



OOPS WITH C++

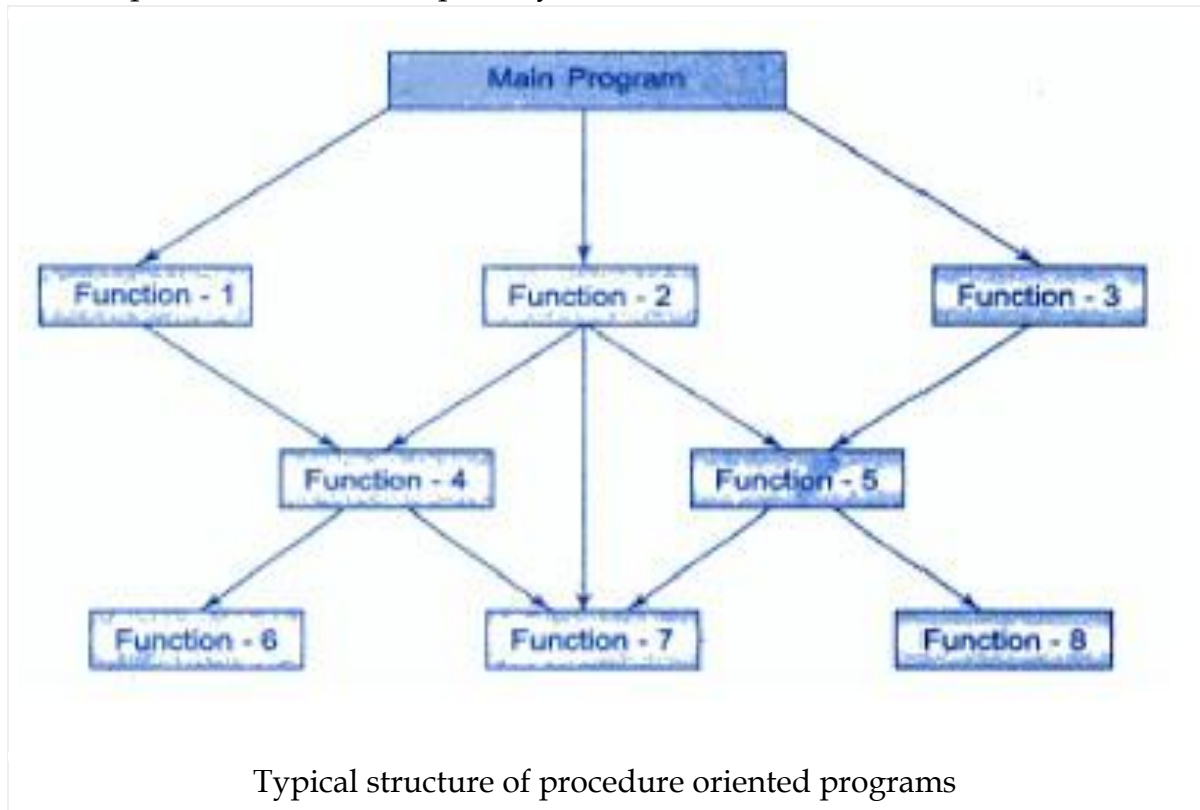
Publisher : **"SHREE MEDHA COLLEGE"**

UNIT-I

PRINCIPLES OF OBJECT ORIENTED PROGRAMMING

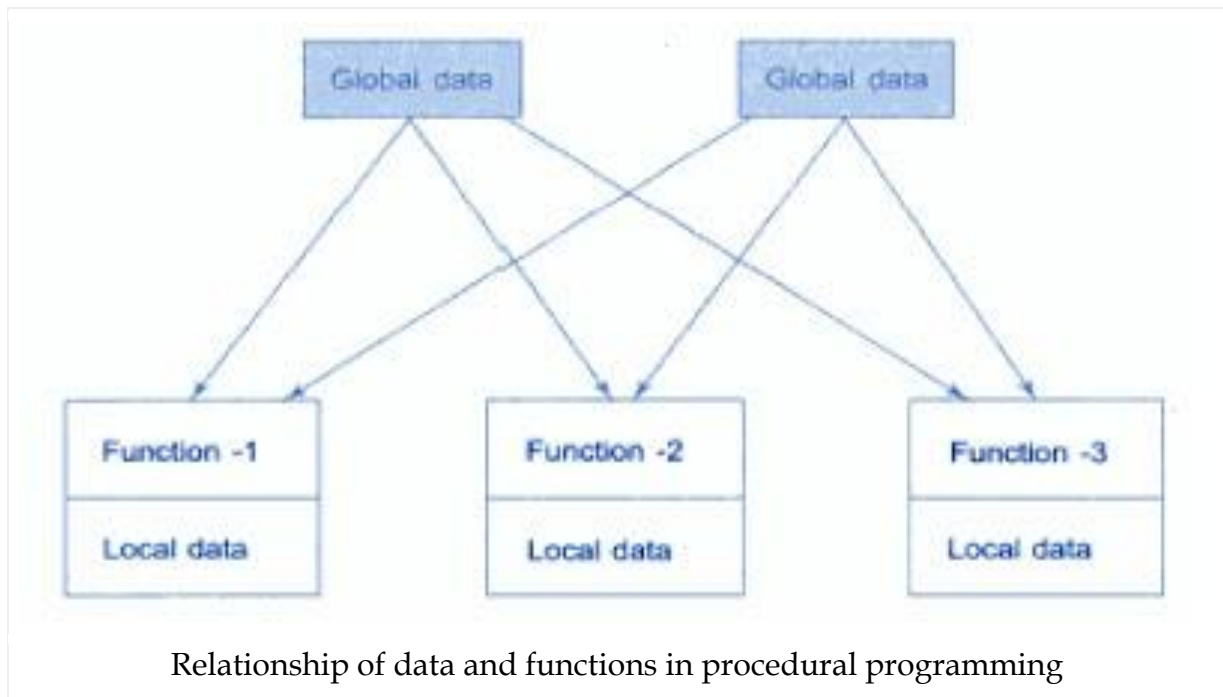
PROCEDURE ORIENTED PROGRAMMING (POP) :-

Conventional programming using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure oriented programming (POP). In the procedure oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions.



Procedure oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as functions. While we concentrate on the development, very little attention is given to the data that are being used by various functions.

In a multi-function program, many important data items are placed as global so that they may be accessed by all functions. Each function may have its own local data. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.



Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really corresponding to the elements of the problem.

Some characteristics of Procedure Oriented Programming are :-

- 1) Emphasis is on doing things(algorithms).
- 2) Large programs are divided into smaller programs known as functions.
- 3) Most of the functions share global data.
- 4) Data more openly around the system from function to function.
- 5) Functions transform data from one form to another.
- 6) Employs top-down approach in program design.

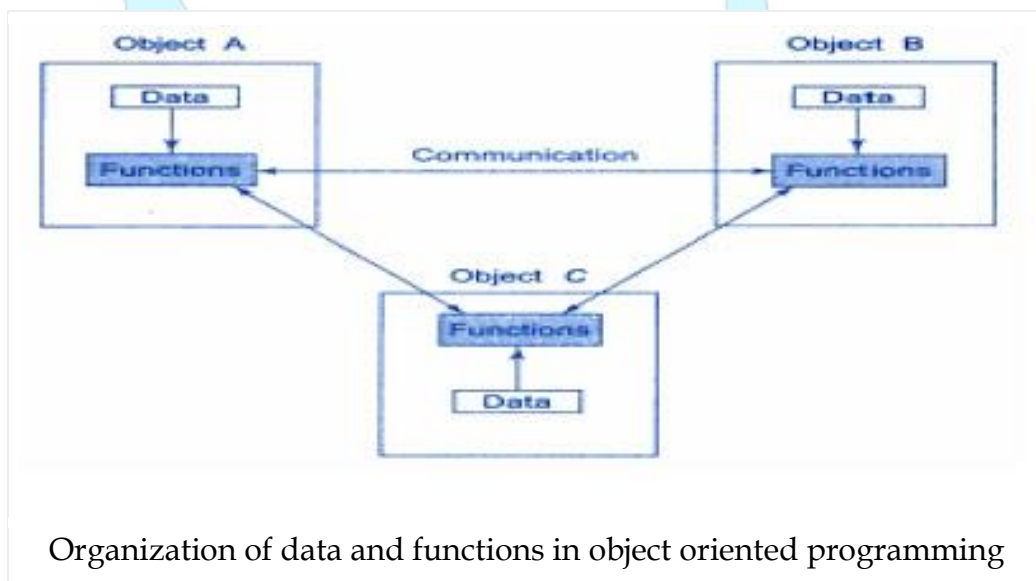
Disadvantages of Procedural languages

- 1) Procedural languages are difficult to relate with the real world objects.
- 2) Procedural codes are very **difficult to maintain**, if the code grows larger.
- 3) A procedural language does not have **automatic memory management** as like in Java. Hence, it makes the programmer to concern more about the memory management of the program.
- 4) The data, which is used in procedural languages are exposed to the whole program. So, there is **no security** for the data.

