquire() | |
| ----- | |

Features of Object Oriented Language

#1 Objects

- Example of objects:
- Physical Objects:
 - Automobiles in traffic flow simulation
 - Countries in Economic model
 - Air craft in traffic – control system .
- Computer user environment objects.
 - Window, menus, icons, etc.
- Data storage constructs.
 - Stacks, Trees, etc.
- Human entities:
 - employees, student, teacher, etc.
- Geometric objects:
 - point, line, Triangle, etc.
- **NOTE:** Objects mainly serve the following purposes:
 - Understanding the real world and a practical base for designers
 - Decomposition of a problem into objects depends on the nature of problem.

Features of Object Oriented Language

#2 Classes

- *A class is a collection of objects of similar type.*
- *For example: manager, peon, secretary, clerk are member of the class employee and class vehicle includes objects car, bus, etc.*
- *It defines a data type, much like a struct in C programming language and built in data type(int char, float etc).*
- *It specifies what data and functions will be included in objects of that class.*
- *Defining class doesn't create an object but class is the description of object's attributes and behaviors.*

Features of Object Oriented Language

#2 Classes

- *Person Class:* Attributes: Name, Age, Sex, etc.
 - Behaviors: Speak(), Listen(), Walk()
- *Vehicle Class:* Attributes: Name, model, color, height, etc,
 - Behaviors: Start(), Stop(), Accelerate(), etc.
- *When class is defined, objects are created as*
<classname> <objectname> ;
- *If employee has been defined as a class, then the statement*
employee manager;
will create an object manager belonging to the class employee.

Features of Object Oriented Language

#2 Classes

- *Each class describes a possibly infinite set of individual objects, each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operations with other instances of the class.*
- *The following points gives the idea of class:*
 - *A class is a template that unites data and operations.*
 - *A class is an abstraction of the real world entities with similar properties.*
 - *Ideally, the class is an implementation of abstract data type.*

Features of Object Oriented Language

#3 Encapsulation and Data Abstraction

- *The wrapping up of data and function into a single unit is called encapsulation.*
- *Encapsulation is most striking feature of a class.*
- *The data is not accessible from outside of class.*
- *Only member function can access data on that class.*
- *The insulation of data from direct access by the program is called data hiding.*
- *That is data can be hidden making them private so that it is safe from accidental alteration.*

Features of Object Oriented Language

#3 Encapsulation and Data Abstraction

- *Abstraction is representing essential features of an object without including the background details or explanation.*
- *It focuses the outside view of an object, separating its essential behavior from its implementation.*
- *The class is a construct in C++ for creating user-defined data types called Abstract Data Types (ADT)*

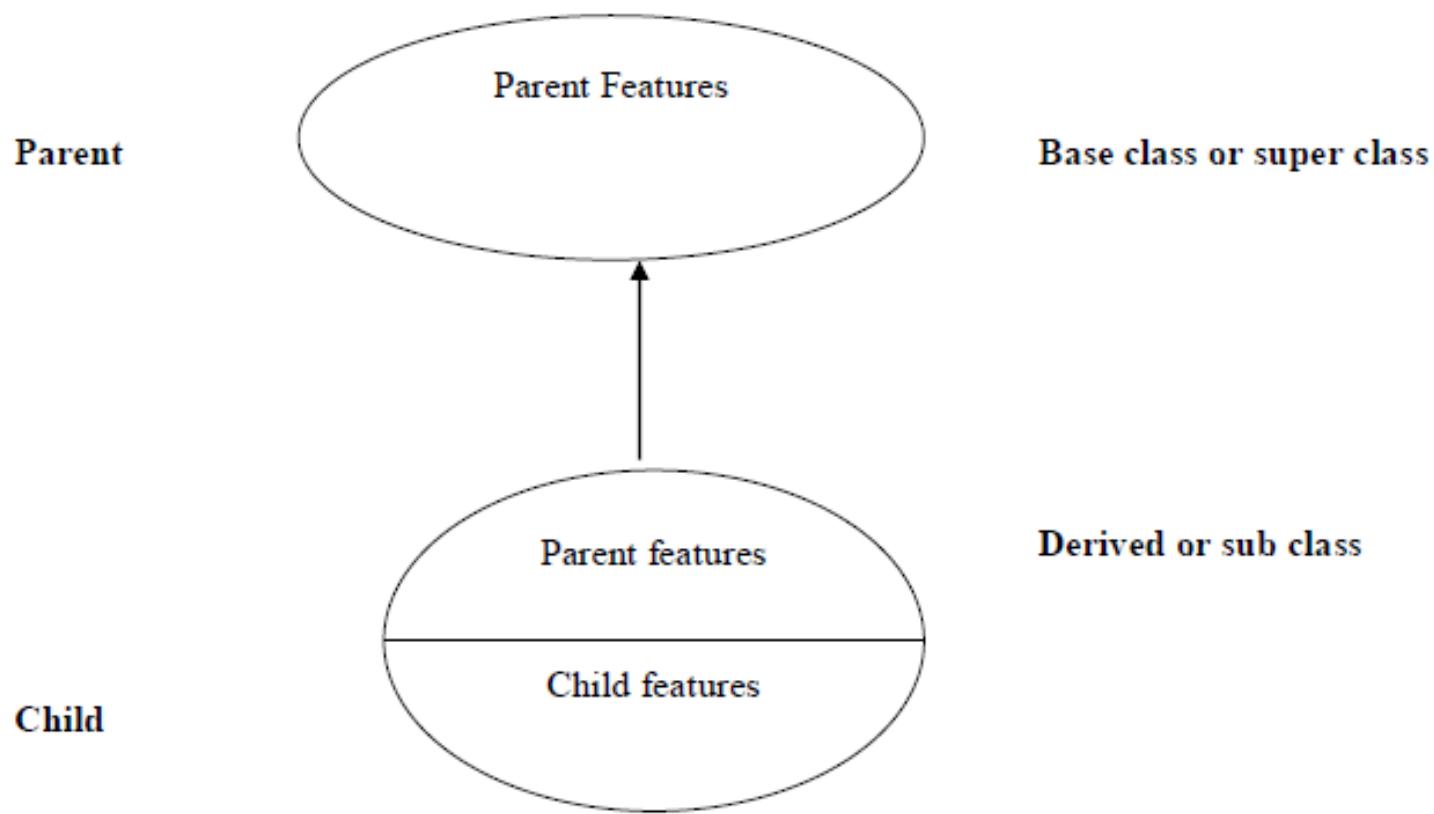
Features of Object Oriented Language

#4 Inheritance

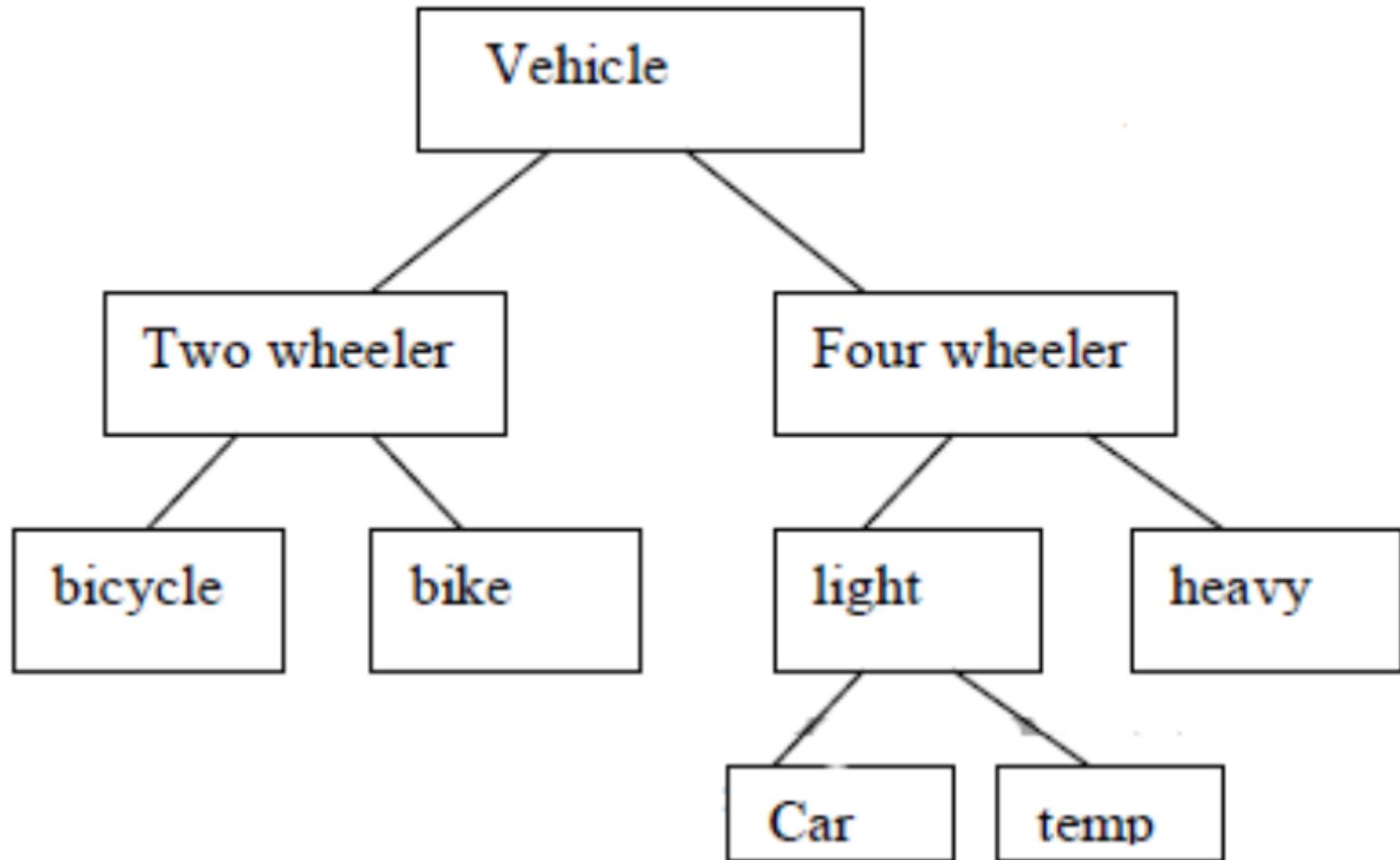
- *Inheritance is the process by which objects of one class acquire the characteristics of object of another class.*
- *In OOP, the concept of inheritance provides the idea of reusability.*
- *We can use additional features to an existing class without modifying it.*
- *This is possible by deriving a new class (derived class) from the existing one (base class).*
- *This process of deriving a new class from the existing base class is called inheritance .*
- *It supports the concept of hierarchical classification.*
- *It allows the extension and reuse of existing code without having to rewrite the code.*

Features of Object Oriented Language

#4 Inheritance

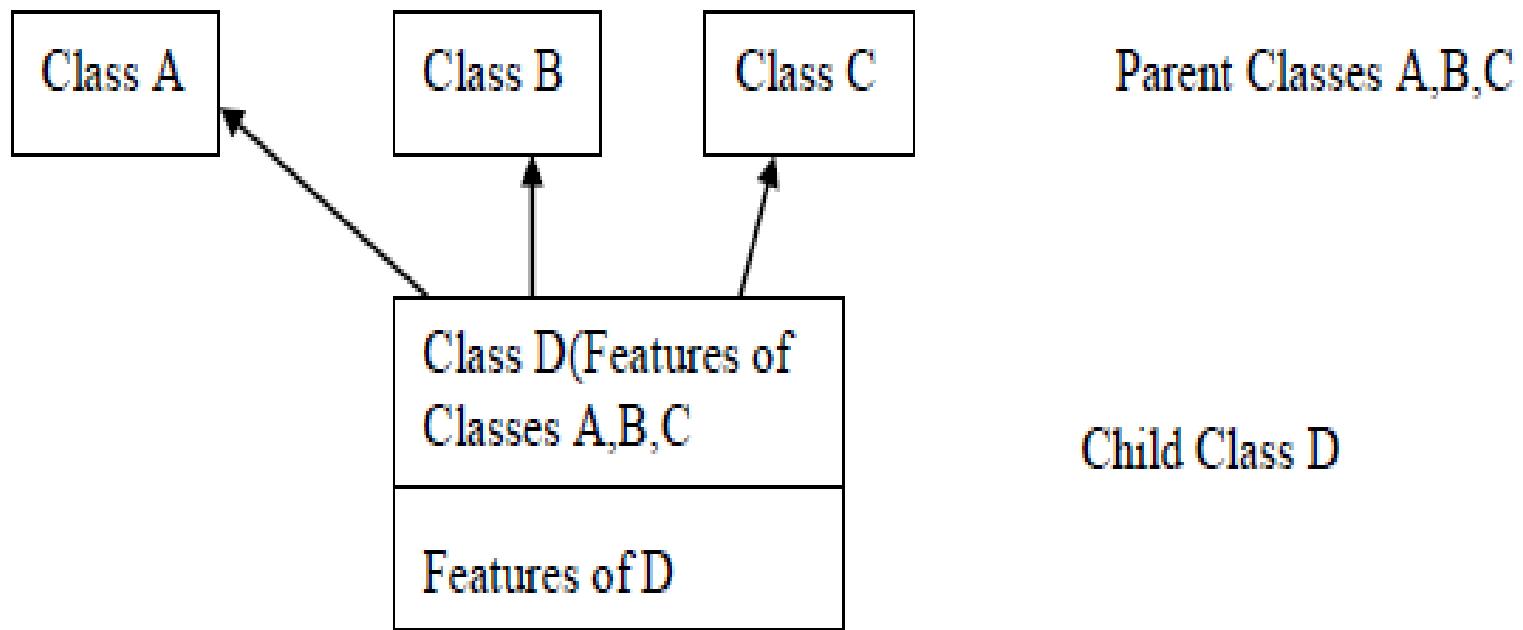


Features of Object Oriented Language



Features of Object Oriented Language

#4 Inheritance



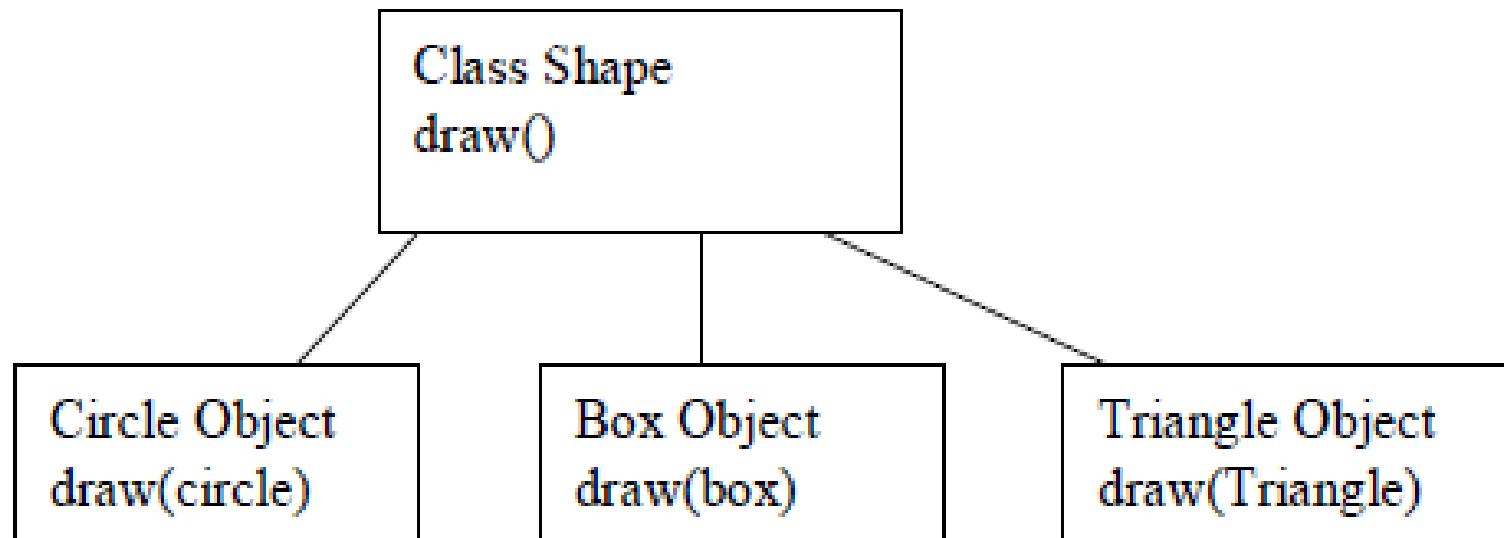
Features of Object Oriented Language

#5 Polymorphism

- *Polymorphism means “having many forms”.*
- *The polymorphism allows different objects to respond to the same message in different ways, the response specific to the type of object.*
- *Polymorphism is important when object oriented programs dynamically creating and destroying the objects in runtime.*
- *Example of polymorphism in OOP is operator overloading, function overloading.*
- *For example operator symbol ‘+’ is used for arithmetic operation between two numbers, however by overloading (means given additional job) it can be used over Complex Object like currency that has Rs and Paisa as its attributes, complex number that has real part and imaginary part as attributes.*
- *By overloading same operator ‘+’ can be used for different purpose like concatenation of strings.*

Features of Object Oriented Language

#5 Polymorphism



Features of Object Oriented Language

#5 Polymorphism

- *Dynamic Binding:*
 - *Binding refers to the linking a function call to the code to be executed in response to the call.*
 - *Dynamic binding means that the code associated with a given function call is not known until the time of the call at run time.*
 - *It is associated with polymorphism & inheritance.*

Features of Object Oriented Language

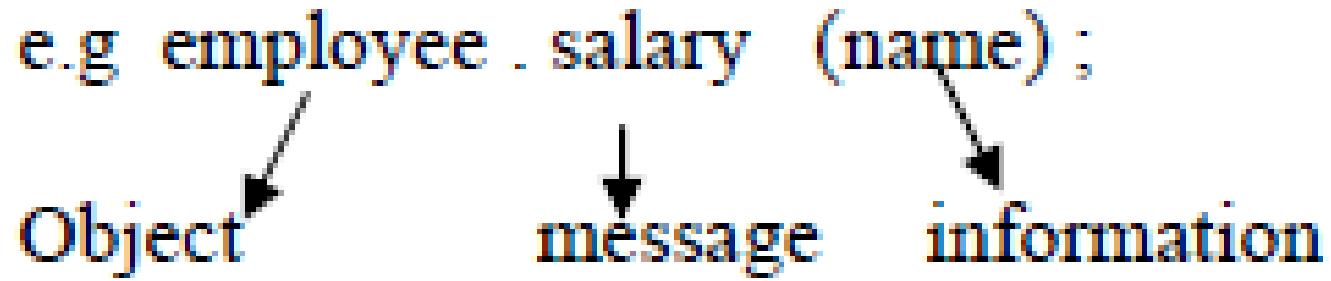
#6 Message Passing

- An Object-Oriented program consists of set of objects that communicate with each other.
- Object communicates with each other by sending and receiving message (information).
- A message for an object is a request for execution of a procedure and therefore will invoke a function or procedure in receiving object that generate the desired result.
- Message passing involves specifying the name of the object name of the function (message) and the information to be sent .

Features of Object Oriented Language

#6 Message Passing

e.g **employee . salary (name) ;**



Object **message** **information**

Advantages of OOP

- *Object oriented programming contributes greater programmer productivity, better quality of software and lesser maintenance cost.*
- *The main advantages are:*
 - *Making the use of inheritance, redundant code is eliminated and the existing class is extended.*
 - *Through data hiding, programmer can build secure programs that cannot be invaded by code in other parts of the program.*
 - *It is possible to have multiple instances of an object to co-exist without any interference.*
 - *System can be easily upgraded from small to large systems.* 8/27/2024 8:37 PM

Advantages of OOP

- *Object oriented programming contributes greater programmer productivity, better quality of software and lesser maintenance cost.*
- *The main advantages are:*
 - *Software complexity can be easily managed.*
 - *Message passing technique for communication between objects makes the interface description with external system much simpler.*
 - *Aids trapping in an existing pattern of human thought into programming.*
 - *Code reusability is much easier than conventional programming languages.*

Disadvantages of OOP

- *Compiler and runtime overhead. Object oriented program required greater processing overhead – demands more resources.*
- *An object's natural environment is in RAM as a dynamic entity but traditional data storage in files or databases*
- *Re-orientation of software developer to object-oriented thinking.*
- *Requires the mastery in software engineering and programming methodology.*
- *Benefits only in long run while managing large software projects.*
- *The message passing between many objects in a complex application can be difficult to trace & debug.*

Application of OOP

- *The most popular application of OOP, up to now, has been area of user interface design such as Windows.*
- *Real Business systems are often much more complex attributes & behaviors .*
- *OOP can simplify such complex problem.*
- *The areas of application of OOP include*
 - *Real time systems*
 - *Simulation & modeling*
 - *Object-Oriented databases*
 - *Hypertext, hypermedia*
 - *AI & expert system*
 - *Neural Networks & parallel programming*
 - *Decision support & Office automation system*
 - *CAM/CAD systems.*
 - *Computer Based training and Educational Systems*

Object Oriented Programming Languages

- *SmallTalk*
- *Eiffel*
- *Java*
- *C++*

Object Oriented Programming Languages

#1 SmallTalk

- Developed by Alan Kay at Xerox Palo Alto Research center (PARC) in 1970's
- 100% Object Oriented Language
- The syntax is very unusual and this leads to learning difficulties for programmers who are used to conventional language syntax.
- Language of this type allocate memory space for object on the heap and dynamic garbage collector required .

Object Oriented Programming Languages

#2 Eiffel

- *Eiffel was designed by a Frenchman named Bertrand Meyer in the late 1980's.*
- *Syntax is very elegant & simple, fewer reserved word than Pascal .*
- *Compiler normally generates “C” source which is then compiled using c compiler which can lead long compile time.*
- *All Eiffel object are created on the heap storage*
- *Pure object oriented*
- *Not very popular as a language for mainstream application development.*

Object Oriented Programming Languages

#3 Java

- Designed by SUN (Stanford University Net) Microsystems, released in 1996 and is a pure Object Oriented language.
- Syntax is taken from C++ but many differences.
- All object are represented by references and there is no pointer type.
- The compiler generates platform independent byte code which is executed at run time by interpreter.
- A very large library of classes for creating event driven GUI is included with JAVA compiler
- JAVA is a logical successor to C++ can also be called as C++-+-(C-Plus-Plus-Minus-Plus-Plus i.e. remove some difficult to use features of C++ and Add some good features)

Object Oriented Programming Languages

#4 C++

- Developed by Bjarne Stroustrup at Bell Lab in New Jersey in early 1980's as extension of C.
- Employs the basic syntax of the earlier C language which was developed in Bell Lab by Kernigan & Ritchie.
- One of most popular languages used for software development.
- By default, C++ creates objects on the system's stack in the same way as the fundamental data type.
- Unlike Java, SmallTalk, and Eiffel; C++ is not a pure Object Oriented language. i.e. it can be written in a conventional 'C'.

