

*"Real Programmers, do not comment their code.
If it was hard to write, it should be
Hard to understand."*

-Anan



**"Life is a difficult game. You can win it only by retaining
your birthright to be a person".**

-Dr.A.P.J

[CORE COURSE II PROGRAMMING IN C++]

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

CORE COURSE II

PROGRAMMING IN C++

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

Unit I :

Principles of Object- Oriented Programming – Beginning with C++ - Tokens, Expressions and Control Structures – Functions in C++

Unit II :

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

Unit III:

Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

Unit IV:

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

Unit V :

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

Text Book

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

Reference Books

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

Basic Regards Programming:

- ✓ The process of preparing and feeding the instruction into the computer for execution is referred to as *programming*.
- ✓ The following are the steps adopted to *develop a program* in any programming language.
 - i. Problem definition
 - ii. Collection of information
 - iii. Selection of solution
 - iv. Development of algorithm
 - v. Writing program code
 - vi. Debugging & testing
 - vii. Documentation
 - viii. Program Maintenance
- ✓ *Software design* is a creative procedure, hence they have various design procedures. Two design approaches to solve any problems are
 - i. Top-Down design approach
 - ii. Bottom-Up Design approach
- ✓ *Programming methodologies* is a complex filed with many methodologies, names, many goals & means to reach them. Some of the important programming methodologies are as follows:
 - i. Step Wise Refinement
 - ii. Modularity
 - iii. Structured Programming.
- ✓ Three design tools emerged from structured programming, they are as follows
 - i. Flow chart
 - ii. Nassi-Shneiderman (NS) Diagrams
 - iii. Pseudocode.
- ✓ A flowchart is s pictorial presentation of the flow of data decrying a process/
programs/ project being studied.

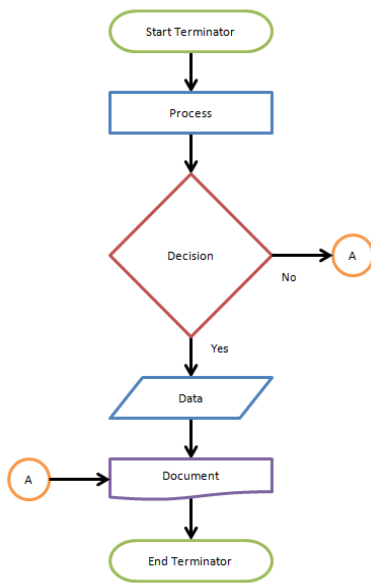


Fig1.1 Sample Flowchart

✓ *Pseudo code* is neither a programming language nor a natural language, but an informal language written in English, particularly used for developing algorithms.

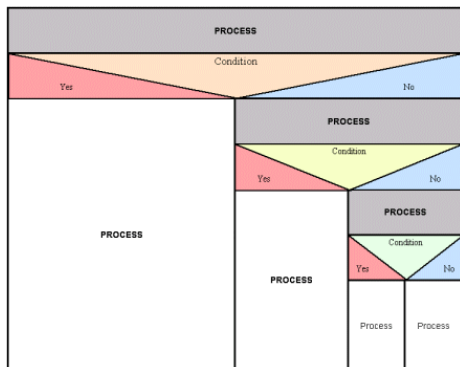
Pseudocode

- For example, for making a cup of tea:

```

Organise everything together;
Plug in kettle;
Put teabag in cup;
Put water into kettle;
Wait for kettle to boil;
Add water to cup;
Remove teabag with spoon/fork;
Add milk and/or sugar;
Serve;
  
```

Nassi-Shneiderman(NS) diagrams are similar to flowcharts, is a pictorial representation of flow of logic with in a program.



✓ An *algorithm* is step by step recipe for solving an instance of a problem.

WHAT IS C++:

- ✓ C++ is a *general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.*
- ✓ C++ is a middle-level language, as it *encapsulates both high and low level language features.*

HISTORY FOR C++:

- ✓ C is a general purpose *structured programming language* developed at AT & T bell Laboratories in 1972 by *Dennis Ritchie.*
- ✓ In *1979, Bjarne Stroustrup* at AT & T bell Laboratories developed another *High Level Programming language* called *c++* as the name implies *c++* is an enhancement of C .
- ✓ Stroustrup initially called the language “*C with Classes*” however in 1983 it was renamed as *c++.*
- ✓ C++ is not merely an extension of *c* , rather it incorporates several new fundamental concepts that form a basis for *Object – oriented Programming.*

APPLICATIONS OF C++:

- ✓ C++ allows us to create hierarchy- related objects, where we can build special object oriented libraries which can be used later by many programmers.
- ✓ C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- ✓ C++ is able to map the real-world problem properly, the part of *c++* gives the language the ability to get close to the machine – level details.

USAGE OF C++

- ✓ By the help of C++ programming language, we can develop different types of secured and robust applications:
 - Window application
 - Client-Server application
 - Device drivers
 - Embedded firmware etc

C vs C++

No.	C	C++
1)	C follows the procedural style programming .	C++ is multi-paradigm. It supports both procedural and object oriented .
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the top-down approach .	C++ follows the bottom-up approach .
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into procedures and modules	C++ programs are divided into functions and classes .
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.

INTRODUCTION TO C++:

Basics of OOP:

- ✓ **Object – oriented programming (OOP)** is the most dramatic innovation in software development in the last decade.
- ✓ **OOP** offers a new and powerful way to cope with the complexity in programs of procedural languages.
- ✓ **OOP** treats data as a critical element in the program development and does not allow it flow freely around the system. It ties **data** more closely to the function that operate on it and **protects it from accidental modifications** from outside functions.
- ✓ C++ is a superset of C. To support the principles of Object- Oriented Programming have the most important features that add on to C, They are as follows:

Basic concepts of OOP

- Objects
- Class
- Data Abstraction
- Data Encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message Passing

The three pillars of object-oriented development:

- **Encapsulation, Inheritance, and Polymorphism.**

BENEFITS OF OBJECT – ORIENTED PROGRAMMING:

- Data Security is enforced
- Inheritance save time
- User – Defined data types can be easily constructed
- Inheritance emphasizes inventions of new data types.
- Large complexity in software development can be easily managed.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program
- It is easy to partition the work in a project based on objects.
- Through inheritance we can eliminate redundant code and extend the use of existing classes.
- Object oriented systems can be easily upgraded from small to large system.

OBJECT ORIENTED LANGUAGES:

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

OBJECT- BASED PROGRAMMING LANGUAGES:

- ✓ Is the style of programming that primarily supports *encapsulation and object identity*. Major features that are required for object-based programming are:

- I. *Data encapsulation*
- II. *Data hiding and access mechanisms*
- III. *Automatic initialization and clear – up of objects*
- IV. *Operator Overloading*

OBJECT- ORIENTED PROGRAMMING LANGUAGES:

- ✓ Incorporates all of *object – based programming* features along with two additional features, namely

- I. *Inheritance*
- II. *Dynamic binding*

Object- based features + inheritance + dynamic binding.

Difference Between Object Based and Object Oriented Languages		
	Object Based Language	Object Oriented Language
Support of features	Object Based Language does not support all the features of OOps	Object Oriented Language supports all the features of OOps.
Inheritance	Object Based Language Does Not Support OOps feature i.e. Inheritance.	Object Oriented Language supports all the Features of OOps including Inheritance.
Sample	Visual Basic is an Object based Programming Language because you can use class and Object here but can not inherit one class from another class i.e. It does not support Inheritance.	Java is an Object Oriented Languages because it supports all the concepts of OOps like Data Encapsulation, Polymorphism, Inheritance, Data Abstraction , Dynamic Binding etc.
Example	Javascript, VB are example of Object Based Language.	C#, Java, VB, Net are example of Object Oriented Languages.

APPLICATIONS OF OOP:

- ✓ OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Hundreds of Windowing systems have been developed, using the OOP Techniques. The promising areas for applications of OOP Include:

Areas for applications of OOP

- Real time systems
- Simulation and modeling
- Object oriented database
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural network and parallel programming
- Decision support system
- Office automation system
- CIM / CAM / CAD systems

PRINCIPLES OF OBJECT:

OBJECTS:

- ✓ **Objects** are the basic run time entities in object -oriented system.
- ✓ The fundamental concept behind the object -oriented languages is to combine both data and functions together such a unit is called as an *objects*.
- ✓ They may represent user defined data such as *vectors, time and lists*.
- ✓ **Memory** is allocated to *objects* and not for *classes*.
- ✓ When a program is executed, the objects interact by sending an message to another one.

For eg: if “*customer*” and “*account*” are two objects in a program, then the customer object may *send a message* to the *account object* requesting for the *bank balance*.

CLASSES:

- ✓ **Classes** are the fundamental unit of *object -oriented programming*.
- ✓ A *class* is thus a *collection of objects of similar type*.
- ✓ The entire set of *data* and *code* of an *data* and *code* of an *object* can be made a user defined data type with the *help of classes*.
- ✓ The c++ class mechanism is a new way of creating and implementing a user defined data type according to the needs of the problem to be solved.
- ✓ **Once a class** has been defined, we can create a *many number of objects* belonging to *that class*.

FOR EG: fruit mango;

Will create an *object Mango* belonging to the *class fruit*.

DATA ABSTRACTION :

- ✓ **Abstraction** refers to the act of *representing essential features without including the background details (or) explanations*.
- ✓ **Classes** use the concept of *abstraction* and are defined as a list of *abstract attributes* such as *size, weight* and *cost* and *functions* to operate on these *attributes*.
- ✓ These *attributes* are sometimes called as *data members* because they *hold information*. The functions that operate on these data are called as *methods* or *member functions*.
- ✓ **Abstraction** is the ability to *create* and *define user defined data types* using *built-in data types*.

DATA ENCAPSULATION:

- ✓ The *wrapping up of data and functions* into a *single unit* (called class) is known as *encapsulation*.
- ✓ It is the *process of combining member functions* and the *data it* manipulates by *logically binding* the data keeping them *safe from outside interference*.
- ✓ The **insulation** of *data* from *direct access* by the *program* is called as *Data Hiding or information hiding*.

INHERITANCE:

- ✓ It is the *process* by which *one objects* of *one class* acquire the *properties* of *objects of another class*.
- ✓ *Inheritance* involves the creation of *new type* from *existing type* in some *hierarchical* fashion. it supports the concepts of *hierarchical classification*.
- ✓ The *existing class* is called as the *Base class* and the *new class* is called as *derived class*
- ✓ *Inheritance* is used to *reduce* the *source code* of an *OOP*

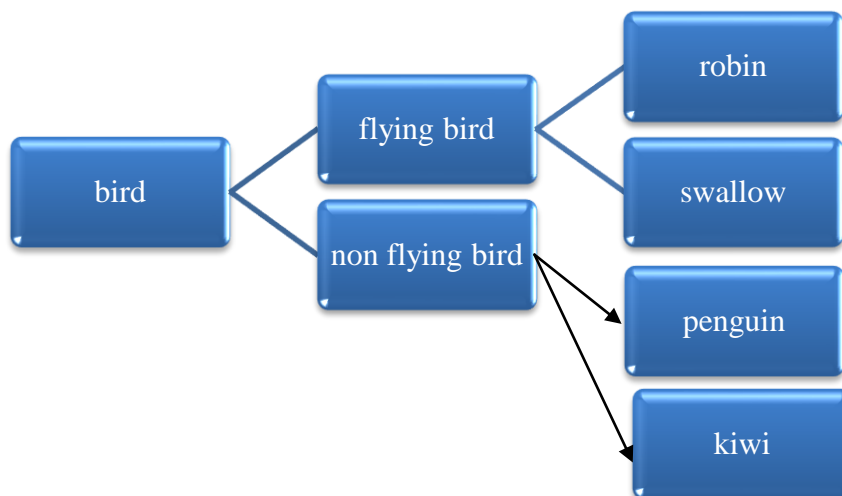


Fig shows property of inheritance

POLYMORPHISM:

- ✓ *Polymorphism* is an *Greek* term means the *ability* to take *more than one* form.
- ✓ **An** operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in operations.
- ✓ **For eg:** consider the operation of addition of two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

- ✓ The *process* of *making* an *operator* to *exhibit different behaviors* in different *instances* are called as *operator overloading*
- ✓ The correct *functions* to be invoked is determined by *the number , type* and *sequence* of *arguments* in *function* is referred to as *function overloading or function polymorphism*.

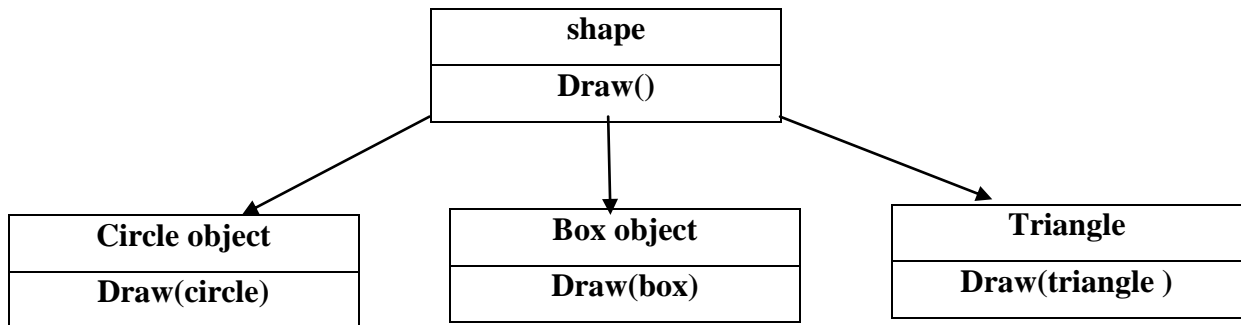


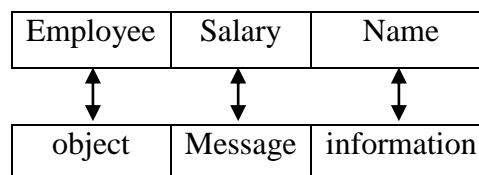
Fig shows polymorphism

DYNAMIC BINDING:

- ✓ *Binding* refers to the *linking of procedure call* to the *code to be executes* in *response to the call*.
- ✓ *Dynamic binding* means that the *code associated* with a *given procedure call* is *not known until* the time if the *call at run time*.
- ✓ It is associated with *polymorphism* and *inheritance*

MESSAGE PASSING:

- ✓ An OOP consists of a *set of objects* that *communicate with each other*. The following basic steps includes for communication
 - 1) *Creating classes that define objects and their behavior*
 - 2) *Creating objects from class definitions*
 - 3) *Establishing communication among the objects*.
- ✓ The concept of *message passing* makes it easier to *talk about building systems* that *directly model* or *simulate their real world counterparts*.
- ✓ Objects have a *life cycle*. They can be *created and destroyed*. Communication with an object is *feasible as long as it is alive*.



STRUCTURE OF C++ PROGRAM:

- ✓ Every c++ program contains a number of building blocks. These building blocks should be written in a correct order and procedure.

Include files
Class Declaration
Member functions definitions
Main function program

Structure of C++ Program

`#include<iostream.h>` → Include files / preprocessor directives

`void main()` → class declaration / function name

`{`
`int x,y,sum` → member function definitions / variable declaration

`cout<<" enter the values of x and y :"` // output statement
`cin>>x>>y;` // input statement } main function program

`Sum=x+y;`

`Cout<<"the sum of x and y="<<sum;` // output statement

`}` → end of the main / end of the program

Output:

Enter the values of x and y : 25 30

The sum of x and y = 35

Sample example Addition program using 2 variables

i) Include file

- ✓ In C++ number of functions, classes and variables are defined and stored in a special file called **header file**.
- ✓ There are so *many header files available*.
- ✓ Each header file has *related functions, classes and data*.
- ✓ If we want to use the predefined functions, classes and variables and data the appropriate header file should be included at the beginning of the program.
E.g.: *iostream.h, string.h, conio.h*

ii) Class Definition

- ✓ It is a user defined data type having defined functions and data.

Class test

```
{  
}
```

iii) Member function definition

- ✓ Member function can be defined inside or outside the class.

Void add()

```
(  
}
```

iv) Main function

- ✓ The *execution of the program begins from void main() function*.
- ✓ The *parentheses following the word main* are the distinguishing feature of the function, without *the parentheses the compiler* would think that main referred to a variable (or) *some other program element*.
- ✓ Where *void proceeding the function main ()* indicates that this particular function does not have a return value.

void main()

```
{  
}
```

TOKENS:

✓ *Tokens* are the *smallest individual unit in a program*. C++ has the following tokens

- 1) Keywords
- 2) Identifiers
- 3) Constants
- 4) Data types
- 5) Operators

KEYWORDS:

✓ *Keywords* also *referred* to as *reserved words* are words *whose meaning has been already defined in the c++ compiler*.

✓ All *keywords* have *fixed meaning* and their *meanings cannot be changed*.

✓ All *keywords must be compulsorily written in lowercase*.

✓ Eg's of keywords are as follows:

**auto double int struct break else long switch bool catch
delete dynamic_cast virtual what_t public false new export if goto**

IDENTIFIERS:

✓ It refers to the name of variables, functions, array, classes .etc., created by the programmer.

RULERS FOR CONSTRUCTING IDENTIFIERS :

- i. Only alphabetic characters, digits and underscores are permitted.
- ii. The name cannot start with a digit.
- iii. Upper and lowercases letters are distinct.
- iv. A declared keyword can't be used as a variable name.

CONSTANTS :

✓ It refer to *fixed values* that *do not change during the execution of a program*.