OBJECT ORIENTED PROGRAMMING (OOP) :-

The major motivating factor in the invention of object oriented is to remove some of the flaws encountered in the procedural oriented approach. Object oriented programming treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modifications from outside functions.

Object oriented programming allows a decomposition of a problem into a number entity called objects and then builds data and functions around these objects. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.



The object oriented programming can be defined as an "approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand ". Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Some characteristics of Object Oriented Programming are :-

- 1) Emphasis is on data rather than procedures or algorithms.
- 2) Programs are divided into what are known as objects.
- 3) Data structures are designed such that characterize the objects.
- 4) Functions that operate on the data are tied together in the data structure.
- 5) Data is hidden and cannot be accessed by external functions.
- 6) Objects may communicate with each other through functions.
- 7) New data and functions can be easily added whenever necessary.
- 8) Follows bottom-up approach in program design.

Advantages/Benefits of Object Oriented Programming over Procedure Oriented Programming: -

1. **Reusability:** In OOP's programs functions and modules that are written by a user can be reused by other users without any modification.
2. **Inheritance:** Through this we can eliminate redundant code and extend the use of existing classes.
3. **Data Hiding:** The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs.
4. **Reduced complexity of a problem:** The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.
5. **Easy to maintain and Upgrade:** OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
6. **Message Passing:** The technique of message communication between objects makes the interface with external systems easier.
7. **Modifiability:** it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods;

Difference between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called functions .	In OOP, program is divided into parts called objects .
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world .
Approach	POP follows Top Down approach .	OOP follows Bottom Up approach .
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.

Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP is: C, VB, FORTRAN, and Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

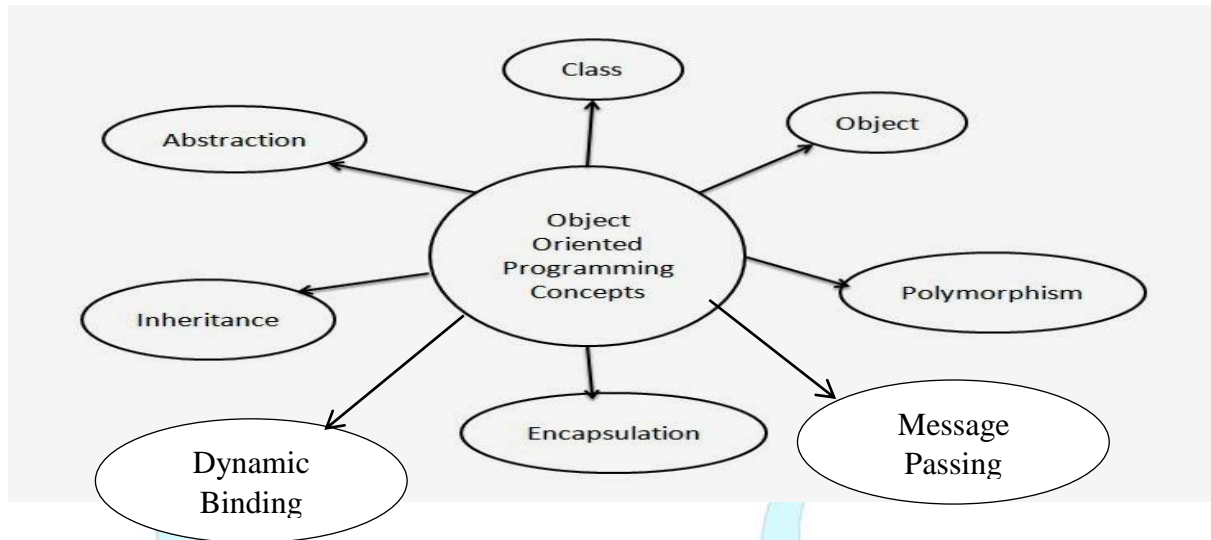
CONCEPTS OF OOP'S

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

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

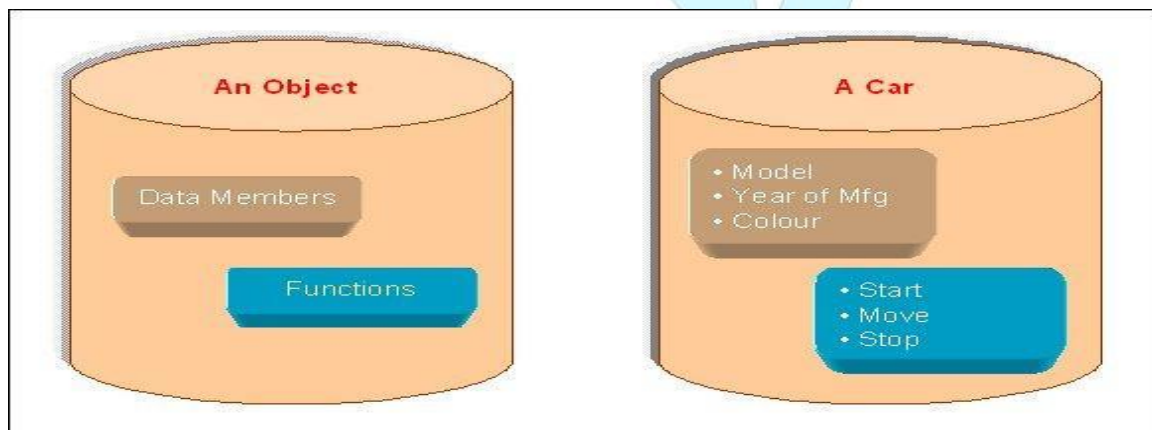
There are a few principle concepts that form the foundation of object-oriented programming:

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



OBJECT:

Any real world entity which can have some characteristics or which can perform some work is called as Object. Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of a class. Each instance of a class can hold its own relevant data.



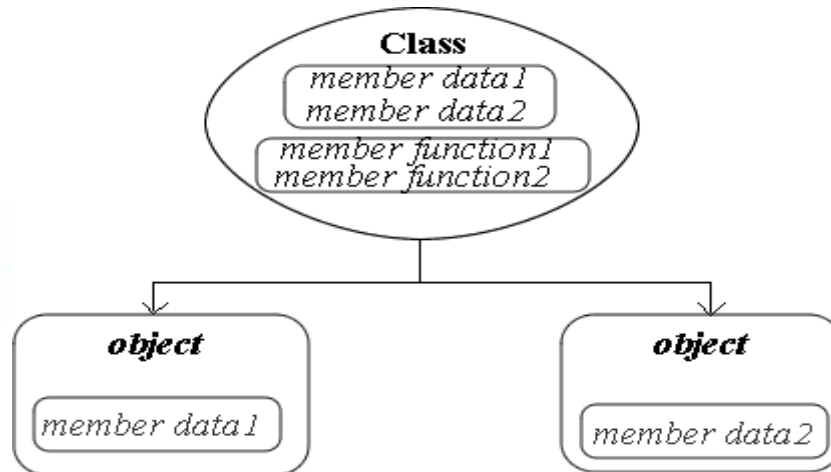
An Object is a collection of data members and associated member functions also known as methods.