END OF UNIT ONE

Syllabus

- **UNIT 2: Basics of C++ Programming**
 - *C++ Program Structure*
 - *Character Set and Tokens*
 - *Data Type*
 - *Type Conversion*
 - *Preprocessor Directives*
 - *Namespace*
 - *Input / Output Streams and Manipulators*
 - *Dynamic Memory Allocation with new and delete*
 - *Control Statements*
 - *Functions: Function Overloading, Inline Functions, Default Argument, Pass by Reference, Return by Reference, Scope and Storage Class.*
 - *Pointers: Pointer Variables Declaration & Initialization, Operators in Pointers, Pointers and Arrays, Pointer and Function.*

UNIT 2

BASICS OF C++ PROGRAMMING

History of C++

- C++ is an object oriented programming language.
- It was called “C with class”.
- C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early eighties.
- Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both languages and create a more powerful language that could support object oriented programming features and still remain power of C.
- The result was C++.

History of C++

- *The major addition was class to original C language, so it was also called “C with classes”.*
- *Then in 1993, the name was changed to C++.*
- *The increment operator ++ to C suggest an enhancement in C language.*
- *C++ can be regarded as superset of C (or C is subset of C++).*
- *Almost all C programs are also true in C++ programs.*

C++ Program Structure

*/*Sample program that reads value from user and returns area and perimeter of the rectangle*/*

```
#include <iostream>                                // for cout and cin
using namespace std;
int main()
{
    int length, width;
    int perimeter, area;                            // declarations
    cout << "Length = ";                          // prompt user
    cin >> length;                               // enter length
    cout << "Width = ";                          // prompt user
    cin >> width;                                // input width
    perimeter = 2*(length + width);               // compute perimeter
    area = length*width;                          // compute area
    cout << endl
    << "Perimeter is " << perimeter;
    cout << endl
    << "Area is " << area << endl;            // output results
    return 0;                                    // end of main program
}
```

C++ Program Structure

- **Single Line Comment (//**
 - Any text from the symbols // until the end of the line is ignored by the compiler.
 - This facility allows the programmer to insert Comments in the program.
 - Any program that is not very simple should also have further comments indicating the major steps carried out and explaining any particularly complex piece of programming.
 - This is essential if the program has to be extended or corrected at a later date.
 - This is a kind of documentation.
 - Also C comment type /*-----*/ is also a valid comment type in C++.

C++ Program Structure

- *The line `#include<iostream>` must start in column one.*
- *It causes the compiler to include the text of the named file (in this case `iostream`) in the program at this point.*
- *The file `iostream` is a system supplied file which has definitions in it which are required if the program is going to use stream input or output.*
- *All programs will include this file.*
- *This statement is a preprocessor directive i.e. it gives information to the compiler but does not cause any executable code to be produced.*

C++ Program Structure

- The actual program consists of the function `main()` which commences at the line `int main()`.
- All programs must have a function `main()`.
- Note that the opening brace `{`) marks the beginning of the body of the function, while the closing brace `}`) indicates the end of the body of the function.
- The word `int` indicates that `main()` returns an integer value.
- Running the program consists of obeying the statements in the body of the function `main()`.

C++ Program Structure

- *The body of the function main contains the actual code which is executed by the computer and is enclosed, as noted above, in braces {}.*
- *Every statement which instructs the computer to do something is terminated by a semi-colon.*
- *Symbols such as `main()`, {} etc. are not instructions to do something and hence are not followed by a semi-colon.*
- *Preprocessor directives are instruction to the compiler itself but program statements are instruction to the computer.*

C++ Program Structure

- Sequences of characters enclosed in double quotes are literal strings.
- Thus instructions such as `cout << "Length = "` send the quoted characters to the output stream `cout`.
- The special identifier `endl` when sent to an output stream will cause a newline to be taken on output.

C++ Program Structure

- All variables that are used in a program must be declared and given a type.
- In this case all the variables are of type **int**, i.e. whole numbers.
- Thus the statement **int length, width;** declares to the compiler that integer variables length and width are going to be used by the program.
- The compiler reserves space in memory for these variables.

C++ Program Structure

- *Values can be given to variables by the assignment statement, e.g. the statement `area = length * width;` evaluates the expression on the right-hand side of the equals sign using the current values of length and width and assigns the resulting value to the variable area.*
- *Layout of the program is quite arbitrary, i.e. new lines, spaces etc. can be inserted wherever desired and will be ignored by the compiler.*
- *The prime aim of additional spaces, new lines, etc. is to make the program more readable.*
- *However superfluous spaces or new lines must not be inserted in words like main, cout, in variable names or in strings.*

Character Set of C++

- Character set is a set of valid characters that a language can recognize.
- A character represents any letter, digits, or any other sign.
- C++ has the following character set:
 - Letters
 - Digits
 - Special Characters / Symbols
 - White Spaces

Character Set of C++

- *Letters:*
 - Letters refer to the 26 alphabets that is used in the English language.
 - Both uppercase (**A-Z**) and lowercase (**a-z**) alphabets can be used in C++.
- *Digits:*
 - Digits from **0-9** or any combination of these can be used in any C++ code.

Character Set of C++

- *Special Characters / Symbols:*
 - Apart from the usual letters and digits, we have a set of special characters that can be used in C++.
 - All these characters are used for various purposes.
 - These are + - * / ^ \ () { } [] = != < > <= >= . ‘ “ \$, ; : % ! & ? _ # @ Space
- *White Spaces:*
 - These are a special set of characters that are mainly used for the purpose of formatting the text in the C++ program.
 - They are also called formatting characters.
 - Some common examples are blank space, horizontal tab, new line, carriage return, vertical tab, form feed, etc.

Tokens

- *The smallest individual units in a program are known as tokens.*
- *C++ has the following tokens:*
 - *Keywords*
 - *Identifiers*
 - *Constants*
 - *Operators*

Tokens

Keywords

- *Keywords are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.*

| | | | | | | | |
|--------------|---------------|------------------|-----------------|-----------------|---------------|---------------|-----------------|
| <i>asm</i> | <i>double</i> | <i>new</i> | <i>switch</i> | <i>class</i> | <i>friend</i> | <i>return</i> | <i>union</i> |
| <i>auto</i> | <i>else</i> | <i>operator</i> | <i>template</i> | <i>const</i> | <i>goto</i> | <i>short</i> | <i>unsigned</i> |
| <i>break</i> | <i>enum</i> | <i>private</i> | <i>this</i> | <i>continue</i> | <i>if</i> | <i>signed</i> | <i>virtual</i> |
| <i>case</i> | <i>extern</i> | <i>protected</i> | <i>throw</i> | <i>default</i> | <i>inline</i> | <i>sizeof</i> | <i>void</i> |
| <i>catch</i> | <i>float</i> | <i>public</i> | <i>try</i> | <i>delete</i> | <i>int</i> | <i>static</i> | <i>volatile</i> |
| <i>char</i> | <i>for</i> | <i>register</i> | <i>typedef</i> | <i>do</i> | <i>long</i> | <i>struct</i> | <i>while</i> |

Tokens

Identifiers

- *Identifiers refers to the names of variables, functions, arrays, classes, etc. created by the programmer.*
- *They are the fundamental requirement of any language.*
- *Each language has its own rules for naming those identifiers.*
- *The following rules are common to both C and C++*
 - *Only alphabetic characters, digits and underscores are permitted.*
 - *The name cannot start with a digit.*
 - *Case sensitive*
 - *A reserved keyword cannot be used as a variable name.*
- *A major difference between C and C++ is the limit on the length of a name.*
- *While ANSI C recognizes only the first 32 characters in a name, C++ places no limit on its length and therefore all the characters in a name are significant.*

Tokens

Constants

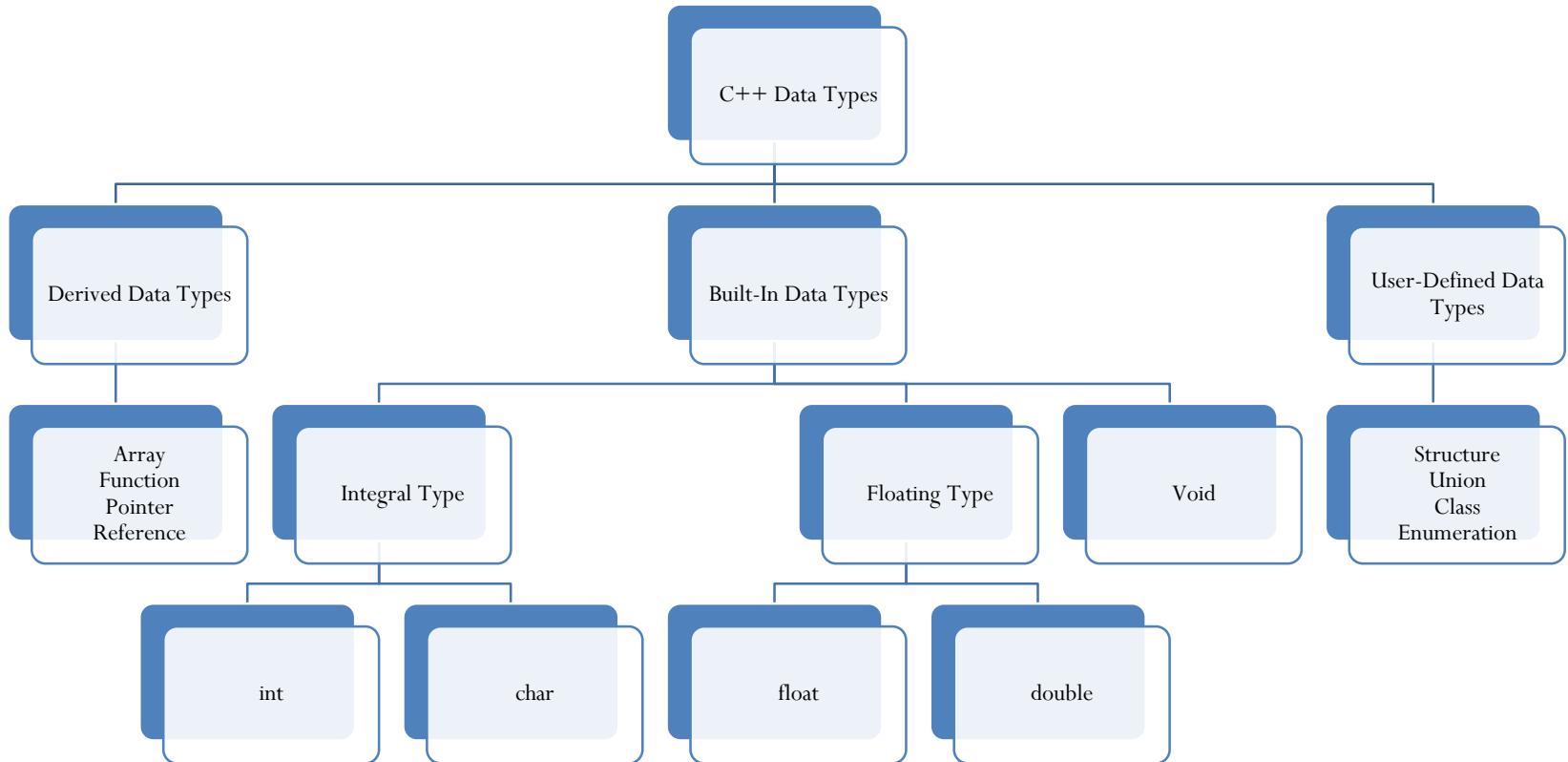
- *A constant is any expression that has a fixed value.*
- *They can be divided in integer numbers, floating-point numbers, characters, and strings.*
- *In addition to decimal numbers, C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16).*
- *If we want to express an octal number we must precede it with a 0 character.*
- *And to express a hexadecimal number we have to precede it with the characters 0x.*
- *For example, the following literal constants are all equivalent to each other:*
 - 75 //decimal
 - 0113 //octal
 - 0x4b //hexadecimal

Tokens

Operators

- *Operators are symbols or combination of symbols that directs the computer to perform some operation upon operands.*
- *Operators can be classified as:*
 - *Arithmetic Operators*
 - *Assignment Operators*
 - *Unary Operators*
 - *Comparison Operators*
 - *Shift Operators*
 - *Bit-wise Operators*
 - *Logical Operators*
 - *Conditional Operators*

C++ Data Types



Type Conversion

- *Converting an expression of a given type into another type is known as type-casting.*
- *There are two types of type conversion:*
 - *Automatic Conversion (Implicit Type Conversion)*
 - *Type Casting (Explicit Type Conversion)*

Type Conversion

Automatic Type Conversion (Implicit Type Conversion)

- When two operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically.
- This is called automatic type conversion or type promotion or implicit type conversion.

- For example:

int x=3;

float y=2.5,z;

z=x+y; // 5.5

Type Conversion

Type Casting (Explicit Type Conversion)

- *Type casting is the forceful conversion from one type to another type.*
- *Sometimes, a programmer needs to convert a value from one type to another forcefully in a situation where the compiler will not do it automatically.*
- *For this, C++ permits explicit type conversion of variables or expressions as follows:*

int a=10000;

*int b=long(a)*5/2; // correct*

*int b=a*5/2; // incorrect*

Preprocessor Directives

- Preprocessor directives are lines included in the code of programs preceded by a hash sign (#).
- These lines are not program statements but directives for the preprocessor.
- The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.
- These preprocessor directives extend only across a single line of code.
- As soon as a newline character is found, the preprocessor directive ends.

Preprocessor Directives

- *The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).*
- *The `#define` Directive – creates symbolic constants.*
- *The `#include` Directive – replaces it by the entire content of the specified header or files.*

Namespaces

- Namespaces allow to group entities under a name.
- This way the global scope can be divided in “sub-scopes”, each one with its own name.
- The format of namespaces is:
namespace identifier
{
 entities
}
- Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace.

Namespaces

- For example:

```
namespace myNamespace
{
    int a,b;
}
```

- In this case, the variables *a* and *b* are normal variables declared within a namespace called *myNamespace*.
- In order to access these variables from outside the *myNamespace* namespace we have to use the scope resolution operator (`::`).

Namespaces

- For example, to access the previous variables from outside `myNamespace` we can write:

`myNamespace::a`

`myNamespace::b`

- The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors.

```
#include <iostream>  
  
using namespace std;  
  
namespace A  
{  
  
    int x = 5;  
  
    void printX()  
    {  
  
        cout<<x<<endl;  
        // function statements goes here  
    }  
  
}
```

```
namespace B  
{  
  
    int x=10;  
  
    void printX()  
    {  
  
        cout<<x<<endl;  
        // function statements goes here  
    }  
  
}  
  
int main()  
{  
  
    A::printX() ;  
    B::printX();  
    return 0;  
}
```

Scope Resolution Operator (::)

- C++ supports a mechanism to access a global variable from a function in which a local variable is defined with the same name as a global variable.
- It is achieved using the scope resolution operator.
`:: GlobalVariableName`
- The global variable to be accessed must be preceded by the scope resolution operator.
- It directs the compiler to access a global variable, instead of one defined as a local variable.
- The scope resolution operator permits a program to reference an identifier in the global scope that has been hidden by another identifier with the same name in the local scope.

Scope Resolution Operator (::)

```
#include<iostream>
using namespace std;
int x=5;
int main()
{
    int x=15;
    cout<<"Local data x="<<x<<" & Global data x="<<::x<<endl;
    {
        int x=25;
        cout<<"Local data x="<<x<<" & Global data x="<<::x<<endl;
    }
    cout<<"Local data x="<<x<<" & Global data x="<<::x<<endl;
    cout<<"Global + Local = "<<::x+x;
    return 0;
}
```

Reference Variable

- C++ introduces a new kind of variable known as reference variable.
- A reference variable provides an alias (Alternative name) of the variable that is previously defined.
- For example, if we make the variable sum a reference to the variable total, the sum and total can be used interchangeably to represent that variable.
- Syntax for defining reference variable

Data_type &reference_name = variable_name

Reference Variable

- Example:

```
int total=100 ;  
int &sum=total;
```

- Here **total** is **int** variable already declared.
- **Sum** is the alias for variable **total**.
- Both the variable refer to the same data **100** in the memory.
- **cout<<total;** and **cout<<sum;** gives the same output **100**.
- And **total= total+100;** **cout<<sum;** gives output **200**.

Reference Variable

- *A reference variable must be initialized at the time of declaration.*
- *This establishes the correspondence between the reference and the data object which it means.*
- *The initialization of reference variable is completely different from assignment to it.*
- *A major application of the reference variables is in passing arguments to function.*

Reference Variable

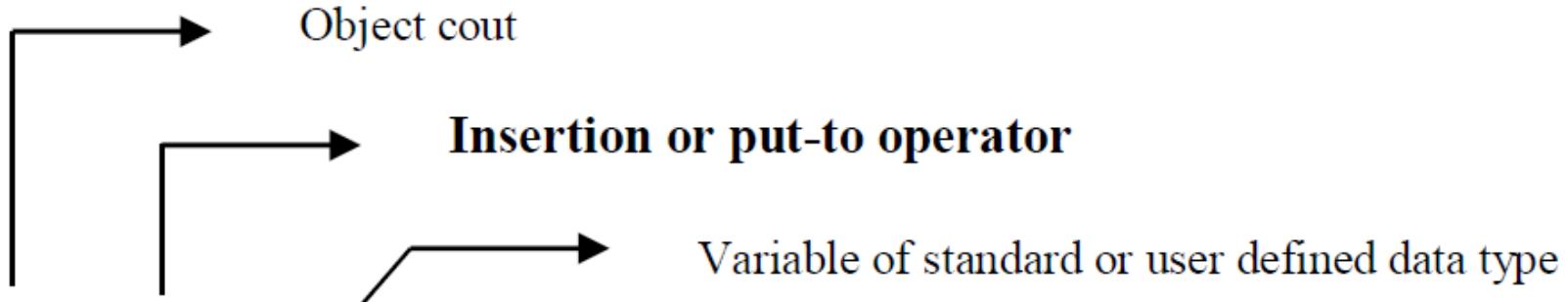
```
//An example of reference
#include<iostream>
using namespace std;
int main()
{
    int x=5;
    int &y=x;
    //y is alias of x
    cout<<"x="<<x<<" and y="<<y<<endl;
    y++;
    //y is reference of x;
    cout<<"x="<<x<<" and y="<<y<<endl;
    x++;
    cout<<"x="<<x<<" and y="<<y<<endl;
    return 0;
}
```

Input & Output

- C++ Supports rich set of functions for performing input and output operations.
- The syntax using these I/O functions is totally consistent of the device with I/O operations are performed.
- C++'s new features for handling I/O operations are called streams.
- Streams are abstractions that refer to data flow.
- Streams in C++ are :
 - Output Stream
 - Input Stream

```
cout << variable;
```

Output Stream



Output Stream

- *The output stream allows us to write operations on output devices such as screen, disk etc.*
- *Output on the standard stream is performed using the **`cout`** object.*
- *C++ uses the bit-wise-left-shift operator for performing console output operation.*
- *The syntax for the standard output stream operation is as follows:
`cout<<variable;`*
- *The word **`cout`** is followed by the symbol **`<<`**, called the insertion or put to operator, and then with the items (variables, constants, expressions) that are to be output.*
- *Variables can be of any basic data types.*

Output Stream

- The following are example of stream output operations:

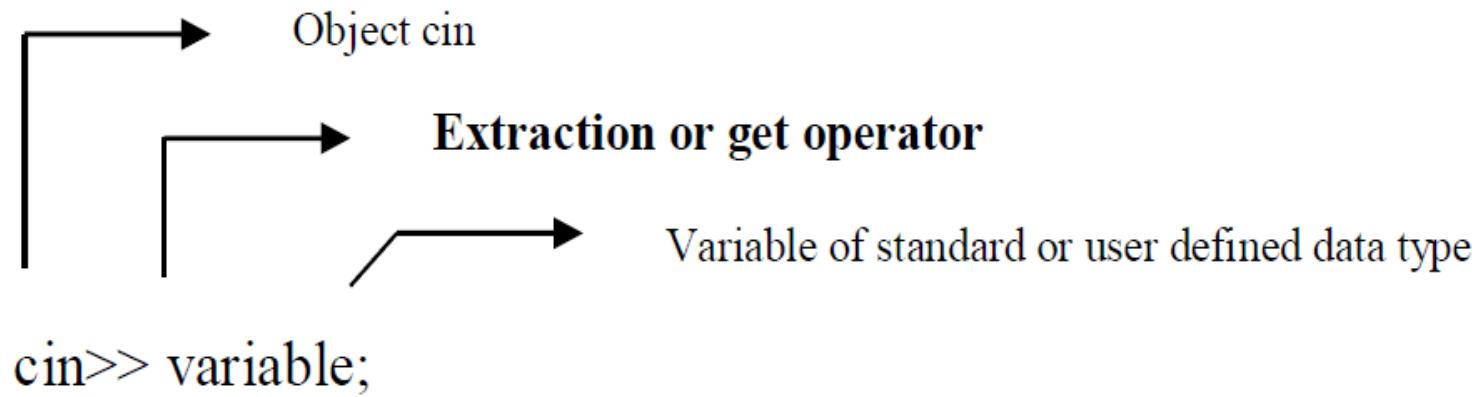
```
cout<<"Hello World!";
```

```
int age;  
cout<<age;
```

```
float weight;  
cout<<weight;
```

- More than one item can be displayed using a single cout output stream object.
- Such output operations in C++ are called cascaded output operations.
- For example: `cout<<"Age is "<<age<<" years";`
- This cout object will display all the items from left to right. If value of age is 30 then this stream prints – *Age is 30 years*

Input Stream



Input Stream

- The input stream allows us to perform read operations with input devices such as keyboard, disk, etc.
- Input from the standard stream is performed using the **cin** object.
- C++ uses the bit-wise right shift operator for performing console input operation.
- The syntax for the standard input stream operation is as follows:
cin>>variable;
- The word **cin** is followed by the symbol **>>**, and then with the items (variables, constants, expressions) into which input data is to be stored.