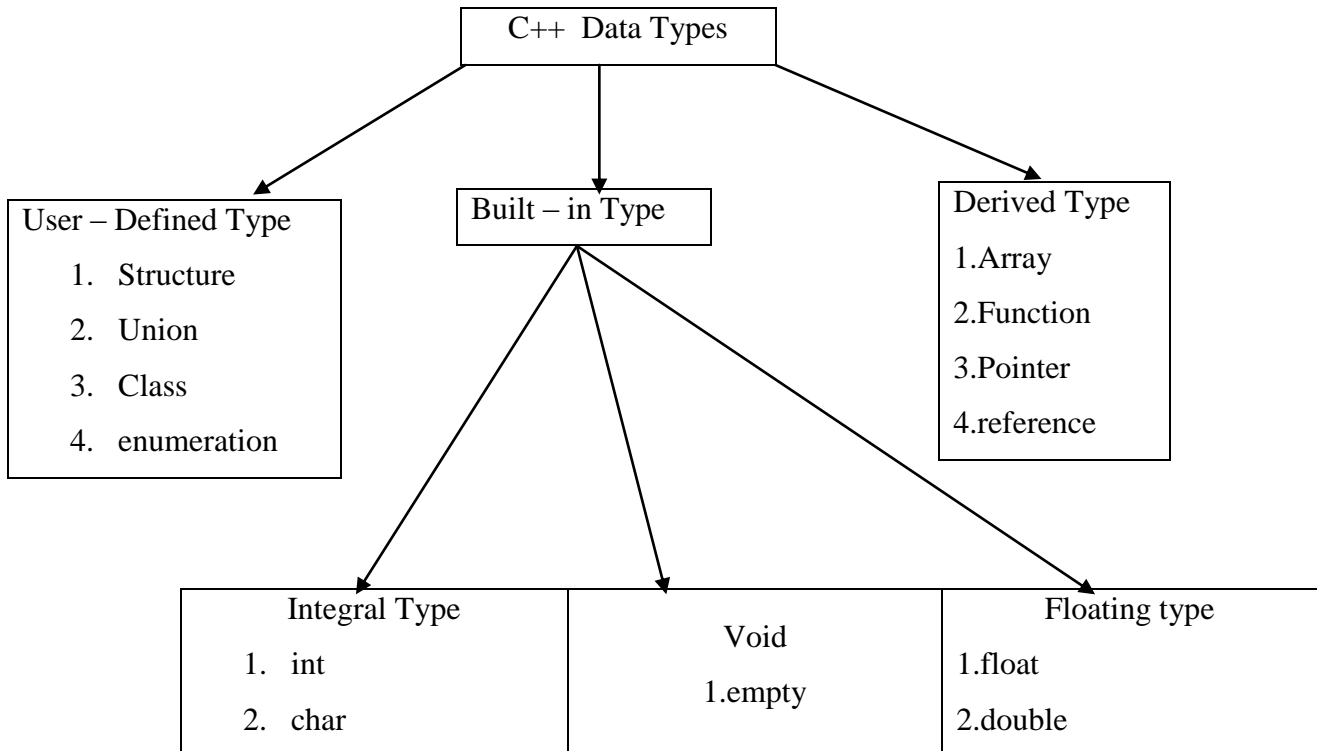
✓ They include *integers, characters, floating point numbers and strings*.

Examples are as follows:

123	//decimal integer constant
12.34	// floating point integer constant
037	//octal integer constant
0X2	//hexa decimal integer constant
"C++"	//string constant
'a'	//character constant

DATA TYPES:

- ✓ C++ language is *rich in its data type*. It supports a wide variety of *data types* each of which *may be represented differently within the computer's memory*, which allows the programmer to select the *appropriate type* to the *needs of the applications*.



Hierarchy of c++ data types

USER DEFINED DATA TYPES:

STRUCTURE:

- ✓ It is often required to group *logically related data items together*.
- ✓ While *array* are used to group *together similar type data elements*, where *structures* are used for *grouping together elements of dissimilar types*.

The general form of a structure definition is as follows:

```
struct name
{
    data_type member1;
    data_type member2;
    .....
    .....
};
```

```
struct book
{
    Charrtitle[25];
    Charauthor[25];
    int pages;
    float price;
};
```

UNIONS:

- ✓ These are conceptually *similar to structures* as they allows us to *group together dissimilar type elements inside a single unit*.
- ✓ The *size of the union* is equal to the *size of the its largest member element*.

```
Union reslut
{
    Int marks;
    Char grade;
    float percentage;
};
```

Difference between Structure and Union

STRUCTURE	UNION
Every memory has its own memory sapce.	All the members use the same memory space to store the values.
It can handle all the members(or) a few as required at a time.	It can handle only one member at a time,as all the members use the same space.
Keyword struct is used.	Keyword union is used.
It may be initialized with all its members.	Only its first member may be initialized.
Any member can be accessed at any time without the loss of data.	Only one member can be accessed at any time with the loss of previous data.
Different interpretation for the same memory location are not possible.	Different interpretations for the same
More storage space is required.	Minimum storage space is required.
Syntax: structure strut-name { }var1..varn;	Syntax: union union-name { }var1..varn;

CLASSES:

- ✓ The *class variables* are known as *objects*, which are the central focus of OOP.

ENUMERATED DATA TYPES:

- ✓ Is another *user – defined data* type which provides a way for *attaching names to the numbers*, thereby increasing comprehensibility of the code.
- ✓ The *enum keyword* automatically *enumerates* a list of words by *assigning the values 0,1,2, and so...on*.
- ✓ This is an *alternative way for symbolic constant*.
- ✓ The *Syntax of enum statement* is similar to that of *struct statement*.

EG:

```
enum shape {circle,square,triangle};  
enum colour {red,blue,green};  
enum position {off,on };
```

DERIVED DATA TYPES:

```
1.Array  
2.Function  
3.Pointer  
4.reference
```

ARRAY:

- ✓ Array are defined as a group of data items that share a common name with the same data type.
- ✓ Individual data elements are called as Elements.
- ✓ Similar to the others array should be declared before all executable statements, for example

```
int a[10];  
char ab[10] [10];  
float abc[10] [10] [10];
```

Explanation:

- ✓ The *1st declaration* declares a single dimensional integer array "*a*" that can store 10 integers (i.e.,0-9) in consecutive memory locations.*2nd Declaration shows a 2D character array "ab"* that can store *10* string constant in consecutive memory.*3rd declaration shows 3D point array "abc"* that can store *1000 elements(10*10*10)*

INITIALIZATION OF ARRAYS:

- ✓ Array can be initialized when they are declared.
- ✓ This is done by adding an equal sign followed by the required values to be initializes after the declaration between braces separated with commas

for eg

```
int a[5]={1,2,3,4,5};
char name [8]={‘h’,’i’};
char name[8]={“bharath”};
char names [3] [10] = { (“BCA”) , {“CS”) , {“IT”)};
```

FUNCTIONS:

- ✓ A *functions* is a *self-contained program segment (block of segments)* that performs some *specific well- defined task when called*.
- ✓ C++ allows programmers to define their own functions for carrying out various individual tasks.

Function declaration :

- ✓ A function has to be declared before using it, in a manner similar to variables and constants.
- ✓ A function declaration tells the compiler about a function's name, return type, and parameters and how to call the function.

The general form :

```
return_type function_name( parameter list );
```

Function Definition:

- ✓ The function definition is the actual body of the function.
- ✓ The function definition consists of two parts namely,
 1. function header
 2. function body.

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

```
return_type function_name( parameter list )
{
    body of the function
}
```

Where

- ✓ A return type *function may return a value*. The return_type is the *data type* of the value the *function returns*. Some functions *perform the desired operations* without returning a *value*. *In this case, the return_type is the keyword void*.
- ✓ **Function Name:** This is the *actual name of the function*.
- ✓ **Parameters:** A parameter is like a *placeholder*. When a *function is invoked*, you pass a *value to the parameter*. This value is referred to as *actual parameter or argument*. The parameter list refers to the *type, order, and number of the parameters of a function*. *Parameters are optional; that is, a function may contain no parameters*.
- ✓ **Function Body:** The function *body contains a collection of statements* that define what *the function does*.

Actual Parameters/ arguments:

- ✓ The *Parameters in the invocation* are called the actual parameters / arguments.

Formal Parameters / arguments:

- ✓ The *actual parameters are represented* in the *invoked function by its* are called as formal parameters / arguments.
- ✓ Arguments can be passed to functions in one of two ways: using
 1. call-by-value
 2. call-by-reference.

CALL BY REFERENCE

- ✓ When using *call-by-value*, a *copy of the argument* is passed to the *function*. *Call-by-reference passes the address of the argument to the function*. By default, *C++ uses call-by-value*.
- ✓ Provision of the *reference variables in c++* permits us to pass parameter to the *functions by reference*.
- ✓ When we pass *arguments by reference*, the *formal arguments in the called function become* aliases to the *actual arguments in the calling function*. This means that when the function is working with its own arguments, it is actually working on the original data.

Example :

```
#include <iostream.h>
#include<conio.h>
void swap(int &x, int &y); // function declaration
int main (){
int a = 10, b=20;
cout << "Before swapping"<<endl;
cout<< "value of a : " << a <<" value of b : " << b << endl;
swap(a, b); //calling a function to swap the values.
cout << "After swapping"<<endl;
cout<<" value of a : " << a<< "value of b : " << b << endl;
return 0;
}
void swap(int &x, int &y) { //function definition to swap the values.
int temp;
temp = x;
x = y;
y = temp;
}
```

Output:

Before swapping value of a:10 value of b:20

After swapping value of a:20 value of b:10

CALL BY VALUE

- ✓ The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
- ✓ In this case, changes made to the parameter inside the function have no effect on the argument.
- ✓ By default, C++ uses call by value to pass arguments.
- ✓ In general, this means that code within a function cannot alter the arguments used to call the function.

Consider the function **swap()** definition as follows.

```
void swap(int x, int y)
{
int temp;
temp = x; /* save the value of x */
x = y; /* put y into x */
y = temp; /* put x into y */
return;
}
```

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Listed below are the special types of member functions that can be used within the class.

- Simple member function
- Static Member function
- Const function
- Inline function
- Friend function

SIMPLE MEMBER FUNCTIONS

- ✓ These are simple functions of C++ with or without return type and with or without parameters. The basic structure of a simple member function is:

Syntax:

```
return_type functionName(parameter_list)
{
    // function body;
}
```

STATIC MEMBER FUNCTIONS

- ✓ The *keyword 'static'* is used with such member functions. *Static is mainly used to hold its positions*. These functions work for the whole class rather than for a *particular object of the class*.

Example:

```
class X {  
  
public:  
  
    static void k(){};  
  
};  
  
int main()  
  
{  
  
    G::k(); // calling member function directly with class name  
  
}
```

- ✓ *The static member functions cannot access ordinary data members and member functions, but can only access the static data members and static member functions of a class.*

CONST MEMBER FUNCTION

- ✓ Const keyword makes *variable constant*, which means *once defined, their value cannot be changed*. The basic syntax of const member function is:

Example:

```
void fun() const{}
```

INLINE FUNCTIONS:

- ✓ It is an *function* that is *expanded in line when it is invoked*. Hence inline functions are recommended only for functions having few statements.
- ✓ The general form of inline function is as follows :

```
inline<function_header>  
{  
.....  
.....  
}
```

} → Body of the functions

FRIEND FUNCTION

- ✓ In C++ a *function or an entire class* may be declared to be a *friend of another class or function*. A friend function *can also be used for function overloading*.
- ✓ *Friend function declaration* can appear *anywhere* in the *class*. But a good practice would be where the class ends.
- ✓ An *ordinary function* that is not the *member function* of a class has no privilege to access the *private data members*, but the *friend function* does have the *capability to access any private data members*.

VIRTUAL FUNCTION

- ✓ A *virtual function* is a special form of *member function* that is declared within a base *class and redefined by a derived class*.
- ✓ The keyword *virtual* is used to *create a virtual function*, precede the function's declaration in the *base class*.
- ✓ If a *class includes* a virtual function and if *it gets inherited*, the virtual class redefines a *virtual function to go with its own need*.
- ✓ In other words, a virtual function is a function which gets override in the derived class and instructs the *C++ compiler for executing late binding on that function*.
- ✓ A function *call is resolved* at runtime in late binding and so compiler determines the type of *object at runtime*.

RECURSION :

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

Let's see a simple example of recursion.

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

```
return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1
```

javaTpoint.com

$1 * 2 * 3 * 4 * 5 = 120$

Fig: Recursion

POINTER

- ✓ A **pointer** is a *programming language object*, whose *value refers to* (or "points to") *another value stored* elsewhere in the *computer memory using its memory address*.

OPERATORS:

- ✓ An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- ✓ C++ is rich in built-in operators. Generally, there are six type of operators:
 - a. **Arithmetic operator**
 - b. **Relation operator**
 - c. **logical operator**
 - d. **Assignment operator**
 - e. **Conditional operator**
 - f. **Increment and Decrement Operators**

a) **Arithmetical operators**

Arithmetical operators +, -, *, /, and % are used to performs an arithmetic (numeric) operation.

Operator Meaning

- + Addition
- Subtraction
- * Multiplication
- / Division
- % Modulus

You can use the operators +, -, *, and / with both integral and floating-point data types. Modulus or remainder % operator is used only with the integral data type.

b) Relational operators

The relational operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

Relational Operators

< Less than

<= Less than or equal to

== Equal to

> Greater than

>= Greater than or equal to

!= Not equal to

c) Logical operators

The logical operators are used to combine one or more relational expression. The logical operators are

Operators Meaning

|| OR

&& AND

! NOT

d) Assignment operator

- ✓ The assignment operator '=' is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side.

For example: m = 5;

- ✓ The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

x = y = z = 32;

This code stores the value 32 in each of the three variables x, y, and z. In addition to standard

assignment operator shown above, C++ also support compound assignment operators.

Compound Assignment Operators

Operator Example Equivalent to

`+= A += 2 A = A + 2`

`-= A -= 2 A = A - 2`

`%= A %= 2 A = A % 2`

`/= A /= 2 A = A / 2`

`*= A *= 2 A = A * 2`

e) Increment and Decrement Operators

- ✓ C++ provides two special operators viz '+' and '-' for incrementing and decrementing the value of a variable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

f) Conditional operator

- ✓ The conditional operator ?: is called ternary operator as it requires three operands. The format of the conditional operator is :

Conditional_ expression ? expression1 : expression2;

If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2

is evaluated.

```
int a = 5, b = 6;
```

```
big = (a > b) ? a : b;
```

The condition evaluates to false, therefore big gets the value from b and it becomes 6.

The comma operator

- ✓ The comma operator gives left to right evaluation of expressions. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

```
int a = 1, b = 2, c = 3, i; // comma acts as separator, not as an operator
```

`i = (a, b);` // stores b into i would first assign the value of a to i, and then assign value of b to variable i. So, at the end, variable i would contain the value 2.

Explicit type casting operator

- ✓ *Type casting operators allow to convert* a value of a given *type to another type*. There are several ways to do this in C++.
- ✓ The *simplest one, which has been inherited from the C language*, is to precede the expression to be *converted by the new type enclosed between parentheses (())*:

```
1 int i;  
2 float f = 3.14;  
3 i = (int) f;
```

The previous code converts the floating-point number 3.14 to an integer value (3); the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of casting types are valid in C++.

Precedence of operators

- ✓ A single expression may have multiple operators. For example:

```
x = 5 + 7 % 2;
```

✓

In C++, the above expression always assigns 6 to variable x, because the % operator has a higher precedence than the + operator, and is always evaluated before.

- ✓ Parts of the expressions can be enclosed in parenthesis to override this precedence order, or to make explicitly clear the intended effect. Notice the difference:

```
1 x = 5 + (7 % 2); // x = 6 (same as without parenthesis)  
2 x = (5 + 7) % 2; // x = 0
```

CORE COURSE II PROGRAMMING IN C++

From greatest to smallest priority, C++ operators are evaluated in the following order:

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		.->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof (<i>type</i>)	parameter pack C-style type-casting	
4	Pointer-to-member	.* ->*	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-

CORE COURSE II PROGRAMMING IN C++

				right
12	Inclusive or		bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction		logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += - =	assignment / compound assignment	Right-to-left
		>>= <<= &= ^= =		
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

- ✓ When an *expression has two operators with the same precedence level, grouping* determines which one is evaluated first: either left-to-right or right-to-left.

EXPRESSIONS:

- ✓ Is a *combination of operators, constants and variables* arranged as per the rules of the *language*.
- ✓ An *expression* may *consist of one or more operands*, and *zero* or more operators to *produce a value*.
- ✓ It can be broadly classified into following types they are as follows:
 1. Constant expression
 2. Integral expression
 3. Float expression
 4. Pointer expression
 5. Relational expression
 6. Logical expression
 7. Bitwise expression
 - 8.

CONSTANT EXPRESSION:

- ✓ *It consist of only values.*

For eg: 15 (or) 20+5/2.0

INTEGRAL EXPRESSION

- ✓ *It consist of those produce integer results after implementing all the automatic explicit type conversion.*

For eg:

M

M*n-5

Where m and n are integer variables.

FLOAT EXPRESSIONS:

- ✓ *Are those which after all conversions , produce floating point results*

Foe eg:

X+y

X*y/10

5+float(10)

10.75

Where x and y are floating point variables.