Object is used for run the class or invokes the class. So many objects can create for a single class. Objects are the basic run-time entities in an object oriented system. They may represent a person, a place or any item that the program has to handle.

Example: Man, Table, Bank Account, Bird, Car etc.,

CLASS:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and are referred to as Methods.



For example consider we have a Class of Cars under which Santro Xing, Alto and Wagener represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Color, Top Speed, Engine Power etc., which form Properties of the Car class and the associated actions i.e., object functions like Start, Move, and Stop form the Methods of Car Class.

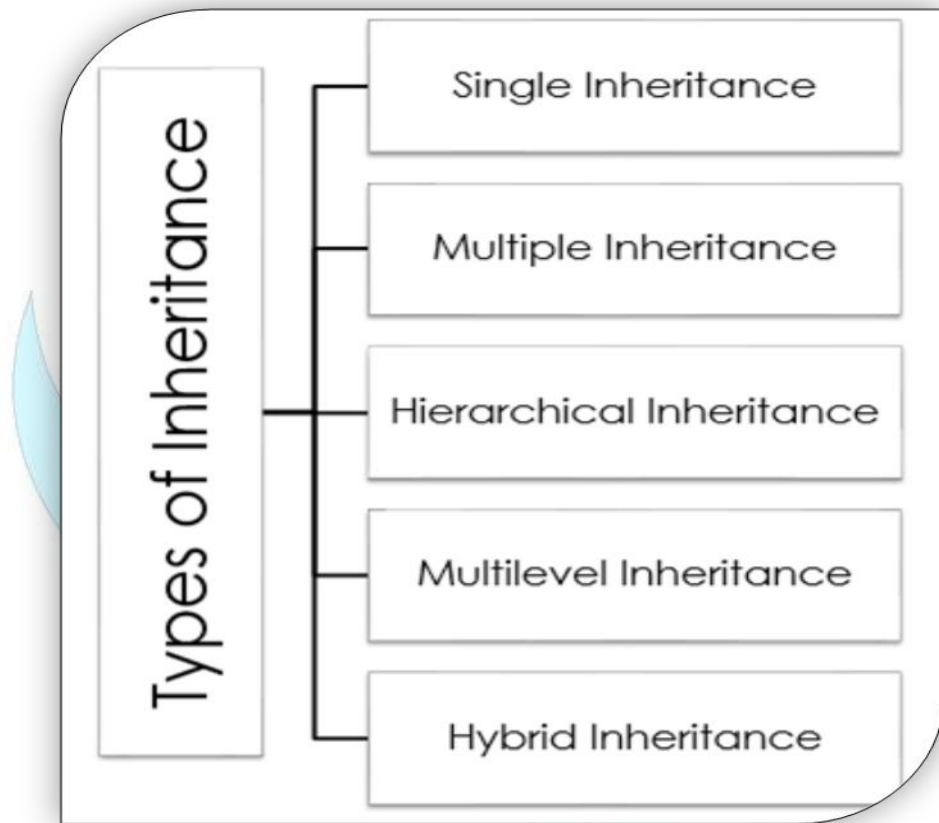
No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

INHERITANCE OR REUSABILITY

The mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance.

Inheritance lets you create new classes from existing class. Any new class that you create from an existing class is called **derived class**; existing class is called **base class**.

The inheritance relationship enables a derived class to inherit features from its base class. Furthermore, the derived class can add new features of its own. Therefore, rather than create completely new classes from scratch, you can take advantage of inheritance and reduce software complexity.

TYPES OF INHERITANCE

Single Inheritance: It is the inheritance hierarchy wherein one derived class inherits from one base class.

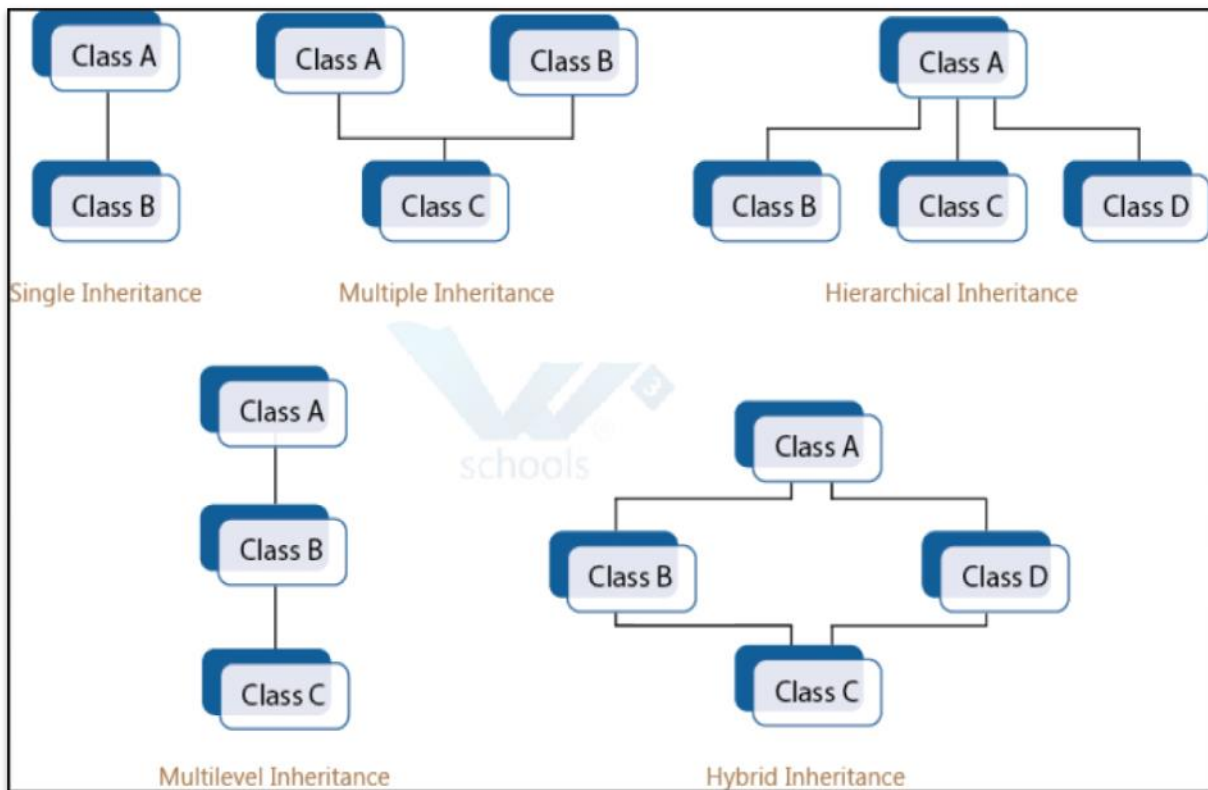
Multiple Inheritance: It is the inheritance hierarchy wherein one derived class inherits from multiple base classes.

Hierarchical Inheritance: It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

Multilevel Inheritance: It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

Hybrid Inheritance: The inheritance hierarchy that reflects any legal combination of other four types of inheritance.

SHREE MEDHA COLLEGE

**DATA ABSTRACTION:**

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

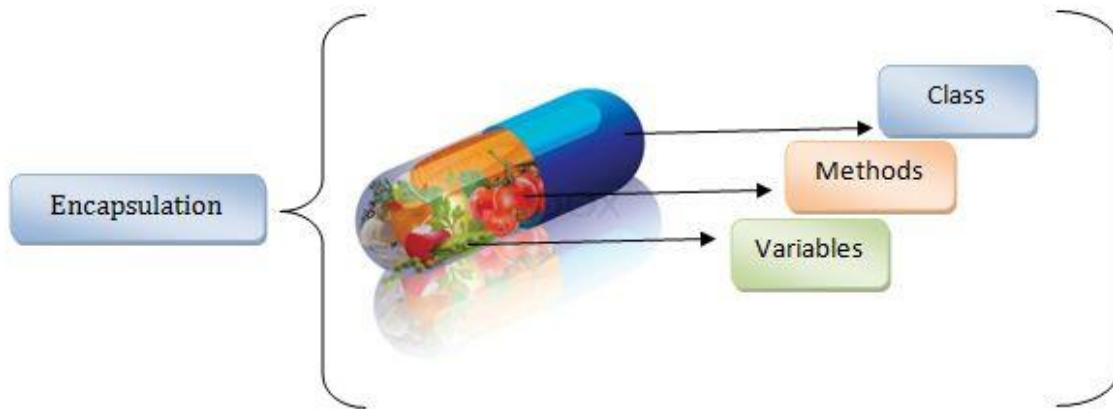
Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

DATA ENCAPSULATION OR DATA/INFORMATION HIDING OR SECURITY:

Encapsulation is an Object Oriented Programming concept. Encapsulation is a process of binding or wrapping the data and the codes that operates on the data into a single entity. This keeps the data safe from outside interface and misuse. One way to think about encapsulation is as a protective wrapper that prevents code and data from being arbitrarily accessed by other code defined outside the wrapper. Data encapsulation led to the important OOP concept of **data hiding**.