Input Stream

- The following are example of stream input operations:

```
int amount;  
cin>>amount;
```

```
float weight;  
cin>>weight;
```

```
char name[20];  
cin>>name;
```

- Input of more than one item can also be performed using the `cin` input stream object.
- Such input operation in C++ are called cascaded input operations.
- For example, reading the name of a person his address, age can be performed by the `cin` as: `cin>> name>>address>>age;`
- The `cin` object reads the items from left to right.
- The complete syntax of the standard input streams operations is as follows:
`cin>>var1>>var2>>var3>>.....>>varN;`
- e.g. `cin>>i>>j>>k>>l;`

Manipulators

- Manipulators are instructions to the output stream that modify the output in various ways. For example `endl` , `setw` etc.
- The **`endl`** Manipulator:
 - The `endl` manipulator causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line.
 - It has same effect as sending the '`\n`' character but somewhat different.
 - It is a manipulator that sends a newline to the stream and flushes the stream (puts out all pending characters that have been stored in the internal stream buffer but not yet output).
 - Unlike '`\n`' it also causes the output buffer to be flushed but this happens invisibly.
 - For example:

```
cout << endl << "Perimeter is " << perimeter;  
cout << endl << "Area is " << area << endl;
```

Manipulators

- *The **setw** Manipulator:*
 - *To use setw manipulator the “iomanip.h” header file must be included.*
 - *The setw manipulator causes the number or string that follows it in the stream to be printed within a field n characters wide, where n is the argument used with setw as setw(n).*
 - *The value is right justified within the field.*
- *The **setprecision** Manipulator:*
 - *This manipulator sets the **n** digits of precision to the right of the decimal point to the floating point output, where **n** is the argument to setprecision(n).*

Manipulators

```
//demonstrates setw manipulator
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
    long pop1=5425678,pop2=47000,pop3=76890;
    cout<<"LOCATION"<<setw(12)<< "POPULATION"<<endl
        <<"Patan"<<setw(15)<<pop1<<endl
        <<"Khotang" <<setw(13)<<pop2<<endl
        <<"Butwal" <<setw(14) <<pop3<<endl;
    return 0;
}
```

Manipulators

```
// demonstrates setprecision manipulator
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float num = 2.71828;
    cout << "Original number is: " << num << endl;
    cout << "The number with 3 significant figures is: ";
    cout << setprecision(3) << num;
    return 0;
}
```

Manipulators

- Manipulators come in two flavors: those that take arguments and those that don't take arguments.

| Manipulators | Purpose |
|--------------|---|
| ws | Turn on whitespace skipping on input |
| dec | Convert to decimal |
| oct | Convert to octal |
| hex | Convert to hexadecimal |
| endl | Insert newline and flush the output stream |
| ends | Insert null character to terminate an output string |
| flush | Flush the output stream |
| lock | Lock the file handle |
| unlock | Unlock the file handle |

Manipulators

```
//demonstrates manipulators
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
    int pop1=54321;
    cout<<"Numbers"<<setw(20)<< "Results"<<endl
        <<"Hexadecimal"<<setw(16)<<hex<<pop1<<endl
        <<"Octal" <<setw(22)<<oct<<pop1<<endl
        <<"Decimal" <<setw(20) <<dec<<pop1<<endl;
    return 0;
}
```

Memory Management with New & Delete

- Along with `malloc()`, `calloc()`, and `free()` functions, C++ also defines two unary operators `new` and `delete` that perform the task of allocating and freeing the memory.
- Since these operators manipulate memory on the free store, they are also known as free store operators.
- A data object created inside a block with `new` will remain in existence until it is explicitly destroyed by using `delete`.
- It is a good programming practice to match every occurrence of `new` operator with corresponding `delete` operator otherwise if program runs for long time and `new` operator is used repeatedly without matched `delete` operator all memory space of heap will be exhausted and hence may cause the problem of system crash.

Memory Management with New & Delete

- The general form of the new operator is:

pointer_variable = new data_type;

*For example: float *q; q = new float;*

*cin>>*q*

- We can also initialize the memory using the new operator. This is done as follows:

pointer_variable = new data_type(value);

For example:

*int *p = new int(25);*

- When an object is no longer needed, it is destroyed to release the memory space for reuse. The general form is:

delete pointer_variable;

For example:

delete p;

- If we want to free a dynamically allocated array we must use the following form of delete:

delete [size] pointer_variable;

For example:

delete [] p;

Memory Management with New & Delete

- The `new` operator has several advantages over the function `malloc()`:
 - It automatically computes the size of the data objects.
 - It automatically returns the correct pointer type so there is no need to use a type cast.
 - It is possible to initialize the object while creating the memory space.
 - Both `new` and `delete` operators can be overloaded.

Control Structure

- Like C, C++ also supports the following three control structures:
 - Sequence structure (straight line)
 - Selection structure (branching or decision)
 - Loop structure (iteration or repetition)

Sequence Structure

- In this structure, the sequence statements are executed one after another from top to bottom.
statement 1;
statement 2;
statement 3;
.....
statement n;
- In this case statement 1 is executed before statement 2, statement 2 is executed before statement 3, and so on.

Selection Structure

- *This structure makes one-time decision, causing a one-time jump to a different part of the program, depending on the value of an expression.*
- *The two structures of this type are:*
 - ***if***
 - ***switch*.**

Selection Structure – if statement

- There are three forms of if statement.
 - Simple if:

- The syntax is:

```
if(expression)
{
    statement(s);
}
```

- This statement lets your program to execute statement(s) if the value of the expression is true (non-zero). If there is only one statement, it is not necessary to use braces.

- For example:

```
int x = 5, y = 10;
if(x<10)
    cout<<"X is less than ten";
if(y > 10)
    cout<<"wow y is greater than ten";
```

- Output:

Here since only the condition at the first if statement is true i.e. $x < 10$, the output is:

X is less than ten

Selection Structure – *if statement*

- There are three forms of *if statement*.

- **The if-else statement:**

- The syntax is:

```
if(expression)
{
    statement(s);
} else
{
    statement(s);
}
```

- If the value of the expression is true, the program executes the statement(s) following if otherwise the statement(s) following else.
 - If there is only one statement, it is not necessary to use braces.

Selection Structure - *if statement*

- There are three forms of if statement.

- The if-else statement:

- For example:

```
int x = 3;  
if(x>2)  
    cout<<"condition true";  
else  
    cout<<"condition false";
```

- Output:

Since $x = 3$, $x > 2$ is true so the output is - condition true.

If we had $x = 1$, then $x > 2$ would have been false giving us an output as – condition false.

Selection Structure – *if statement*

- There are three forms of *if statement*.

- *The if-else-if ladder:*

- The syntax is:

```
if(expression) {  
    Statement(s);  
} else if(expression) {  
    Statement(s);  
} else if(expression) {  
    statement(s);  
}  
...  
} else {  
    Statement(s);  
}
```

- In this case, the program goes down until one of the expressions is true.
 - It then executes the following statement(s) and exits.
 - If none of the expressions are true, the program executes the statement(s) following *else*.
 - If there is only one statement, it is not necessary to use braces.

Selection Structure - *if statement*

- There are three forms of if statement.

- The if-else-if ladder:

- For example:

```
int age // this is taken as input
if( age < 18 )
    cout<< "You are a child";
else if( age < 55 )
    cout<< "You are an adult";
else
    cout<< "You are a senior";
```

- Output:

Determine the output if age = 24.

Selection Structure – switch statement

- The syntax form is:

```
switch (expression) {  
    case value 1:  
        statement(s);  
        break;  
    case value 2:  
        statement(s)  
        break;  
    .....  
    case value n:  
        statement(s);  
        break;  
    default:  
        statement(s);  
}
```

Selection Structure – switch statement

- In this case, the value of the expression is compared with each of the constant values in the case statements.
- If a match is found, the program executes the statement(s) following the matched case statement.
- If none of the constants matches the value of the expression, then the program executes the statement(s) following default statement.
- However, the default statement is optional.
- The value of the expression must be of type integer or character and each of the values specified in the case statement must be of a type compatible with the expression.
- Each case value must be a unique literal. Duplicate case values are not allowed.
- The break statement is used to terminate the entire switch statement.

Selection Structure – switch statement

- Consider the following example:

```
switch( month )
{
    case 4:
    case 6:
    case 9:
    case 11:
        cout<< "30 days";
        break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        cout<< "31 days";
        break;
    case 2:
        cout<< "if leap year 29 days else 28 days";
        break;
    default:
        cout<< "not a number for months must be 1-12";
}
```

Loop Structure

- These structures repeatedly execute a section of your program a certain number of times.
- The repetition continues while a condition is true.
- When the condition becomes false, the loop ends and control passes to the statement following the loop.
- C++ provides three kinds of loops:
 - **for** loop
 - **while** loop
 - **do while** loop.

Loop Structure – *for loop*

- The general form is:
for(initialization; test expression; iteration)
{
 Statement(s);
}
- First before loop starts, the initialization statement is executed that initializes the loop control variable or variables in the loop.
- Second the test expression is evaluated, if it is true, the program executes the statement(s).
- Finally, the iteration (usually an expression that increments or decrements the loop control variables) statement will be executed, and the test expression will be evaluated, this continues until the test expression is false.
- The curly braces are unnecessary if only a single statement is being repeated.
- When we have the fixed number of the iteration known then we usually (although not always) use for loop.

Loop Structure – *for loop*

- *For example*

```
for( int i=10; i > 1; i-- )  
    cout<<i;
```

- *Output:*

10 9 8 7 6 5 4 3 2

Loop Structure – while loop

- The general form is:

```
while(test expression)
{
    statement(s);
}
```
- Here the program executes the statement(s) as long as the test expression is true.
- When the test expression becomes false, the while loop stops executing the statement(s).
- The curly braces are unnecessary if only a single statement is being repeated.
- This loop structure is usually used if we don't know the number of iterations before the loop starts.

Loop Structure – while loop

- *For example*

```
int i=1;  
while( i <= 10 )  
{  
    cout<< "i=" << + i;  
    i++;  
}
```

- *Output:*

- The above code prints the number 1 to 10 as $i = 1, i = 2, \dots, i = 10$, when i reaches 11 the condition $i \leq 10$ becomes false and the loop terminates.

- *Remember:*

- Do not forget to increment i , otherwise loop will never terminate
 - i.e. code goes into infinite loop.
 - So when using loop, remember to guarantee its termination.

Loop Structure – *do while loop*

- *The general form is:*

```
do  
{  
statement(s);  
} while(test expression);
```

- *In this case, the program executes the statement(s) first and then evaluates the test expression.*
- *If this expression is true, the loop will repeat.*
- *Otherwise, the loop terminates.*
- *The curly braces are unnecessary if only a single statement is being repeated.*
- *This loop always executes statement(s) at least once, because its test expression is at the bottom of the loop.*

Loop Structure – do while loop

- *For example*

```
int i = 10;  
do{  
    cout<< "Hello";  
    i++;  
} while( i < 10 );
```

- *Output:*

“Hello” is printed, though $i < 10$ is false.

Jump Statements

- For the transfer of control from one part of the program to another, C++ supports three jump statements:
 - ***Break***
 - ***Continue***
 - ***return***

Jump Statements – *break*

- The use of *break* statement causes the immediate termination of the *switch* statement and the loop (all type of loops) from the point of *break* statement.
- The control then passes to the statements following the *switch* statement and the loop.
- **For example**

```
for( int i = 1; i <= 10; i++ )  
{  
    if( i == 5 )  
        break;  
    cout<< " i = "<< i;  
}
```
- **Output:**
 - *i = 1 i = 2 i = 3 i = 4*
 - In the code above the loop is terminated immediately after the value of *i* is 5.
- **Remember:**
 - In case of nested loop if *break* statement is in inner loop only the inner loop is terminated.

Jump Statements – continue

- Continue statement causes the execution of the current iteration of the loop to stop, and then continue at the next iteration of the loop.

- Example:

```
for( int i=1; i<= 10; i++ )  
{  
    if( i == 5 || i == 7 )  
    {  
        continue;  
    }  
    cout<<i<<"";  
}
```

- Output:

1 2 3 4 6 8 9 10

Jump Statements – *return*

- *It is used to transfer the program control back to the caller of the method.*
- *There are two forms of return statement.*
- *First with general form **return expression**; returns the value whereas the second with the form **return**; returns no value but only the control to the caller.*

Function – Default Arguments

- When declaring a function we can specify a default value for each of the last parameters which are called default arguments.
- This value will be used if the corresponding argument is left blank when calling to the function.
- To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration.
- If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

Function – Default Arguments

- For example:

```
#include <iostream>
using namespace std;
int divide (int a, int b=2)
{
    int r;
    r=a/b;
    return (r);
}
int main ()
{
    cout << divide (12);
    cout << endl;
    cout << divide (20,4);
    return 0;
}
```

- As we can see in the body of the program, there are two calls to function divide.
- If we do not pass second parameter in the function divide, the function sets the second parameter as 2.
- If we do pass the second parameter in the function divide, the function replaces the second parameter 2 by the passed argument.

Inline Functions

- *Function call is a costly operation.*
- *During the function call execution, our system takes overheads like: Saving the values of registers, Saving the return address, Pushing arguments in the stack, Jumping to the called function, Loading registers with new values, Returning to the calling function, and reloading the registers with previously stored values.*
- *For large functions this overhead is negligible but for small function taking such large overhead is not justifiable.*
- *To solve this problem concept of inline function is introduced in C++.*

Inline Functions

- *The functions which are expanded inline by the compiler each time it's call is appeared instead of jumping to the called function as usual is called inline function.*
- *This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.*

```
#include <iostream>
#include<conio.h>
using namespace std;
inline void sum(int a,int b)
{
    int s;
    s= a+b;
    cout<<"Sum="<<s<<endl;
}
int main()
{
    int x,y;
    cout<<"Enter two numbers"<<endl;
    cin>>x>>y;
    sum(x,y);
    return 0;
}
```

- Here at the time of function call instead of jumping to the called function, function call statement is replaced by the body of the function.
- So there is no function call overhead.

Overloaded Functions

- In C++ two different functions can have the same name if their parameter types or number are different.
- That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters.
- This is called *function overloading*.

- For example:

```
#include <iostream>
using namespace std;
int mul (int a, int b)
{
    return (a*b);
}
float mul (float a, float b)
{
    return (a*b);
}
int main ()
{
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << mul (x,y);
    cout << "\n";
    cout << mul(n,m);
    cout << "\n";
    return 0;
}
```

- In the first call to “mul” the two arguments passed are of type `int`, therefore, the function with the first prototype is called.
- This function returns the result of multiplying both parameters.
- While the second call passes two arguments of type `float`, so the function with the second prototype is called.
- Thus behavior of a call to `mul` depends on the type of the arguments passed because the function has been overloaded.
- Notice that a function cannot be overloaded only by its return type.
- At least one of its parameters must have a different type.

Arguments passed by value

- In case of pass by value, copies of the arguments are passed to the function not the variables themselves.
- For example, suppose that we called our function addition using the following code:

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r=a+b;
    return (r);
}
int main ()
{
    int z;
    int x=5, y=3;
    z = addition (x,y);
    cout << "The result is " << z;
    return 0;
}
```

Arguments passed by reference

- In case of pass by reference, Address of the variables (variable itself) not copies of the arguments are passed to the function.
- For example, suppose that we called our function addition using the following code:

```
#include <iostream>
using namespace std;
int addition (int &a, int &b)
{
    int r;
    r=a+b;
    return (r);
}
int main ()
{
    int z;
    int x=5, y=3;
    z = addition (x,y);
    cout << "The result is " << z;
    return 0;
}
```

Return by Reference

- If we return address (reference) rather than value from a function then it is called return by reference.

```
#include<iostream>
#include<conio.h>
using namespace std;
int &min(int &x, int &y)
{
    if(x<y)
        return x;
    else
        return y;
}
int main()
{
    int a,b,r;
    cout<<"Enter two numbers"<<endl;
    cin>>a>>b;
    min(a,b)=0;
    cout<<"a="<<a<<endl<<"b="<<b;
    return 0;
}
```

Scope of Variables

- All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable.
- For most of the cases its between the curly braces, in which variable is declared that a variable exists, not outside it.
- There are three types of variable scope in C++:
 - Local Scope
 - Global Scope
 - Class Scope

Scope of Variables

- *There are three types of variable scope in C++:*
 - **Local Scope**
 - *A variable declared within a block of code enclosed by braces ({}) is accessible only within the block, and only after the point of declaration.*
 - *Outside that they are unavailable and leads to compile time error.*
 - **Global Scope**
 - *Any variable declared outside all blocks or classes has global scope.*
 - *It is accessible anywhere in the file after its declaration.*
 - **Class Scope**
 - *Members of classes have class scope.*
 - *Variables with class scope are accessible in all of the methods of the class.*
 - *Details of class scope will be considered in upcoming chapters.*

Scope of Variables – Example

```
#include<iostream>
using namespace std;
int g=20; //global variable
int main()
{
    int i=10; //local variable
    if(i<20) //if condition scope starts
    {
        int n=100; //local variable
        cout<<"Inside if"<<endl;
        cout<<"g= "<<g<<endl;
        cout<<"i= "<<i<<endl;
        cout<<"n= "<<n<<endl;
    } //if condition scope ends
    cout<<"Below if"<<endl;
    cout<<"g= "<<g<<endl;
    cout<<"i= "<<i<<endl;
    cout<<"n= "<<n<<endl; //ERROR (cannot be accessed here)
    return 0;
}
```

Storage Classes

- *Storage class of a variable also defines the lifetime and scope (visibility) of a variable.*
- *Lifetime* means the duration till which the variable remains active and *visibility* defines in which module of the program the variable is accessible.
- *There are five types of storage classes in C++. They are:*
 - *Automatic*
 - *External*
 - *Static*
 - *Register*
 - *Mutable*

Storage Classes

- ***Automatic Storage Class***

- *The keyword `auto` is used to declare automatic variables.*
- *However, if a variable is declared without any keyword inside a function, it is automatic by default.*
- *This variable is visible only within the function it is declared and its lifetime is same as the lifetime of the function as well.*
- *Once the execution of function is finished, the variable is destroyed.*

Storage Classes

- ***External Storage Class***

- *External storage class assigns variable a reference to a global variable declared outside the given program.*
- *The keyword `extern` is used to declare external variables.*
- *They are visible throughout the program and its lifetime is same as the lifetime of the program where it is declared.*
- *This visible to the functions present in the program.*

Storage Classes

- **Static Storage Class**
 - *Static storage class ensures a variable has the visibility mode of a local variable but lifetime of an external variable.*
 - *It can be used only within the function where it is declared but destroyed only after the program execution has finished.*
 - *When a function is called, the variable defined as static inside the function retains its previous value and operates on it.*
 - *This is mostly used to save values in a recursive function.*

Storage Classes

- **Register Storage Class**

- Register storage assigns a variable's storage in the CPU registers rather than primary memory.
- It has its lifetime and visibility same as automatic variable.
- The purpose of creating register variable is to increase access speed and makes program run faster.
- If there is no space available in register, these variables are stored in main memory and act similar to variables of automatic storage class.
- So only those variables which requires fast access should be made register.
- Keyword register is used for specifying register variables.

Storage Classes

- ***Mutable Storage Class***
 - *The mutable specifier applies only to objects.*
 - *It allows a member of an object to override const member function.*
 - *That is, a mutable member can be modified by a const member function.*

Storage Classes – Example

```
#include<iostream>
using namespace std;
int g; // global variable, initially holds 0
void test()
{
    static int s; // static variable, initially holds 0
    register int r; // register variable
    r=5;
    s=s+r*2;
    cout<<"Inside test"<<endl;
    cout<<"g= "<<g<<endl;
    cout<<"s= "<<s<<endl;
    cout<<"r= "<<r<<endl;
}
int main()
{
    int a; // or declare as: auto int a; automatic variable
    g=25;
    a=17;
    test();
    cout<<"Inside main"<<endl;
    cout<<"a= "<<a<<endl;
    cout<<"g= "<<g<<endl;
    test();
    return 0;
}
```

Pointers

- *A pointer is a variable whose value is the address of another variable.*
- *Like any variable or constant, you must declare a pointer before you can work with it.*
- *The general form of a pointer variable declaration is:*
*type *var-name;*
- *Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable.*
- *The asterisk you used to declare a pointer is the same asterisk that you use for multiplication.*
- *However, in this statement the asterisk is being used to designate a variable as a pointer.*

Pointers

- Following are the valid pointer declaration:

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

- The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address.
- The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Address-of and Dereferencing Operator

- The address of a variable can be obtained by preceding the name of a variable with an ampersand (&) sign, known as address-of operator.
- For example:
`int a=10;`
`int *p=&a; // pointer declaration and initialization`
- This would assign the address of variable *a* to pointer variable *p*.
- We are no longer assigning the content of the variable itself to *p*, but its address.
- The actual address of a variable in memory cannot be known before runtime.
- In order to help clarify the concepts lets assume that *a* is placed during runtime in the memory address 1776 and *p* is placed in memory address 3005.

Pointer – Example

```
#include<iostream>
using namespace std;
int main()
{
    int v=20;          // actual variable declaration
    int *p; //pointer variable
    p=&v; //store address of variable in pointer variable
    cout<<"Value of v variable: "<<v<<endl;
    cout<<"Address stored in p variable: "<<p<<endl;
    cout<<"Value of *p variable: "<<*p<<endl;
    return 0;
}
```

Pointer and Arrays

- *The concept of arrays is related to that of pointers.*
- *In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type.*
- *For example, consider these two declarations:*
`int x[20];`
`int *p;`
- *Now the following assignment operation would be valid:*
`p=x;`
- *After that, pointer p and x would be equivalent and would have very similar properties.*
- *The main difference being that p can be assigned a different address, whereas x can never be assigned anything, and will always represent the same block of 20 elements of type int.*
- *Therefore, the following assignment would not be valid:*
`x=p; // not valid`

Pointer & Array – Example

```
#include<iostream>
int MAX=3;
using namespace std;
int main()
{
    int i;
    int var[MAX]={10,100,1000};
    int *ptr;
    ptr=var;
    for(i=0;i<MAX;i++)
    {
        cout<<"Value of var["<<i<<"]="<<*ptr<<endl;
        ptr++;           // point to the next location
    }
    return 0;
}
```

END OF UNIT TWO

Syllabus

- **UNIT 3: Classes & Objects**

- *A simple Class and Object*
- *Accessing members of class*
- *Initialization of class object*
- *Constructor & Destructor*
- *Objects as Function Arguments*
- *Returning Objects from Functions*
- *Structures and Classes*
- *Memory allocation for Objects*
- *Static data members*
- *Member functions defined outside the class*

UNIT 3

CLASSES & OBJECTS

Introduction

- Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into a single unit called classes.
- The data components of the class are called **data members** and the function components are called **member functions**.
- The data and functions of a class are intimately tied together.
- Class is a blueprint of real world objects.

Introduction

- *A programmer can create any number of objects of the same class.*
- *Classes have the property of information hiding.*
- *It allows data and functions to be hidden, if necessary, from external use.*
- *Classes are also referred to as programmer-defined data types.*

Structure in C & C++

- ***Structures in C***

- *One of the unique features of C language*
- *Provides method for packing together the data of different types*
- *Convenient tool for handling a group of logically related data items*
- *We cannot add or subtract two structure variables in C*
- *Data hiding is not permitted in C*

Structure in C & C++

- **Structures in C++**
 - C++ structure supports all features of structures in C
 - We can easily add or subtract two structure variables
 - C++ structure can have both data and function. Data are called data member while functions are called the member function.
 - In C++, structure names are stand-alone. i.e. keyword struct may be omitted in the declaration of structure variable.
 - In C++, a structure member can be declared as “**Private**” so that they cannot be accessed directly by the external function.

Extensions to C structure by C++

- Allows adding functions as a member of structure

```
class Employee
{
    .....
    .....
    void getdata()
    {
        .....
        .....
    }
    .....
    .....
}
```

Extensions to C structure by C++

- Allows us to use the access specifiers *private* , *public*, and *protected* to use inside the class

```
class Employee
{
    public:
        int eid, sal;
    private:
        void getdata()
    {
        //function body
    }
    .....
    .....
}
```

Extensions to C structure by C++

- Allows us to use structures similar to that of primitive data types while defining variables

```
struct Employee e;           // C style  
Employee e;                 // C++ style
```

Example - 1

```
#include<iostream>
using namespace std;
class Employee
{
public:
    int eid,sal;
private:
    void getdata( )
    {
        cout<<"Enter ID and Salary of an employee"<<endl;
        cin>>eid>>sal;
    }
    void display( )
    {
        cout<<"p ID:"<<eid<<endl<<"Salary:"<<sal<<endl;
    }
};
int main()
{
    Employee e;
    e.getdata( );
    cout<<"Employee Details::::"<<endl;
    e.display( );
    return 0;
}
```

Specifying a Class

- *The specification starts with the keyword **class** followed by the class name.*
- *Like structure, its body is delimited by braces terminated by a semicolon.*
- *The body of the class contains the keywords **private**, **public**, and **protected**.*

Specifying a Class

- *Private data and functions can only be accessed from within the member functions of that class.*
- *Public data or functions, on the other hand are accessible from outside the class.*
- *Usually the data within a class is private and functions are public.*
- *The data is hidden so it will be safe from accidental manipulation, while the functions that operated on the data are public so they can be accessed from outside the class.*

```
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        void getdata()
        {
            int l,b;
            cout<<"Enter length and breadth of a rectangle"<<endl;
            cin>>l>>b;
        }
        void showdata()
        {
            int l,b;
            cout<<"Length = "<<l<<endl<<"Breadth = "<<b<<endl;
        }
        int findarea()
        {
            int l,b;
            return l * b;
        }
};
```

Access Modifiers

- Access modifiers are constructs that defines the scope and visibility of members of a class.
- There are three access modifiers:
 - **Private**
 - Private members are accessible only inside the class and not from any other location outside of the class.
 - **Protected**
 - Protected members are accessible from the class as well as from the child class but not from any other location outside of the class.
 - **Public**
 - Public members are accessible from any location of the program.

Creating Objects

- *The class declaration does not define any objects but only specifies what they will contain.*
- *Once a class has been declared, we can create variables (objects) of that type by using the class name (like any other built-in type variables).*
- *For example: `rectangle r;` creates a variable (object) `r` of type `rectangle`.*
- *We can create any number of objects from the same class.*
- *For example: `rectangle r1, r2, r3;`*

Creating Objects

- Objects can also be created when a class is defined by placing their names immediately after the closing braces. For example,

```
class rectangle
{
    .....
    .....
}r1, r2, r3;
```

Accessing Class Members

- When an object of the class is created then the members are accessed using the ‘.’ dot operator.

- For example:

```
r.setdata(4,2);
```

```
r.showdata();
```

```
cout<<"Area = "<<r.findarea()<<endl;
```

```
cout<<"Perimeter = "<<r.findperimeter();
```

- Private class members cannot be accessed in this way from outside of the class.

- For example, if the following statements are written inside the main function, the program generates compiler error.