POINTER EXPRESSIONS:

- ✓ *Which produce address value.*

For eg:

&m

Ptr

Ptr+1

“xyz”

Where m is a variable and ptr is a pointer

RELATIONAL EXPRESSIONS:

- ✓ *It yields of type **bool** which takes a value true or false*

LOGICAL EXPRESSION:

- ✓ *It combine two or more relational expressions and produce bool type results*

For eg:

a>b && x==10

BITWISE EXPRESSION

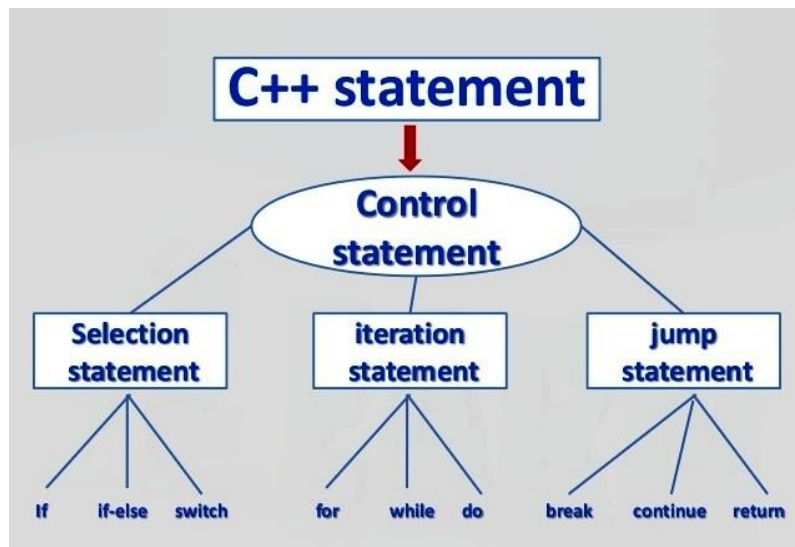
- ✓ Are used to manipulate data at bit level .
- ✓ This are basically used for testing or shifting bits.

For eg:

X<<3 // shift three bit position to left

Y>>1 // shift one bit position to right

STATEMENTS IN C++:



Selection statements:

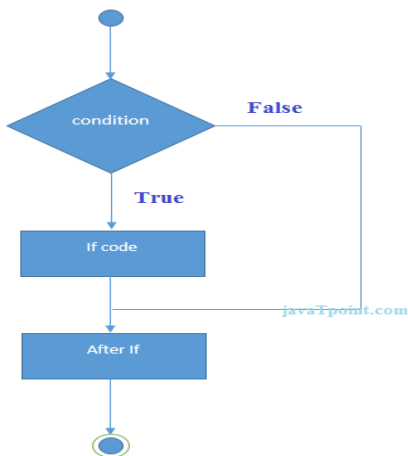
IF STATEMENT:

- ✓ The C++ if statement *tests the condition. It is executed if*

condition is true.

Syntax:

```
if(condition)
{
//code to be executed
}
```



```
include <iostream>
using namespace std;
int main () {
int num = 10;
if (num % 2 == 0)
{
cout<<"It is even nu
mber";
}
return 0;
}
```

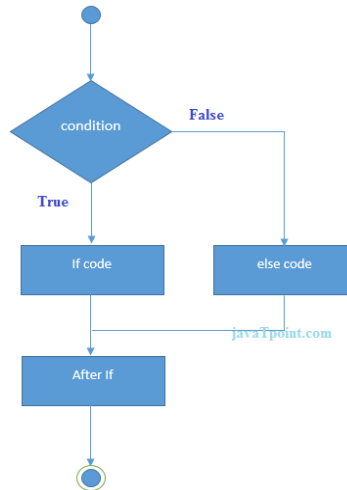
Output:/p>

It is even number

IF-ELSE STATEMENT

- ✓ The C++ if-else statement also *tests the condition*. It executes if *block if condition is true otherwise else block is executed*.

```
if(condition)
{
//code if condition is true
}else{
//code if condition is false
}
```

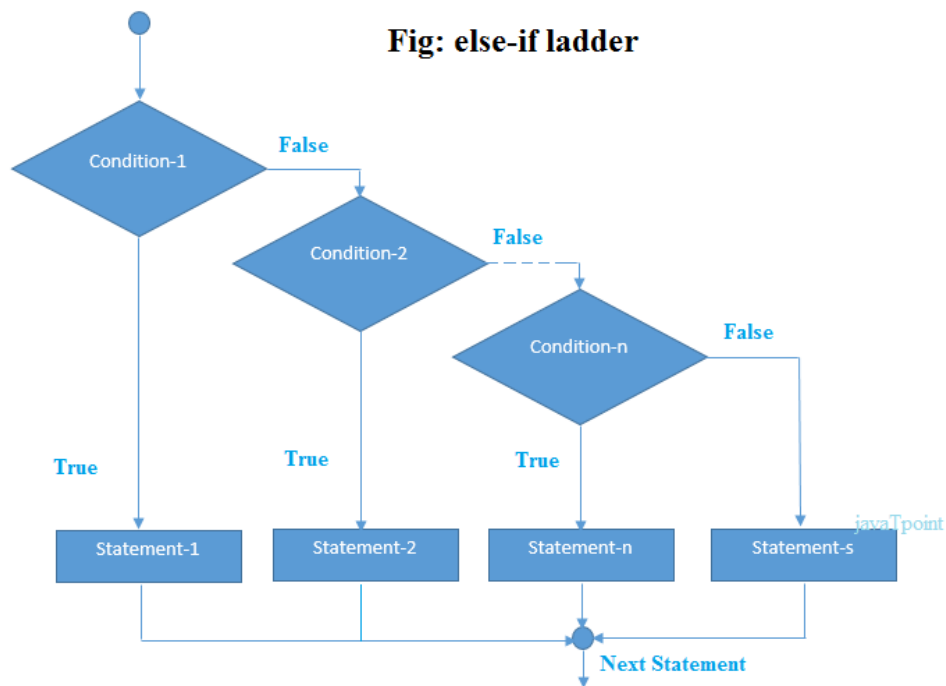


IF-ELSE-IF LADDER STATEMENT

- ✓ *The C++ if-else-if ladder statement executes one condition from multiple statements.*

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
..
else{
//code to be executed if all the conditions are false
}
```

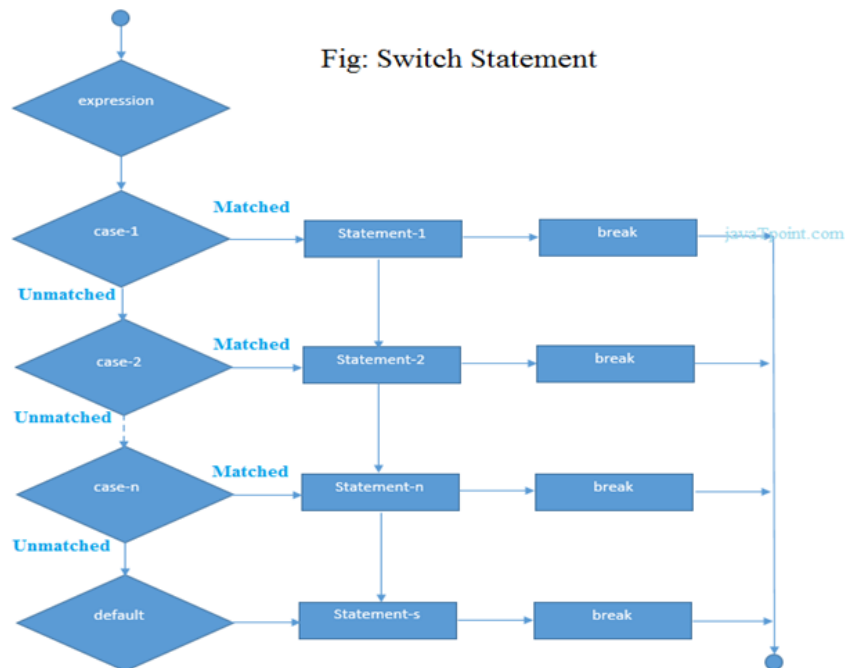

Fig: else-if ladder



C++ SWITCH

- ✓ *The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.*

```
switch(expression){  
case value1:  
//code to be executed;  
break;  
case value2:  
//code to be executed;  
break;  
.....  
default:  
//code to be executed if all cases are not matched;  
break;  
}
```



ITERATION STATEMENT:

- ✓ The *loop / iteration statement* execute the set of *statements repeatedly* until the *condition is true*.
- ✓ They are three statements. They are as follows:
 - *For*
 - *While*
 - *dowhile*

For loop:

- ✓ The C++ *for loop* is used *to iterate a part* of the program *several times*. If the number of *iteration is fixed*, it is recommended to use *for loop than while or do-while loops*.
- ✓ The C++ for loop is *same as C/C#*. We can *initialize variable, check condition* and *increment/decrement value*.

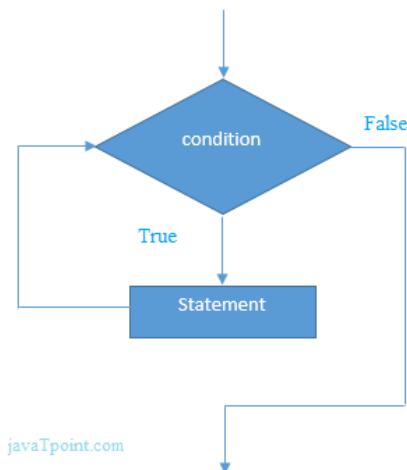
```
#include <iostream>
using namespace std;
int main() {
    for(int i=1;i<=10;i++){
        cout<<i <<"\n";
    }
}
```

o/p:
1 2 3 4 5 6 7 8 9 10

```
for(initialization; condition; increment;)
{
    Statements;
}
```

While loop:

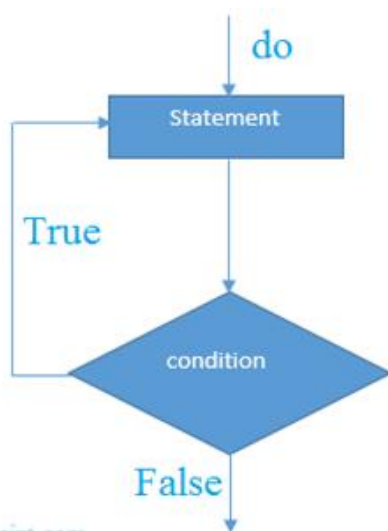
- ✓ In C++, while loop is used to *iterate a part of the program several times*. If the number of *iteration is n*
- ✓ *of fixed, it is recommended to use while loop than for loop.*



```
while(condition)
{
    statements;
}
```

C++ DO-WHILE LOOP

- ✓ The C++ *do-while loop* is used to *iterate a part of the program several times*.
- ✓ If the number of *iteration* is not *fixed* and you must have *to execute the loop* at least once, it is recommended *to use do-while loop*.
- ✓ The C++ *do-while loop is executed* at least *once because condition* is checked after *loop body*.



```
do
{
    Statements;
} while(condition);
```

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    do{
        cout<<i<<"\n";
        i++;
    } while (i <= 10);
}
```

o/p:
1 2 3 4 5 6 7 8 9 10

JUMP STATEMENTS :

BREAK:

- ✓ The C++ break is used to *break loop* or *switch statement*. It *breaks the current flow* of the *program at the given condition*. In case of *inner loop*, it *breaks only inner loop*.

```
jump-statement;  
break;
```

CONTINUE

- ✓ The C++ *continue statement* is used to *continue loop*.
- ✓ It continues *the current flow* of *the program* and *skips the remaining* code at specified condition.
- ✓ In case of inner loop, *it continues only inner loop*.

```
jump-statement;  
continue;
```

GOTO STATEMENT

- ✓ The C++ *goto* statement is also *known as jump statement*. It is used to *transfer control* to the *other part of the program*.
- ✓ It *unconditionally* jumps to the *specified label*.
- ✓ It can be used to *transfer control* from deeply *nested loop* or *switch case label*.

Standard output (cout)

- ✓ On most program environments, the standard output by default is the screen, and the *C++ stream object defined to access it is cout*.
- ✓ For formatted output operations, cout is used together with the *insertion operator*, which is written as << (i.e., two "less than" signs).

```
1 cout << "Output sentence"; // prints Output sentence on screen
2 cout << 120;           // prints number 120 on screen
3 cout << x;             // prints the value of x on screen
```

Standard input (cin)

- ✓ In most program environments, the standard input by default is the keyboard, and the *C++ stream object defined to access it is cin*.
- ✓ For formatted input operations, cin is used together with the extraction operator, which is written as >> (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored. For example:

```
1 int age;
2 cin >> age;
```

C++ STORAGE CLASSES

- ✓ *Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.*
- ✓ *Lifetime* refers to the *period during* which the *variable remains* active and *visibility* refers to the *module of a program* in which the variable is accessible.

There are five types of storage classes, which can be used in a C++ program

1. Automatic
2. Register
3. Static
4. External
5. Mutable

CORE COURSE II PROGRAMMING IN C++

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

UNIT II

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

2.1 CLASSES AND OBJECTS:

2.1.1 CLASSES:

- ✓ A *class* is a c++ mechanism that *enables user to create and implement user-defined data* type, according to the *needs of the problem to be solved*.
- ✓ It allows the *data to be hidden* , if necessary *from external use*.
- ✓ A *class* is a *user defined data type*. It is a *template of an object*.
- ✓ A class contain *data member and member function*.
- ✓ Generally a class is specification has two parts they are as follows:
 - *Class Declaration*
 - *Class function Definitions*
- ✓ The *class declaration describes the type and scope of its member*. The *class function definitions describe how the class functions are implemented*.
- ✓ The general form of *class declaration is as follows*:

```
Class class_name
{
Private:
    Variable declaration;
    Function declaration;
Public:
    Variable declaration;
    Function declaration;
}
```

- ✓ Here the *keyword class* specifies that follows is an *abstract data of type class_name*.
- ✓ The *functions, variables* are collectively called as *class members*.
- ✓ They are usually groped under two section namely
 - *Public*
 - *Private*

CLASS MEMBERS:

- ✓ A class may *declare data members, member function, within its scope*, referred to as *class members*.
- ✓ A class may also declare type names with its scope, referred to as nested type.

- ✓ A class member should not be declared with storage classes as *auto,extern (or) register* but it may be declared as *static*.
- ✓ Data and functions are members.
- ✓ Data Members and methods must be declared within the class definition.
- ✓ Private member functions and data can not be accessed out side of class and *only accessed within a class*.
- ✓ The public access_specifier allows functions or data to be accessible to other *parts of your program*.
- ✓ The *protected access_specifier* is needed only *when inheritance is involved*.

Example:

```
Class A
{
    int i; // i is a data member of class A
    int j; // j is a data member of class A
    int i; // Error redefinition of i
}
```

- ✓ A member cannot be redeclared within a class.
- ✓ No member can be added elsewhere other than in the class definition.

OBJECTS:

CREATING OBJECTS:

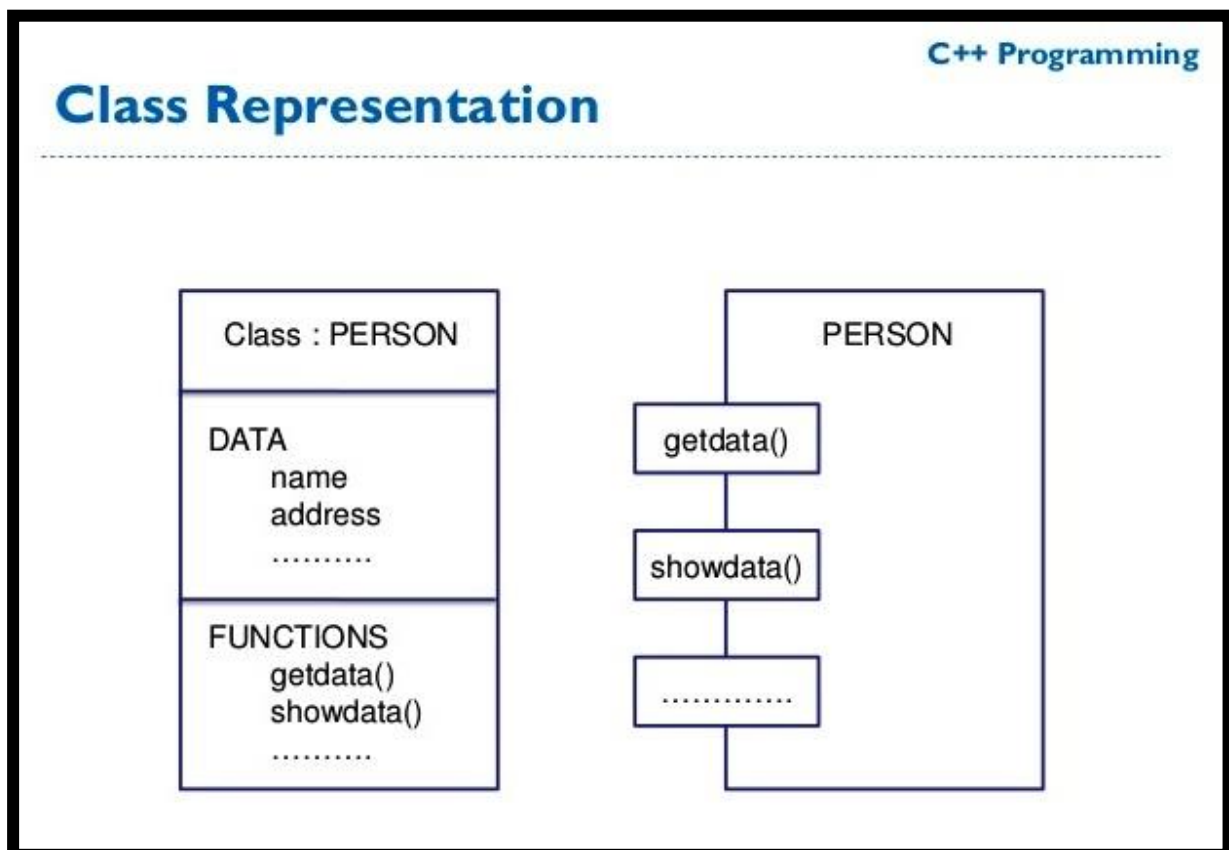
- ✓ In C++, *Object is a real world entity*, for example, *chair, car, pen, mobile, laptop* etc.
- ✓ In other words, *object is an entity that has state and behavior*. Here, state means data and *behavior means functionality*.
- ✓ Object is a *runtime entity, it is created at runtime*.
- ✓ Object is an *instance of a class*. All the *members of the class can be accessed through object*.