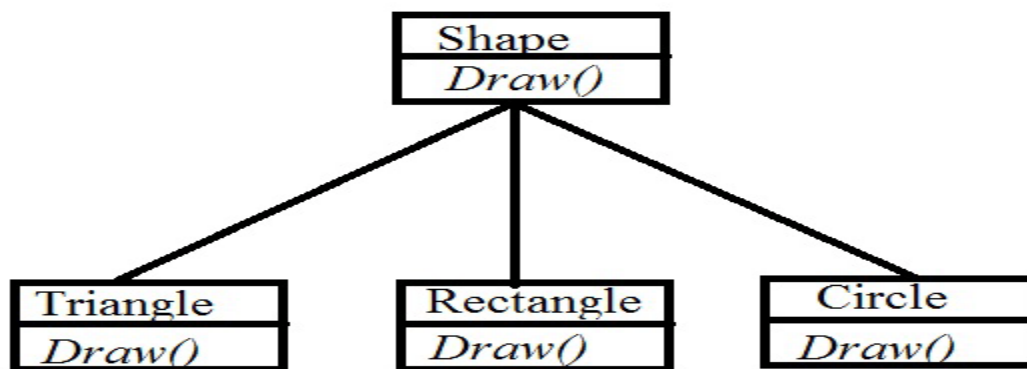
**Example:****TV operation**

It is encapsulated with cover and we can operate with remote and no need to open TV and change the channel.

Here everything is in private except remote so that anyone can access not to operate and change the things in TV.

POLYMORPHISM

"Poly" means "many" and "morph" means "form". Polymorphism is the ability of an object (or reference) to assume (be replaced by) or become many different forms of object. Example: function overloading, function overriding, virtual functions.

**Example1:**

A Teacher behaves to student.

A Teacher behaves to his/her seniors.

Here teacher is an object but attitude is different in different situation.

Example2:

Below is a simple example of the above concept of polymorphism:

6 + 10

The above refers to integer addition.

The same + operator can also be used for floating point addition:

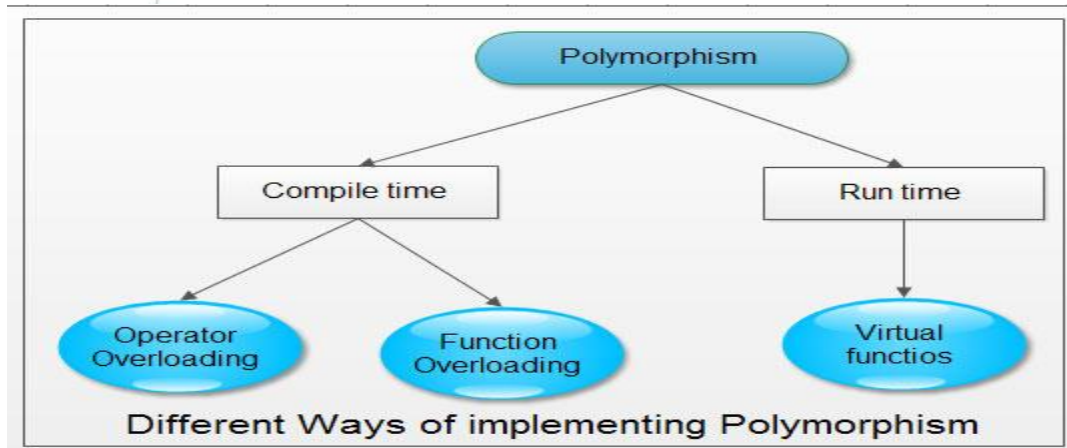
7.15 + 3.78

TYPES OF POLYMORPHISM

There are two types of polymorphism one is compile time polymorphism and the other is run time polymorphism. Compile time polymorphism is functions and operators overloading. Runtime time polymorphism is done using inheritance and virtual functions. Here are some ways how we implement polymorphism in Object Oriented programming languages.

Compile time polymorphism -> Operator Overloading, Function Overloading

Run time polymorphism -> Interface and abstract methods, Virtual member functions.



Compile Time Polymorphism:

The compiler is able to select the appropriate function for a particular call at compile-time itself. This is known as compile-time polymorphism. The *compile-time* polymorphism is implemented with templates.

- Function name overloading
- Operator overloading

FUNCTION OVERLOADING

Using a single function name to perform different types of tasks is known as function overloading. Using the concept of function overloading, design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

OPERATOR OVERLOADING

The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

Run-time:

The appropriate member function could be selected while the programming is running. This is known as run-time polymorphism. The *run-time* polymorphism is implemented with inheritance and virtual functions.

- Virtual functions

VIRTUAL FUNCTIONS

A function qualified by the virtual keyword. When a virtual function is called via a pointer, the class of the object pointed to determine which function definition will be used. Virtual functions implement polymorphism, whereby objects belonging to different classes can respond to the same message in different ways.

DYNAMIC BINDING

Binding refers connecting the function call to a function definition/implementation. Or It is a link between the function call and function definition.

Types of Binding

- Static Binding.
- Dynamic Binding.

STATIC BINDING:

By default, matching of function call with the correct function definition happens at compile time. This is called static binding or early binding or compile-time binding. Static binding is achieved using function overloading and operator overloading. Even though there are two or more functions with same name, compiler uniquely identifies each function depending on the parameters passed to those functions.

DYNAMIC BINDING:

C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time; this is called dynamic binding or late binding or run-time binding. Dynamic binding is achieved using virtual functions. Base class pointer points to derived class object. And a function is declared virtual in base class, then the matching function is identified at run-time using virtual table entry.

MESSAGE PASSING

Message passing nothing but sending and receiving of information by the objects same as people exchange information. So this helps in building systems that simulate real life. Following are the basic steps in message passing.

- Creating classes that define objects and its behavior.
- Creating objects from class definitions
- Establishing communication among objects

In OOPs, Message Passing involves specifying the name of objects, the name of the function, and the information to be sent.

For example: student.mark (name). Here student is object, mark is message, and name is information.

APPLICATIONS OF OOP:

Applications of OOP are beginning to gain importance in many areas. The most important application is user interface design. Real business systems are more complex and contain many attributes and methods, but OOP applications can simplify a complex problem.

1. User interface design such as windows, menu.
2. Real Time Systems
3. Simulation and Modeling
4. Object oriented databases
5. AI and Expert System
6. Neural Networks and parallel programming
7. Decision support and office automation system
8. Computer animation
9. To design Compiler
10. To access relational data base
11. To develop administrative tools and system tools
12. To develop computer games

DIFFERENCE BETWEEN ABSTRACTION AND ENCAPSULATION: -

ABSTRACTION	ENCAPSULATION
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. Abstraction - Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. Encapsulation - Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

OBJECT ORIENTED LANGUAGES (OOL)

Object-oriented language was primarily designed to reduce complexity in typical procedural languages through data binding and encapsulation techniques. In object-oriented language, the objects created provide limited or no access to other functions or methods within the program. This enables only authorized or inherited methods/functions to access a particular object.

Object-oriented language typically supports the following features, at minimum:

- The ability to create classes and their associated objects
- Encapsulation
- Inheritance

Depending on the features they support, they can be classified into two categories

- 1) Object based programming languages.
- 2) Object oriented programming.