```
r.length = length;
```

```
r1.breadth = breadth;
```

A Complete Example

```
#include<iostream>
#include<conio.h>
using namespace std;
class rectangle
{
private:
int length,breadth;
public:
void setdata(int l, int b)
{
length = l;
breadth = b;
}
void showdata()
{
cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
}
int findarea()
{
return length * breadth;
}
int findperimeter()
{
return (2 * (length + breadth));
}
};
```

```
int main()
{
rectangle r;
r.setdata(4, 2);
r.showdata();
cout<<"Area= "<<r.findarea()<<endl;
cout<<"Perimeter= "<<r.findperimeter();
return 0;
}
```

Defining Member Functions Outside of the Class

- The member functions that are declared inside a class have to be defined separately outside the class.
- The general form of this definition is:
return-type class-name ::function-name(argument declaration)
{
 Function body
}
- The symbol `::` is called the binary scope resolution operator.

A Complete Example

```
#include<iostream>
#include<conio.h>
using namespace std;
class rectangle
{
private:
    int length, breadth;
public:
    void setdata(int l, int b);
    void showdata();
    int findarea();
    int findperimeter();
};

void rectangle :: setdata(int l, int b)
{
    length = l; breadth = b;
}
void rectangle :: showdata()
{
    cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
}
int rectangle :: findarea()
{
    return length * breadth;
}
int rectangle :: findperimeter()
{
    return (2 * (length + breadth));
}
int main()
{
    rectangle r;
    r.setdata(4, 2);
    r.showdata();
    cout<<"Area= "<<r.findarea()<<endl;
    cout<<"Perimeter= "<<r.findperimeter();
    return 0;
}
```

Scope Resolution Operator (::)

- The scope resolution operator can also be used to uncover hidden global variable.
- In this case it is called unary scope resolution operator.
- The general form is: :: variable-name
- For Example:

```
#include<iostream>
using namespace std;
int m = 4;
int main()
{
    int m = 2;
    cout<<m<<endl;
    cout<<::m;
    return 0;
}
```

- Output:

2
4

Memory Allocation for Objects

- For each object, the memory space for data members is allocated separately because the data members will hold different data values for different objects.
- However, all the objects in a given class use the same member functions.
- Hence, the member functions are created and placed in memory only once when they are defined as a part of a class specification and no separate space is allocated for member functions when objects are created.

Memory Allocation for Objects

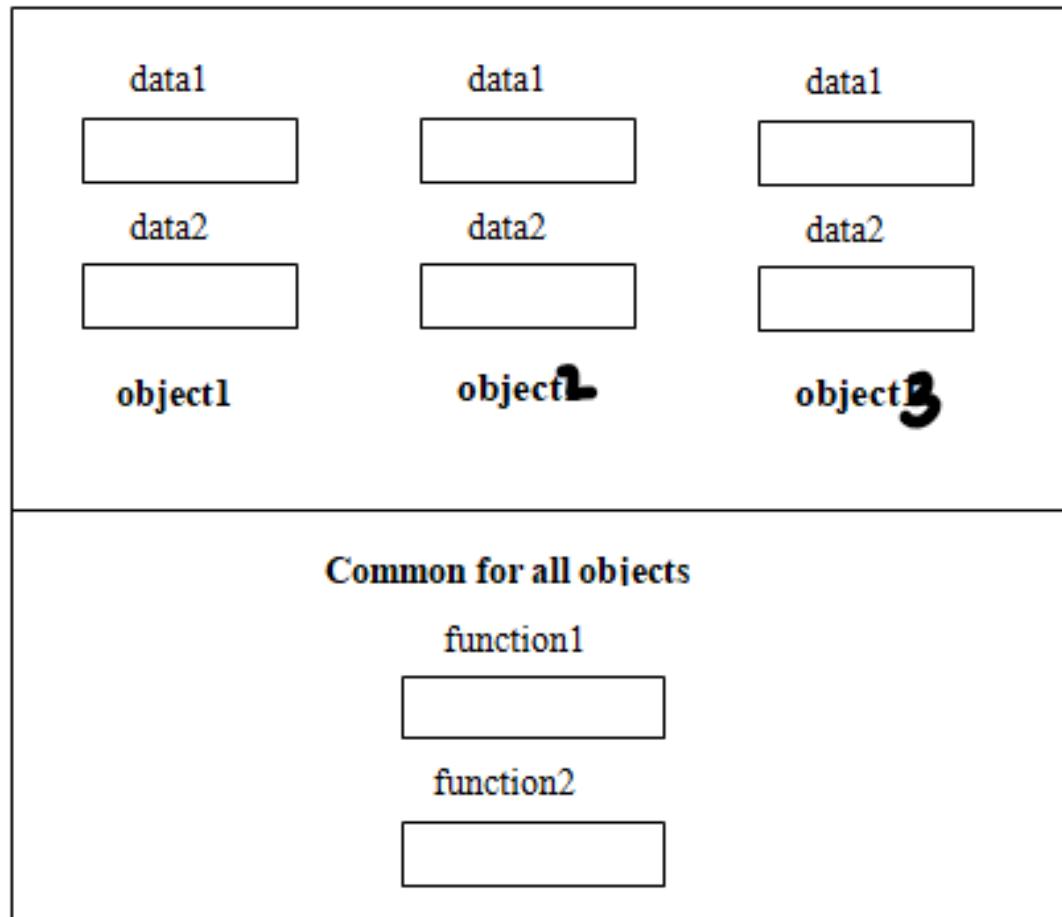


Fig: Objects in Memory

Nesting of Member Functions

- A member function can be called by using its name inside another member function of the same class.
- This is known as nesting of member functions.
- **Remember:**
 - Like private data member, some situations may require certain member functions to be hidden from the outside call.
 - A private member function can only be called by another member function that is a member function of its class.

Nesting of Member Functions

```
#include<iostream>
using namespace std;
class nest
{
int a;
int square_num()
{
    return a*a;
}
public:
void input_num()
{
    cout<<"Enter a number:"<<endl;
    cin>>a;
}
int cube_num()
{
    return a*a*a;
}
void disp_num()
{
    int sq=square_num(); //nesting of member function
    int cu=cube_num(); //nesting of member function
    cout<<"The square of "<<a<<" is " <<sq<<endl;
    cout<<"The cube of "<<a<<" is " <<cu;
}
};
```

```
int main()
{
nest n1;
n1.input_num();
n1.disp_num();
return 0;
}
```

Static Data Members

- *If a data member in a class is defined as **static**, then only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.*
- *Hence, these data members are normally used to maintain values common to the entire class and are also called class variables.*

```
#include<iostream>
using namespace std;
class rectangle
{
    private:
        int length;
        int breadth;
        static int count; // static data member
    public:
        void setdata(int l, int b)
        {
            length = l;
            breadth = b;
            count++;
        }
        void displaycount()
        {
            cout<<count<<endl;
        }
};
int rectangle :: count;
```

```
int main()
{
    rectangle r1, r2, r3;
    r1.displaycount();
    r2.displaycount();
    r3.displaycount();
    r1.setdata(15,6);
    r2.setdata(9,8);
    r3.setdata(12,9);
    r1.displaycount();
    r2.displaycount();
    r3.displaycount();
    return 0;
}
```

Static Member Functions

- Like static member variables, we can also have static member functions.
- A static member function can have access to only other static members (functions or variables) declared in the same class and can be called using the class name (instead of objects) as follows:

class-name ::function-name;

```
#include<iostream>
using namespace std;
class rectangle
{
    private:
        int length;
        int breadth;
        static int count;
    public:
        void setdata(int l, int b)
        {
            length = l;
            breadth = b;
            count++;
        }
        static void displaycount()
        {
            cout<<count<<endl;
        }
};
int rectangle :: count;
```

```
int main()
{
    rectangle r1, r2, r3;
    rectangle :: displaycount();
    r1.setdata(6, 5);
    r2.setdata(8, 4);
    r3.setdata(15, 7);
    rectangle :: displaycount();
    return 0;
}
```

Example

```
/* program that adds two distance objects each having two private data members feet  
(type int) and inches (type float). */  
  
#include<iostream>  
  
#include<conio.h>  
  
using namespace std;  
  
class dist  
{  
  
private:  
    int feet;  
    int inches;  
  
public:  
    void setdata(int f,int i)  
    {  
        feet=f;  
        inches=i;  
    }  
    dist adddinstance(dist d2)  
    {  
        dist d3;  
        d3.feet =feet + d2.feet;  
        d3.inches =inches + d2.inches;  
        d3.feet=d3.feet + d3.inches/12;  
        d3.inches=d3.inches%12;  
        return d3;  
    }  
    void display()  
    {  
        cout<<"(" <<feet << ", " <<inches << ")" <<endl;  
    }  
};
```

```
int main()  
{  
    dist d1, d2, d3;  
  
    d1.setdata(5,6);  
  
    d2.setdata(7,8);  
  
    d3 = d1.adddinstance(d2);  
  
    cout<<"d1 = "; d1.display();  
  
    cout<<"d2 = "; d2.display();  
  
    cout<<"d3 = "; d3.display();  
  
    return 0;  
}
```

Objects as Function Arguments

- Like any other data type, an object may be used as a function argument in three ways:
 - *Pass-by-Value*
 - *Pass-by-Reference*
 - *Pass-by-Pointer*

Objects as Function Arguments

- **Pass-by-Value**

- In this method, a copy of the object is passed to the function.
- Any changes made to the object inside the function do not affect the object used in the function call.
- For example:

```
distance addDistance(distance d)
{
    distance dd;
    dd.feet = feet + d.feet;
    dd.inches = inches + d.inches;
    dd.feet = dd.feet + dd.inches / 12
    dd.inches = dd.inches % 12;
    return dd;
}
```

Objects as Function Arguments

- **Pass-by-Reference**

- *In this method, an address of the object is passed to the function.*
- *The function works directly on the actual object used in the function call.*
- *This means that any changes made to the object inside the function will reflect in the actual object.*
- *For example:*

```
distance adddistance(distance &d2)
{
    distance d3;
    d3.feet = feet + d2.feet;
    d3.inches = inches + d2.inches;
    d3.feet=dd.feet + dd.inches / 12
    d3.inches=dd.inches%12;
    return d3;
}
```

Objects as Function Arguments

- **Pass-by-Pointer**

- Like pass-by-reference method, pass-by-pointer method can also be used to work directly on the actual object used in the function call.

- For example:

```
distance adddistance(distance *d2)
{
    distance d3;
    d3.feet = feet + d2->feet;
    d3.inches = inches + d2->inches;
    d3.feet=dd.feet + dd.inches / 12
    d3.inches=dd.inches%12;
    return d3;
}
```

- This function must be called as:

```
d3=d1.addInstance(&d2);
```

Returning Objects

- A function cannot only receive objects as arguments but also can return them.
- A function can return objects by the following three ways:
 - Return-by-Value
 - Return-by-Reference
 - Return-by-Pointer

Returning Objects

- **Return-by-Value**

- In this method, a copy of the object is returned to the function call.

- For example:

```
distance adddistance(distance d2)
{
    distance d3;
    d3.feet = feet + d2.feet;
    d3.inches = inches + d2.inches;
    d3.feet = dd.feet + dd.inches / 12
    d3.inches = dd.inches % 12;
    return d3;
}
```

Returning Objects

- **Return-by-Reference**

- In this method, an address of the object is returned to the function call.
- We cannot return automatic variables by reference.
- For example, in the following function d3 is not an automatic variable and is returned by reference.

```
distance& adddistance(distance d2, distance& d3)
{
    d3.feet = feet + d2.feet;
    d3.inches = inches + d2.inches;
    d3.feet=d3.feet +d3.inches / 12
    d3.inches=d3.inches%12;
    return d3;
}
```

- This function must be called as follows:

```
d3=d1.adddistance(d2,d3);
```

Returning Objects

- **Return-by-Pointer**

- Like return-by-reference method, return-by-pointer returns the address of the object to the function call.

- For example, in the following function d3 is returned by pointer.

```
distance* adddistance(distance d2)
{
    distance* d3;
    d3->feet = feet + d2.feet;
    d3->inches = inches + d2.inches;
    d3.feet=dd.feet + dd.inches/12
    d3.inches=dd.inches%12;

    return d3;
}
```

- In the main function, d3 must be declared as a pointer variable and the members of d3 must be accessed as follows:

```
distance d1, d2, *d3;
cout<<"d3 = "; d3->display();
```

Friend Functions

- *The concepts of data hiding and encapsulation dictate that private members of a class cannot be accessed from outside the class, that is, non-member functions of a class cannot access the non-public members (data members and member functions) of a class.*
- *However, we can achieve this by using friend functions.*
- *To make an outside function friendly to a class, we simply declare the function as a **friend** of the class.*

```
#include<iostream>
using namespace std;
class sample
{
    private:
        int a;
        int b;
    friend float mean(sample s);
public:
    void setvalue()
    {
        a = 25;
        b = 40;
    }
};
```

```
float mean(sample s)
{
    return float(s.a + s.b)/2;
}

int main()
{
    sample x;
    x.setvalue();
    cout<<"Mean      value      = "
         <<mean(x);
    return 0;
}
```

Friend Classes

- *The member functions of a class can all be made friends of another class when we make the former entire class a friend of later.*
- *For example, if in a class **alpha** the entire class **beta** is declared as a **friend**, then all the members of **beta** can access the private data of **alpha**.*

```
#include<iostream>
using namespace std;
class alpha
{
private:
    int x;
public:
    void setdata(int d)
    {
        x = d;
    }
    friend class beta;
};
```

```
class beta
{
public:
    void func(alpha a)
    {
        cout<<a.x<<endl;
    }
};

int main()
{
    alpha a;
    a.setdata(99);
    beta b;
    b.func(a);
    return 0;
}
```

Constructors

- *A constructor is a special member function that is executed automatically whenever an object is created.*
- *It is used for automatic initialization.*
- *Automatic initialization is the process of initializing object's data members when it is first created, without making a separate call to a member function.*
- *The name of the constructor is same as the class name.*
- *There are three types of constructor:*
 - *Default Constructor*
 - *Parameterized Constructor*
 - *Copy Constructor*

Constructors

- For example:

```
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle()      // constructor
    {
        length = 0;
        breadth = 0;
    }
    .......
    .......
};
```

Characteristics of Constructors

- Constructors should be *defined or declared in the public section.*
- They do not have return types.
- They cannot be inherited but a derived class can call the base class constructor.
- Like functions, they can have default arguments.
- Constructors cannot be **virtual**.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make ‘implicit calls’ to the new and delete operators when a memory allocation is required.

Default Constructor

- *A constructor that accepts no parameters is called default constructor.*
- *If a class does not include any constructor, the compiler supplies a default constructor.*
- *If we create an object by using the declaration **rectangle r1;** default constructor is invoked.*

Default Constructor

```
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle()      // Default Constructor
    {
        length = 0;
        breadth = 0;
    }
    .......
    .......
};
```

Parameterized Constructor

- *Unlike a default constructor, a constructor may have arguments.*
- *The constructors that take arguments are called parameterized constructors.*
- *For example, the constructor used in the following example is the parameterized constructor and takes two arguments both of type **int**.*

Parameterized Constructor

```
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle(int l, int b)      // Parameterized Constructor
        {
            length = l;
            breadth = b;
        }
        .......;
        .......;
};
```

Copy Constructor

- A copy constructor is used to declare and initialize an object with another object of the same type.
- For example, the statement **rectangle r2(r1);** creates new object **r2** and performs member-by-member copy of **r1** into **r2**.
- Another form of this statement is **rectangle r2 = r1;**
- The process of initializing through assignment operator is known as copy initialization.
- A copy constructor takes reference to an object of the same class as its argument.
- We cannot pass the argument by value to a copy constructor.

Copy Constructor

```
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle(rectangle &r)      // Copy Constructor
    {
        length = r.length;
        breadth = r.breadth;
    }
    .......
    .......
};
```

Example of Constructors

```
#include<iostream>
using namespace std;
class Item
{
    int code, price;
public:
    Item()      //Default Constructor
    {
        code= price =0;
    }
    Item(int c,int p)          //Parameterized Constructor
    {
        code=c;
        price=p;
    }
    Item(Item &x)            //Copy Constructor
    {
        code=x.code;
        price= x.price;
    }
    void display()
    {
        cout<<"Code:::"<<code<<endl<<"Price:::"<<price<<endl;
    }
};
```

```
int main()
{
    Item I1;
    Item I2(102,300);
    Item I3(I2);
    I1.display();
    I2.display();
    I3.display();
    return 0;
}
```

Constructor Overloading

- We can define more than one constructor in a class either with different number of arguments or with different type of argument which is called constructor overloading.

```
// constructor overloading
#include <iostream>
using namespace std;
class Account
{
    private:
        int accno; float balance;
    public:
        Account()
            { accno=1024, balance=5000.55; }
        Account(int acc)
            { accno=acc; balance=0.0; }
        Account(int acc, float bal)
            { accno=acc; balance=bal; }
        void display()
        {
            cout<<"Account no. = "<<accno<<endl;
            cout<<"Balance = "<<balance<<endl;
        }
};
```

```
int main()
{
    Account acc1;
    Account acc2(100);
    Account acc3(200, 8000.50);
    cout<<endl<<"Account
information"<<endl;
    acc1.display();
    acc2.display();
    acc3.display();
    return 0;
}
```

Destructors

- A destructor is a special member function that is executed automatically just before lifetime of an object is finished.
- A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde (~).
- Like constructors, destructors do not have a return value.
- They also take no arguments.
- Hence, we can use only one destructor in a class.
- The most common use of destructors is to deallocate memory that was allocated for the object by the constructor.

Destructors

```
#include<iostream>
using namespace std;
class Test
{
    private:
    int x,y;
    public:
    Test()      //constructor
    {
        cout<<"Memory is allocated"<<endl;
    }
    ~Test()     //destructor
    {
        cout<<"Memory is deallocated"<<endl;
    }
};
int main()
{
    Test p;    //life time of p finishes here, and destructor is called
    return 0;
}
```

END OF UNIT THREE

Syllabus

- ***UNIT 4: Operator Overloading***
 - *Fundamental of Operator Overloading*
 - *Restriction on Operator Overloading*
 - *Operator Functions as a Class Members*
 - *Overview of Unary and Binary Operator*
 - *Prefix and Postfix Unary Operator Overloading*
 - *Overloading Binary Operator (Arithmetic, Comparison, Assignment)*
 - *Data Conversion*

UNIT 4

OPERATOR OVERLOADING

Introduction

- *Operator overloading is one of the most exciting features of C++.*
- *The term operator overloading refers to giving the normal C++ operators additional meaning so that we can use them with user-defined data types.*
- *For example, C++ allows us to add two variables of user-defined types with the same syntax that is applied to the basic type.*
- *We can overload all the C++ operators except the following:*
 - *Class member access operators (., .*)*
 - *Scope resolution operator (::)*
 - *Size operator (sizeof)*
 - *Conditional operator (?:)*

Introduction

- Although the semantics of an operator can be extended, we cannot change its syntax and semantics that govern its use such as the number of operands, precedence, and associativity.
- Operator overloading is done with the help of a special function, called operator function.
- The general form of an operator function is:
return-type operator op(arguments)
{
Function body
}

Introduction

- Where return-type is the type returned by the operation, operator is the keyword, and op is the operator symbol.
- For example, to add two objects of type distance each having data members feet of type int and inches of type float, we can overload + operator as follows:

distance operator +(distance d2)

{

//function body

}

- And we can call this operator function with the same syntax that is applied to its basic types as follows:

d3 = d1 + d2;

Introduction

- *Operator functions must be either member functions or friend functions.*
- *A basic difference between them is that*
 - *A friend function will have only one argument for unary operators and two for binary operators*
 - *A member function has no argument for unary operators and only one for binary operators.*
- *This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function.*
- *This is not the case with friend functions.*

Introduction

- *We cannot use friend functions to overload certain operators.*
- *These operators are:*
 - = *assignment operator*
 - () *function call operator*
 - [] *subscripting operator*
 - *class member access operator*

Overloading Unary Operators

- Let us consider the increment (++) operator.
- This operator increases the value of an operand by 1 when applied to a basic data item.
- This operator should increase the value of each data member when applied to an object.