- ✓ Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1; //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

- ✓ Once the class declared we can create variables of that type by using the class name



- ✓ *Declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage.*
- ✓ Objects can also be created when a class is defined by placing their names immediately after closing brace.

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

ACCESSING CLASS MEMBER:

- ✓ The *main()* cannot contain statements that access *class members directly*. Class members can be *accessed only by an object of that class*.
- ✓ To access *class members*, use the *dot (.) operator*. The *dot operator links the name of an object with the name of a member*.
- ✓ The general form of the dot operator is shown here:

```
Object-name.function-name (actual arguments);
```

- ✓ For eg:

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to number and 75.5 to cost of the object x by implementing the getdata() function.

DEFINING MEMBER FUNCTIONS:

- ✓ Member function can be *defined in to two places* they are as follows:
 - *Outside the class definition*
 - *Inside the class definition*
- ✓ The *general form of an member function* is as follows:

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

C++ Programming

Member Functions

Defining Function inside Class

```
class demo
{
    int no1, no2, add;
public:
    //inline function
    void getdata() //inside
    class function
    {
        cout << "Enter 2 nos: "
    ;
        cin >> no1 >> no2;
    }
    void showdata()
    {
        cout << "Sum is: " <<
        add;
    }
};
```

▶ www.ecti.co.in

Defining Function outside Class

```
class demo
{
    int no1, no2, add;
public:
    void getdata(); //prototype
    declaration
    void showdata()
    {
        cout << "Sum is: " <<
        add;
    }
};
void demo :: getdata()
{
    cout << "Enter 2 nos: " ;
    cin >> no1 >> no2;
}
```

ENVISION
Computer Training Institute

//

Program to illustrate the working of objects and class in C++ Programming

```
#include <iostream>
using namespace std;
class Test
{
    private:
        int data1;
        float data2;
    public:
        void insertIntegerData(int d)
        {
            data1 = d;
            cout << "Number: " << data1;
        }
        float insertFloatData()
        {
            cout << "\nEnter data: ";
            cin >> data2;
            return data2;
        }
};

int main()
{
    Test o1, o2;
    float secondDataOfObject2;
    o1.insertIntegerData(12);
    secondDataOfObject2 = o2.insertFloatData();

    cout << "You entered " << secondDataOfObject2;
    return 0;
}
```

Number: 12

Enter data: 23.3

You entered 23.3

ARRAY WITHIN A CLASS:

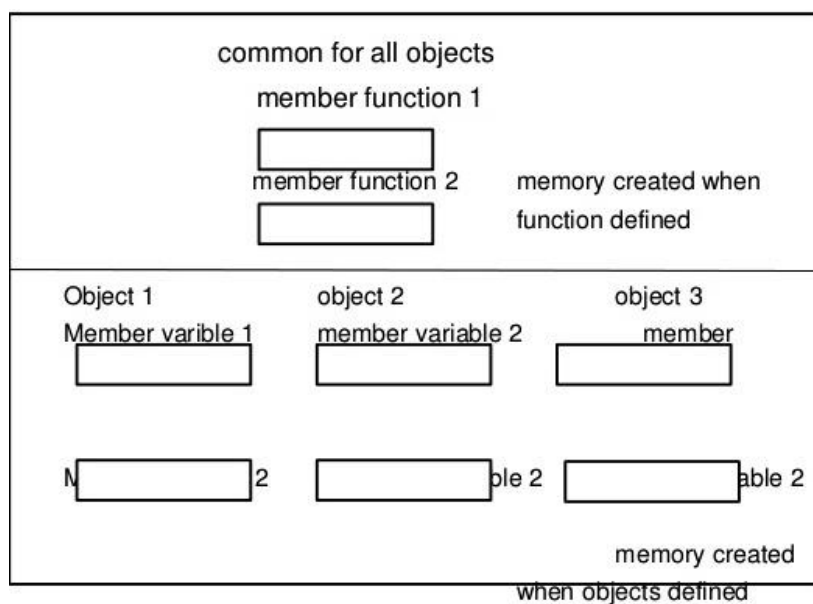
- ✓ The array can be *used as member variable in a class.*

Array within class:

- The arrays can be used as member variable in a class.
- An array is collection of same data type or group of data item that store in a common name.
- Syntax:
`data_type name[size]={list of value};`
 Like
`int number[4]={1,2,3,4};`

MEMORY ALLOCATION FOR OBJECTS:

- ✓ Actually, the *member functions* are *created* and *placed* in the *memory space only once* when *they area defined as a part of class specification.*
- ✓ Since all the *objects belonging* to that class use the *same member functions*, no *separate space* is allocated for *member functions* when the *objects are created.*
- ✓ Only space for *member variable* is allocated *separately for each object.*



Static Class Members

- ✓ *Static class members* can be defined using **static** keyword.
- ✓ When *we declare a member of a class* as static it means *no matter how many objects of the class are created*, there is *only one copy of the static member which is shared by all objects of the class*.

A) Static data member b) Static member functions

a) Static data member

The member variables in a class can be declared as static. The general form is,

```
static datatype membervariable;
```

Characteristics

- At *once the object for the class* are declared, the *static member variables* are *initialized to zero*.
- For each *static variable separate memory locations* are allocated and it is common for all *objects in the class*.
- Each *static member variable* must be defined outside the class using scope *resolution operator*.

```
datatype classname::membervariable;
```

- For *static member variable we can give initial value*.

```
datatype classname::membervariable=value;
```

b) Static member function

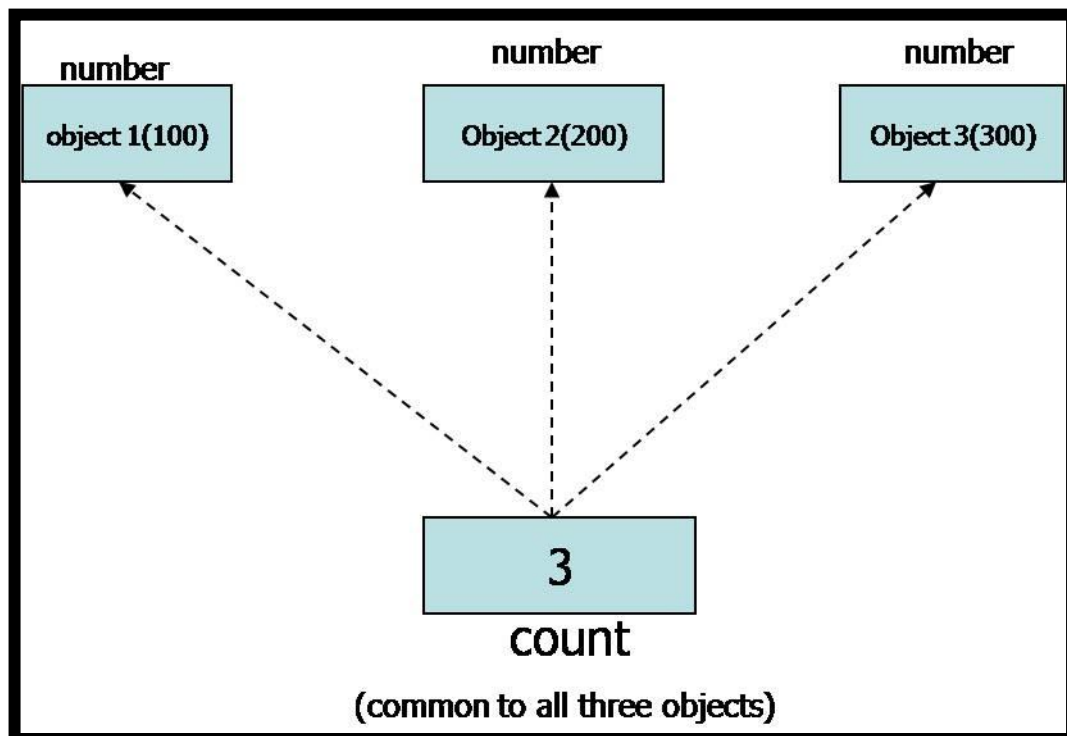
- ✓ It can be declared using static. *It can only access the* static members *declared in the same class*. Static member function can be called using *classname*.

General form: `classname::memberfunction;`

Example:

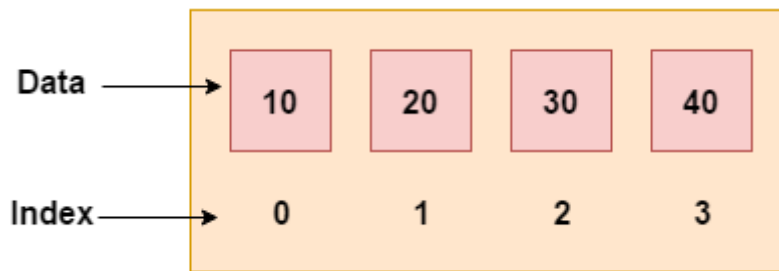
```
#include<iostream.h>
#include<conio.h>
class sample
{
private:
static int st;
public:
void incre()
{
```

```
st++;
}
static void print()
{
cout<<"static = "<<st;
}
};
int sample::st;
void main()
{
clrscr();
sample s1,s2;
sample::print();
s1.incre();
s2.incre();
sample::print();
getch();
}
```



C++ ARRAYS

- ✓ Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.
- ✓ In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.



ADVANTAGES OF C++ ARRAY

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

C++ Array of Objects

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of class or object. As we know, an array is a collection of similar type, therefore an array can be a collection of class type.

• Syntax for Array of object

```
class class-name {  
    datatype var1;  
    datatype varN;  
  
    method1();  
    methodN();  
};  
class-name obj[ size ];
```

Eg:

```
Employee manager [3];           // array of manager
```

```
Employee foreman [15];         // array of foreman
```

```
Employee worker [75];          // array of worker
```

Friend Function

- ✓ In a *class*, *private member* variables can be accessed only by the *member functions in that class*.
- ✓ Friend function can be *used to access* all private and *protected members of the class for which it is a friend*.
- ✓ Friend function should *be a non-member class*.

General form: `friend return_type function_name(args);`

Characteristics:

- *The keyword friend can only be used inside the class.*
- *More than one friend function can be declared in a class.*
- *A function can be friend to more than one class.*
- *Friend function definition should not contain the scope operator.*

CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS

- ✓ A *constructor* is a “*special “ member* whose task is to *initialize the objects of its class*.
- ✓ A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.
- ✓ Constructor *functions are invoked automatically* when an object for *a class is created*.

CHARACTERISTICS OF CONSTRUCTOR:

- A constructor has the same name as the class itself.
- It is invoked automatically when the objects are created.
- It should be defined public.
- They can have default argument.
- It can be overload.
- It has no return type even void.
- We cannot refer to their address.
- They cannot be inherited.
- They cannot be virtual.

Types of Constructor

- i) Default Constructor / Constructor with no argument
- ii) Parameterized Constructor
- iii) Multiple Constructor / Overloading Constructor
- iv) Copy Constructor

DEFAULT CONSTRUCTOR / CONSTRUCTOR WITH NO ARGUMENT :

- ✓ A constructor which *has no argument* is known as *default constructor*. It is invoked at the *time of creating object*.

For example:

```
#include <iostream>
using namespace std;
class Employee
{
    public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

o/p:

```
Default Constructor Invoked
Default Constructor Invoked
```

PARAMETERIZED CONSTRUCTOR

- ✓ A *constructor which has parameters* is called *parameterized constructor*. It is used to provide *different values to distinct objects*.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
    public:
        int id;//data member (also instance variable)
        string name;//data member(also instance variable)
        float salary;
        Employee(int i, string n, float s)
        {
            id = i;
            name = n;
            salary = s;
        }
        void display()
        {
            cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an
object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

O/P:

```
101  Sonoo  890000
102  Nakul  59000
```

MULTIPLE CONSTRUCTOR / OVERLOADING CONSTRUCTOR :

✓ If a class has *more than one constructor*, then it is called Multiple Constructor.

Example:

```
#include<iostream.h>
#include<conio.h>
class time
{
private:
int hours;
int minutes
int seconds;
public:
time() // default constructor
{
hours=0;
minutes=0;
seconds=0;
}
time(int h) // one argument constructor
{
hours=h;
minutes=0;
seconds=0;
}
time(int h,int m) // two argument constructor
{
hours=h;
minutes=m;
seconds=0;
}
time(int h,int m,int s) // three argument constructor
{
hours=h;
minutes=m;
seconds=s;
}
void showtime()
{
cout<<hours;
cout<<minutes;
cout<<seconds;
}
};
void main()
{
clrscr();
time t1;
time t2(12);
```

```
time t3(12,15);
time t4(12,15,15);
t1.showtime();
t2.showtime();
t3.showtime();
t4.showtime();
getch();
```

Copy Constructor

- ✓ It is used to *declare and initialize an object* with the values *from another object*.

```
#include<iostream.h>           {
#include<conio.h>              cout<<a;
class data                     }
{                               };
private:                       void main()
int a;                          {
public:                          clrscr();
data(int x)                     data d1(100);
{                                 d1.print();
a=x;                             data d2=d1;
}                                 d2.print();
data(data &ob)                getch();
{                                 }
a=ob.a;
}
void print()
```

CONSTRUCTORS WITH DEFAULT ARGUMENTS:

- ✓ *It is possible to define constructors with default arguments.*

for eg:

```
complex (float real, float imag=0);
```

The default value of the argument imag is 0. Then the statement

```
Complex c(5.0);
```

Assigns the values 5.0 to the real variable and 0.0 to imag (by default)

DYNAMIC INITIALIZATION OF OBJECTS:

- ✓ Objects can be *initialized dynamically too*. That is to say, *the initial value of an object may be provided during run time*.

- ✓ One advantage of *dynamic initialization* is that we can provide various initialization formats, using *overloaded constructors*.

STATIC V/S DYNAMIC INITIALIZATION

Static initialization means when value of variable is already known and static initialization takes place at compile time.

For ex. `int a=10;`

Dynamic initialization means when value of a variable is decided at run time i.e., inside execution

For dynamic initialization we use **CIN**

DESTRUCTORS

- ✓ As the name implies, is used to *destroy the objects* that have been *created by a constructor*
- ✓ A destructor works *opposite to constructor*; it *destructs the objects of classes*. It can be defined *only once in a class*. Like constructors, it is *invoked automatically*.
- ✓ A destructor is *defined like constructor*. It must have same name as class. *But it is prefixed with a tilde sign (~)*.

```
~destructorname ()
```

```
{
```

```
Statement;
```

```
}
```

Characteristics:

- Destructor has the same name as the classname.
- No argument can be provided to a destructor.
- It won't return any value.
- They cannot be inherited.
- It may not be static.

OPERATOR OVERLOADING:

- ✓ It is an *important technique that c++ permits* us to *add 2 variables* of user defined types with *same syntax* that is applied to *basic types*.
- ✓ This means that *c++* has the *ability to provide the operators* with a *special meaning for data type*. The mechanism of *giving such special meaning* to an *operator* is known *as operator overloading*.

Operator Overloading

- Almost any operator can be overloaded in C++.
- However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows.
 - 1.scope operator - ::
 - 2.sizeof
 - 3.member selector - .
 - 4.member pointer selector - *
 - 5.ternary operator - ?:

Defining operator overloading

- The general form of an operator function is:

```
return-type class-name :: operator op (argList)
{
    function body // task defined.
}
```

 - where **return-type** is the type of value returned by the specified operation.
 - **op** is the operator being overloaded.
 - **operator op** is the function name, where operator is a keyword.

Nitesh Dubli, Lecturer @ Patkar-Varde College, Goregaon(W).

PROCESS OF OPERATOR OVERLOADING:

- ✓ Create a *class that define* the *data type* that is to be used in the *overloading operation*.
- ✓ Declare the operator function operator *op()* in the part of the class.

- ✓ It may *be either friend or member function.*
- ✓ Define the operator function *to implement the required operations.*

Overloaded operator function can be invoked by expression such as

op X or X op

For unary operators

X op y

For binary operators

Operator op(x)

C++ OVERLOADING (FUNCTION AND OPERATOR)

- ✓ If we create *two or more members having same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:*
 - methods,
 - constructors, and
 - indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- **Function overloading**
- **Operators overloading**

C++ FUNCTION OVERLOADING

- ✓ *Having two or more function with same name but different in parameters, is known as function overloading in C++.*
- ✓ *The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for same action.*

OVERLOADING UNARY OPERATORS:

Overloading Unary Operators

- Overloading unary operators
 - Can overload as non-static member function with no arguments
 - Can overload as global function with one argument
 - Argument must be class object or reference to class object
 - Remember, static functions only access static data

EXAMPLE FOR UNARY OPERATOR :

Overloading Unary Operators (++/--)

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

class complex
{
    int a,b,c;
public:
    complex(){}
    void getvalue()
    {
        cout<<"Enter the Two Numbers:";
        cin>>a>>b;
    }
    void operator++()
    {
        a++;
        b++;
    }
    void operator--()
    {
        a--;
        b--;
    }
    void display()
    {
        cout<<a<<" +\t"<<b<<"i"<<endl;
    }
};
```


Binary Operator Overloading

- At least one parameter must be of the enclosing type.
- You may overload as many times as you like with different parameter types.
- You may return any type.
- No “ref” or “out” parameters.
- Two Types:-
 1. Binary Arithmetic Operators
 2. Binary Comparison Operators

Overloading Unary Operator using friend Functions

- When you overload a unary operator using a friend function you would have to pass one argument to the friend function.
- **Beware:** When you use friend functions, they will not have the ‘this’ pointer.
- If you attempt to modify some value of an object passed as argument, then the friend function actually only operates on a copy (it does not act on the original object).
- This is because it is passed by value (and not as reference).