OBJECT BASED PROGRAMMING LANGUAGES

Object based programming is a style of programming that preliminary support encapsulation & object entity. Major feature that are required for the object based programming are data Encapsulation, Automatic Initialization & clear up of objects, operator overloading, function overloading.

Languages that support programming with objects are said to be object based programming language. They do not support inheritance & dynamic binding. Example **ADA**

OBJECT ORIENTED PROGRAMMING

OOP incorporates all object based programming language features along with two additional features i.e., Object based inheritance & Dynamic binding etc.

Example: C++, Java, Simula, Small Talk, Eiffel etc.

INTRODUCTION & HISTORY OF C++

C++ is a general-purpose programming language. It has imperative, oriented and generic programming features, while also providing the facilities for low-level memory manipulation. In the early 1980's, also at Bell Laboratories, another programming language was created which was based upon the C language. This new language was developed by Bjarne Stroustrup and was called C++. Stroustrup states that the purpose of C++ is to make writing good programs easier and more pleasant for the individual programmer. When he designed C++, he added OOP (Object Oriented Programming) features to C without significantly changing the C component. Thus *C++ is a "relative" (called a superset)* of C, meaning that any valid C program is also a valid C++ program. C++ is also called as C with classes.

Standardization

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998 ^[12]	C++98
2003	ISO/IEC 14882:2003 ^[13]	C++03
2007	ISO/IEC TR 19768:2007 ^[14]	C++07/TR1
2011	ISO/IEC 14882:2011 ^[4]	C++11
2014	ISO/IEC 14882:2014 ^[15]	C++14
2017	to be determined	C++17

WHY C++ OR FEATURES/ADVANTAGES OF CPP

C++ has certain characteristics over other programming languages. The most remarkable are:

Object-oriented programming

The possibility to orientate programming to objects allows the programmer to design applications from a point of view more like a communication between objects rather than on a structured sequence of code. In addition it allows a greater reusability of code in a more logical and productive way.

Portability

You can practically compile the same C++ code in almost any type of computer and operating system without making any changes. C++ is the most used and ported programming language in the world.

Brevity

Code written in C++ is very short in comparison with other languages, since the use of special characters is preferred to key words, saving some effort to the programmer (and prolonging the life of our keyboards!).

Modular programming

An application's body in C++ can be made up of several source code files that are compiled separately and then linked together. Saving time since it is not necessary to recompile the complete application when making a single change but only the file that contains it. In addition, this characteristic allows to link C++ code with code produced in other languages, such as Assembler or C.

C Compatibility

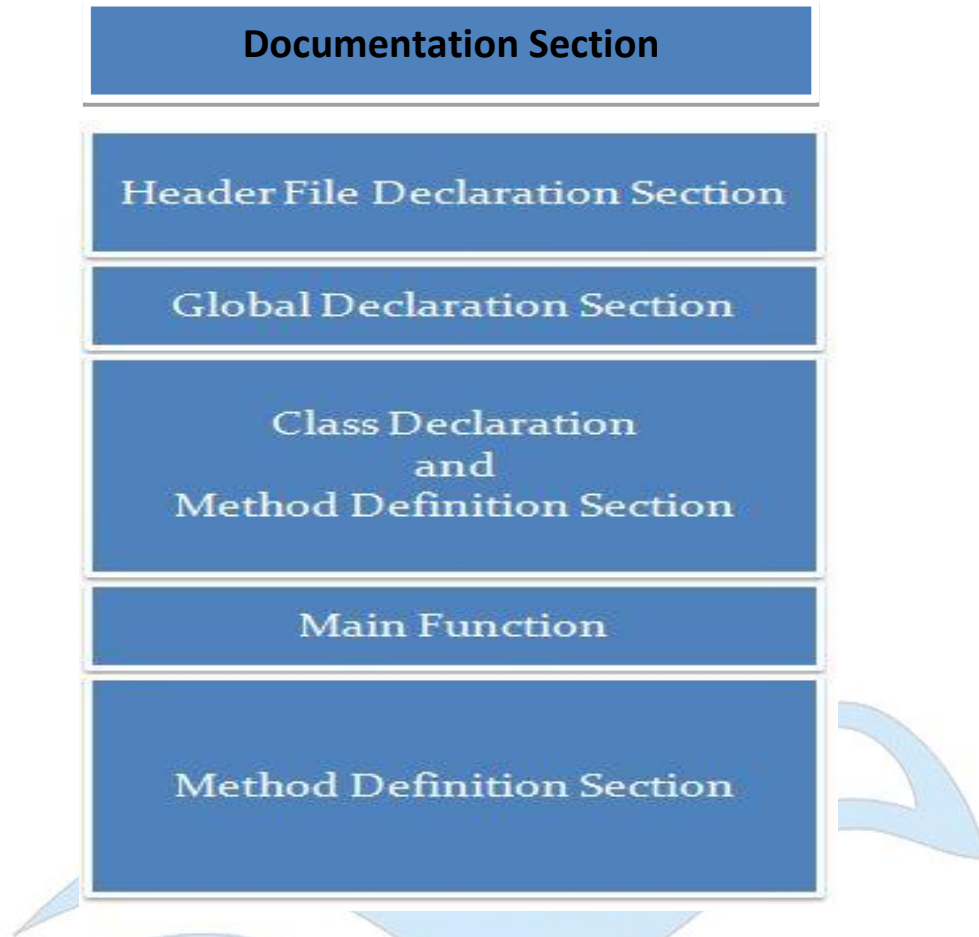
C++ is backwards compatible with the C language. Any code written in C can easily be included in a C++ program without making any change.

Speed

The resulting code from a C++ compilation is very efficient, due indeed to its duality as high-level and low-level language and to the reduced size of the language itself.

BASIC STRUCTURE OF C++ PROGRAM

As C++ is a programming language so it follows a predefined structure. The program is divided into many sections; it is important to know the need of every section. The easiest way to understand the basic structure of C++ program is by writing a program. The basic C++ program is as follows:



The basic structure of C++ program mentioned above can be divided into following sections:

Documentation Section: This section comprises of comments. As the name suggests, this section is used to improve the readability and understanding of the program. // (Double Slash) represents comments in C++ program. Comments can be of single line or multiple lines. Double Slash comments are used to represent single line comments. For multiple line comment, you can begin with /* and end with */. For example:

```
// Text line number (Single line)
```

```
/* Text line number 1
```

```
Text line number 2
```

```
Text line number 3 */ (Multi line)
```

Linking and Directives Section: The program written above begins with #include<iostream.h>. <iostream.h> represents header file which includes the functionalities of predefined functions. In linking section, the compiler in-built

functions such as `cout<<`, `cin>>` etc are linked with INCLUDE subdirectory's header file `<iostream.h>`. The '#' symbol tells about "address to" or "link to". `iostream` is input/output stream which includes declarations of standard input-output library in C++. We can also use other header files as per the requirement in the program like `#include<conio.h>`, `#include<math.h>` etc.

Global Declaration Section: There are certain programs which requires variables that can be used in more than one function, so then the variables can be declared outside the `main()` function or respective functions. Then those variables become accessible in any of the functions, hence named as Global Variables as their scope becomes global to the program.