```
#include<iostream>
#include<iomanip>
using namespace std;
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle(int l, int b)
        {
            length = l;
            breadth = b;
        }
        void operator ++()
        {
            ++length;
            ++breadth;
        }
        void display()
        {
            cout << "Length      =      " << length << endl << "Breadth      =      "
                << breadth;
        }
};
```

```
int main()
{
    rectangle r1(5, 6);
    ++r1;
    r1.display();
    return 0;
}
```

```
#include<iostream>
#include<iomanip>
using namespace std;
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle(int l, int b)
        {
            length = l;
            breadth = b;
        }
        void operator ++(int)
        {
            ++length;
            ++breadth;
        }
        void display()
        {
            cout<<"Length      =      "<<length<<endl<<"Breadth      =
            " <<breadth;
        }
};
```

```
int main()
{
    rectangle r1(5, 6);
    r1++;
    r1.display();
    return 0;
}
```

Overloading Unary Operators using Friend Function

- It is also possible to overload unary operators using a friend function.
- Since no object is used to invoke the friend function, no object is passed to the friend function implicitly.
- Therefore when overloading unary operator by using friend function, it must take an object as argument.
- You can overload pre-increment operator using friend function.

```
#include<iostream>
#include<iomanip>
using namespace std;
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle(int l, int b)
        {
            length = l;
            breadth = b;
        }
        friend void operator ++(rectangle&);
        void display()
        {
            cout<<"Length = "<<length<<endl<<"Breadth =
            " <<breadth;
        }
};
```

```
void operator ++(rectangle&r)
{
    ++r.length;
    ++r.breadth;
}

int main()
{
    rectangle r1(5, 6);
    ++r1;
    r1.display();
    return 0;
}
```

Overloading Unary Operators

Negation Operator

- Minus is the operator which can be binary as well as unary.
- If we write expression like “ $z=x-y$ ”, minus (-) operator acts as binary operator.
- But if we write expression like “ $x=-y$ ”, minus (-) operator acts as unary operator which makes value of y negative and puts that value in x .
- You can overload negation (-) operator to make value of object negative.

```
#include<iostream>
#include<iomanip>
using namespace std;
class point
{
private:
    int x,y;
public:
    void getdata()
    {
        cout<<"Enter x and y coordinate"<<endl;
        cin>>x>>y;
    }
    void display()
    {
        cout<<"("<<x<< , " <<y<<")"<<endl;
    }
    point operator - ()
    {
        point t;
        t.x=-x;
        t.y=-y;
        return t;
    }
};
```

Here, the return type is as the name of the class because we are returning multiple value at the end.

```
int main()
{
    point p,q;
    p.getdata();
    q=-p;
    cout<<"q=";
    q.display();
    return 0;
}
```

Overloading Binary Operators

- Let us consider the addition (+) operator.
- This operator adds the values of two operands when applied to a basic data item.
- This operator should add the values of corresponding data members when applied to two objects.

Overloading Binary Operators

```
#include <iostream>
using namespace std;
class Distance
{
public:
    int feet, inch;
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }
}
```

```
Distance operator + ( Distance d2)
{
    int ft = feet + d2.feet;
    int in = inch + d2.inch;
    ft=ft+in/12;
    in=in%12;
    return Distance(ft, in);
}
int main()
{
    Distance d1(8, 9);
    Distance d2(10, 5);
    Distance d3;
    d3 = d1 + d2;
    cout << "\nTotal Feet & Inches: " << d3.feet
    << "" << d3.inch << "\n";
    return 0;
}
```

Overloading Binary Operators Using Friend Function

```
#include <iostream>
using namespace std;
class Distance
{
public:
    int feet, inch;
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }
    friend Distance operator+(Distance&, Distance&);
};
Distance operator+(Distance& d1, Distance& d2) //  
Call by reference
{
```

```
Distance d3;
d3.feet = d1.feet + d2.feet;
d3.inch = d1.inch + d2.inch;
d3.feet = d3.feet + d3.inch/12;
d3.inch = d3.inch%12;
return d3;
}
int main()
{
    Distance d1(8, 9);
    Distance d2(10, 5);
    Distance d3;
    d3 = d1 + d2;
    cout << "\nTotal Feet & Inches: " <<
    d3.feet << " " << d3.inch << "\n";
    return 0;
}
```

Overloading Comparison Operator

- Overloading comparison operator is almost similar to overloading plus (+) operator except that it must return value of an integer type.
- This is because result of comparison is always true or false.
- C++ treats true as non-zero value and false as zero.
- You can overload < (less than) operator to compare two objects.

Overloading Comparison Operator

```
class Time
{
    int hr,min;
public:
void getdata()
{
    cout<<"Enter hour and minute"<<endl;
    cin>>hr>>min;
}
void display()
{
    cout<<hr<<" hr "<<min<<" min "<<endl;
}
int operator < (Time t)
{
    Time temp;
    if(hr<t.hr)
        return 1;
    else if(hr == t.hr && min< t.min)
        return 1;
    else
        return 0;
}
};
```

```
int main()
{
    Time t1,t2,t3;
    t1.getdata();
    t2.getdata();
    if(t1<t2)
    {
        cout<<"t1 is smaller"<<endl;
    }
    else
    {
        cout<<"t2 is smaller"<<endl;
    }
    return 0;
}
```

```
class String
{
    char s[20];
public:
void getdata()
{
    cout<<"Enter a string"<<endl;
    cin>>s;
}
void display()
{
    cout<<"s="<<s<<endl;
}
String operator + (String t)
{
    String temp;
    strcpy(temp.s,s);
    strcat(temp.s, t.s);
    return temp;
}
```

```
int main()
{
String s1,s2,s3;
s1.getdata();
s2.getdata();
s3=s1+s2;
s3.display();
return 0;
}
```

Overloading Assignment Operators

- We can overload assignment ($=$) operator in C++.
- By overloading assignment operator, all values of one object can be copied to another object by using assignment operator.
- Assignment operator must be overloaded by a non-static member function only.
- If the overloading function for the assignment operator is not written in the class, the compiler generates the function to overload the assignment operator.

Overloading Assignment Operator

```
#include<iostream>
#include<iomanip>
using namespace std;
class Marks
{
    int m1,m2;
public:
    Marks()
    {
        m1=0;
        m2=0;
    }
    Marks(int i, int j)
    {
        m1=i;
        m2=j;
    }
    void operator = (Marks &M)
    {
        m1=M.m1;
        m2=M.m2;
    }
    void display()
    {
        cout<<"Marks in 1st subject: "<<m1<<endl;
        cout<<"Marks in 2nd subject: "<<m2<<endl;
    }
};
```

```
int main()
{
    Marks mark1(45,89);
    Marks mark2(36,59);
    cout<<"Marks of first student: "<<endl;
    mark1.display();
    cout<<endl<<"Marks of second student:
    "<<endl;
    mark2.display();
    mark1=mark2;
    cout<<endl<<**mark1.display is now
    displaying marks of second student **"<<endl;
    mark1.display();
    return 0;
}
```

Nameless Temporary Objects

- We can also use nameless temporary objects to add two distance objects whose purpose is to provide a return value for the function.

- For example,

distance operator + (distance d2)

{

int ft = feet + d2.feet;

int in = inches + d2.inches;

ft=ft+in/12;

in=int%12;

return distance(ft, in); // an unnamed temporary object

}

Type Conversion

- *The type conversions are automatic as long as the data types involved are built-in types.*
- *If the data types are user defined, the compiler does not support automatic type conversion and therefore, we must design the conversion routines by ourselves.*
- *Three types of situations might arise in the data conversion in this case.*
 - *Conversion from basic type to class type*
 - *Conversion from class type to basic type*
 - *Conversion from one class type to another class type*

Conversion: Basic type to Class type

```
#include<iostream>
using namespace std;
class Time
{
    int hrs,min;
public:
    Time(int);
    void display();
};

Time ::Time(int t)
{
    cout<<"Basic Type to Class Type
Conversion"<<endl;
    hrs=t/60;
    min=t%60;
}
```

```
#include<iostream>
using namespace std;
class number{
int n;
public:
void disp(){
cout<<n;
}
number(int n){
this->n = n;
};
int main(){
number num = 100;
num.disp();
return 0;
}
```

```
void Time::display()
{
    cout<<hrs<< " Hours(s)" <<endl;
    cout<<min<< " Minutes" <<endl;
}

int main()
{
    int duration;
    cout<<"Enter time duration in
minutes:"<<endl;
    cin>>duration;
    Time t1=duration;
    t1.display();
    return 0;
}
```

Conversion: Class type to Basic type

```
#include<iostream>
#include<iomanip>
using namespace std;
class Time
{
    int hrs,min;
public:
    Time(int ,int); // constructor
    operator int(); // casting operator function
    ~Time() // destructor
    {
        cout<<"Destructor called..."<<endl;
    }
};

Time::Time(int a,int b)
{
    cout<<"Constructor called with two parameters..."<<endl;
    hrs=a;
    min=b;
}

Time ::operator int()
{
    cout<<"Class Type to Basic Type Conversion..."<<endl;
    return(hrs*60+min);
}
```

```
int main()
{
    int h,m,duration;
    cout<<"Enter Hours ";
    cin>>h;
    cout<<"Enter Minutes ";
    cin>>m;
    Time t(h,m); // construct object
    duration = t; // casting conversion OR duration = (int)t
    cout<<"Total Minutes are "<<duration<<endl;
    cout<<"2nd method operator overloading "<<endl;
    duration = t.operator int();
    cout<<"Total Minutes are "<<duration<<endl;
    return 0;
}
```

```
#include<iostream>
using namespace std;
class number{
    int n;
public:
    void disp(){
        cout<<n;
    }
    number(int n){
        this->n = n;
    }
    operator int(){
        return n;
    }
};

int main(){
    number num;
    num = 100;
    int x = num; /* Here primary variable want's to assign the value that was in class varibale or object */
    cout<<"The value of x:"<<x;
    return 0;
}
```

Conversion: Class type to Class type

```
#include<iostream>
#include<iomanip>
using namespace std;
class inventory1
{
    int ino,qty;
    float rate;
public:
    inventory1(int n,int q,float r)
    {
        ino=n;
        qty=q;
        rate=r;
    }
    int getino()
    {return(ino);}
    float getamt()
    {return(qty*rate);}
    void display()
    {
        cout<<endl<<"ino = "<<ino<<" qty = "<<qty<<" rate
        ="<<rate;
    }
};
```

```
class inventory2
{
    int ino;
    float amount;
public:
    void operator=(inventory1 I)
    {
        ino=I.getino();
        amount=I.getamt();
    }
    void display()
    {
        cout<<endl<<"ino = "<<ino<<" amount = "<<amount;
    }
};
int main()
{
    inventory1 I1(1001,30,75);
    inventory2 I2;
    I2=I1; //inventory2 I2=I1;
    I1.display();
    I2.display();
    return 0;
}
```

#include<iostream>
using namespace std;
//Rectangle should be declare d first class Rectangle { int width,length,area; public:
 Rectangle(int w,int l) { width=w; length=l; area=w*length; }
 void output() { cout<<"\nLength : "<<length<<"\nWidth : "<<width<<"\nArea of

END OF UNIT FOUR

Syllabus

- **UNIT 5: Inheritance**
 - *Introduction to inheritance*
 - *Derived class and Base class*
 - *Access specifiers (private, protected, and public)*
 - *Overriding member functions*
 - *Types of inheritance – Simple, Multiple, Hierarchical, Multilevel, Hybrid*
 - *Abstract base class*
 - *Public and Private Inheritance*
 - *Constructor and Destructor in derived classes*
 - *Ambiguity in multiple inheritance*
 - *Aggregation (class with in class)*

UNIT 5

INHERITANCE

Introduction

- *Inheritance (or derivation) is the process of creating new classes, called **derived classes**, from existing classes, called **base classes**.*
- *The derived class inherits all the properties from the base class and can add its own properties as well.*
- *The inherited properties may be hidden (if private in the base class) or visible (if public or protected in the base class) in the derived class.*
- *Inheritance uses the concept of code **reusability**.*

Introduction

- Once a base class is written and debugged, we can reuse the properties of the base class in other classes by using the concept of inheritance.
- Reusing existing code saves time and money and increases program's reliability.
- An important result of reusability is the ease of distributing classes.
- A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Defining Derived Class

- A derived class is specified by defining its relationship with the base class in addition to its own details.

- The general syntax of defining a derived class is as follows:

```
class derivedClassName : accessSpecifier baseClassName
```

{

```
.....;
```

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

}

Defining Derived Class

- The colon (:) indicates that the **derivedClassName** class is derived from the **baseClassName** class.
- The access specifier of the visibility mode is optional and, if present, may be public, private, or protected; By default it is private.
- Visibility mode describes the accessibility status of derived features.

Defining Derived Class

```
class xyz // base class
{
    // members of xyz
};

class ABC : public xyz // public derivation
{
    // members of ABC
};

class MNO : xyz // private derivation (by default)
{
    // members of ABC
};
```

Types of Inheritance

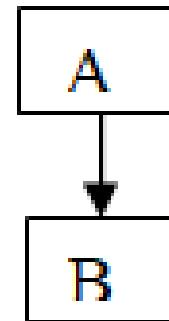
- A class can inherit properties from one or more classes and from one or more levels.
- On the basis of this concept, there are five types of inheritance.
 - Single inheritance
 - Multiple Inheritance
 - Hierarchical Inheritance
 - Multilevel Inheritance
 - Hybrid Inheritance

Single Inheritance

- In single inheritance, a class is derived from only one base class.
- For example:

```
class A
{
    members of A
};

class B : public A
{
    members of B
};
```



Single Inheritance

```
#include <iostream>
using namespace std;
class Vehicle           // base class
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle // sub class derived from a single base classes
{
    // member of class Car
};

int main()
{
    Car obj;           // creating object of sub class will invoke the constructor of base classes
    return 0;
}
```

Multiple Inheritance

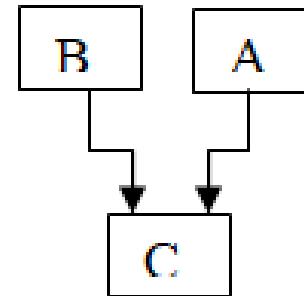
- In this inheritance, a class is derived from more than one base class.

- For example:

```
class A
{
    members of A
};

class B
{
    members of B
};

class C : public A, public B
{
    members of C
};
```



```
#include <iostream>
using namespace std;
class Vehicle // first base class
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler // second base class
{
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle" << endl;
    }
};
```

```
// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler
{
    // member of class Car
};

int main()
{
    Car obj;
    return 0;
}
```

Hierarchical Inheritance

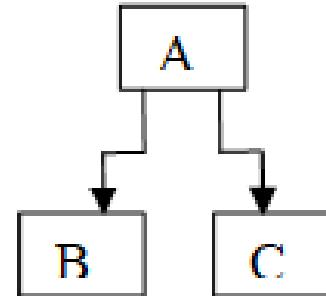
- In this type, two or more classes inherit the properties of one base class.

- For example:

```
class A
{
    members of A
};

class B : public A
{
    members of B
};

class C : public A
{
    members of C
};
```



Hierarchical Inheritance

```
#include <iostream> // second sub class
using namespace std;
class Vehicle // base class
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle // first sub class
{
    // members of class Car
};

class Bus: public Vehicle
{
    // members of class Bus
};

int main()
{
    Car obj1;
    Bus obj2;
    return 0;
}
```

Multilevel Inheritance

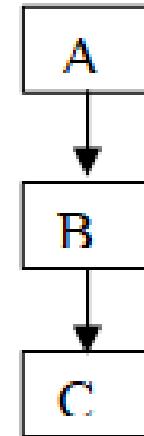
- The mechanism of deriving a class from another derived class is known as multilevel inheritance.
- The process can be extended to an arbitrary number of levels.

- For example:

```
class A
{
    members of A
};

class B : public A
{
    members of B
};

class C : public B
{
    members of C
};
```



Multilevel Inheritance

```
#include <iostream>
using namespace std;
class Vehicle // base class
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class fourWheeler: public Vehicle
{
public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles" << endl;
    }
};
```

```
class Car: public fourWheeler
{
public:
    Car()
    {
        cout << "Car has 4 Wheels" << endl;
    }
};

int main()
{
    Car obj;
    return 0;
}
```

Hybrid Inheritance

- This type of inheritance includes more than one type of inheritance mentioned previously.

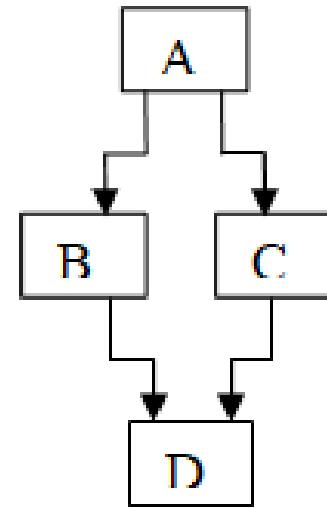
- For example:

```
class A
{
    members of A
};

class B : public A
{
    members of B
};

class C : public A
{
    members of C
};

class D : public B, public C
{
    members of D
};
```



```
#include <iostream>
using namespace std;
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
class Fare
{
public:
    Fare()
    {
        cout << "Fare of Vehicle\n";
    }
};
```

```
class Car: public Vehicle
{
    // member of class Car
};

class Bus: public Vehicle, public Fare
{
    // member of class Bus
};

int main()
{
    Car obj1;
    Bus obj2;
    return 0;
}
```

Protected Access Specifier

- If a class member is **private**, it can be accessed only from within the class where it lies.
- However, if a class member is **public**, it can be accessed from within the class and from outside the class.
- A **protected** member, on the other hand, can be accessed from within the class where it lies and from any class derived from this class. It can't be accessed from outside these classes.

Protected Access Specifier

| Access Specifier | Accessible from Own Class | Accessible from Derived Class | Accessible from Objects outside the class |
|------------------|---------------------------|-------------------------------|---|
| Public | Yes | Yes | Yes |
| Protected | Yes | Yes | No |
| Private | Yes | No | No |

NOTE: Only public and protected members are visible in the derived class.

Public, Protected, & Private Inheritance

- A derived class can be defined by specifying its relationship with the base class in addition to its own details.
- The general form is:

class derived-class-name : visibility-mode base-class-name

{

Members of derived classes

Public, Protected, & Private Inheritance

- *The visibility mode is optional.*
- *If present, may be private, protected, or public. The default visibility mode is private.*
- *For example,*
class ABC : [private] [protected] [public] XYZ
{
members of ABC
};

Public, Protected, & Private Inheritance

- *The visibility mode specifies the visibility of inherited members.*
- *If the visibility mode is **private***
 - *Public and protected members of the base class become private members in the derived class.*
 - *Therefore these members can only be accessed in the derived class.*
 - *They are inaccessible from outside this derived class.*

Public, Protected, & Private Inheritance

- *The visibility mode specifies the visibility of inherited members.*
- *If the visibility mode is **protected***
 - *Both public and protected members of the base class become protected members in the derived class.*
 - *Therefore these members can only be accessed in the derived class and from any class derived from this class.*
 - *They are inaccessible from outside these classes.*

Public, Protected, & Private Inheritance

- *The visibility mode specifies the visibility of inherited members.*
- *If the visibility mode is **public***
 - *Public and protected members of the base class do not change.*

Public, Protected, & Private Inheritance

| Base Class Visibility | Derived Class Visibility | | |
|-----------------------|--------------------------|----------------------|--------------------|
| | Public Derivation | Protected Derivation | Private Derivation |
| Private | Not visible | Not visible | Not visible |
| Protected | Protected | Protected | Private |
| Public | Public | Protected | Private |

NOTE: If we want to disallow the further inheritance of the members of base class from derived class, then private derivation is used.

Public, Protected, & Private Inheritance

/ C++ Implementation to show that a derived class doesn't inherit access to private data members. However, it does inherit a full parent object. */*

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};
```

```
class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A // 'private' is default
for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Public, Protected, & Private Inheritance

```
class Base
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class Derived : public Base
{
    public:
        void getdata()
        {
            cout<<"Enter x, y, and z:"<<endl;
            cin>>x;
            cin>>y;
            cin>>z;
        }
};
```

```
int main()
{
    Derived d;
    d.getdata();
    cout<<"x= "<<d.x<<endl;
    cout<<"y= "<<d.y<<endl;
    cout<<"z= "<<d.z<<endl;
    return 0;
}
```

Derived Class Constructors

- When applying inheritance, we usually create objects using the derived class.
- If the base class contains a constructor it can be called from the initializer list in the derived class constructor.

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

class A
{
protected:
    int adata;
public:
    A(int a)
    {
        adata = a;
    }
};
```

```
class B : public A
{
    int bdata;
public:
    B(int a, int b) : A(a)
    {
        bdata = b;
    }
    void showdata()
    {
        cout<<"adata = "<<adata<<endl <<"bdata =
        "<<bdata;
    }
};

int main()
{
    B b(5, 6);
    b.showdata();
    return 0;
}
```

Derived Class Constructors

```
class A
{
protected:
    int adata;
};

class B : public A
{
public:
    B(int a, int b)
    {
        adata = a;
        bdata = b;
    }
};

void showdata()
{
    cout<<"adata = "<<adata<<endl
    <<"bdata = "<<bdata;
};

int main()
{
    B b(5, 6);
    b.showdata();
    return 0;
}
```

Order of Execution of Constructors

- *The base class constructor is executed first and then the constructor in the derived class is executed.*
- *In case of multiple inheritance, the base class constructors are executed in the order in which they appear in the definition of the derived class.*
- *Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.*

Order of Execution of Constructors

- Furthermore, the constructors for virtual base classes are invoked before any non-virtual base classes.
- If there are multiple virtual base classes, they are invoked in the order in which they are declared in the derived class.

```
class A
{
public:
    A()
    {
        cout<<"Class A Constructor"<<endl;
    }
};

class B:public A
{
public:
    B()
    {
        cout<<"Class B Constructor"<<endl;
    }
};
```

```
class C: public B
{
public:
    C()
    {
        cout<<"Class C Constructor"<<endl;
    }
};

int main()
{
    C c;
    return 0;
}
```

Order of Execution of Destructors

- *The derived class destructor is executed first and then the destructor in the base class is executed.*
- *In case of multiple inheritances, the derived class destructors are executed in the order in which they appear in the definition of the derived class.*
- *Similarly, in a multilevel inheritance, the destructors will be executed in the order of inheritance.*

```
class A
{
public:
    ~A()
    {
        cout<<"Class A Destructor"<<endl;
    }
};

class B:public A
{
public:
    ~B()
    {
        cout<<"Class B Destructor"<<endl;
    }
};
```

```
class C: public B
{
public:
    ~C()
    {
        cout<<"Class C Destructor"<<endl;
    }
};

int main()
{
    C c;
    return 0;
}
```

Overriding Member Function

- *We can use member functions in a derived class that override those in the base class.*
- *In this case, both base and derived class functions have same name, same number of parameters, and similar type of parameters.*

```
class A
{
public:
void show()
{
    cout<<"This is class A";
}
};

class B : public A
{
public:
void show()
{
    cout<<"This is class B"<<endl;
}
};
```

```
int main()
{
    B b;
    b.show(); // invokes the member function
              // from class B
    b.A :: show(); // invokes the member function
                    // from class A
    return 0;
}
```

Ambiguities in Inheritance

- Suppose two base classes have an exactly similar member.
- Also, suppose a class derived from both of these base classes do not have this member.
- Then, if we try to access this member from the objects of the derived class, it will be ambiguous.

```
class A
{
public:
void show()
{
    cout<<"This is class A"<<endl;
}
};

class B
{
public:
void show()
{
    cout<<"This is class B"<<endl;
}
};
```

```
class C : public A, public B
{
    // member body
};

int main()
{
    C c;
    c.show();
    // ambiguous – will not compile
    c.A :: show(); // will compile
    c.B :: show(); // will compile
    return 0;
}
```

```
class A
{
public:
void show()
{
    cout<<"This is class A"<<endl;
}
};

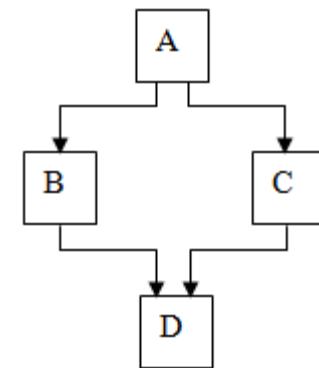
class B
{
public:
void show()
{
    cout<<"This is class B"<<endl;
}
};
```

```
class C : public A, public B
{
public:
void show()
{
    A :: show();
    B :: show();
}
};

int main()
{
C c;
c.show();           // will compile
c.A :: show();     // will compile
c.B :: show();     // will compile
return 0;
}
```

Virtual Base Classes

- Consider a situation where we derive a class **D** from two classes **B** and **C** that are each derived from the same class **A**.
- This creates a diamond-shaped inheritance tree(called hybrid multipath inheritance) and all the public and protected members from class **A** inherited into class **D** twice once through the path $A \rightarrow B \rightarrow D$ and gain through the path $A \rightarrow C \rightarrow D$.
- This causes ambiguity and should be avoided.