RULES OF OPERATOR OVERLOADING IN C++

Rules for overloading operators



- Only existing operators can be overloaded. New operators cannot be created.
- At least one operand (user defined type).
- Cannot change the basic meaning of the operator.
- Follow the syntax rules of the original operators.
- Some operators cannot be overloaded.
- Cannot use friend function to overload certain operators.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- Binary arithmetic operators must explicitly return a value.

MANIPULATION OF STRING USING OPERATORS:

Manipulation of Strings using Operators

- There are lot of limitations in string manipulation in C as well as in C++.
- Implementation of strings require character arrays, pointers and string functions.
- C++ permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to other built-in data types.
- ANSI C++ committee has added a new class called string to the C++ class library that supports all kinds of string manipulations.

20

By:-Gourav Kottawar

Manipulating Strings

String Function

Functions:	Description
strlen	Find the Length of String
strlwr	Converts a String to Lowercase
strupr	Converts a String to Uppercase
strcat	Appends one String at the end of Another String
strncat	Appends first n Characters of a String at the end of Another string
strcpy	Copies a String to Another String

String conversion

atof	Convert string to double (function)
atoi	Convert string to integer (function)
atol	Convert string to long integer (function)
atoll <small>C++11</small>	Convert string to long long integer (function)
strtod	Convert string to double (function)
strtof <small>C++11</small>	Convert string to float (function)
strtol	Convert string to long integer (function)
strtold <small>C++11</small>	Convert string to long double (function)
strtoll <small>C++11</small>	Convert string to long long integer (function)
strtoul	Convert string to unsigned long integer (function)
strtoull <small>C++11</small>	Convert string to unsigned long long integer (function)

TYPE CONVERSIONS:

- ✓ The *assignment operation causes the automatic type conversion.*
- ✓ The type of *data to the right of an assignment operator is automatically converted to the type of the variable on the left.*

For eg:

```
int m;  
float x=3.145;  
m=x
```

Here convert *x* to an integer before its value is *assigned* to *m*. Thus the *fractional part is truncated*. The type conversions are automatic as long as the data types involved are built in types.

- ✓ *Three types of situation* might arise in the data conversion between incompatible types they are as follows:
 - *Conversion from basic type to class type.*
 - *Conversion from class type to basic type.*
 - *Conversion from one class type to another class type.*

CONVERSION FROM BASIC TYPE TO CLASS TYPE

Basic to Class Type

A constructor to build a string type object from a char * type variable.

```
string :: string(char *a)  
{  
    length = strlen(a);  
    P = new char[length+1];  
    strcpy(P,a);  
}
```

The variables length and p are data members of the class string.

CONVERSION FROM CLASS TYPE TO BASIC TYPE

Class To Basic Type

A constructor function do not support type conversion from a class type to a basic type.

An overloaded **casting operator** is used to convert a class type data to a basic type.

It is also referred to as **conversion function**.

```
operator typename( )  
{  
    ...  
    ... ( function statements )  
    ...  
}
```

This function converts a **class type** data to **typename**.

CONVERSION FROM ONE CLASS TYPE TO ANOTHER CLASS TYPE

Learners Support Publications www.lsp4you.com

One Class To Another Class Type

continue...



operator typename()

- Converts the class object of which it is a member to typename.
- The typename may be a built-in type or a user-defined one.
- In the case of conversions between objects, typename refers to the destination class.
- When a class needs to be converted, a casting operator function can be used at the source class.
- The conversion takes place in the source class and the result is given to the destination class object.

syntax for Conversion from one class type to another class type

Learners Support Publications www.lsp4you.com

One Class To Another Class Type



```
objX = objY ; // objects of different types
```

- **objX** is an object of class **X** and **objY** is an object of class **Y**.
- The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**.
- Conversion is takes place from **class Y** to **class X**.
- **Y** is known as **source class**.
- **X** is known as **destination class**.

“You have to dream before
Your dreams can come true”

-Dr.A.P.J.Abdul Kalam

UNIT III

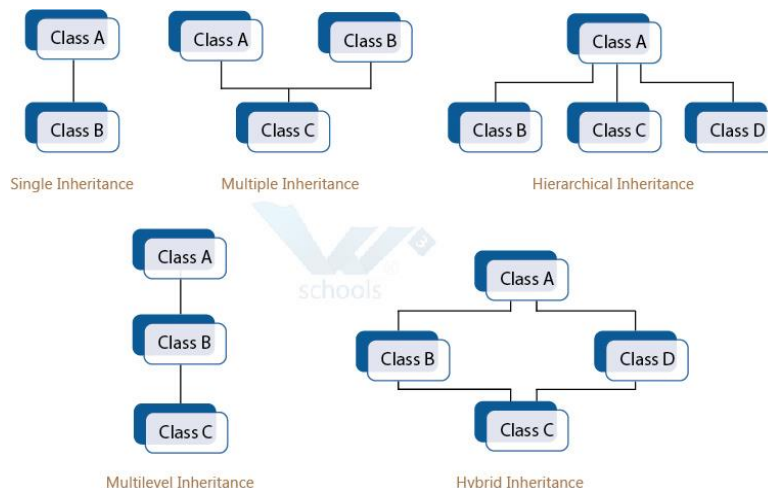
Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

INHERITANCE:

- ✓ The c++ *classes can be reused in several ways*. Once a *class has been* written and tested it can be *adopted by other programmers* to *suit their requirements*.
- ✓ The *mechanism of deriving a new class from an old one* is called as *inheritance*.
- ✓ The *class which from which we derive* is referred to as *the base class, parent class or super class*
- ✓ The *class that we derive* is referred as *the derived class, child class or sub class*.
- ✓ The *derived class* is always as *big as the base class*, since the derived class inherits all the *properties of the base class* and *can add properties of its own*.
- ✓ The *act of creating a derived class by is something* is referred as *sub classing*.

FEATURES/ADVANTAGES OF INHERITANCE

- Reusability
- Extensibility
- Saves time and effort
- Reliability
- Overriding
- ✓ The inheritance mechanism can be broadly classified into five types they area as follows:
 - Single inheritance
 - Multiple inheritance
 - Hierarchical inheritance
 - Multilevel inheritance
 - Hybrid inheritance



SINGLE INHERITANCE:

- ✓ In *single inheritance*, there is *only one base class and one derived class*. The *Derived class gets inherited from its base class*.
- ✓ This is the *simplest form of inheritance*. In the above figure, fig(a) is the diagram for single inheritance.

Syntax:

```
class base_class
{
.....
};

class derived_class:visibilitymode base_class
{
.....
};
```

MULTIPLE INHERITANCE

- ✓ In this type of inheritance, a *single derived class may inherit from two or more base classes*.
- ✓ In the above list of figures, *fig(b) is the structure of Multiple Inheritance*.

```
class base_class1
{
.....
};

class base_class2
{
.....
};

class derived_class:visibilitymode base_class1,base_class2
{
.....
};
```


HIERARCHICAL INHERITANCE

- ✓ In this type of *inheritance*, *multiple derived classes get inherited from a single base class*.
- ✓ In the above list of figures, *fig(c) is the structure of Hierarchical Inheritance*.

Syntax:

```
class base_classname
{
properties; methods;
};
class derived_class1 : visibility_mode base_classname
{
properties; methods;
};
    class derived_class2 : visibility_mode base_classname
    {
properties; methods;
    };
    ... ..
class derived_classN : visibility_mode base_classname
{
properties; methods;
};
```

MULTILEVEL INHERITANCE

- ✓ The *classes can also be derived from the classes* that are *already derived*. This type of *inheritance is called multilevel inheritance*.
- ✓ In the above list of figures, *fig(d) is the structure of Multilevel Inheritance*

Syntax:

```
class base_class
{
.....
};
```

```
class derived_class1:visibilitymode base_class
{
.....
};

class derived_class2:visibilitymode derived_class1
{
.....
};
```

HYBRID INHERITANCE

- ✓ This is a *Mixture of two or More Inheritance and in this Inheritance*, a Code May *Contains two or Three types of inheritance in Single Code*.
- ✓ In the above figure, the *fig(e) is the diagram for Hybrid inheritance*.

```
class base1
{
.....
};

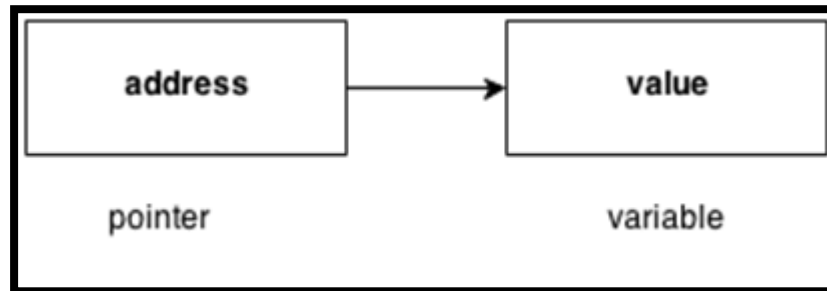
class derived1:visibilitymode base1
{
.....
};

class base2
{
.....
};

class derived2:visibilitymode derived1,visibilitymode base2
{
....
};
```

POINTERS

- ✓ Pointers are *extremely powerful programming tool* that can make things easier and help to *increase the efficiency of a program* and allow programmers to *handle an unlimited amount of data*.
- ✓ In other words, a *pointer variable holds the address of a memory location*.



DECLARING A POINTER

- ✓ The pointer in C++ language can be *declared using * (asterisk symbol)*.

Syntax:

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

- ✓ Following are the valid pointer declaration :

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

ADVANTAGE OF POINTER

- 1) Pointer *reduces the code and improves the performance*, it is used to *retrieving strings, trees etc. and used with arrays, structures and functions*.
- 2) We can return *multiple values from function using pointer*.
- 3) It makes you able to access *any memory location in the computer's memory*.

POINTER WITH ARRAYS AND STRINGS:

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

- ✓ In c++ language, we can *dynamically allocate memory using malloc() and calloc() functions where pointer is used.*

2) Arrays, Functions and Structures

- ✓ Pointers in c++ language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

For eg:

```
#include <iostream>
using namespace std;
int main()
{
int number=30;
int * p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
return 0;
}
```

O/p:

Address of number variable is:0x7ffccc8724c4

Address of p variable is:0x7ffccc8724c4

Value of p variable is:30

Types of pointer:

There are following few important pointer concepts which should be clear to a C++ programmer :

Sr.No	Concept & Description
1	Null Pointers C++ supports null pointer, which is a constant with a value of zero defined in several standard libraries.
2	Pointer Arithmetic There are four arithmetic operators that can be used on pointers: ++, --, +, -
3	Pointers vs Arrays There is a close relationship between pointers and arrays.
4	Array of Pointers You can define arrays to hold a number of pointers.
5	Pointer to Pointer C++ allows you to have pointer on a pointer and so on.
6	Passing Pointers to Functions Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
7	Return Pointer from Functions C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

MANIPULATION OF POINTERS:

- ✓ We can manipulate a *pointer with the indirection operator i.e “*”* which is known as *dereference operator*.
- ✓ With this operator, we can indirectly access the *data variable content*.

Syntax:

```
*pointer_varaiable
```

For eg:

```
#include<iostream.h>

int main()
{
    int a=10;

    int *ptr;

    clrscr();

    ptr=&a;

    cout<<"the value of a is: "<<*ptr;

    *ptr=*ptr+a;

    cout<<"\n The revised value of a is "<<a;

    getch();

    return(0);
}
```

O/p:

```
The value of a is           :      10
```

```
The revised value of a is   :      20
```

ARRAY OF POINTERS:

- ✓ An array of pointers point to an array of data items.
- ✓ Each element of the pointer array points to an item of the data array.
- ✓ Data items can be accessed wither directly or by dereferencing the elements of pointer array.
- ✓ We can declare an array of pointer as follows:

```
int *inarray[10];
```

POINTERS TO FUNCTIONS:

- ✓ The *pointer to function is known as Callback function*.
- ✓ We can use these *function pointers to refer to a function*.
- ✓ Using function pointer, we can allow a c++ *program to select a function dynamically at run time*.
- ✓ Here the *function is passed as a pointer*.
- ✓ C++ provides *2 types* of function pointers 1. *Function pointers that point to static member functions* and 2. *Function pointers that point to non-static member functions*.
- ✓ These 2 function pointers are *incompatible with each other*
- ✓ We can declare a function pointer in c++ as follows:

```
data _type (*function_name) ();
```

Example:

```
int(*num_function(int x));
```

POINTER TO OBJECTS:

- ✓ As stated earlier , a pointer can point to an object created by a class. Consider the following statement
 - **item x;**
- ✓ where *item is a class and x is an object* defined to be of type item. Similarity we can define a pointer *it_ptr* of type item as *item*it_ptr;*

this Pointer:

- ✓ c++ uses a unique *keyword* called *this* to represent an *object that invokes a member function*.
- ✓ *this is a pointer* that points to the *object for which this function was called*.
- ✓ For eg: the *function call A.max()* will set *the pointer this to the address of the object A*.

Example:

```
class abc
{
    int a;
    ... ;
    ... ;
};
```

Here the private variable 'a' can be used directly inside a member function, like

a=123;

We can also use the following statement to do the same job as *this* →

VIRTUAL FUNCTIONS:

- ✓ A *virtual function* is a special form of *member function* that is declared within a base *class and redefined by a derived class*.
- ✓ The keyword "*virtual*" is used to create a *virtual function*, precede the *function's declaration in the base class*.
- ✓ If a class includes *a virtual function and if it gets inherited*, the virtual class redefines a *virtual function to go with its own need*.
- ✓ In other words, a *virtual function is a function* which gets *override in the derived class and instructs the C++ compiler for executing late binding on that function*.
- ✓ A *function call is resolved at runtime in late binding and so compiler determines the type of object at runtime*.

RULES FOR VIRTUAL FUNCTIONS:

- 1) The virtual functions *must be members of some class*.
- 2) They *cannot be* static members.
- 3) They are *accessed* by using *object pointers*.
- 4) A virtual function *can be a friend of another class*.
- 5) Virtual function in a *base class* must be defined *even though it may not be used*.
- 6) The prototype of *base class version* of a virtual function and all the derived class *versions must be identical*.
- 7) We cannot have *virtual constructors*, *but we can have virtual destructors*.
- 8) Base *pointer can point to any type of derived* object. But we cannot use a pointer of a derived *class to access an object of base type*.
- 9) Incrementing and decrementing of the *base pointer* will *not point* to next object of *derived class*.
- 10) It will get *incremented or decremented only relative to its base type*.
- 11) If a *virtual function* is defined in the base class, *it not be redefined in the derived* class, then *calls will invoke the base function*.

PURE VIRTUAL FUNCTIONS:

- ✓ The function inside the base class is seldom used for performing any task.
- ✓ It only serves a placeholder.
- ✓ For example: if we have not defined any object of class media and therefore the function display() in the base class has been defined "empty".
- ✓ Such functions are called "do-nothing" functions.
- ✓ A "do-nothing" function may be defined as follows:

Virtual void display ()=0;

ABSTRACT BASE CLASS

- ✓ 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*.

CHARACTERISTICS

- It *cannot be* instantiated, but *pointers* and *references* of *Abstract class type* can be *created*.
- It can have *normal functions and variables along with a pure virtual function*.

- They are mainly used for *Up casting*, so that its *derived classes* can use its interface.
- Classes inheriting an *Abstract class must implement* all pure virtual functions, or else they will *become abstract too*.

POLYMORPHISM :

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

There are two types of polymorphism in C++:

- **Compile time polymorphism:** It is *achieved by function overloading* and operator *overloading* which is also known as *static binding or early binding*.
- **Runtime polymorphism:** It is *achieved by method overriding* which is also known as *dynamic binding or late binding*.