Class Definition: *Classes* are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are defined using keyword `class`, with the following syntax:

```
class class_name
{
    access_specifier_1:
    data member;
    access_specifier_2:
    member functions;
    ...
} object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain *members*, which can either be data or function declarations, and optionally *access specifiers*.

main() Section: This is the section in which the program coding is written. Basically, it acts as a container for C++ program. The execution of the C++ program begins with `main ()` function and it is independent of the location of `main ()` function in the program. `Main ()` is a function as represented by parenthesis "()". This is because it is a function declaration. The body of the `main ()` function can be found right after these parenthesis, the body is enclosed in braces "{}".

Body of main () Section: The body of the `main()` function begins with "{".

- o **Local Variable Declaration:** In this the variables which are used in the body of the `main ()` functions are declared. These are called the local variables as their scope is limited within the `main ()` function only, unless they are

declared globally outside the main () function. " **int a=10, b=34;**" in the above program represents local variables. We can also declare the variables anywhere in the C++. It is also referred as flexible declaration.

- **Statements to Execute:** This section includes statements for reading, writing and executing data using I/O functions, library functions, formulas, conditional statements etc. cin & cout are used for input & output statements in C++.cin>> is used for input & cout<< for output.
- **return 0;** in the above program causes the function to finish and 0 represents that function has been executed with zero errors. This is considered as most usual way to end a C++ program.
- Finally, the body of the main () function ends with "}".

User Defined Functions (Methods): There are certain functions that are called by calling statements from the main () function. Every function includes local variable declaration section and executable statement section similar to main program.

ENUMERATED DATA TYPE

C++ allows programmers to create their own data types. Perhaps the simplest method for doing so is via an enumerated type. An **enumerated type** is a data type where every possible value is defined as a symbolic constant (called an **enumerator**). Enumerated types are declared via the **enum** keyword. Let's look at an example:

```
// define a new enum named Color
enum Color
{
    // Here are the enumerators
    // These define all the possible values this type can hold
    COLOR_BLACK,
    COLOR_RED,
    COLOR_BLUE,
    COLOR_GREEN,
    COLOR_WHITE,
    COLOR_CYAN,
    COLOR_YELLOW,
    COLOR_MAGENTA
};
// Declare a variable of enumerated type Color
Color eColor = COLOR_WHITE;
```

Defining an enumerated type does not allocate any memory. When a variable of the enumerated type is declared (such as eColor in the example above), memory is allocated for that variable at that time.

Enum variables are the same size as an int variable. This is because each enumerator is automatically assigned an integer value based on its position in the enumeration list. By default, the first enumerator is assigned the integer value 0, and each subsequent enumerator has a value one greater than the previous enumerator:

```
enum Color
{
    COLOR_BLACK, // assigned 0
    COLOR_RED, // assigned 1
    COLOR_BLUE, // assigned 2
    COLOR_GREEN, // assigned 3
    COLOR_WHITE, // assigned 4
    COLOR_CYAN, // assigned 5
    COLOR_YELLOW, // assigned 6
    COLOR_MAGENTA // assigned 7
};

Color eColor = COLOR_WHITE;
cout << eColor;
```

The cout statement above prints the value 4.

It is possible to explicitly define the value of enumerator. These integer values can be positive or negative and can be non-unique. Any non-defined enumerators are given a value one greater than the previous enumerator.

```
// define a new enum named Animal
enum Animal
{
    ANIMAL_CAT = -3,
    ANIMAL_DOG, // assigned -2
    ANIMAL_PIG, // assigned -1
    ANIMAL_HORSE = 5,
    ANIMAL_GIRAFFE = 5,
    ANIMAL_CHICKEN // assigned 6
};
```

Because enumerated values evaluate to integers, they can be assigned to integer variables:

```
int nValue = ANIMAL_PIG;
```

However, the compiler will not implicitly cast an integer to an enumerated value. The following will produce a compiler error:

```
Animal eAnimal = 5; // will cause compiler error
```

It is possible to use a static_cast to force the compiler to put an integer value into an enumerated type, though it's generally bad style to do so:

```
Animal eAnimal = static_cast<Animal>(5); // compiler won't complain, but bad style
```

Each enumerated type is considered a distinct type. Consequently, trying to assign enumerators from one enum type to another enum type will cause a compile error:

Animal eAnimal = COLOR_BLUE; // will cause compile error

Enumerated types are incredibly useful for code documentation and readability purposes when you need to represent a specific number of states.

For example, functions often return integers to the caller to represent error codes when something went wrong inside the function. Typically, small negative numbers are used to represent different possible error codes. For example:

```
int ParseFile()
{
    if (!OpenFile())
        return -1;
    if (!ReadFile())
        return -2;
    if (!Parsefile())
        return -3;

    return 0; // success
}
```

However, using magic numbers like this isn't very descriptive. An alternative method would be through use of an enumerated type:

```
enum ParseResult
{
    SUCCESS = 0,
    ERROR_OPENING_FILE = -1,
    ERROR_READING_FILE = -2,
    ERROR_PARSING_FILE = -3,
};

ParseResult ParseFile()
{
    if (!OpenFile())
        return ERROR_OPENING_FILE;
    if (!ReadFile())
        return ERROR_READING_FILE;
    if (!Parsefile())
        return ERROR_PARSING_FILE;
    return SUCCESS;
}
```

This is much easier to read and understand than using magic number return values. Furthermore, the caller can test the function's return value against the appropriate enumerator, which is easier to understand than testing the return result for a specific integer value.

```
if (ParseFile() == SUCCESS)
{
    // do something
}
else
{
    // print error message
}
```

Another use for enum is as array indices, because enumerator indices are more descriptive than integer indices. We will cover this in more detail in the section on arrays.

Finally, as with constant variables, enumerated types show up in the debugger, making them more useful than #defined values in this regard.

REFERENCE VARIABLES

A reference variable allows us to create an alternative name for already defined variable. It is introduced in C++. Once you define a reference variable that references already defined variable then you can use any of them alternatively in the program. Both refer to the same memory location. Thus if you change value of any of the variable it will affect both variables because one variable reference to another variable.

The general syntax of creating reference variable is given below: Data-type &

Referace_Name=Variable_Name;

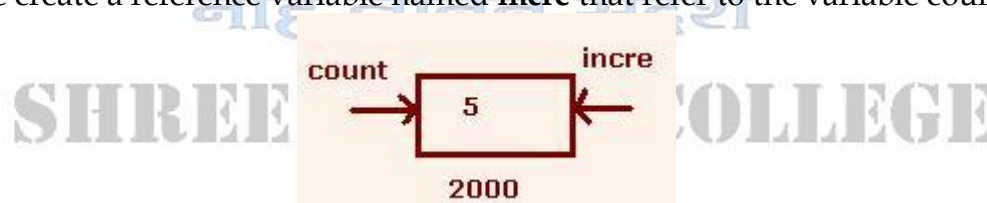
The reference variable must initialize at the time of declaration.

Example:

```
int count;
count = 5;
Int & incre = count;
```

Here, We have already defined variable named **count**.

Then we create a reference variable named **incre** that refer to the variable count.



Since both variable refer to same memory location if you change value of variable count using following statement:

Count=count + 5;

Will change the value of variable count and incre to 10.

The major use of reference variable is in function. If we want to pass reference of the variable at the time of function call so that the function works with original variable we can use reference variable.

MANIPULATORS

- **Manipulators** are functions that are used to format or modify the output stream in various ways.
- They have a special characteristic that they are used along with insertion (<<) operator to change the format of the data.
- There are many manipulators in C++. But as of now, we will be dealing only with the **endl** and **setw** manipulators.
- To be able to use manipulators in our program, we must include **<iomanip>** header file in our source program.
- But here is an exception – the endl manipulator can be used without including the <iomanip> file.

Now let us see how these two manipulators work.

1. endl

The endl manipulator works the same way as the '\n' character in C++. That is the endl manipulator outputs the subsequent data or text in the next line.

But the difference is that endl also flushes the output buffer when it is used in a program.

Here is a sample program that shows how endl works.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    int num;
    cout<<"Enter your roll number: ";
    cin>>num;

    cout<<"Hello roll number "<<num<<endl;
    cout<<"Welcome to your new class!!";
    getch();
    return 0;
}
```

In the above program, endl operator is used in the fourth statement inside main() function.

After this statement, the string "Welcome to your new class!!" is printed in the next line on the output screen.

SETW MANIPULATOR

This manipulator is used to set the width of the output in a program. It takes up an argument 'n' which is the width of the field in which the output is to be displayed.

By default, the output in the field is right-aligned.

Here is a sample program to show how it works.

```
<em>#include<iostream>
#include<conio.h>
#include<iomanip>
using namespace std;
int main()
{
int num1,num2,num3;
cout<<"Enter three numbers:\n";
cin>>num1>>num2>>num3;
cout<<"\nDisplaying the three numbers\n"
    <<"Num1:"<<setw(8)<<num1<<endl
    <<"Num2:"<<setw(8)<<num2<<endl
    <<"Num3:"<<setw(8)<<num3<<endl;
getch();
return 0;
}
</em>
```

In the program above, we ask the user to input three integers and then we display them using setw() manipulator.