Virtual Base Classes

- We can remove this kind of ambiguity by using the concept of virtual base class.
- For this we make direct base classes (B and C) virtual base classes as follows:

class A

{ };

class B : virtual public A

{ };

class C : public virtual A

{ };

class D : public B, public C

{ };

- In this case, class D inherits only one copy from the classes C and D.
- Note: The keywords **virtual** and **public** may be used in either order.

```
#include<iostream>
using namespace std;
class A
{
protected:
    int adata;
};
class B : public A
{
    // data members
};
class C : public A
{
    // data members
};
```

```
class D : public B, public C
{
public:
    D(int a)
    {
        adata = a;
    }
    int getdata()
    {
        return adata;
    }
    int main()
    {
        D d(5);
        cout << d.getdata();
        return 0;
    }
}
```

```
#include<iostream>
using namespace std;
class A
{
protected:
    int adata;
};
class B : virtual public A
{
    //data members
};
class C : public virtual A
{
    //data members
};
```

```
class D : public B, public C
{
public:
    D(int a)
    {
        adata = a;
    }
    int getdata()
    {
        return adata;
    }
    int main()
    {
        D d(5);
        cout<<d.getdata();
        return 0;
    }
}
```

Containership (Aggregation)

- *Inheritance is often called a “kind of” relationship.*
- *In inheritance, if a class B is derived from a class A , we can say “ B is a kind of A ”.*
- *This is because B has all the characteristics of A , and in addition some of its own.*
- *For example, we can say that bulldog is a kind of dog: A bulldog has the characteristics shared by all dogs but has some distinctive characteristics of its own.*

Containership (Aggregation)

- *There is another type of relationship, called a “has a” relationship, or containership.*
- *We say that a bulldog has a large head, meaning that each bulldog includes an instance of a large head.*
- *In object oriented programming, has a relationship occurs when one object is contained in another.*

```
#include<iostream>
using namespace std;
class Employee
{
    int eid, sal;
public:
void getdata()
{
    cout<<"Enter id and salary of
employee"<<endl;
    cin>>eid>>sal;
}
void display()
{
    cout<<"Emp
ID:"<<eid<<endl<<"Salary:"<<sal<<en
dl;
}
};
```

```
class Company
{
    int cid;
    char cname[20];
    Employee e;
public:
void getdata()
{
    cout<<"Enter id and name of the company:"<<endl;
    cin>>cid>>cname;
    e.getdata();
}
void display()
{
    cout<<"Comp ID:"<<cid<<endl<<"Comp Name:"<<cname<<endl;
    e.display();
}
int main()
{
    Company c;
    c.getdata();
    c.display();
    return 0;
}
```

Local Classes

- Classes can also be defined and used inside a function or a block.
- Such classes are called local classes. For example,

```
void test (int a)
{
    .....
    class A
    {
        .....
    };
    .....
    A a; //create object of type A
    .....
}
```

- Local classes can use global variables and static variables declared inside the function but cannot use automatic local variables.
- The global variables should be used with the scope resolution operator,

Local Classes

```
#include <iostream>
using namespace std;
void fun()
{
    class Test // local to fun
    {
        public:
        void method()
        {
            cout << "Local Class method() called";
        }
    };
    Test t;
    t.method();
}
int main()
{
    fun();
    return 0;
}
```

Abstract Class

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation.
- Such a class is called as abstract class.
- An abstract class is a class which contains at least one pure virtual function in it.
- Abstract classes are used to provide an interface for its sub classes.
- Classes inheriting an abstract class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Abstract Class

- *For example, let `SHAPE` be a base class.*
- *We cannot provide implementation of function `DRAW()` in `SHAPE`, but we know every derived class must have implementation of `DRAW()`.*
- *We cannot create an object of abstract classes.*

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0; // Pure
    Virtual Function
};

class Derived : public Base
{
public:
    void show()
    {
        cout << "Implementation of Virtual
        Function in Derived class";
    }
};
```

```
int main()
{
    Base obj; // Compile Time Error
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

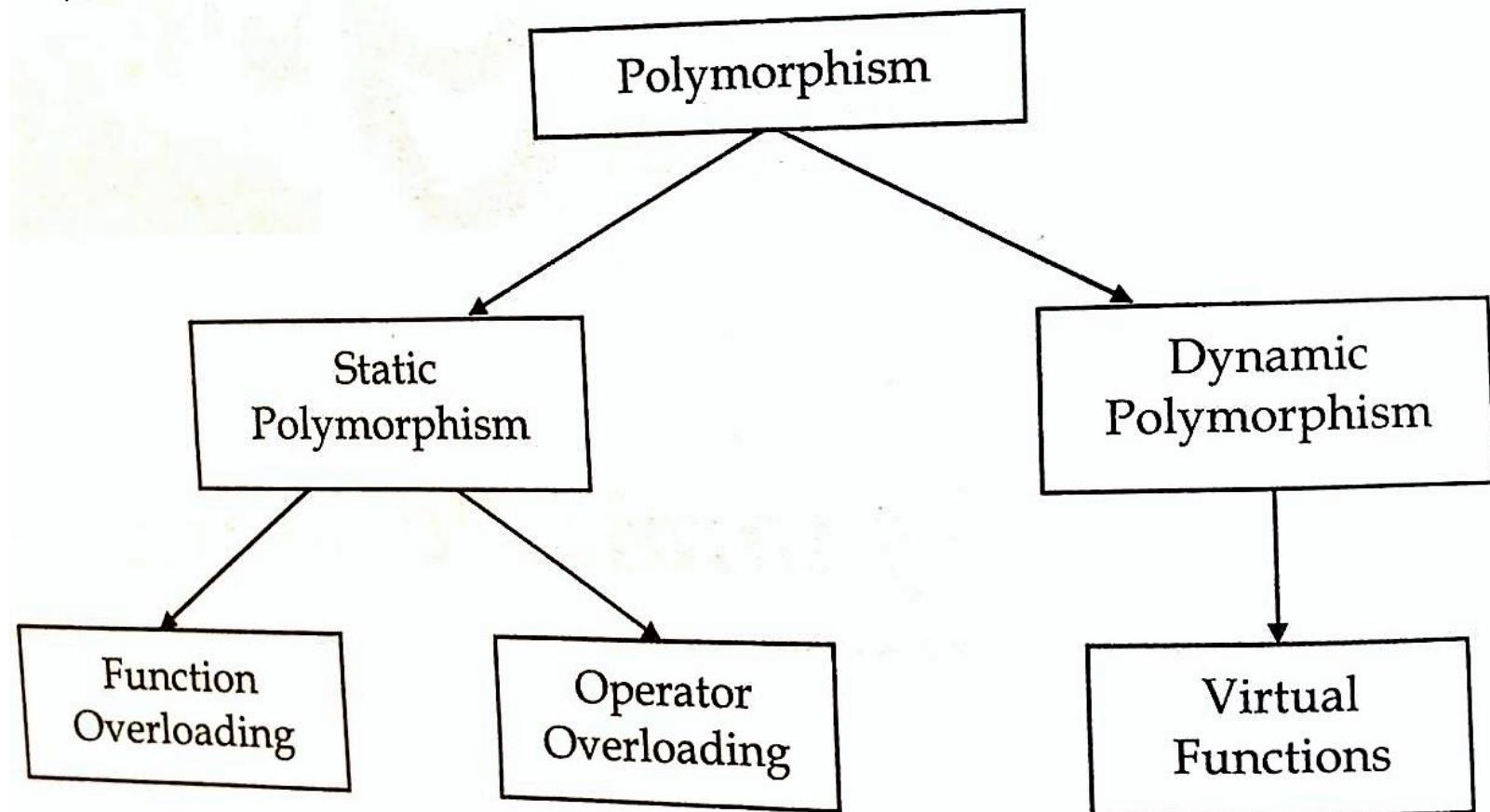
END OF UNIT FIVE

Syllabus

- ***UNIT 6:Virtual Function, Polymorphism, and miscellaneous C++ Features***
 - *Concept of Virtual Functions*
 - *Difference between normal member function accessed with pointers and virtual member function accessed with pointers*
 - *Late Binding, Abstract Class, and Pure Virtual Functions*
 - *Virtual Destructors, Virtual Base Class*
 - *Friend Function, Friend Class*
 - *Static Function*
 - *Assignment and Copy Initialization, Copy Constructor*
 - *This Pointer*
 - *Concrete Classes (vs. Abstract Class)*
 - *Polymorphism and its roles*

UNIT 6

***VIRTUAL FUNCTION, POLYMORPHISM, &
MISCELLANEOUS C++ FEATURES***



Polymorphism

- *Polymorphism means state of having many forms.*
- *We have already seen that polymorphism is implemented using the concept of overloaded functions and operators.*
- *In this case, polymorphism is called **early binding** or **static binding** or **static linking**.*
- *This is also called **compile time polymorphism** because the compiler knows the information needed to call the function at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself.*

Polymorphism

- *There is also another kind of polymorphism called **run time polymorphism**.*
- *In this type, the selection of appropriate function is done dynamically at run time.*
- *So, this is also called **late binding** or **dynamic binding**.*
- *C++ supports a mechanism known as **virtual functions** to achieve run time polymorphism.*
- *Run time polymorphism also requires the use of pointers to objects.*

Virtual Functions

- *Virtual means existing in appearance but not in reality.*
- *When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class.*
- *Furthermore, when we use virtual functions, different functions can be executed by the same function call.*
- *The information regarding which function to invoke is determined at run time.*
- *We should use virtual functions and pointers to objects to achieve run time polymorphism.*

Virtual Functions

- For this, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes.
- The function in the base class is declared as virtual using the keyword `virtual`.
- When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer.

```
class Base
{
public:
    virtual void print()
    {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base
{
public:
    void print()
    {
        cout << "Derived Function" << endl;
    }
};
```

```
int main()
{
    Derived derived1;
    // pointer of Base type that points to derived1
    Base* base1 = &derived1;
    // calls member function of Derived class
    base1->print();
    return 0;
}
```

Abstract Classes and Pure Virtual Function

- An abstract class is one that is not used to create objects.
- It is used only to act as a base class to be inherited by other classes.
- We can make abstract classes by placing at least one pure virtual function in the base class.
- A pure virtual function is one with the expression = 0 added to the declaration — that is, a function declared in a base class that has no definition relative to the base class.
- In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function.

Abstract Classes and Pure Virtual Function

```
class A
{
protected:
int data;
public:
A(int d)
{
data = d;
}
virtual void show() = 0;
};

class B : public A
{
public:
B(int d) : A(d)
{
}
void show()
{
cout<<data<<endl;
}
};
```

```
class C : public A
{
public:
C(int d) : A(d)
{
}
void show()
{
cout<<data;
}
};

int main()
{
A *a;
B b(5);
C c(6);
a = &b;
a->show();
a = &c;
a->show();
return 0;
}
```

Friend Function

- *The concept of encapsulation and data hiding dictate that non-member functions should not be allowed to access an object's private and protected members.*
- *This policy is, if you are not a member you cannot get it.*
- *Sometimes this feature leads to considerable inconvenience in programming.*
- *If we want to use a function to operate on objects of two different classes, then a function outside a class should be allowed to access and manipulate the private members of the class.*

Friend Function

- In C++, this is achieved by using the concept of friend function.
- Private member of a class cannot be accessed from outside the class.
- Non member function of a class cannot access the member of a class.
- But using friend function we can achieve this.

Friend Function

- *The function declaration must be prefixed by the keyword friend whereas the function definition must not.*
- *The function could be defined anywhere in the program similar to any normal C++ function.*
- *Function definition does not use either the keyword friend or scope resolution operator ::*
- *A friend function is not a member of any classes but has the full access to the member of class within which it is declared as friend.*

Friend Function

Class test

{

.....

.....

.....

Public:

.....

.....

friend void friendfunc(); // declaration

};

Friend Function

- A friend function has the following characteristics.
 - It is not in the scope of the class within which it has been declared as friend.
 - Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a normal C++ function without the help of any object.
 - Unlike member functions, it cannot access member name directly and has to use an object name and dot operator with each member name. i.e. `t1.x`.
 - It can be declared as public or private part of the class, the meaning is same.
 - Normally, it takes objects as arguments.

```
class sample
{
    private:
        int a;
        int b;
    friend float mean(sample s);
public:
    void setvalue()
    {
        a = 25;
        b = 40;
    }
};
```

```
float mean(sample s)
{
    return float(s.a + s.b)/2;
}

int main()
{
    sample x;
    x.setvalue();
    cout<<"Mean      value      = "
         <<mean(x);
    return 0;
}
```

Friend Classes

- *The member functions of a class can all be made friends of another class when we make the former entire class a friend of later.*
- *For example, if in a class **alpha** the entire class **beta** is declared as a **friend**, then all the members of **beta** can access the private data of **alpha**.*

```
#include<iostream>
using namespace std;
class alpha
{
private:
    int x;
public:
    void setdata(int d)
    {
        x = d;
    }
    friend class beta;
};
```

```
class beta
{
public:
    void func(alpha a)
    {
        cout<<a.x<<endl;
    }
};

int main()
{
    alpha a;
    a.setdata(99);
    beta b;
    b.func(a);
    return 0;
}
```

Virtual Destructors

- Since destructor are member functions, they can be made virtual with placing keyword `virtual` before it.
- The syntax is
`virtual ~classname () ; // virtual destructor.`
- The destructor in base class should always be virtual.
- If we use `delete` with a base class object to destroy the derived class object, then it calls the member function destructor for base class.
- This causes the base class object to be destroyed.
- Hence making destructor of base class virtual, we can prevent such disoperation.

```
class Base
{
public:
    ~Base()           // non virtual
    //virtual ~Base()
    {
        cout<<"Base Destroyed";
    }
};

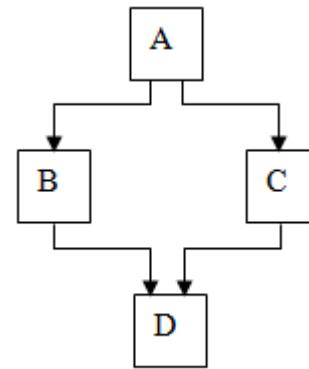
class Derv1:public Base
{
public:
    ~Derv1()
    {
        cout<<"Derived1 destroyed\n";
    }
};
```

```
/* class Derv2:public Base
{
public:
    ~Derv2()
    {
        cout<<"Derived2 destroyed\n";
    }
}; */

int main()
{
    Base * pBase = new Derv1();
    delete pBase;
    return 0;
}
```

Virtual Base Class

- In multiple inheritance, if a base class parent derives its two child class then another class is derived from two child.



- When member function of class D want to access data member of parent class A, then problem arises due to ambiguity.
- To resolve such ambiguity we use virtual base class.

Virtual Base Class

- A virtual base class is one from which classes are derived virtually as

```
class A
{
    // body of class A
};

class B: virtual public A
{
    // Body of B
};

class C:virtual public A
{
    // Body of class C
};

class D: public B; public C.
{
    // Body of class D
};
```

```
#include <iostream>
using namespace std;
class student
{
protected:
int roll;
public:
void getno(int a)
{
    roll=a;
}
void putno()
{
    cout<<"\nRollNumber is:"<<roll;
}
};
```

```
class test:virtual public student
{
protected:
float part1,part2;
public:
void getmark(float a,float b)
{
    part1=a;
    part2=b;
}
void putmark()
{
    cout<<"\nPart1="<<part1;
    cout<<"\nPart2="<<part2;
}
};
```

```
class sport:virtual public student
{
protected:
int score;
public:
void getscore(int a)
{
    score=a;
}
void putscore()
{
    cout<<"\nScore:"<<score;
}
};
```

```
class result:public test, public sport
{
float total;
public:
void display()
{
    total=part1+part2+score;
    putno();
    putmark();
    putscore();
    cout<<"\nTotal Score:"<<total;
}
};
```

Virtual Base Class

```
int main()
{
    result student1;
    student1.getno(999);
    student1.getmark(25,54);
    student1.getscore(7);
    student1.display();
    return 0;
}
```

```
#include<iostream>
using namespace std;
class Shape
{
protected:
float p,l,b;
public:
void setvalue(int x, int y)
{
    l=x;
    b=y;
}
};
```

```
class Square:public Shape
{
public:
void find_perimeter()
{
    p=4*l;
    cout<<"Perimeter = "<<p<<endl;
}
int main()
{
    Shape *bp;
    Square s;
    bp=&s;
    bp->setvalue(5,2);
    s.find_perimeter();
    return 0;
}
```

Virtual Functions (Use of Pointer)

```
class Base
{
public:
virtual void print()
{
    cout << "Base Function" << endl;
}
};

class Derived : public Base
{
public:
void print()
{
    cout << "Derived Function" << endl;
}
};
```

```
int main()
{
    Derived derived1;
    // pointer of Base type that points to derived1
    Base* base1 = &derived1;
    // calls member function of Derived class
    base1->print();
    return 0;
}
```

Normal Member Function (Use of Pointer)

```
class A
{
public:
void show()
{
    cout<<"This is class A"<<endl;
}
};

class B:public A
{
public:
void show()
{
    cout<<"This is class B"<<endl;
}
};

class C:public A
{
public:
void show()
{
    cout<<"This is class C"<<endl;
}
};
```

```
int main()
{
A *p,a;
B b;
C c;
p=&b;
p->show();
b.show();
p=&c;
p->show();
c.show();
p=&a;
p->show();
a.show();
return 0;
}
```

Abstract Class & Concrete Class

- An abstract class is a class containing at least one pure virtual function.
- They cannot be instantiated into an object directly.
- However, we can create pointers and references to an abstract class.
- Abstract classes are designed to be specifically used as base classes.
- Only a subclass of an abstract class can be instantiated directly if all inherited pure virtual methods have been implemented by that class.

Abstract Class & Concrete Class

```
class ABC
{
public:
    virtual Fun1()=0;
    virtual Fun2()=0;
};
```

- With above abstract class, we cannot create it's object but we can create pointer to ABC class as below:

```
ABC a;           // invalid
ABC *p;          // valid
```

Abstract Class & Concrete Class

- *A concrete class is a class that can be used to create an object.*
- *It has no pure virtual functions.*
- *Concrete class can be used to create an object.*
- *You must extend an abstract class and make a concrete class to be able to then create an object.*

Abstract Class & Concrete Class

```
#include<iostream>
using namespace std;
class Polygon // abstract class
{
protected:
int width, height;
public:
void setvalues(int a,int b)
{
    width=a;
    height=b;
}
virtual int area(void)=0;
};
class Rectangle:public Polygon // concrete class
{
public:
int area(void)
{
    return (width*height);
}
};
```

```
class Triangle:public Polygon // concrete class
{
public:
int area(void)
{
    return(width*height/2);
}
};

int main()
{
    Rectangle rect;
    Triangle trgl;
    Polygon *poly=&rect;
    poly->setvalues(4,5);
    cout<<"Area of Rectangle = "<<poly->area()<<endl;
    poly=&trgl;
    poly->setvalues(6,5);
    cout<<"Area of Triangle = "<<poly->area()<<endl;
    return 0;
}
```

Static Function Members

- *By declaring a function member as static, you make it independent of any particular object of the class.*
- *A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.*
- *A static member function can only access static data member, other static member functions and any other functions from outside the class.*
- *You could use a static member function to determine whether some objects of the class have been created or not.*

```
class Box
{
public:
    static int objectCount;
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double
        h = 2.0)
    {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
        // Increase every time object is created
        objectCount++;
    }
}
```

```
double Volume()
{
    return length * breadth * height;
}

static int getCount()
{
    return objectCount;
}

private:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};
```

Static Function Members

```
int Box::objectCount = 0; // Initialize static member of class Box
int main(void)
{
    // Print total number of objects before creating object.
    cout << "Initial Stage Count: " << Box::getCount() << endl;

    Box Box1(3.3, 1.2, 1.5); // Declare box1
    Box Box2(8.5, 6.0, 2.0); // Declare box2

    // Print total number of objects after creating object.
    cout << "Final Stage Count: " << Box::getCount() << endl;
    return 0;
}
```

Constructors

- *A constructor is a special member function that is executed automatically whenever an object is created.*
- *It is used for automatic initialization.*
- *Automatic initialization is the process of initializing object's data members when it is first created, without making a separate call to a member function.*
- *The name of the constructor is same as the class name.*
- *There are three types of constructor:*
 - *Default Constructor*
 - *Parameterized Constructor*
 - *Copy Constructor*

Copy Constructor

- A copy constructor is used to declare and initialize an object with another object of the same type.
- For example, the statement **rectangle r2(r1);** creates new object **r2** and performs member-by-member copy of **r1** into **r2**.
- Another form of this statement is **rectangle r2 = r1;**
- The process of initializing through assignment operator is known as copy initialization.
- A copy constructor takes reference to an object of the same class as its argument.
- We cannot pass the argument by value to a copy constructor.

Copy Constructor

```
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle(rectangle &r)      // Copy Constructor
        {
            length = r.length;
            breadth = r.breadth;
        }
        .......;
        .......;
};
```

Example of Constructors

```
class Item
{
    int code, price;
public:
    Item()      //Default Constructor
    {
        code= price =0;
    }
    Item(int c,int p)          //Parameterized Constructor
    {
        code=c;
        price=p;
    }
    Item(Item &x)            //Copy Constructor
    {
        code=x.code;
        price= x.price;
    }
    void display()
    {
        cout<<"Code:: "<<code<<endl<<"Price:: "<<price<<endl;
    }
};
```

```
int main()
{
    Item I1;
    Item I2(102,300);
    Item I3(I2);
    I1.display();
    I2.display();
    I3.display();
    return 0;
}
```

this Pointer

- Every object in C++ has access to its own address through an important pointer called **this** pointer.
- The **this** pointer is an implicit parameter to all member functions.
- Therefore, inside a member function, this may be used to refer to the invoking object.
- Friend functions do not have a **this** pointer, because friends are not members of a class.
- Only member functions have a **this** pointer.

this Pointer

- In C++, **this** pointer is used to represent the address of an object inside a member function.
- For example, consider an object **obj** calling one of its member function say **method()** as **obj.method()**.
- Then, **this** pointer will hold the address of object **obj** inside the member function **method()**.