In programming background, *polymorphism can be broadly divided into two parts*. These are:

1. Static Polymorphism
2. Dynamic Polymorphism.

STATIC POLYMORPHISM

- ✓ In static polymorphism or early binding, there you will get two subcategories like:
 - *Function overloading which is the process of using the same name for two or more functions.*
 - *Operator overloading which is the process of using the same operator for two or more operands.*

CODE SNIPPET FOR FUNCTION OVERLOADING

```
class funcO1 {  
public:  
    funcO1 ();  
    funcO1 (int i);  
    int add(int a, int b);  
    int add(float a, float b);  
};
```

CODE SNIPPET FOR OPERATOR OVERLOADING

```
class calc {  
public:  
    // + operator overloading technique  
    int operator+(calc g);  
private:  
    int k;  
};
```

DYNAMIC POLYMORPHISM

- ✓ This refers to the entity *which changes its form depending on circumstances at runtime.*
- ✓ This concept can be *adopted as analogous to a chameleon changing its color at the sight of an approaching object.*

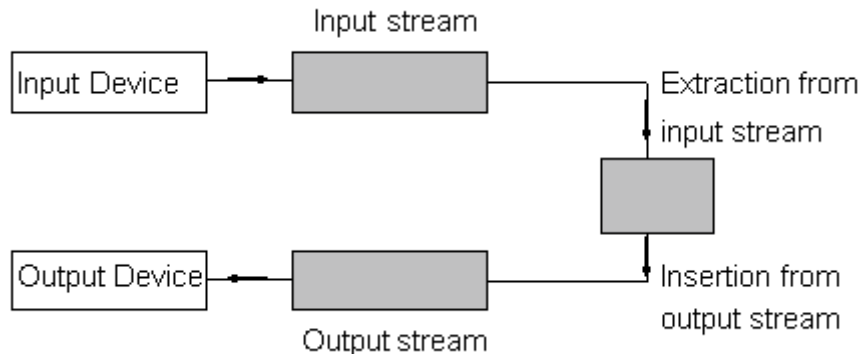
Unit IV

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

Managing Console I/O Operations

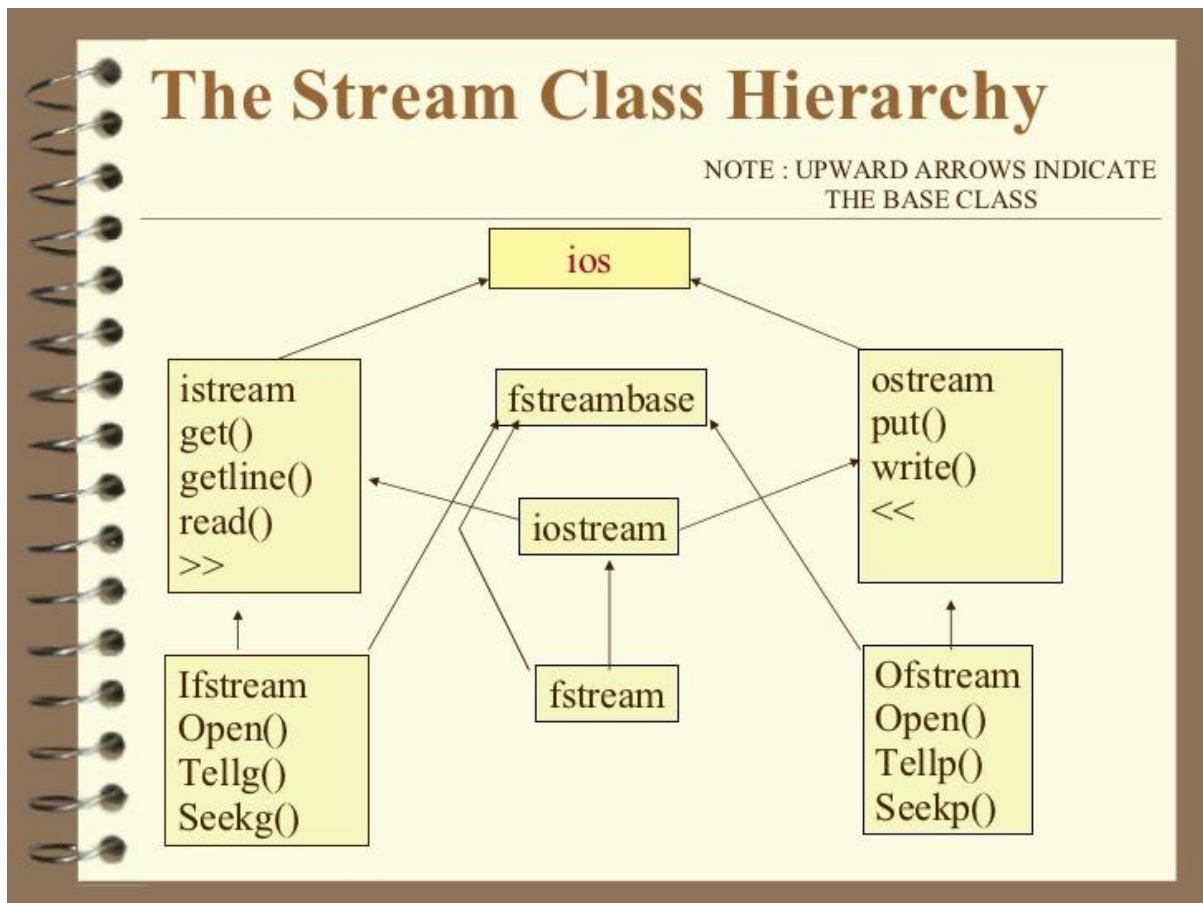
STREAMS:

- ✓ The I/O system in *c++* is *designed* to work with a wide *variety of device including terminals, disks, and tape drives*.
- ✓ Although *each device is very different*, the I/O system supplies *an interface* to the *programmer* that is *independent of the actual device being accessed*. This *Interface* is called *stream*.
- ✓ A stream is a *sequence of bytes*.
- ✓ It acts either as *a Source form* which the *input data* can be obtained or as a *destination* to which the *output data* can be sent.
- ✓ The *source stream* that provides *data to the program* is called the *input stream* .
- ✓ the *destination stream* that *receives the output from the program* is called as *output stream*



C++ STREAMS:

- ✓ The *c++* I/O system contains a *hierarchy of classes* that are used to *define various streams to deal with both the console and disk files*. These classes are called as *stream classes*.
- ✓ The *classes are* declared in the *header file iostream* this file should be included in all the *programs that communicate with the console unit*.



Stream classed for console i/O operations

UNFORMATTED INPUT/OUTPUT OPERATIONS

- ✓ We have already used the *cin and cout* (pre-defined in *iostream* file) *for input and output of data of various types.*
- ✓ This has been made possible by *overloading the operators << and >>* to recognize all the basic C++ types.
 - cin standard input stream
 - cout standard output stream

Example:

```
#include <iostream>

using namespace std;

void main()

{

    int g;

    cin>>g;

    cout << "Output is: "<< g;

}
```

put() and get() functions

- ✓ The classes *istream* and *ostream* defines two *member functions* *get()* and *put()* respectively to *handle single character input/output operations*.

Get() function is of two types:

1. get(char *)
2. get(void)

Both of them can be used to fetch a character including a blank space, tab or new-line character.

Code snippet

```
char ch;

cin.get(ch);

while(ch != '\n')

{

    cout<<ch;

    cin.get(ch);

}
```

- ✓ Similarly, the function `put()`, a member of `ostream` class can be used to output a line of text character by character.

Example:

```
cout.put ('g');  
  
char ch;  
  
cout.put(ch);
```

`getline()` and `write()`

- ✓ We can *read and display lines* of text more efficiently using the *oriented input/output functions*. They are:

- `getline()`
- `write()`

- ✓ The *`getline()` function reads the entire line of texts that ends with a newline character*. The general form of `getline()` is:

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

- ✓ *The `write()` function displays the entire line of text and the general form of writing this function is:*

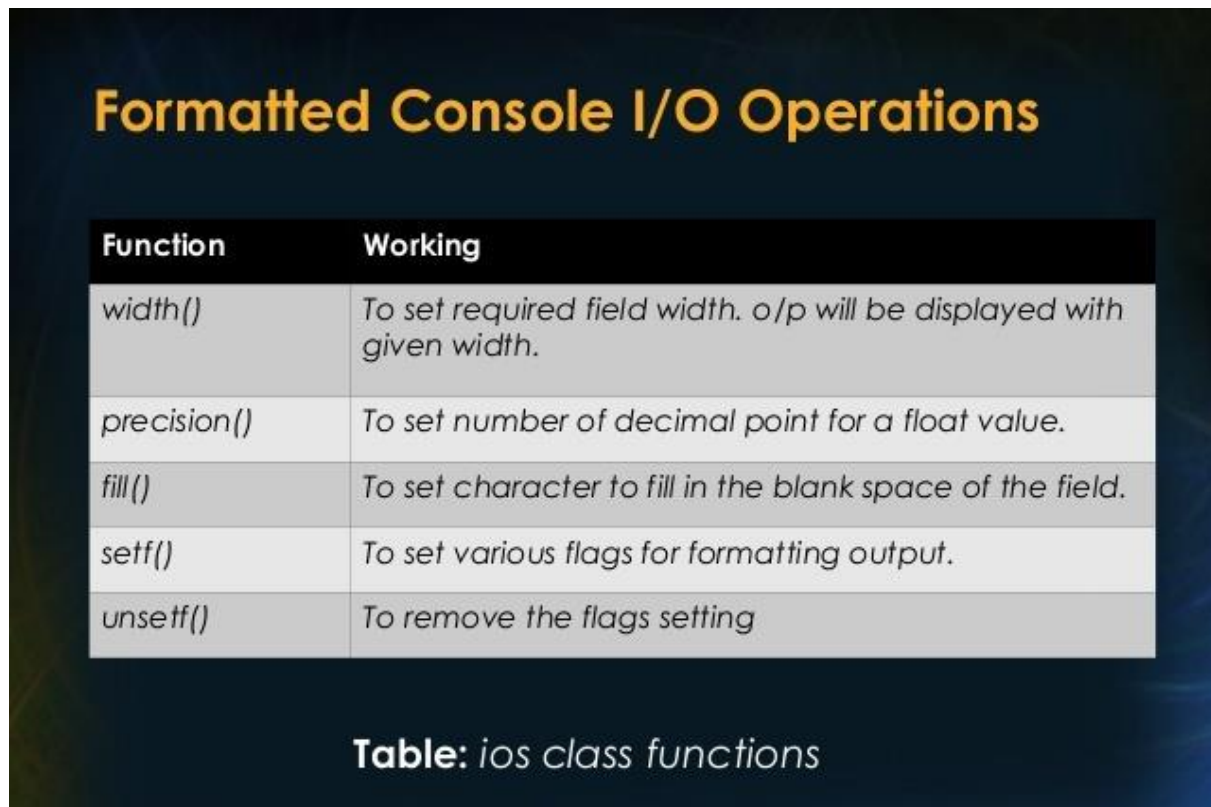
```
cout.write (line, size);
```

FORMATTED CONSOLE I/O OPERATIONS:

- ✓ C++ supports a number of features that could be used for *formatting the output*.

These features include:

- *ios class function and flags*
 - *Manipulators*
 - *User-defined output functions*
- ✓ The *ios* class contains a large of member functions that would help us to *format the output in a number of ways*. The most important ones among them are as follows:



Function	Working
<code>width()</code>	To set required field width. o/p will be displayed with given width.
<code>precision()</code>	To set number of decimal point for a float value.
<code>fill()</code>	To set character to fill in the blank space of the field.
<code>setf()</code>	To set various flags for formatting output.
<code>unsetf()</code>	To remove the flags setting

Table: ios class functions

Defining field width :`width()`

- ✓ We can use the *width()* function to define the *width of a field necessary* for the output of an item. Since it is a member function, we have to use an *object to invoke* it as follows:

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

- Where *w* is the *field width (number of columns)*. The output will be printed in *a field of w characters* wide at the right *end of the field*.

Setting Precision: precision()

- ✓ We can specify the *number of digits* to be displayed *after the decimal point* while *printing the floating point numbers*. This can be done by *using the precision()* member function as follows:

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

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

Example program for width and precision Manipulators:

```
#include<iostream>
void main()
{
float pi=22.0/7.0;
int I;
cout<<"value of pi:\n";
for(i=1;i<=10;i++)
{
cout.width(i+1);
cout.precision(i);
cout<<pi<<"\n";
}
}
```

```
o/p:
value of PI:
3.1
3.14
3.143
3.1429
3.14286
3.142857
3.1428571
3.14285707
3.142857075
3.1428570747
```

Filling and padding : fill()

- ✓ The unused *positions of the field* are filled with *white spaces, by default*. We can use *fill() function* to fill *the used positions by any desired character*.

```
cout.fill(ch)
```

for eg:

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

output:

↓

*	*	*	*	*	*	5	2	5	0
---	---	---	---	---	---	---	---	---	---

Formatting flags, bit – fields and setf();

- ✓ The setf() , a member *function of the ios class, can provide formatted operations.*
- ✓ The setf() (*setf stands for set flags*)

```
cout.setf (args1 , arg2)
```

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

Flag Value	Bit Field	Effect Produced
Ios::left	Ios::adjustfield	Left-justified output
Ios::right	Ios::adjustfield	Right-adjust output
Ios::internal	Ios::adjustfield	Padding occurs between the sign or base indicator and the number, when the number output fails to fill the full width of the field.
Ios::dec	Ios::basefield	Decimal conversion
Ios::oct	Ios::basefield	Octal conversion
Ios::hex	Ios::basefield	Hexadecimal conversion
Ios::scientific	Ios::floatfield	Use exponential floating notation.
Ios::fixed	Ios::floatfield	Use ordinary floating notation.

Managing output with manipulators:

- ✓ The *header file iomanip* provides a *set of function* called *manipulators* which can be used to *manipulate the output formats*.
- ✓ They provide the same features as that of the *ios member functions and flags*.
- ✓ Some *manipulators* are more convenient to use than their *counterparts in the class ios*.

```
cout<<manip1<<manip2<<manip3<<item;
cout<<manip1<<item1<<manip2<<item2;
```

This kind of *concatenation* is useful when we want *to display several columns of output*.

- Manipulators and their meanings

Manipulators	Meaning	Equivalent
setw(int w)	Set the field width to w	width()
setprecision(int d)	Set floating point precision to d	precision()
setfill(int c)	Set the fill character to c	fill()

```
cout << setw(5) << setprecision(2) << 1.2345  
      << setw(10) << setprecision(4) << sqrt(2);
```

- We can jointly use the manipulators and the ios functions in a program.

WORKING WITH FILES:

- ✓ A *file* is a *collection of related* data stored in *particular area on the disk*.
- ✓ The program can be *designed to perform* the *read and write operations on these files*.
- ✓ The program typically *involves either or both* of the *following kinds of data communication*.
 - Data transfer b/w the console unit and the program.
 - Data transfer b/w the program and a disk file
- ✓ The *stream* that *supplied data to the program* is known as *input stream* and one that *receives data from the program* is known as *output stream*.

CLASSES FOR FILE STREAM OPERATIONS :

- ✓ The i/o system of c++ contains a *set of classes* that define the *file handling methods*.
- ✓ These include *ifstream, ofstream and fstream*
 - To *create an input stream*, declare an object of *type ifstream*.
 - To *create an output stream*, declare an object of *type ofstream*.
 - To *create an input/output stream*, declare an object of *type fstream*.

For example, this fragment creates one input stream, one output stream and one stream capable of both input and output:

```
ifstream in; // input;  
fstream out; // output;  
fstream io; // input and output
```

- ✓ To perform file I/O operation. We must include a header file <fstream.h> which contains all classes supports file operations.

- ✓ **ifstream class: (input file operation)**

This class provides input operations or methods which contains open(), getline(), get(), read(), tellg(), seekg().

Example :

```
    ifstream f1;  
    Fstream infile;  
    f1.open("employee.dat");
```

- ✓ **ofstream class: (output file operation)**

This class provides output operations it supports the following methods.

```
    Open()      tell()  
    Put()      write()  Seekp()
```

Example :

```
    ofstream outfile;  
    Outfile open ("student-dat");
```

- ✓ **fstream class:**

It provides all function supports I/O operation

details of file stream classes :

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains Openprot constant used in the open() of file stream classes. Also contain close() and open() as members.
fstreambase	Provides operations common to file streams. Serves as a base for fstream , ifstream and ofstream class. Contains open() and close() functions.
ifstream	Provides input operations. Contains open() with default input mode. Inherits the functions get() , getline() , read() , seekg() , tellg() functions from istream .
ofstream	Provides output operations. Contains open() with default output mode. Inherits put() , seekp() , tellp() and write() functions from ostream .
fstream	Provides support for simultaneous input and output operations. Contains open with default input mode. Inherits all the functions from istream and ostream classes through iostream .

OPENING AND CLOSING A FILE :

- ✓ A *file must be opened before you can read from it or write to it.* Either *ofstream* or *fstream* object may be used to open a file for writing. And *ifstream* object is used to open a file for reading purpose only.
- ✓ Following is the standard syntax for **open()** function, which is a member of **fstream**, **ifstream**, and **ofstream** objects.

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

- ✓ Here, the *first argument specifies the name and location of the file to be opened* and *the second argument of the open() member function defines the mode in which the file should be opened.*

CLOSING A FILE

- ✓ When a *C++ program terminates* it automatically *flushes all the streams*, release all the *allocated memory and close all the opened files*.
- ✓ But it is *always a good practice* that a programmer should close all the *opened files before program termination*.

Following is the standard *syntax for close() function*, which is a member of *fstream, ifstream, and ofstream objects*.

```
void close();
```

READING FROM A FILE

- ✓ We read information *from a file into your program using the stream extraction operator (>>)* just as you use that *operator to input information from the keyboard*.

The only difference is that you use an *ifstream or ofstream* object instead of the **cin** object.

SEQUENTIAL INPUT AND OUTPUT OPERATIONS:

- ✓ The file stream classes support a number of member functions for performing the input and output operations on files.
- ✓ One pairs of functions, *put() and get()* are designed *for handling a single character at a time*.
- ✓ Another pair of functions, *write(),read()* are designed *to write and read blocks of binary data*.

put() and get() Functions:

- ✓ The function *put()* *writes a single character* to the associated stream. Similarly, the function *get()* *reads a single character from the associated stream*.
- ✓ The following program illustrates how the functions work on a file. The program requests for a string.
- ✓ On receiving the string, the program writes it, character by character, to the file using the **put()** function in a for loop. The length of string is used to terminate the for loop.

The program then displays the contents of file on the screen. It uses the function `get()` to fetch a character from the file and continues to do so until the end-of-file condition is reached. The character read from the files is displayed on screen using the operator `<<`.