Notice how we have cascaded the '<<' operator to display the three numbers. We could have used individual cout statements for displaying each number.

BASIC INPUT/OUTPUT

The example programs of the previous sections provided little interaction with the user, if any at all. They simply printed simple values on screen, but the standard library provides many additional ways to interact with the user via its input/output features. This section will present a short introduction to some of the most useful.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file. A *stream* is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs:

Stream	description
Cin	standard input stream
Cout	standard output stream
Cerr	standard error (output) stream
Clog	standard logging (output) stream

We are going to see in more detail only cout and cin (the standard output and input streams); cerr and clog are also output streams, so they essentially work like cout, with the only difference being that they identify streams for specific purposes: error messages and logging; which, in many cases, in most environment setups, they actually do the exact same thing: they print on screen, although they can also be individually redirected.

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).

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

The << operator inserts the data that follows it into the stream that precedes it. In the examples above, it inserted the literal string Output sentence, the number 120, and the value of variable x into the standard output stream cout. Notice that the sentence in the first statement is enclosed in double quotes (") because it is a string literal, while in the last one, x is not. The double quoting is what makes the difference; when the text is enclosed between them, the text is printed literally; when they are not, the text is interpreted as the identifier of a variable, and its value is printed instead. For example, these two sentences have very different results:

```
1 cout << "Hello"; // prints Hello
2 cout << Hello;   // prints the content of variable Hello
```

Multiple insertion operations (<<) may be chained in a single statement:

```
cout << "This " << " is a " << "single C++ statement";
```

This last statement would print the text This is a single C++ statement. Chaining

insertions is especially useful to mix literals and variables in a single statement:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

Assuming the *age* variable contains the value 24 and the *zipcode* variable contains 90064, the output of the previous statement would be:

I am 24 years old and my zipcode is 90064

What cout does not do automatically is add line breaks at the end, unless instructed to do so. For example, take the following two statements inserting into cout:

```
cout << "This is a sentence.";
```

```
cout << "This is another sentence.";
```

The output would be in a single line, without any line breaks in between. Something like:

This is a sentence. This is another sentence.

To insert a line break, a new-line character shall be inserted at the exact position the line should be broken. In C++, a new-line character can be specified as `\n` (i.e., a backslash character followed by a lowercase n). For example:

```
1 cout << "First sentence.\n";
```

```
2 cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

First sentence.

Second sentence.

Third sentence.

Alternatively, the `endl` manipulator can also be used to break lines. For example:

```
cout << "First sentence." << endl;
```

```
cout << "Second sentence." << endl;
```

This	would	print:
First		sentence.
Second		sentence.

The `endl` manipulator produces a newline character, exactly as the insertion of `'\n'` does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly *fully buffered* streams, and `cout` is (generally) not a *fully buffered* stream. Still, it is generally a good idea to use `endl` only when flushing the stream would be a feature and `'\n'` when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.

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:

```
int age;  
cin >> age;
```

The first statement declares a variable of type `int` called `age`, and the second extracts from `cin` a value to be stored in it. This operation makes the program wait for input from `cin`; generally, this means that the program will wait for the user to enter some sequence with the keyboard. In this case, note that the characters introduced using the keyboard are only transmitted to the program when the ENTER (or RETURN) key is pressed. Once the statement with the extraction operation on `cin` is reached, the program will wait for as long as needed until some input is introduced.

The extraction operation on `cin` uses the type of the variable after the `>>` operator to determine how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

```
// i/o example
```

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

```
void main ()
```

```
{
```

```
    int i;
```

```
    cout << "Please enter an integer value:
```

```
";
```

```
    cin >> i;
```

```
    cout << "The value you entered is "
```

```
<< i;
```

```
    cout << " and its double is " << i*2 << " Please enter an integer value: 702
```

```
".\n";
```

```
}
```

Edit & Run

The value you entered is 702 and its double is 1404.

As you can see, extracting from `cin` seems to make the task of getting input from the standard input pretty simple and straightforward. But this method also has a big drawback. What happens in the example above if the user enters something else that cannot be interpreted as an integer? Well, in this case, the extraction operation fails. And this, by default, lets the program continue without setting a value for variable `i`, producing undetermined results if the value of `i` is used later.

This is very poor program behavior. Most programs are expected to behave in an expected manner no matter what the user types, handling invalid values appropriately. Only very simple programs should rely on values extracted directly from `cin` without further checking. A little later we will see how *stringstreams* can be used to have better control over user input. Extractions on `cin` can also be chained to request more than one datum in a single statement:

```
cin >> a >> b;
```

This is equivalent to:


```
1 cin >> a;  
2 cin >> b;
```

In both cases, the user is expected to introduce two values, one for variable a, and another for variable b. Any kind of space is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.

cin and strings

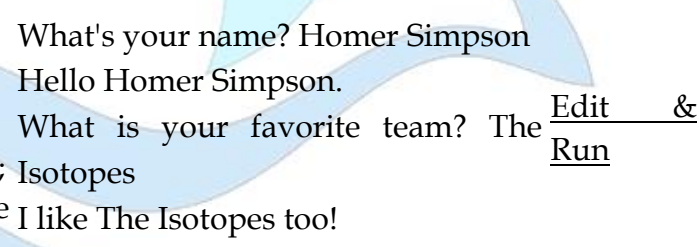
The extraction operator can be used on cin to get strings of characters in the same way as with fundamental data types:

```
1 string mystring;  
2 cin >> mystring;
```

However, cin extraction always considers spaces (whitespaces, tabs, new-line...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.

To get an entire line from cin, there exists a function, called getline, that takes the stream (cin) as first argument, and the string variable as second. For example:

```
// cin with strings  
#include <iostream.h>  
#include <string.h>  
void main ()  
{  
    string mystr;           What's your name? Homer Simpson  
    cout << "What's your name? ";   Hello Homer Simpson.  
    getline (cin, mystr);          What is your favorite team? The  
    cout << "Hello " << mystr << ".\n"; Isotopes  
    cout << "What is your favorite I like The Isotopes too!  
team? ";  
    getline (cin, mystr);  
    cout << "I like " << mystr << "  
too!\n";  
}
```

 Edit & Run

Notice how in both calls to getline, we used the same string identifier (mystr). What the program does in the second call is simply replace the previous content with the new one that is introduced.

The standard behavior that most users expect from a console program is that each time the program queries the user for input, the user introduces the field, and then presses ENTER (or RETURN). That is to say, input is generally expected to happen in terms of lines on console programs, and this can be achieved by using getline to obtain input from the user. Therefore, unless you have a strong reason not to, you should always use getline to get input in your console programs instead of extracting from cin.

stringstream

The standard header `<sstream>` defines a type called `stringstream` that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on `cin` and `cout`. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

This declares a string with initialized to a value of "1204", and a variable of type `int`. Then, the third line uses this variable to extract from a `stringstream` constructed from the string. This piece of code stores the numerical value 1204 in the variable called `myint`.

```
// stringstreams
#include <iostream.h>
#include <string.h>
#include <sstream.h>
void main ()
{
    string mystr;
    float price=0;
    int quantity=0;
    cout << "Enter price: ";
    getline (cin,mystr);
    stringstream(mystr) >> price;
    cout << "Enter quantity: ";
    getline (cin,mystr);
    stringstream(mystr) >> quantity;
    cout << "Total price: " << price*quantity << endl;
}
```

Enter price: 22.25

Enter quantity: 7 [Edit & Run](#)

Total price: 155.75

In this example, we acquire numeric values from the *standard input* indirectly: Instead of extracting numeric values directly from `cin`, we get lines from it into a string object (`mystr`), and then we extract the values from this string into the variables `price` and `quantity`. Once these are numerical values, arithmetic operations can be performed on them, such as multiplying them to obtain a total price.

With this approach of getting entire lines and extracting their contents, we separate the process of getting user input from its interpretation as data, allowing the input process to be what the user expects, and at the same time gaining more control over the transformation of its content into useful data by the program.

TYPE CONVERSIONS

Implicit conversion

Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
```

```
int b;
```

```
b=a;
```

Here, the value of a is promoted from short to int without the need of any explicit operator. This is known as a *standard conversion*. Standard conversions affect fundamental data types, and allow the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions.