- i. This pointer stores the address of calling object it means if we print this then we get the address of the calling obj.
- ii. It refer to the current class instance variable. Where instance variables are kind of those variables which are declare in the class but outside of the constructor or block.

```
class sample
{
    int a,b;
public:
    void input(int a,int b)
    {
        this->a=a+b;
        this->b=a-b;
    }
    void output()
    {
        cout<<"a = "<<a<<endl<<"b = "<<b;
    }
};
```

```
int main()
{
    sample x;
    x.input(5,8);
    x.output();
    return 0;
}
```

Run-time vs. Compile-time

- Run time is the time period where the executable code is running **WHEREAS** compile time is the time period where the code typed is converted into executable code.
- In run time, errors can be detected only after the execution of the program **WHEREAS** in compile time, errors are detected before the execution of the program
- In run time, errors that occur during the execution of a program are called run time errors which aren't detected by the compiler **WHEREAS** errors that occur during compile time are detected by the compiler and either are semantics error or syntax error.

Real Time Examples of Polymorphism

- *A person can have various roles and responsibilities at the same time.*
- *A woman plays multiple roles in her life such as a mother, wife, daughter, daughter in law, sister, etc.*
- *A man behaves as an employee in an office, son or husband at home, customer at a mall, etc.*
- *A mobile is one device but offers various features such as camera, radio, etc.*

SUMMARY

- *Polymorphism in C++*
- *Types of Polymorphism – Compile Time & Run Time*
- *Compile Time Polymorphism*
 - *Function Overloading*
 - *Operator Overloading*
- *Run Time Polymorphism*
 - *Function Overriding*
 - *Virtual Function*
- *Pure Virtual Function*
- *Difference between Run Time & Compile Time*
- *Real Time Example of Polymorphism in C++*

END OF UNIT SIX

Syllabus

- ***UNIT 7: Function Templates and Exception Handling***
 - *Function Templates*
 - *Function Templates with Multiple Arguments*
 - *Class Templates*
 - *Templates and Inheritance*
 - *Exceptional Handling (Try, Throw, & Catch)*
 - *Multiple exceptions, Exceptions with Arguments*
 - *Use of Exceptional Handling*

UNIT 7

**FUNCTION TEMPLATES &
EXCEPTION HANDLING**

Function Templates

- Function templates are special functions that can operate with generic types.
- This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.
- In C++ this can be achieved using template parameters.
- A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function.
- These function templates can use these parameters as if they were any other regular type.

Function Templates

- The format for declaring function templates with type parameters is:
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
- The only difference between both prototypes is the use of either the keyword **class** or the keyword **typename**.
- Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

Function Templates

- For example, to create a template function that returns the greater one of two objects we could use:

template <class Type>

Type GetMax (Type a, Type b)

{

return (a>b?a:b);

}

- To use this function template we use the following format for the function call:

function_name <type> (parameters);

Function Templates

- For example, to call *GetMax* to compare two integer values of type *int* we can write:

int x,y;

GetMax <int> (x,y);

- When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of *myType* by the type passed as the actual template parameter (*int* in this case) and then calls it.
- This process is automatically performed by the compiler and is invisible to the programmer.

```
#include <iostream>
using namespace std;
template <class T>
T GetMax (T a, T b)
{
    T result;
    result = (a>b)? a : b;
    return (result);
}
```

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

Function Templates

- In the previous example we used the function template ***GetMax()*** twice.
- The first time with arguments of type ***int*** and the second one with arguments of type ***long***. The compiler has instantiated and then called each time the appropriate version of the function.
- As you can see, the type ***T*** is used within the ***GetMax()*** template function even to declare new objects of that type:
T result;
- Therefore, ***result*** will be an object of the same type as the parameters ***a*** and ***b*** when the function template is instantiated with a specific type.

Function Templates

- We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets.

- For example:

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

- In this case, our function template **GetMin()** accepts two parameters of different types and returns an object of the same type as the first parameter (**T**) that is passed.
- For example, after that declaration we could call **GetMin()** with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

Function Templates (with arguments)

```
#include <iostream>
using namespace std;
template <class T, class U>
T GetMax (T a, U b)
{
    T result;
    result = (a>b)? a : b;
    return (result);
}

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

Class Templates

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

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

Class Templates

- The class that we have just defined serves to store two elements of any valid type.
- For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

mypair<int> myobject (115, 36);

- This same class would also be used to create an object to store any other type:

mypair<double> myfloats (3.0, 2.18);

```
template <class T>
class mypair
{
    T a, b;
public:
    mypair (T first, T second)
    {
        a=first; b=second;
    }
    T getmax ()
    {
        T retval;
        retval = a>b? a : b;
        return retval;
    }
};
```

```
int main ( )
{
    mypair <int> myobject (100,
    125);
    cout << myobject.getmax();
    return 0;
}
```

Exceptions

- *Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing.*
- *Exceptions might include conditions such as division by zero, access to an array outside of its bounds, running out of memory or disk space, not being able to open a file, trying to initialize an object to an impossible value etc.*
- *When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively.*
- *C++ provides built-in language features to detect and handle exceptions, which are basically runtime errors.*

Exceptions

- *The purpose of the exception handling mechanism is to provide means to detect and report an “exceptional circumstance” so that appropriate action can be taken.*
- *The mechanism suggests a separate error handling code that performs the following tasks:*
 - *Find the problem (**Hit the exception**).*
 - *Inform that an error has occurred (**Throw the exception**).*
 - *Receive the error information (**Catch the exception**).*
 - *Take corrective action (**Handle the exception**).*
- *The error handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.*

Exception Handling Mechanism

- *Exception handling mechanism in C++ is basically built upon three keywords:*
 - *try*
 - *throw*
 - *catch*
- *The keyword **try** is used to surround a block of statements, which may generate exceptions.*
- *This block of statements is known as try block.*

Exception Handling Mechanism

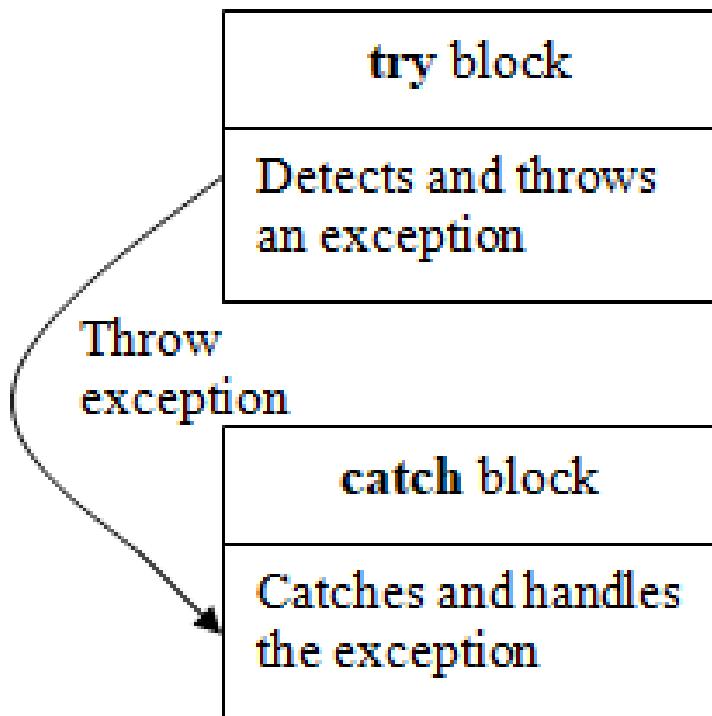
- *Exception handling mechanism in C++ is basically built upon three keywords:*
 - *try*
 - *throw*
 - *catch*
- *When an exception is detected, it is thrown using the **throw** statement situated either in the try block or in functions that are invoked from within the try block.*
- *This is called throwing an exception and the point at which the **throw** is executed is called the throw point.*

Exception Handling Mechanism

- *Exception handling mechanism in C++ is basically built upon three keywords:*
 - *try*
 - *throw*
 - *catch*
- *A catch block defined by the keyword **catch** catches the exception thrown by the **throw** statement and handles it appropriately.*
- *This block is also called exception handler.*
- *The catch block that catches an exception must immediately follow the try block that throws an exception.*

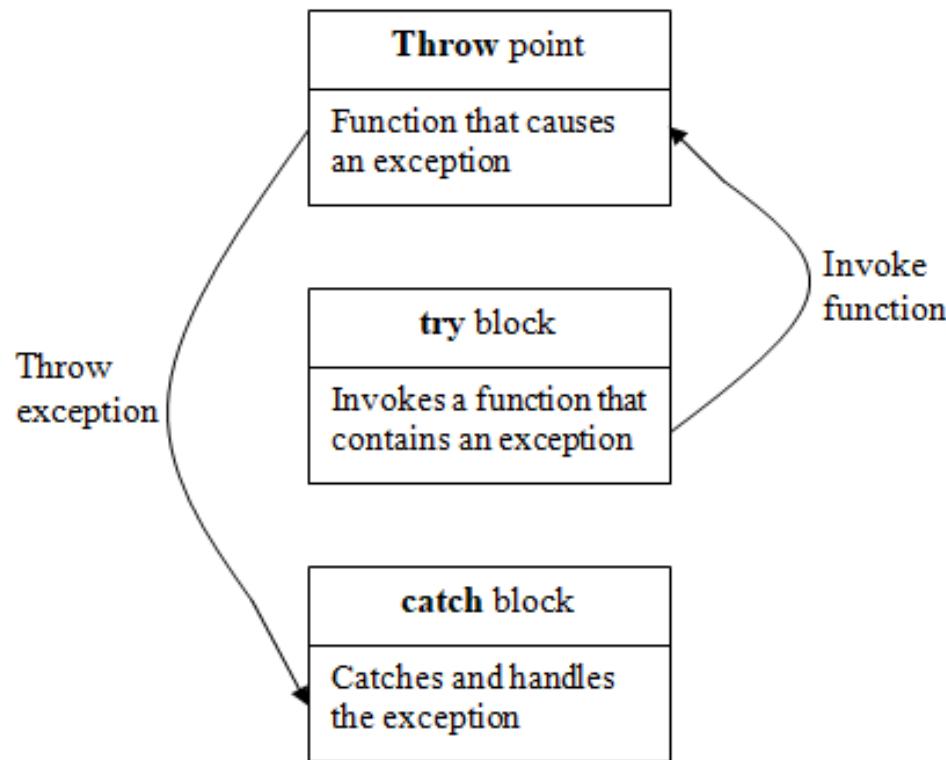
Exception Handling Mechanism

- The figure below shows the exception handling mechanism if try block throws an exception.



Exception Handling Mechanism

- The figure below shows the exception handling mechanism if function invoked by try block throws an exception.



Exception Handling - 1

```
// Try block throwing exception
#include<iostream>
using namespace std;
int main()
{
    int a, b;
    cout<<"Enter values of a & b:\n";
    cin>>a>>b;
    try
    {
        if(b == 0)
            throw b;
        else
            cout<<"Result = "<<(float)a/b;
    }
    catch(int i)
    {
        cout<<"Divide by zero exception: b = "<<i;
    }
    cout<<endl<<"END";
    return 0;
}
```

Here if exception occurs in the try block, it is thrown and the program control leaves in the try block and enters the catch block.

Exception Handling – 2

```
//Function invoked by try block throwing exception
#include<iostream>
using namespace std;
void divide(int a, int b)
{
    if(b == 0)
        throw b;
    else
        cout<<"Result="<<(float)a/b;
}
int main()
{
    int a, b;
    cout<<"Enter values of a & b:\n";
    cin>>a>>b;
    try
    {
        divide(a, b);
    }
    catch(int i)
    {
        cout<<"Divide by zero exception: b = "<<i;
    }
    cout<<endl<<"END";
    return 0;
}
```

Here try block invokes the function divide.

If exception occurs in this function, it is thrown and control leaves this function and enters the catch block.

Exception Handling Mechanism

- Note that, exceptions are used to transmit information about the problem.
- If the type of exception thrown matches the argument type in the **catch** statement, then only catch block is executed for handling the exception.
- After that, control goes to the statement immediately after the catch block.
- If they do not match, the program is aborted with the help of **abort()** function, which is invoked by default.
- In this case, statements following the catch block are not executed.
- When no exception is detected and thrown, the control goes to the statement immediately after the catch block skipping the catch block.

Throwing Mechanism

- When an exception is detected, it is thrown using the **throw** statement in one of the following forms:
`throw(exception);`
`throw; // used for re-throwing an exception (discussed later)`
- The operand exception may be of any type (built-in or user-defined), including constants.
- When an exception is thrown, the catch statement associated with the try block will catch it.
- That is, the control exits the current try block, and is transferred to the catch block after the try block.
- Throw point can be in a deeply nested scope within a try block or in a deeply nested function call.
- In any case, control is transferred to the catch statement.

Catching Mechanism

- *Code for handling exceptions is included in catch blocks.*
- *The general form of catch block is:*
catch(type arg)
{
//Body of catch block
}
- *The type indicates the type of exception that catch block handles.*
- *The parameter arg is optional.*
- *If it is named, it can be used in the exception handling code.*

Catching Mechanism

- *The catch statement catches an exception whose type matches with the type of catch argument.*
- *When it is caught, the code in the catch block is executed.*
- *After its execution, the control goes to the statement immediately following the catch block.*
- *If an exception is not caught, abnormal program termination will occur.*
- *The catch block is simply skipped if the catch statement does not catch an exception.*

```
try
{
    // Try block
}
catch(type1 arg)
{
    // Catch block1
}
catch(type2 arg)
{
    // Catch block 2
}
.....
.....
}
catch(typeN arg)
{
    // Catch block N
}
```

- It is possible that a program segment has more than one condition to throw an exception.
- In such cases, we can associate more than one **catch** statement with a **try**.
- When an exception is thrown, the exception handlers are searched in order for an appropriate match.
- The first handler that yields a match is executed.
- After that, the control goes to the first statement after the last **catch** block for that **try** skipping other exception handlers.
- When no match is found, the program is terminated.
- **Note:** It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

```
void test(int x)
{
    try
    {
        if(x == 0) throw x;
        if(x == 1) throw 1.0;
    }
    catch(int m)
    {
        cout<<"Caught an integer"<<endl;
    }
    catch(double d)
    {
        cout<<"Caught a double";
    }
}
```

```
int main()
{
    test(0);
    test(1);
    test(2);
    return 0;
}
```

Catch All Exception

- If we want to catch all possible types of exceptions in a single **catch** block, we use **catch** in the following way:

```
catch(...)  
{  
    // Statements for processing all exceptions  
}
```

Catch All Exception

```
void test(int x)
{
    try
    {
        if(x == 0) throw x;
        if(x == 1) throw 1.0;
    }
    catch(...)
    {
        cout<<"Caught an exception";
    }
}

int main()
{
    test(0);
    test(1);
    test(2);
    return 0;
}
```

Re-throwing an Exception

- A handler may decide to rethrow the exception caught without processing it.
- In such situations, we may simply invoke `throw` without any arguments as shown below:
`throw;`
- This causes the current exception to be thrown to the next enclosing **try/catch** sequence and is caught by a **catch** statement listed after that enclosing **try** block.

```
#include<iostream>
using namespace std;
void divide(int a, int b)
{
    try
    {
        if(b == 0)
            throw b;
        else
            cout<<"Result = "<<(float)a/b;
    }
    catch(int)
    {
        throw;
    }
}
```

```
int main()
{
    int a, b;
    cout<<"Enter values of a & b: "<<endl;
    cin>>a>>b;
    try
    {
        divide(a, b);
    }
    catch(int i)
    {
        cout<<"Divide by zero exception: b = "<<i;
    }
    cout<<endl<<"END";
    return 0;
}
```

Specifying Exceptions

- It is also possible to restrict a function to throw only certain specified exceptions.
- This is achieved by adding a throw list clause to the function definition.
- The general form is as follows:

```
type function(arg-list) throw (type-list)
{
    //Function body
}
```
- The type-list specifies the type of exceptions that may be thrown.
- Throwing any other type of exception will cause abnormal program termination.

Specifying Exceptions

- If we wish to prevent a function from throwing any exception, we may do so by making the type-list empty.
- Hence the specification in this case will be
 - type function(arg-list) throw ()
 - {
 - // Function body
 - }
- Note: A function can only be restricted in what types of exception it throws back to the try block that called it. The restriction applies only when throwing an exception out of the function (and not within the function).

```
#include<iostream>  
  
using namespace std;  
  
void test(int x) throw (int, double)  
{  
    if(x == 0) throw x;  
    if(x == 1) throw 1.0;  
}
```

```
int main()  
{  
    try  
    {  
        test(1);  
    }  
    catch(int m)  
    {  
        cout<<"Caught an integer\n";  
    }  
    catch (double d)  
    {  
        cout<<"Caught a double";  
    }  
    return 0;  
}
```

END OF UNIT SEVEN

Syllabus

- **UNIT 8: File Handling and Streams**
 - Stream Class Hierarchy for Console Input/Output
 - Unformatted Input/Output
 - Formatted Input/Output with *ios* Member Functions
 - Formatting with Manipulators
 - File Input/Output with Streams
 - Opening and Closing Files
 - Read/Write from File
 - File Access Pointers and their Manipulators
 - Sequential, Random Access to File
 - Testing Errors during File Operations
 - Stream Operator Overloading

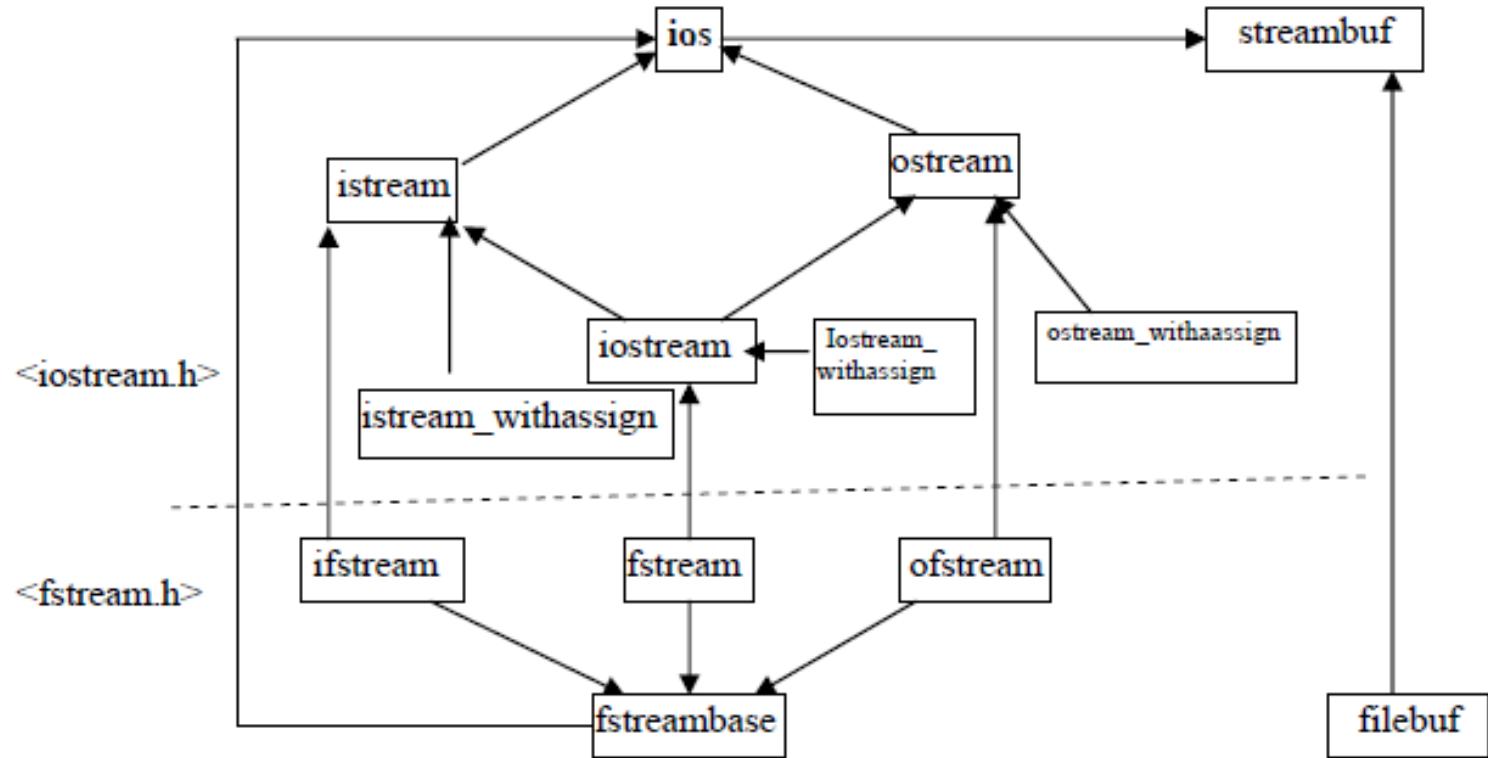
UNIT 8

FILE HANDLING AND STREAMS

Introduction

- *A stream is a name given to flow of data.*
- *In C++ stream is represented by an object of a particular class. For example, cin and cout are input and output stream objects.*
- *There are no any formatting characters in stream like %d, %c etc. in C which removes major source of errors.*
- *Due to overloading operators and functions, we can make them work with our own classes.*

The Stream Class Hierarchy



C++ Class Hierarchy

- ***filebuf***: The class *filebuf* sets the file buffer to read and write.
- ***ios***: *ios* class is parent of all stream classes and contains the majority of C++ stream features.
- ***istream class***: Derived from *ios* and perform input specific activities.
- ***ostream class***: derived from *ios* class and perform output specific activities.

C++ Class Hierarchy

- ***iostream class:*** Derived from both *istream* and *ostream* classes, it can perform both input and output activities and used to derive *iostream_withassign* class.
- ***_withassign classes:*** There are three *_withassign* classes.
 - *istream_withassign*
 - *ostream_withassign*
 - *iostream_withassign*

These classes are much like those of their parent but include the overloaded assignment operators.

C++ Class Hierarchy

- **streambuf**: sets stream buffer i.e. an area in memory to hold the objects actual data. Each object of a class associated with the **streambuf** object so if we copy the stream object it cause the confusion that we are also copying **streambuf** object. So **_withassign** classes can be used if we have to copy otherwise not.
- **fstreambase**: Provides operations common to file streams. Serves as a base for **fstream**, **ifstream** and **ofstream** and contains **open()** and **close()** functions.
- **ifstream**: Contains input operations in file. Contains **open()** with default input mode, inherits **get()**, **getline()**, **read()**, **seekg()**, **tellg()** from **istream**.
- **ofstream**: Provides output operation in file. Contains **open()** with default output mode, inherits **put()**. **Seekp()**, **tellp()** and **write()** from **ostream**.
- **fstream**: Provides support for simultaneous input and output operations. Contains **open()** with default input mode: Inherits all the functions of **istream** and **ostream** through **iostream**.

File I/O with Stream Classes

- In C++, file handling is done by using C++ streams.
- The classes in C++ for file I/O are *ifstream* for input files, *ofstream* for output files, and *fstream* for file used for both input and output operation.
- These classes are derived classes from *istream*, *ostream*, and *iostream* respectively and also from *fstreambase*.
- The header file for *ifstream*, *ofstream* and *fstream* classes is **<fstream.h>**
- To create and write disk file we use *ofstream* class and create object of it.
ofstream outf;

File I/O with Stream Classes

- The creation and opening file for write operation is done either using its constructor or using `open()` member function which had already been defined in `ofstream` class.
- Creating and opening file for write operation is given as:
`ofstream outf("myfile.txt");` // using constructor of ofstream class.
OR
`ofstream outf;`
`outf.open("myfile.txt");` // using open() member function

Writing Text into File

- We use the object of *ofstream* to write text to file created as:

```
outf<<“This is the demonstration of file operation”;
```

```
outf<<“You can write your text”;
```

```
outf<<“The text are written to the disk files”;
```

Writing Text into File

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    // Create and open a text file
    ofstream MyFile("filename.txt");
    // Write to the file
    MyFile << "This is the message written in filename.txt file";
    // display the message
    cout<<"filename.txt successfully created with the custom message";
    // Close the file
    MyFile.close();
}
```

Writing Data into File

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    char ch1='C';
    char ch2='R';
    int roll1=11;
    int roll2=24;
    float sem=2.0;
    char *str1="Texas International College";
    char *str2="B.Sc. CSIT";
    ofstream fout("Test.txt");
    fout<<str1<<endl<<str2<<endl<<ch1<<ch2<<" - Roll Number: "<<roll1<<endl;
    fout<<ch1<<ch2<<" - Roll Number: "<<roll2<<endl;
    cout<<"Data written to file\n";
}
```

Reading Data from File

- To read data from file , we use an object of *ifstream* class File is opened for reading using constructor of *ifstream* class or **open()** member function as;

```
ifstream fin("test.txt"); // constructor
```

OR

```
ifstream fin;  
fin.open("test.txt"); // member function open();
```

Reading data is done as:

```
fin>>ch>>i>>f>>str;
```

which is similar as reading data from keyboard by *cin* object.

Reading Text from File

- To read text from file we use *ifstream* class and file is opened for read operation using constructor or *open()* member function.

- For example:

```
ifstream infile("myfile.txt"); // using constructor
```

or

```
ifstream infile;  
infile.open("myfile.txt");
```

- // Reading from file myfile.txt:

```
while(infile) // or while(!infile.eof()) until end of file  
{  
    infile.getline(buffer,maxlength); // buffer to be defined as char  
    // string of length maxlen  
    cout<<buffer; // for display to screen  
}
```

Reading Text from File

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{
    const int LEN = 100;
    char text[LEN]; //for buffer
    ifstream infile("Test.txt");
    while(infile)      // until end of file Alternate is: while(!infile.eof())
    {
        infile.getline(text,LEN);      // read a line of text
        cout<<endl<<text;          // display line of text
    }
}
```

Reading Text from File

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{
    const int LEN = 100;
    char text[LEN]; //for buffer
    ifstream infile("Test.txt");
    while(!infile.eof())           // until end of file Alternate is: while(infile)
    {
        infile.getline(text,LEN);   // read a line of text
        cout<<endl<<text;         // display line of text
    }
}
```

Reading Text from File

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

int main ()
{
    // Create a text string, which is used to output the text file
    string myText;
    // Read from the text file
    ifstream MyReadFile("filename.txt");
    // Use a while loop together with the getline() function to read the file line by line
    while (getline (MyReadFile, myText))
    {
        cout << myText;           // Output the text from the file
    }
    // Close the file
    MyReadFile.close();
}
```

Writing & Reading Text of File

```
int main ()  
{  
    char data[100];  
    // open a file in write mode.  
    ofstream outfile;  
    outfile.open("file.txt");  
    cout << "Writing to the file" << endl;  
    cout << "Enter your name: ";  
    cin.getline(data, 100);  
    // write inputted data into the file.  
    outfile << data << endl;  
    cout << "Enter your age: ";  
    cin >> data;  
    // again write inputted data into the file.  
    outfile << data << endl;  
    // close the opened file.  
    outfile.close();  
    // open a file in read mode.  
    ifstream infile;  
    infile.open("file.txt");  
    cout << "Reading from the file" << endl;  
    infile >> data;  
    // write the data at the screen.  
    cout << data << endl;  
    // again read the data from the file and  
    // display it.  
    infile >> data;  
    cout << data << endl;  
    // close the opened file.  
    infile.close();  
    return 0;  
}
```

Character I/O in File get() and put() functions

- `put()` and `get()` functions are members of `ostream` and `istream` classes so they are inherited to `ofstream` and `ifstream` objects.
- `put()` is used to write a single character in file and `get()` is used for reading a character from file.

Character I/O in File

get() and put() functions

```
#include<iostream>
#include<fstream>
#include<string.h>
using namespace std;
int main()
{
    char*str="String written to file one
char at a time";
    ofstream fout;
    fout.open("myfile.txt");
    for(int i=0;i<strlen(str);i++)
    {
        fout.put(str[i]);
    }
    cout<<"File write completed";
}
```

```
#include<iostream>
#include<fstream>
#include<string.h>
using namespace std;
int main()
{
    // Reading character wise from the file
    char ch;
    ifstream infile;
    infile.open("myfile.txt");
    while(infile)
    {
        infile.get(ch);
        cout<<ch;
    }
}
```

Working with Multiple File

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
ofstream outfile;
//create file district and open for write
outfile.open("district.txt");
outfile<<"Kathmandu\n";
outfile<<"Lalitpur\n";
outfile<<"Kavreplanchowk\n";
outfile<<"Dhading\n";
outfile.close(); // close the file district after writing
outfile.open("headquarter.txt");
outfile<<"Kathmandu\n";
outfile<<"Patan\n";
outfile<<"Dhulikhel\n";
outfile<<"Trishuli\n";
outfile.close(); // closes the file headqtr
```

```
//Reading the above files
const int LEN = 80;
char text[LEN];
ifstream infile("district.txt"); // opens file district for read
while(infile)
{
    infile.getline(text,LEN);
    cout<<text<<endl;
}
infile.close(); // closes file district after display
infile.open("headquarter.txt");
while(infile)
{
    infile.getline(text,LEN);
    cout<<text<<endl;
}
infile.close(); // closes file headqtr
```

Opening File in Different Mode

- In earlier examples, we have used the `ofstream` and `ifstream` constructors or `open()` member function using only one argument i.e. `filename` e.g. “`test`” etc.
- However this can be done by using two argument – One is `filename` and other is `filemode`.
- Syntax:
`Stream-object.open(“filename”,filemode);`
- The second argument `filemode` is the parameter which is used for what purpose the file is opened.

Opening File in Different Mode

- If we haven't used any filemode argument and only filename with `open()` function, the default mode is as:

`ios::in` for `ifstream` functions means open for reading only.

i.e. `fin.open("test");` is equivalent to `fin.open("test",ios::in);` as default.

`ios::out` for `ofstream` functions means open for writing only.

i.e. `fout.open("test");` is same as `fout.open("test",ios::out);` as default.

Opening File in Different Mode

- Class *fstream* inherits all features of *ifstream* and *ofstream* so we can use *fstream* object for both input/output operation in file.
- When *fstream* class is used , we should mention the second parameter <filemode> with *open()*.
- The file mode parameter can take one or more such predefined constants in *ios* class.

Opening File in Different Mode

- The following are such file mode parameters:

| Parameters | Meanings |
|-----------------------|---------------------------------------|
| <i>ios::app</i> | Append to end of file |
| <i>ios::ate</i> | Go to end-of-file on opening |
| <i>ios::binary</i> | Binary file |
| <i>ios::in</i> | Open file for reading only |
| <i>ios::nocreate</i> | Opens fail if the file does not exist |
| <i>ios::noreplace</i> | Open files if the file already exist |
| <i>ios::out</i> | Open file for writing only |
| <i>ios::trunc</i> | Delete the contents of file if exist |

Opening File in Different Mode

- Opening file in `ios::out` mode also opens in the `ios::trunc` mode default
- `ios::app` and `ios::ate` takes to the end-of-file when opening but only difference is that `ios::app` allows to add data only end-of-file but `ios::ate` allows us to add or modify data at anywhere in the file. In both case file is created if it does not exists.
- Creating a stream of `ostream` default implies `output(write)` mode and `ifstream` implies `input(read)`, but `fstream` stream does not provide default parameter so we must provide the mode parameter with `fstream`.
- The mode can combine two or more parameters using bitwise OR operator (`|`)

For example: `fout.open("test",ios::app | ios::out);`

File Pointers

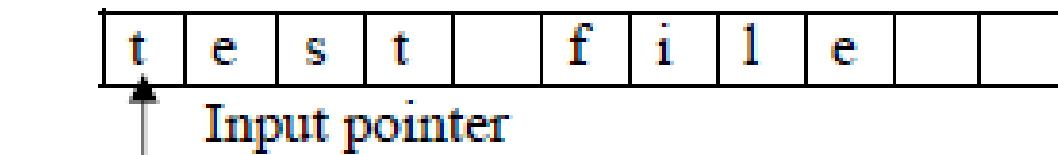
- *The file management system associates two types of pointers with each file.*
 - *get pointer (input pointer)*
 - *put pointer (output pointer)*
- *These pointers facilitate the movement across the file while reading and writing.*
 - *The get pointer specifies a location from where current read operation initiated.*
 - *The put pointer specifies a location from where current write operation initiated.*

File Pointers

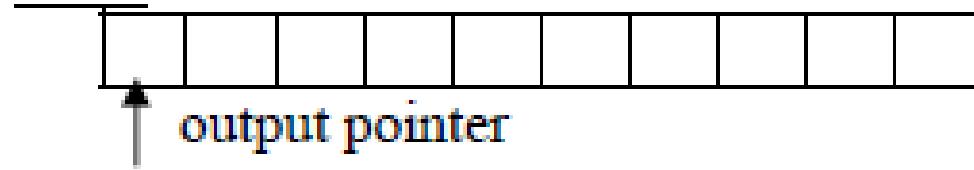
- *The file pointer is set to a suitable location initially depending upon the mode which it is opened.*
- **Read-only Mode:** When a file is opened in read-only mode, the input (get) pointer is initialized to the beginning of the file.
- **Write-only mode:** In this mode, existing contents are deleted if file exists and put pointer is set to beginning of the file.
- **Append mode:** In this mode, existing contents are unchanged and put pointer is set to the end of file so writing can be done from end of file.

File Pointers

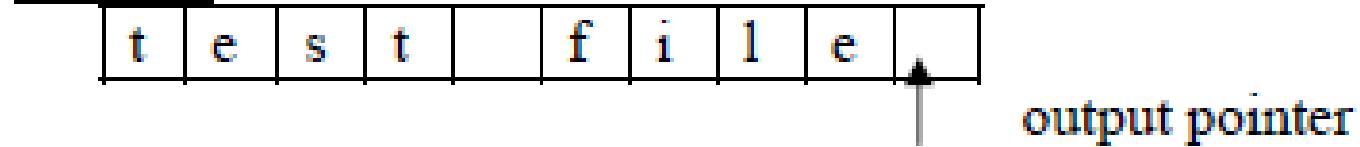
Read



write



Append



Functions Manipulating File Pointers

- C++ I/O system supports 4 functions for setting a file to any desired position inside the file.
- The functions are:

| Function | Member of Class | Action |
|----------------------|-----------------------|---|
| <code>seekg()</code> | <code>ifstream</code> | Moves get file pointer to a specific location |
| <code>seekp()</code> | <code>ofstream</code> | Moves put file pointer to a specific location |
| <code>tellg()</code> | <code>ifstream</code> | Returns the current position of the get pointer |
| <code>tellp()</code> | <code>ofstream</code> | Returns the current position of the put pointer |

- These all four functions are available in `fstream` class by inheritance.

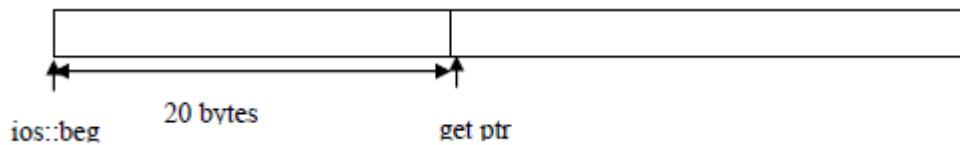
Functions Manipulating File Pointers

- For example:

ifstream infile;

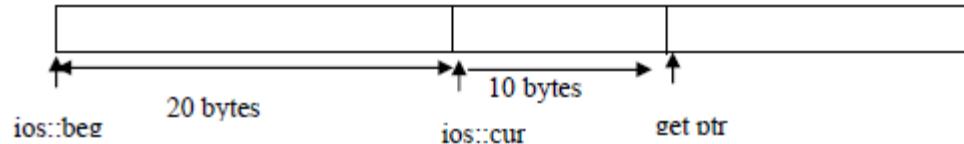
infile.seekg(20, ios::beg);

// moves file ptr to 20th byte in the file and reading starts from 21st item.



infile.seekg(10, ios::cur);

// moves get ptr 10 bytes further from current position



Pointer Offset Calls & their Actions

- Let us assume: *ofstream fout;*

| Seek | Action |
|---------------------------------|---|
| <i>fout.seekg(0, ios::beg)</i> | <i>Go to beginning of the file</i> |
| <i>fout.seekg(0, ios::cur)</i> | <i>Stay at current location</i> |
| <i>fout.seekg(0, ios::end)</i> | <i>Go to the end of file</i> |
| <i>fout.seekg(n, ios::beg)</i> | <i>Move to (n+1) byte from beginning of the file</i> |
| <i>fout.seekg(n, ios::cur)</i> | <i>Move forward by n bytes from current position</i> |
| <i>fout.seekg(-n, ios::cur)</i> | <i>Move backward by n bytes from current position</i> |
| <i>fout.seekg(n, ios::beg)</i> | <i>Move write pointer (n+1) byte location</i> |
| <i>fout.seekg(-n, ios::cur)</i> | <i>Move write pointer n bytes backwards</i> |

Use of seekg() function

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    fstream myFile("test.txt");           // Open a new file for input / output operations
    myFile << "Texas International College"; // Add the characters to the file
    myFile.seekg(6, ios::beg);           // Seek to 6 characters from beginning of the file
    //myFile.seekg(14, ios::cur); // Seek to 14 more characters from the current position
    char A[21];                      // Read the next 21 characters from the file into a buffer
    myFile.read(A, 21);
    A[21] = 0;                        // End the buffer with a null terminating character
    cout << A << endl;               // Output the contents read from the file
    myFile.close();                  // Close the file
}
```

The write() & read() function

- The write () function is a member of stream class fstream and used to write data in file as binary format.
- The read () function is used to read data (binary form) from a file.
- The data representation in binary format in file is same as in system.
- The no of byte required to store data in text form is proportional to its magnitude but in binary form, the size is fixed.

The write() & read() function

- The prototype for read() and write() functions are:
infile.read((char) &variable, sizeof(variable));*
outfile.write((char) &variable, sizeof(variable));*
- The first parameter is a pointer to a memory location at which the data is to be retrieved [read()] or to be written [write()] function.
- The second parameter indicates the number of bytes to be transferred.

The write() & read() function

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int number1=530;
    float number2=100.50;
    // open file in write binary mode, write integer and close
    ofstream ofile("number.bin",ios::binary);
    ofile.write((char*)&number1, sizeof(number1));
    ofile.write((char*)&number2, sizeof(float));
    ofile.close();
    // open file in read binary mode, read integer & close
    ifstream ifile ("number.bin",ios::binary);
    ifile.read((char*)&number1,sizeof(number1));
    ifile.read( (char*) &number2,sizeof(number2));
    cout<<number1<< " "<<number2<<endl;
    ifile.close();
}
```

Writing Object to Disk File

```
class emp
{
    char empname[20];
    int eno;
    float sal;
public:
void getdata()
{
    cout<<"Enter Name: ";
    cin>>empname;
    cout<<"Enter Emp No: "; cin>>eno;
    cout<<"Enter salary: "; cin>>sal;
}
};
```

```
int main()
{
    emp em;
    cout<<"Enter the detail of
employee"<<endl;
em.getdata();
ofstream fout("emp.dat");
fout.write((char*)&em,sizeof(em));
cout<<"Object written to file";
}
```

```
class emp
{
    char empname[20];
    int eno;
    float sal;
public:
    void showdata()
    {
        cout<<"\nName: "<<empname<<endl;
        cout<<"Emp NO: "<<eno<<endl;
        cout<<"Salary: "<<sal<<endl;
    }
};
```

```
int main()
{
    emp em;
    ifstream fin("emp.dat");
    fin.read((char*)&em,sizeof(em));
    cout<<"Object detail from
file";
    em.showdata();
}
```

Writing & Reading Objects

```
#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
class student
{
    char name[20];
    int roll;
    char add[20];
public:
void readdata()
{
    cout<<"Enter name:";cin>>name;
    cout<<"Enter Roll. no.:";cin>>roll;
    cout<<"Enter address:";cin>>add;
}
void showdata()
{
    cout<<setw(10)<<roll<<setiosflags(ios::left)<<setw(10)
    <<name<<setiosflags(ios::left)<<setw(10)<<add<<endl;
}
};
```

```
int main()
{
    int i;
    student s[5];
    fstream file;
    file.open("record.txt", ios::in | ios::out);
    cout<<"enter detail for 5 students:"<<endl;
    for(int i=0;i<5;i++)
    {
        s[i].readdata();
        file.write((char*)&s[i],sizeof(s[i]));
    }
    file.seekg(0); //move pointer begining.
    cout<<"Output from file"<<endl;
    cout<<setiosflags(ios::left)<<setw(10)<<"RollNo"
    <<setiosflags(ios::left)<<setw(10)<<"Name"
    <<setiosflags(ios::left)<<setw(10)<<"Address"<<endl;
    for(i=0;i<5;i++)
    {
        file.read((char*)&s[i],sizeof(s[i]));
        s[i].showdata();
    }
    file.close();
}
```

Writing & Reading Binary Files

```
#include <stdio.h>

int main()
{
    char ch;
    /* Pointers for both binary files*/
    FILE *fpbr, *fpbw;
    /* Open for bin1.exe file in rb mode */
    fpbr = fopen("bin1.exe", "rb");
    /* test logic for successful open*/
    if(fpbr == NULL)
    {
        puts("Input Binary file is having issues while opening");
    }
    /* Opening file bin2.exe in "wb" mode for writing*/
    fpbw = fopen("bin2.exe", "wb");
    /* Ensure bin2.exe opened successfully*/
    if(fpbw == NULL)
    {
        puts("Output binary file is having issues while opening");
    }
    /*Read &Write Logic for binary files*/
    while(1)
    {
        ch = fgetc(fpbr);
        if(ch==EOF)
            break;
        else
            fputc(ch,fpbw);
    }
    /* Closing both the binary files */
    fclose(fpbr);
    fclose(fpbw);
    return 0;
}
```

Command Line Arguments

- C++ supports the features that facilitates the supply of arguments to the `main()` function.
- The arguments are supplied to the main at the time of program execution from command line.
- The main function takes two arguments.
 - First of which is argument count `argc` and
 - second is an array of arguments name `argv[]` as
`main(int argc, char* argv[])`
- Such program is invoked in command prompt as
`C> programname arg1 arg2....`

Command Line Arguments

(Copying from one file to another)

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fs;
    ofstream fd;
    string str;
    char sourcefile[100], destinationfile[100];
    cout << "Enter Source File with Extension:" << endl;
    gets(sourcefile);
    fs.open(sourcefile);
    if (!fs)
    {
        cout << "Error in Opening Source File...!!!";
        exit(1);
    }
    cout << "Enter Destination File with Extension:" << endl;
    gets(destinationfile);
    fd.open(destinationfile);
    if (!fd)
    {
        cout << "Error in Opening Destination File...!!!";
        fs.close();
        exit(2);
    }
    if (fs && fd)
    {
        while (getline(fs, str))
        {
            fd << str << endl;
        }
        cout << "\n\n Source File Data Successfully Copied to Destination
File...!!!";
    }
    else
    {
        cout << "File Cannot Open...!!!";
    }
    cout << "\n\n Open Destination File and Check!!!\n";
    fs.close();
    fd.close();
}
```

Error Handling during the File Operations in C++

- The C++ programming language provides several built-in functions to handle errors during file operations. The file error handling
- The following are the built-in functions to handle file errors.
 - *bad()*
 - *fail()*
 - *good()*
 - *eof()*

Error Handling during the File Operations in C++

- *bad()*
 - It returns a non-zero (true) value if an invalid operation is attempted or an unrecoverable error has occurred. Returns zero if it may be possible to recover from any other error reported and continue operations.
- *fail()*
 - It returns a non-zero (true) value when an input or output operation has failed.
- *good()*
 - It returns a non-zero (true) value when no error has occurred; otherwise returns zero (false).
- *eof()*
 - It returns a non-zero (true) value when end-of-file is encountered while reading; otherwise returns zero (false).

Implementation of bad()

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream file;
    file.open("my_file.txt",
    ios::out);
    string data;
    file >> data;
    if(!file.bad())
    {
        cout << "Operation not success!!!"
        << endl;
        cout << "Status of the badbit: " <<
        file.bad() << endl;
    }
    else
    {
        cout << "Data read from file - " <<
        data << endl;
    }
    return 0;
}
```

Implementation of fail()

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream file;
    file.open("my_file.txt", ios::out);
    string data;
    file >> data;
    if(file.fail())
    {
        cout << "Operation not success!!!" << endl;
        cout << "Status of the failbit: " << file.fail() << endl;
    }
    else
    {
        cout << "Data read from file - " << data
        << endl;
    }
    return 0;
}
```

Implementation of good() & eof()

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream file;
    file.open("my_file.txt", ios::in);
    cout << "goodbit: " << file.good()
        << endl;
    string data;
    cout << endl << "Data read from
file:" << endl;
    while(!file.eof())
    {
        file >> data;
        cout << data << " ";
    }
    cout << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    { real = r; imag = i; }
    friend ostream &operator << (ostream &out, const Complex &c);
    friend istream &operator >> (istream &in, Complex &c);
};
ostream &operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}
```

```
istream &operator >> (istream &in, Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imaginary Part ";
    in >> c.imag;
    return in;
}
int main()
{
    Complex c1;
    cin >> c1;
    cout << "The complex object is ";
    cout << c1;
    return 0;
}
```

END OF UNIT EIGHT

P
R
E
P
A
R
E
D

B
Y
S
U
L
A
V

N
E
P
A
L

P
R
E
P
A
R
E
D

B
Y

S
U
L
A
V

N
E
P
A
L

BEST WISHES & HAPPY CODING 😊