PROGRAM FOR I/O OPERATIONS ON CHARACTERS

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main() {
char string[80];
cout<<"enter a string \n";
cin>>string;
int len =strlen(string);
fstream file;
file.open("TEXT". ios::in | ios::out);
for (int i=0;i<len;i++)
file.put(string[i]);
file.seekg(0);
char ch;
while(file)
{
file.get(ch);
cout<<ch;
} return }
```

WRITE() AND READ () FUNCTIONS:

- ✓ The functions `write()` and `read()`, unlike the functions `put()` and `get()`, *handle the data in binary form*. This means that the *values are stored in the disk file in same format* in which they are *stored in the internal memory*.
- ✓ An *int* character *takes two bytes to store* its value in the *binary form, irrespective of its size*. But a *4 digit int* will take *four bytes* to store it in the *character form*.
- ✓ The binary input and output functions takes the following form:
- ✓

```
infile.read (( char * ) & V, sizeof (V));  
outfile.write (( char * ) & V , sizeof (V));
```

- ✓ These functions take *two arguments*. The first is *the address of the variable V*, and the second is the *length of that variable in bytes*.
- ✓ The address of the *variable must be cast to type char*(i.e pointer to character type)*.
- ✓ The following program illustrates how these two functions are used to save an array of floats numbers and then recover them for display on the screen.

PROGRAM FOR // I/O OPERATIONS ON BINARY FILES

```
#include <iostream.h>  
#include <fstream.h>  
#include <iomanip.h>  
const char * filename = "Binary";  
int main()  
{  
float height[4] = { 175.5,153.0,167.25,160.70};  
ofstream outfile;  
outfile.open(filename);  
outfile.write((char *) & height,sizeof(height));  
outfile.close();  
for (int i=0;i<4;i++)  
height[i]=0;  
ifstream infile;  
infile.open(filename);  
infile.read ((char *) & height,sizeof (height));  
for (i=0;i<4;i++)  
{  
cout.setf(ios::showpoint);  
cout<<setw(10)<<setprecision(2)<<height[i];  
}  
infile.close();  
return 0; }
```


UPDATING A FILE : RANDOM ACCESS:

- ✓ The updating would include more of the following tasks:
 - *Displaying the contents of a file*
 - *Modifying an existing item*
 - *Adding a new item*
 - *Deleting an existing item*
- ✓ These *actions require file pointers* to move to a *particular locations* that corresponds to the item/ *object under consideration*.
- ✓ A *file pointer points to a data element* such as *character in the file*. The pointers are helpful in *lower level operations in files*. There are two types of pointers:
 - *get pointer*
 - *put pointer*
- ✓ The *get pointer* is also called *input pointer*. When *we open a file for reading*, we can *use the get pointer*. The *put pointer is also called output pointer*. When we open a file for writing, *we can use put pointer*.
- ✓ These pointers are *helpful in navigation through a file*. When *we open a file* for reading, the get pointer will be at *location zero and not 1*. *The bytes in the file are numbered from zero*.
- ✓ Therefore, *automatically when* we assign an *object to ifstream* and then *initialize the object with a file name*, the get pointer will be *ready to read the contents from 0th position*.
- ✓ Similarly, when *we want to write we will assign to an ofstream* object a filename. Then, the put pointer will point to the 0th position of the given *file name after it is created*.
- ✓ When we *open a file for appending*, the put pointer will point to the 0th position. But, when we say write, then the pointer *will advance to one position after the last character in the file*.

FILE POINTER FUNCTIONS

- ✓ There are essentially four functions, which help us to *navigate the file as given below Functions*
 - **tellg()** *Returns the current position of the get pointer*
 - **seekg()** *Moves the get pointer to the specified location*
 - **tellp()** *Returns the current position of the put pointer*
 - **seekp()** *Moves the put pointer to the specified location*

FILE POSITION POINTERS

- ✓ *Both istream and ostream* provide member functions for *repositioning the file-position pointer*. These member functions are *seekg ("seek get")* for *istream* and *seekp ("seek put")* for *ostream*.
- ✓ The argument to *seekg and seekp* normally is a *long integer*. A second argument can be specified to indicate the seek direction.
- ✓ The seek direction can be *ios::beg (the default)* for positioning relative to the beginning of a stream, *ios::cur for positioning relative to the current position* in a stream or *ios::end for positioning relative to the end of a stream*.
- ✓ The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are –

COMMAND – LINE ARGUMENTS:

- ✓ The name of the *file containing the program* to be executed and *data and results are the filenames passed to the program as command- line arguments*

▪ `main(int argc, char*argv[]`

The 1st argument *argc* (*known as argument counter*) represents *Number of arguments in the command line* and 2nd *argument argv*(*known as argument vector*) is an array of *char type pointers that points to the command line arguments*.

TEMPLATES:

- ✓ It is an *new concept* which *enables* us to *define generic classed* and functions and thus provides support for *generic programming*.
- ✓ Generic programming is an *approach* where *generic types* are used as *parameters* in *algorithms* so that they work for a *variety of suitable data types and data structures*
- ✓ An template can be consider as *an a kind of macro*.
- ✓ A template is a *blueprint or formula* for creating a *generic class or a function*.
- ✓ There is a single definition of *each container*, such as *vector*, but we can define many different kinds of vectors for example, *vector <int> or vector <string>*.
- ✓ Templates can be classified into two types they are as follows:

- *Function Template*

- *Class Template*

FUNCTION TEMPLATE:

The general form of a template function definition is shown here –

```
template <class type> ret-type func-name(parameter list) {  
    // body of function  
}
```

Here, *type* is a *placeholder name for a data type used by the function*. This name can be used within the *function definition*.

CLASS TEMPLATE

- ✓ Just as we can define function templates, *we can also define class templates*. The general form of a *generic class declaration* is shown here –

```
template <class type> class class-name {  
    .  
    .  
    .  
}
```

Here, *type* is *the placeholder type name*, which will be *specified when a class is instantiated*. You can define more than *one generic data type by using a comma-separated list*.

PREDEFINED C++ MACROS

C++ provides a number of predefined macros mentioned below –

Sr.No	Macro & Description
1	__LINE__ This contains the current line number of the program when it is being compiled
2	__FILE__ This contains the current file name of the program when it is being compiled.

3	<p>__DATE__</p> <p>This contains a string of the form month/day/year that is the date of the translation of the source file into object code.</p>
4	<p>__TIME__</p> <p>This contains a string of the form hour:minute:second that is the time at which the program was compiled.</p>

EXCEPTION HANDLING:

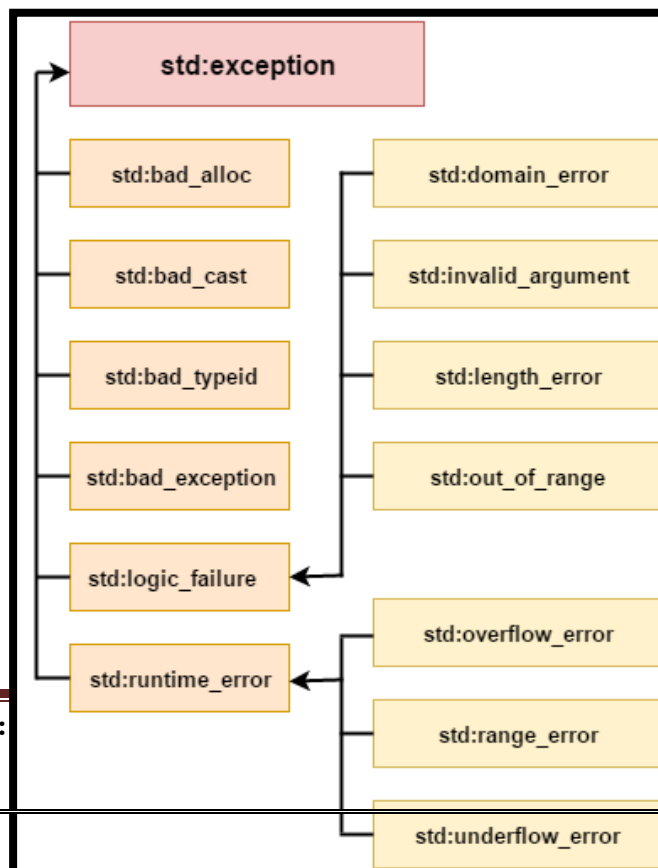
- ✓ Exception Handling in C++ is a *process to handle runtime errors*. We perform *exception handling* so the normal flow of the *application can* be maintained even after *runtime errors*.
- ✓ In C++, exception *is an event or object* which is *thrown at runtime*. All exceptions are derived from *std::exception class*. It is a *runtime error which can be handled*. If we *don't handle the exception*, it prints *exception message* and *terminates the program*.

ADVANTAGE

- ✓ *It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.*

C++ CLASSES

- ✓ In C++ are we can parent-is



EXCEPTION

standard exceptions defined in <exception> class that use inside our programs. The arrangement of child class hierarchy shown below:

CORE COURSE II PROGRAMMING IN C++

All the exception classes in C++ are derived from `std::exception` class. Let's see the list of C++ common exception classes.

Exception	Description
<code>std::exception</code>	It is an exception and parent class of all standard C++ exceptions.
<code>std::logic_failure</code>	It is an exception that can be detected by reading a code.
<code>std::runtime_error</code>	It is an exception that cannot be detected by reading a code.
<code>std::bad_exception</code>	It is used to handle the unexpected exceptions in a c++ program.
<code>std::bad_cast</code>	This exception is generally be thrown by dynamic_cast .
<code>std::bad_typeid</code>	This exception is generally be thrown by typeid .
<code>std::bad_alloc</code>	This exception is generally be thrown by new .

- ✓ C++ Exception Handling Keywords *In C++, we use 3 keywords to perform exception handling:*
 - *try*
 - *catch, and*
 - *throw*

C++ TRY/CATCH

- ✓ In C++ programming, exception handling is performed using try/catch statement. The C++ *try block is used to place the code that may occur exception.*
- ✓ C++ function that can *detect and recover from errors execute from within a try block* which *causes the compiler to pay special attention to generate code for handling exception*

The General form of try block is as follows:

```
try
{
...
...
}
```

CATCH BLOCK

- ✓ When an exception is thrown it is caught by its corresponding catch block, which processes the exception.
- ✓ The catch block immediately following the try block is called as exception handler.

The general form of catch block is as follows:

```
catch (exception Type argument)
{
.....
}
```

RETHROWING AN EXCEPTION:

- ✓ C++ allows *to rethrow* an exception *after partially handling* it or *after determining that the exception handler cannot deal with it all*.
- ✓ A “*throw*” statement can be used for it

C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
    return (x/y);
}
int main () {
    int i = 50;
    int j = 0;
    float k = 0;
    k = division(i, j);
    cout << k << endl;
    return 0;
}
```

```
}
```

output: Floating point exception (core dumped)

C++ TRY/CATCH EXAMPLE

```
#include <iostream>
using namespace std;
float division(int x, int y) {
    if( y == 0 ) {
        throw "Attempted to divide by zero!";
    }
    return (x/y);
}
int main () {
    int i = 25;
    int j = 0;
    float k = 0;
    try {
        k = division(i, j);
        cout << k << endl;
    }catch (const char* e) {
        cerr << e << endl;
    }
    return 0;
}
```

Output:

Attempted to divide by zero!

Unit V

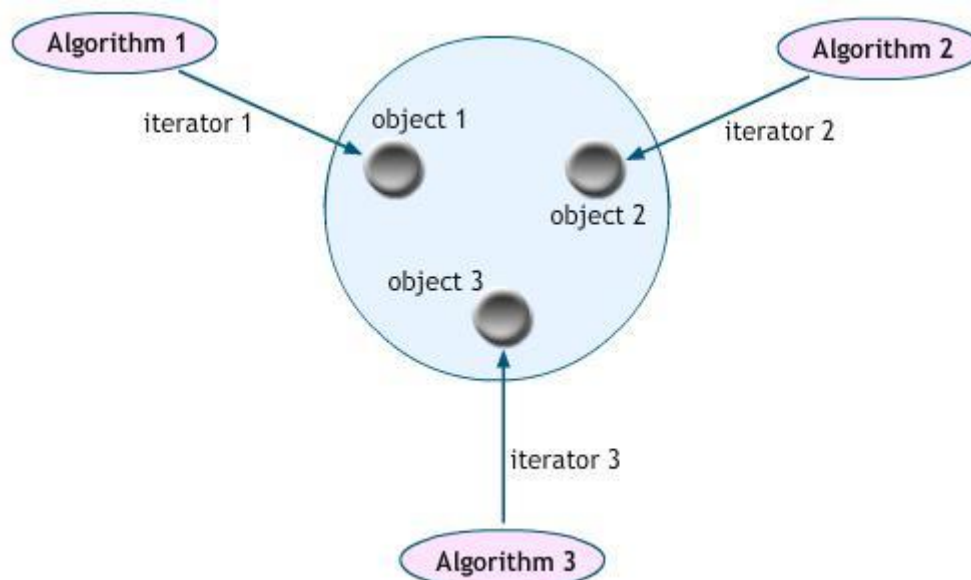
Standard Template Library - Manipulating Strings - Object Oriented Systems Development

Standard Template Library:

- ✓ *Classes* (data structures) and *functions* (algorithms) that could be used as a standard approach for *storing and processing of data*.
- ✓ The collection of these *generic classes and functions* is called as *standard Template Library (STL)*.
- ✓ Using *STL* can *save time, effort and lead to high quality programs*.
- ✓ All these benefits are *possible because* we are basically “*reusing*” the *well-written* and *well-tested* components defined *in STL*.

COMPONENTS OF STL:

- ✓ There are **3 core key** components of STL they are as follows:
 - *Containers*
 - *Algorithms*
 - *Iterators*



Relationship b/w the three STL components

CONTAINERS:

- ✓ A container is a *way to store data*, whether the *data consists of built-in types* such as *int and float, or of class objects*.
- ✓ The STL *makes seven basic kinds of containers available*, as well as three more that are *derived from the basic kinds*.

Containers

- Sequential containers
 - Vector
 - Deque
 - List
- Associative containers
 - Set
 - Map
 - Multiset
 - Multimap
- Container adaptors
 - Stack
 - Queue
 - Priority queue

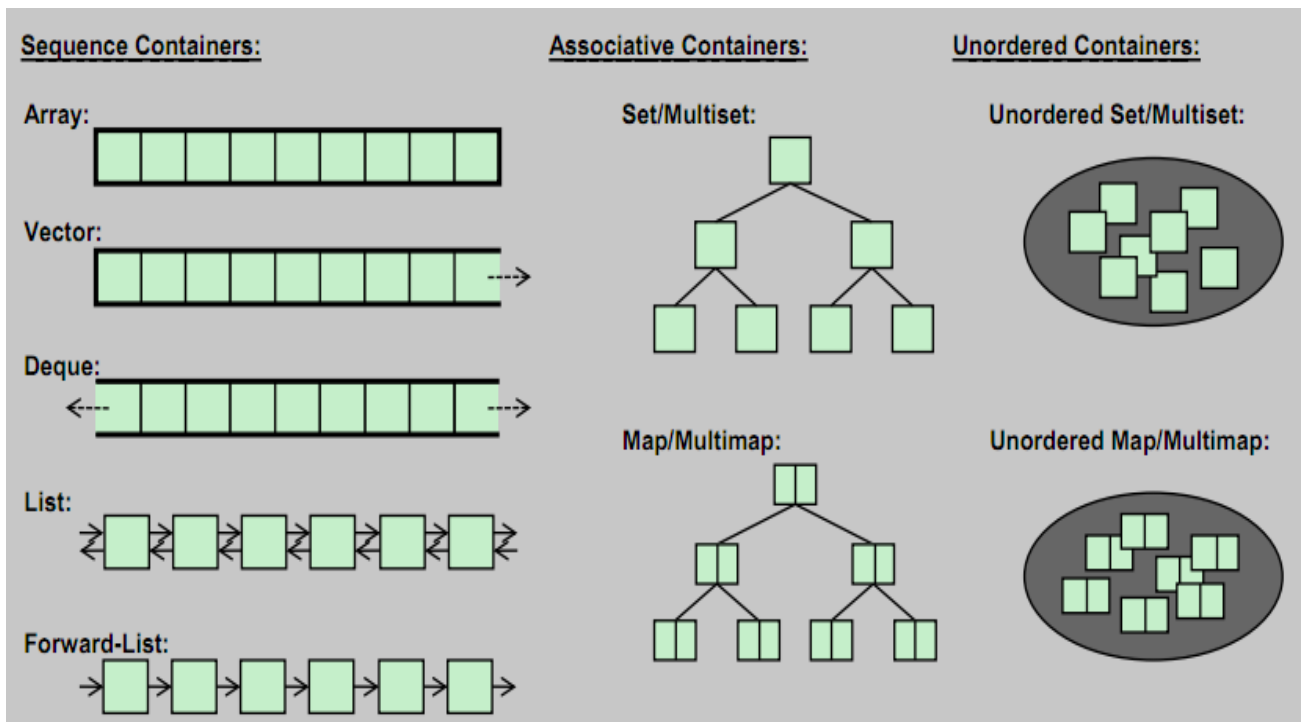


Table 4.1 STL Containers

Container	Type	Description
deque	Sequential	Double-ended queue
list	Sequential	Linear list
map	Associative	Collection of key/value pairs in which each key is associated with exactly one value
multimap	Associative	Collection of key/value pairs in which each key may be associated with more than one value
multiset	Associative	Collection in which each element is not necessarily unique
priority_queue	Adaptor	Priority queue
queue	Adaptor	Queue
set	Associative	Collection in which each element is unique
stack	Adaptor	Stack
vector	Sequential	Dynamic array

Containers supported by stl

SEQUENCE CONTAINERS:

- ✓ A *sequence container* stores a set of *elements* in what you can *visualize as a line*, like *houses on a street*. Each element is related to the other elements by its position along the line. Each element *is preceded by one specific element and followed by another*.
- ✓ The STL provides the *vector* container to avoid these difficulties. This can *be very time-consuming*.
- ✓ The STL provides the *list container*, which is based on the idea of a linked list.
- ✓ The third sequence container is the *deque, which can be thought of as a combination of a stack and a queue*. Both *input and output* take place on the top of the stack.
- ✓ A queue, *on the other hand*, uses a *first-in-first-out arrangement*: data goes in at the *front and comes out at the back*, like a *line of customers in a bank*.
- ✓ A *deque combines* these approaches *so you can insert or delete data from either end*. The word *deque is derived from Double-Ended QUEUE*. It's a *versatile mechanism that's not only useful in its own right*, but can be used as the basis for *stacks and queues*.

ASSOCIATIVE CONTAINERS

- ✓ An associative container is *not sequential*; instead it uses *keys to access data*. The keys, typically *numbers or strings*, are used automatically by the container to arrange the stored elements in a *specific order*.
- ✓ It's like an ordinary *English dictionary*, in which you access data by looking up words arranged in *alphabetical order* and the container converts this key to the *element's location in memory*.
- ✓ There are two kinds of *associative containers in the STL: sets and maps*.
- ✓ These both store data in a *structure called a tree*, which offers *fast searching, insertion, and deletion*.
- ✓ *Sets and maps* are thus very *versatile general data structures* suitable for a wide *variety of applications*. However, it is inefficient to sort them and perform other operations that require random access.
- ✓ *Sets are simpler* and more commonly used *than maps*. A *set stores* a number of items which *contain keys*. The keys are the attributes used to order the items.
- ✓ For example, a set might store objects of *the person class*, which are ordered alphabetically *using their name attributes as* keys. In this situation, you can quickly locate a *desired person* object by searching for the object *with a specified name*.
- ✓ If a set stores values of a basic type such as int, the key is the entire item stored.
- ✓ A map *stores pairs of objects: a key object and a value object*.
- ✓ The *map and set* containers allow *only one key* of a given value to be stored. This makes sense in, *say, a list of employees arranged by unique employee numbers*.

ALGORITHMS

- ✓ An algorithm is a *function* that does *something* to the *items in a container* (or *containers*).
- ✓ We noted, algorithms in the *STL are not member functions or even friends* of *container classes*, as they are in earlier container libraries, *but are standalone* template functions.
- ✓ STL algorithms *reinforce the philosophy of reusability*
- ✓ STL algorithms based on the nature of operations they perform, may be categorized as under:
 - Retrieve or nonmutating algorithm
 - Mutating algorithm

- Sorting algorithm
 - Set Algorithm
 - Relational Algorithm
- ✓ Suppose you create an array of type int, with data in it:
- ```
int arr[8] = {42, 31, 7, 80, 2, 26, 19, 75};
```
- ✓ You can then use the STL sort() algorithm to sort this array by saying
- ```
sort(arr, arr+8);
```
- where arr is the address of the beginning of the array, and arr+8 is the past-the-end address(one item past the end of the array).

ITERATORS

- ✓ Iterators are *pointer-like entities* that are used to *access individual data items* (which *are usually called elements*), in a container.
- ✓ Often they are used to *move sequentially* from *element to element*, a process called *iterating through the container*.
- ✓ We can increment *iterators with the ++ operator* so they *point to the next element*, and *dereference them with the * operator to obtain the value* of the element they *point to*.
- ✓ In the STL *an iterator* is represented by an *object of an iterator class*.
- ✓ Different classes *of iterators must be used with different types of container*.
- ✓ There are *three major classes* of iterators: *forward, bidirectional, and random access*.
- ✓ A *forward iterator* can only *move forward through the container, one item at a time*. Its *++ operator accomplishes* this. It *can't move backward* and it *can't be set to an arbitrary location in the middle of the container*.
- ✓ A *bidirectional iterator* can *move backward as well as forward*, so both its *++ and -- operators* are defined.
- ✓ A *random access iterator*, in addition to *moving backward and forward, can jump to an arbitrary location*

APPLICATIONS OF CONTAINER CLASSES:

- ✓ Most popular containers namely *Vector, list and map*

VECTORS:

- ✓ Is the most widely used *container*. It stores *elements* in *contiguous memory locations* and *enables direct access to* any element using the *subscript operator []*.
- ✓ A vector can change its *size dynamically* and therefore *allocated memory* as needed at *run time*.

LISTS:

- ✓ It supports *bidirectional* , *linear list* and provides an *efficient implementation* for *deletion and insertion operations*.
- ✓ Unlike a vector , which supports *random access* , a list can be *accesses sequentially only*.

MAPS:

- ✓ A map is a *sequence of (key, value)* pairs where a *single value is associated* with each *unique key*.
- ✓ A map is commonly called *an associative array*.

FUNCTION OBJECTS:

- ✓ Is a *function* that has been *wrapped in a class* so that it looks *like an object*.
- ✓ The class has only *one member function* the *overloaded () operator and no data*.
- ✓ This class is *templated* so that it can be used *with different data types*.

MANIPULATING STRINGS

- ✓ ANSI standard C++ introduces a *new class* called “*string*” which is an *improved version of C strings in several ways*.
- ✓ In many cases, the strings object may be *treated like any other built-in data type*.
- ✓ The string is treated as *another container class for C++*.

STRING CLASS IN C++

- ✓ The string class *is huge and includes many constructors, member functions, and operators*.
- ✓ Programmers may use the *constructors, operators and member functions to achieve the following:*

1. Creating string objects
2. Reading string objects from keyboard
3. Displaying string objects to the screen
4. Finding a substring from a string
5. Modifying string
6. Adding objects of string
7. Comparing strings
8. Accessing characters of a string
9. Obtaining the size or length of a string, etc...

IMPORTANT CONSTRUCTORS OBTAINED BY STRING CLASS

- String(): This *constructor* is used for *creating an empty string*
- String(const char *str): This *constructor* is used for *creating string objects* from a null-terminated string
- String(const string *str): This *constructor* is used for *creating a string object from another string object*

IMPORTANT FUNCTIONS SUPPORTED BY STRING CLASS

- 1) append(): This function *appends a part of a string to another string*
- 2) assign(): This function *assigns a partial string*
- 3) at(): This function obtains the character *stored at a specified location*
- 4) begin(): This function *returns a reference to the start of the string*
- 5) capacity(): This function gives the *total element that can be stored*
- 6) compare(): This function *compares a string against the invoking string*
- 7) empty(): This function *returns true if the string is empty*
- 8) end(): This function returns a reference to *the end of the string*

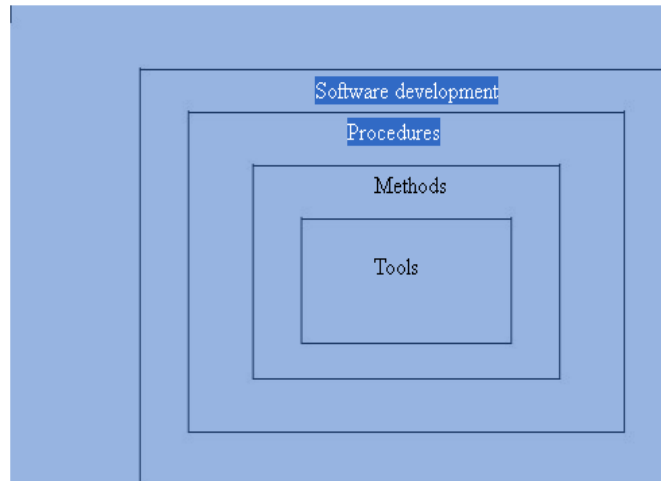
- 9) erase(): This function *removes character as specified*
- 10) find(): This function searches for the *occurrence of a specified substring*
- 11) length(): It gives *the size of a string or the number of elements of a string*
- 12) swap(): This function *swaps the given string with the invoking one*

OPERATORS USED FOR STRING OBJECTS

1. =: assignment
2. +: concatenation
3. ==: Equality
4. !=: Inequality
5. <: Less than
6. <=: Less than or equal
7. >: Greater than
8. >=: Greater than or equal
9. []: Subscription
10. <<: Output
11. >>: Input

OBJECT ORIENTED SYSTEMS DEVELOPMENT:

- ✓ Software engineers have been trying various *tools, methods, and procedures to control the process of software development in order to build high quality software with improved productivity.*
- ✓ The methods provide *“how to s”* for building the software while the tools *provide automated or semi-automated support for the methods.*
- ✓ They are used in all the stages of software development process, namely, *planning, analysis, design, development and maintenance.*
- ✓ The software development procedures integrate the methods and tools together and enable rational and timely development of software systems.



- ✓ They provide guidelines as to *apply the methods and tools*, how to *produce the deliverables at each stage*, what *controls to apply*, and what *milestones to use to assess* the progress.
- ✓ There exist a number of *software development paradigms*, each using a *different set of methods and tools*.
- ✓ The selection of particular *paradigms depends on the nature of the application*, the *programming language* used, and the *controls and deliverables required*.
- ✓ The development of a *successful system depends not* only on the use of the appropriate methods and *techniques but also on the developer's commitment to the objectives of the systems*.
- ✓ A successful system must:
 1. *satisfy the user requirements*,
 2. *be easy to understand by the users and operators*,
 3. *be easy to operate*,
 4. *have a good user interface*,
 5. *be easy to modify, Tools*
 6. *be expandable*,
 7. *have adequate security controls against misuse of data*,
 8. *handle the errors and exceptions satisfactorily*, and
 9. *Be delivered on schedule within the budget*.

PROCEDURE-ORIENTED PARADIGM.

- ✓ Software development is usually *characterized by a series of stages* depicting the various tasks involved in the *development process*.
- ✓ The *classic life cycle* is based on an underlying model, commonly referred to as the “*water fall*” model.
- ✓ This model attempts to break up the *identifiable activities into series of actions*, each of which must be completed before the *next begins*.
- ✓ The activities *include problem definition, requirement analysis, design, coding, testing, and maintenance*.
- ✓ Further refinements to this *model include iteration* back to the previous stages in order to *incorporate any changes or missing links*.

Problem Definition:

- ✓ This activity requires a *precise definition of the problem in user terms*.
- ✓ A clear statement of the problem is *crucial to the success of the software*.
- ✓ It helps not only the development but also the *user to understand the problem better*.

Analysis:

- ✓ This covers a *detailed study of the requirements of both the user and the software*.
- ✓ The activity is basically concerned with what of the system such as
 1. What are the inputs to the systems?
 2. What are the processes required?
 3. What are the outputs expected?
 4. What are the constraints?

Design:

- ✓ The design *phase deals* with various *concepts of system design* such as *data structure, software architecture, and algorithms*.
- ✓ This *phase translates* the requirements into a representation of the software. *This stage answers the questions of how*.

Coding:

- ✓ Coding refers to the *translation of the design* into *machine-readable form*. The more detailed the design, the *easier is the coding, and better its reliability*.

Testing:

- ✓ Once the *code is written*, it should be *tested rigorously* for *correctness of the code* and *results*. Testing may involve the *individual units* and the *whole systems*. It requires a *detailed plan as to what, when and how to test*.

Maintenance:

- ✓ After the software *has been installed*, it may *undergo some changes*. This may occur due to a change in *the user's requirement*, a change in the *operating environment*, or an error in the software that has *been fixed during the testing*.
- ✓ Maintenance ensures that these changes *are incorporated wherever necessary*.

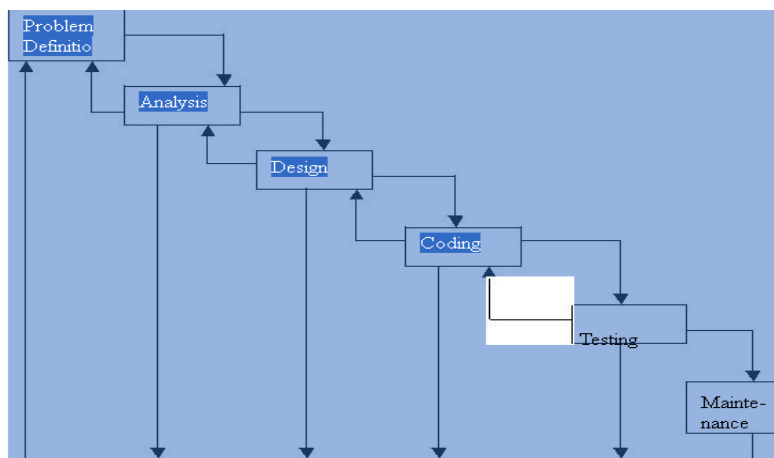


Fig. classic software development life cycle (Embedded 'water-fall' model)

OBJECT-ORIENTED ANALYSIS

- ✓ Object-oriented analysis (OOA) *refers to the methods of specifying requirements* of the software in the terms of *real-world objects*, their *behavior, and their interactions*.
- ✓ Object-oriented design (OOD), on the *other hand*, turns the software requirements into *specifications for objects* and *derives class hierarchies* from *which the objects can be created*.

- ✓ Finally, *object-oriented programming (OOP)* refers to the *implementation of the program using objects, in an object-oriented programming language such as C++.*
- ✓ By developing specifications of the objects found in the *problem space, a clear and well-organized statement of the problem is actually built into application.*
- ✓ These objects form a *high-level layer of definitions* that are written in terms of the problem space. *During the refinement of the definitions* and the implementation of the application objects, *other objects are identified.*
- ✓ All the phases in the *object-oriented approach* work more closely together because of the commonality of the object model. In *one phase*, the problem domain objects are identified, while in the *next phase additional objects required for a particular solution are specified.* The design process is repeated for these implementation-level objects.
- ✓ Object-oriented analysis provides us with simple, yet *powerful, mechanism for identifying objects, the building block of the software to be developed.*
- ✓ The analysis is basically concerned with the decomposition of a problem into its component parts and establishing a logical model to describe the system functions.
- ✓ The object-oriented analysis (OOA) approach consists of the following steps:
 - a. *Understanding the problem.*
 - b. *Drawing the specification of requirement of the user and the software.*
 - c. *Identifying the objects and their attributes.*
 - d. *Identifying the services that each object is expected to provide (interface).*
 - e. *Establishing inter-connections (collaborations) between the objects in terms of services required and services rendered.*

Understanding the problem.

- ✓ 1st step in the *analysis process* is to *understand the problem of the user.*
- ✓ The problem statement should be *refined and redefined in terms of computer system engineering that could suggest a computer based solution.*
- ✓ The problem statement provides the *basis for drawing the requirements specification of both the user and s/w.*

Specification of requirement

- ✓ Once the problem is *Clearly defined*, the next step is to *understand what the proposes system is required to do*.
- ✓ A clear understanding should be *exist between the user and developer* of what is *required*.
- ✓ Based on the requirements, the specifications for the *s/w should be drawn*.
- ✓ The developer should state clearly
 - *What output are required*
 - *What processes are involved to produce these o/p.*
 - *What i/p are necessary*
 - *What resources are required.*

Identifying the objects and their attributes

- ✓ objects can be identified in terms of the real world objects as well as the abstract objects.
- ✓ The application may be analyzed by using one the following approaches:
 - Data flow Diagrams(DFD)
 - Textual analysis (TA)

Identifying the services

- ✓ Once the *objects in the solution* space have been identified, the *next step is to identify the set of services that object should offer*.
- ✓ *Services are identified* by examining *all the verbs and verb phrases in the problem description statements*.

Establishing inter-connections

- ✓ Here we may use an *information flow diagram (IFD)* or an *Entity Relationship diagram (ER)* to enlist this information.
- ✓ Here we *must establish* a correspondence b/w the *service and actual information (messages) that are being communicated*.

STEPS IN OBJECT ORIENTED DESIGN:

- ✓ Design is *concerned with mapping of objects* in the *problem space* into objects in the *solution space* and *creating an overall structure* and *computational models of the system*.
- ✓ **Reusability** of classes from the previous designs, classification of the *object into subsystems* and determination of *appropriate protocols* are some of the consideration of *design stage*.
- ✓ The OOD approach may involve following steps:
 - 1) *Review of objects created in the analysis phase.*
 - 2) *Specification of class dependencies*
 - 3) *Organization of class hierarchies*
 - 4) *Design of Classes*
 - 5) *Design of member function*
 - 6) *Design of Driver Program*

Review of objects created in the analysis phase

- ✓ The main objective of this *review exercise* is to *refine the objects* in terms of their attributed and operations and to *identity other objects that are solution specific*.
- ✓ Some guidelines that might help the review process are:
 - *If only one object is necessary for a service , then it operates only on that objects*
 - *If two or more objects are required for an operation to occur, then it is necessary to identify which object's private part should be known to the operation*
 - *If an operation requires knowledge of more than one type of objects, then the operations is not functionally cohesive and should be rejected,.*

Class dependencies

- ✓ It is important to identify *appreciate classes* to represent the objects in the *solution space* and *establish a relationships*.
- ✓ The major relationships that are important in context of designs are:
 - *Inheritance Relationships*
 - *Containment Relationships*
 - *Use Relationships*

Organization of class hierarchies

- ✓ It involves *identification of common attributes and functions among a group or related classes and then combining them to form a new class.*
- ✓ The new class will serve as the *super class and others as subordinate class.*
- ✓ The *new class may or may not have the meaning of the object by itself.*
- ✓ If the object is created *purely to combine* the common attributes, it is called as *an “abstract class”.*

Design of Classes

- ✓ We have identified *classes, their attributes and minimal set* of operations required by the concepts a *class is representing*
- ✓ For the class to be useful, it must contain the following functions:
 - *Class Management functions*
 - *Class implementation functions*
 - *Class access functions*
 - *Class utility functions*

Design of member functions

- ✓ The member functions define the operations that are performed on the object's data.

Design of the driver program

- ✓ Every c++ program *must contain a main() function code known as “driver program”*
- ✓ The execution *of the program begins and ends here.*
- ✓ The driver program is mainly responsible for
 - *Receiving data values from the user*
 - *Creating objects from the class definitions*
 - *Arranging communication b/w the objects as a sequence of messages for invoking member functions*
 - *Displaying output results in form required by the user*

Implementation

- ✓ It includes *coding and testing*
- ✓ Codes includes *writing code for classes, member functions and main program*, that acts as a *driver in the program*,

Wrapping up

- ✓ Following are some points for your thought and innovation
 1. *Set clear goals and tangible objectives*
 2. *Keep in mind that the proposed system must be flexible m portable and extendable*
 3. *Keep a clear documentation of everything that goes into the system*
 4. *Try to reuse the existing functions and classes*
 5. *Keep functions strongly typed wherever possible*
 6. *Use prototype wherever possible*
 7. *Match design and programming style*
 8. *Keep the system clean, simple, small and efficient as far as possible*

.....the end.....