Converting to int from some smaller integer type, or to double from float is known as *promotion*, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

- If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., -1 becomes the largest value representable by the type, -2 the second largest, ...).
- The conversions from/to bool consider false equivalent to *zero* (for numeric types) and to *null pointer* (for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.
- If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes *undefined behavior*.
- Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is *implementation-specific* (and may not be portable).

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

- *Null pointers* can be converted to pointers of any type
- Pointers to any type can be converted to void pointers.
- *Pointer upcast*: pointers to a derived class can be converted to a pointer of an *accessible* and *unambiguous* base class, without modifying its const or volatile qualification.

Implicit conversions with classes

In the world of classes, implicit conversions can be controlled by means of three member functions:

- **Single-argument constructors:** allow implicit conversion from a particular type to initialize an object.
- **Assignment operator:** allow implicit conversion from a particular type on assignments.
- **Type-cast operator:** allow implicit conversion to a particular type.

For example:

// implicit conversion of classes:

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

```
class A {};
```

```
class B {
```

```
public:
```

```
    // conversion from A (constructor):
```

```
    B (const A& x) {}
```

```
    // conversion from A (assignment):
```

```
    B& operator= (const A& x) {return *this;}
```

```
    // conversion to A (type-cast operator)
```

```
    operator A() {return A();}
```

Edit & Run

```
};
```

```
void main ()
```

```
{
```

```
    A foo;
```

```
    B bar = foo; // calls constructor
```

```
    bar = foo; // calls assignment
```

```
    foo = bar; // calls type-cast operator
```

```
    return 0;
```

```
}
```

The type-cast operator uses a particular syntax: it uses the operator keyword followed by the destination type and an empty set of parentheses. Notice that the return type is the destination type and thus is not specified before the operator keyword.

Keyword explicit

On a function call, C++ allows one implicit conversion to happen for each argument. This may be somewhat problematic for classes, because it is not always what is intended. For example, if we add the following function to the last example:

```
void fn (B arg) {}
```

This function takes an argument of type B, but it could as well be called with an object of type A as argument:

```
fn (foo);
```

This may or may not be what was intended. But, in any case, it can be prevented by marking the affected constructor with the explicit keyword:

```
// explicit:
#include <iostream.h>
class A {};
class B {
public:
    explicit B (const A& x) {}
    B& operator= (const A& x) {return *this;}
    operator A() {return A();}
};

void fn (B x) {}

void main ()
{
    A foo;
    B bar (foo);
    bar = foo;
    foo = bar;
    // fn (foo); // not allowed for explicit ctor.
    fn (bar);
    return 0;
}
```

Edit & Run

Additionally, constructors marked with explicit cannot be called with the assignment-like syntax; In the above example, bar could not have been constructed with:

```
B bar = foo;
```

Type-cast member functions (those described in the previous section) can also be specified as explicit. This prevents implicit conversions in the same way as explicit-specified constructors do for the destination type.

TYPE CASTING

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as *type-casting*. There exist two main syntaxes for generic type-casting: *functional* and *c-like*:

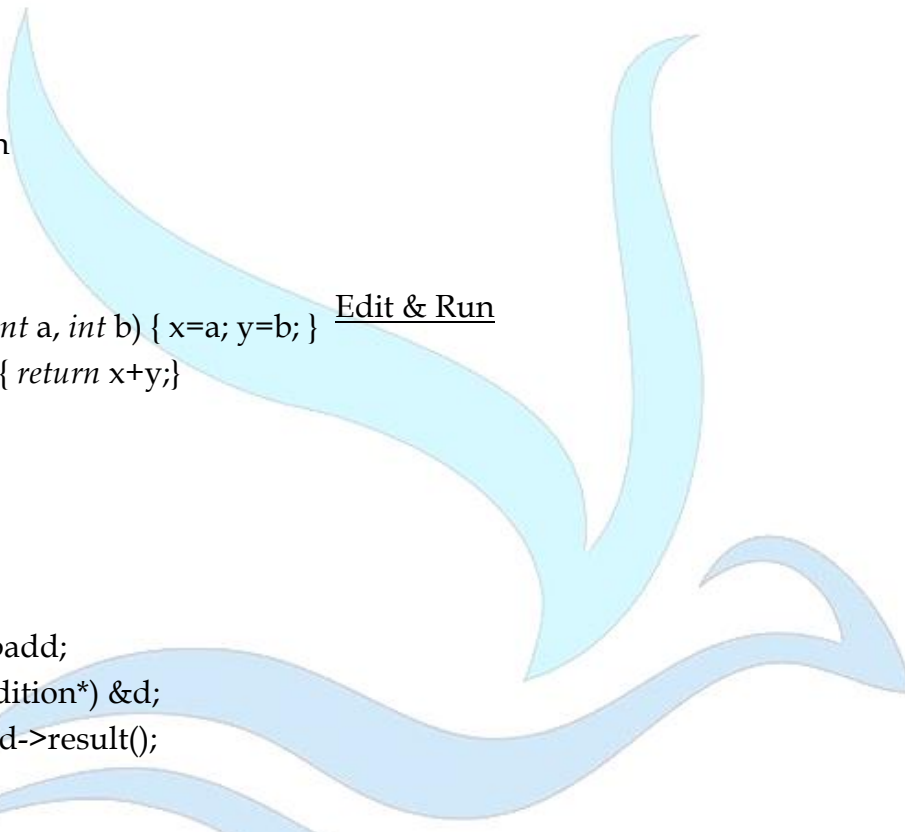
```
double x = 10.3;
int y;
y = int (x); // functional notation
y = (int) x; // c-like cast notation
```

The functionality of these generic forms of type-casting is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that - while being syntactically correct- can cause runtime errors. For example, the following code compiles without errors:

```
// class type-casting
#include <iostream.h>
class Dummy
{
    double i,j;
};

class Addition
{
    int x,y;
public:
    Addition (int a, int b) { x=a; y=b; }
    int result() { return x+y; }
};

void main ()
{
    Dummy d;
    Addition * padd;
    padd = (Addition*) &d;
    cout << padd->result();
}
```



The program declares a pointer to Addition, but then it assigns to it a reference to an object of another unrelated type using explicit type-casting:

```
padd = (Addition*) &d;
```

Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or some other unexpected results.

In order to control these types of conversions between classes, we have four specific casting operators: `dynamic_cast`, `reinterpret_cast`, `static_cast` and `const_cast`. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)

The traditional type-casting equivalents to these expressions would be:

(new_type) expression
new_type (expression)

but each one with its own special characteristics:

dynamic_cast

dynamic_cast can only be used with pointers and references to classes (or with void*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

This naturally includes *pointer upcast* (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an *implicit conversion*.

But dynamic_cast can also *downcast* (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type. For example:

```
// dynamic_cast
```

```
#include <iostream>
```

```
#include <exception>
```

```
using namespace std;
```

```
class Base { virtual void dummy() {} };
```

```
class Derived: public Base { int a; };
```

```
void main ()
```

```
{
```

```
    try
```

```
{
```

```
    Base * pba = new Derived;
```

```
    Base * pbb = new Base;
```

```
    Derived * pd;
```

```
    pd = dynamic_cast<Derived*>(pba);
```

```
    if (pd==0) cout << "Null pointer on first type  
cast.\n";
```

```
    pd = dynamic_cast<Derived*>(pbb);
```

```
    if (pd==0) cout << "Null pointer on second  
type-cast.\n";
```

```
}
```

Null pointer on second Edit &
type-cast. Run

```
catch (exception& e)
{cout << "Exception: " << e.what();}
    return 0;
}
```

Compatibility note: This type of `dynamic_cast` requires *Run-Time Type Information (RTTI)* to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This needs to be enabled for runtime type checking using `dynamic_cast` to work properly with these types.

The code above tries to perform two dynamic casts from pointer objects of type `Base*` (`pba` and `pbb`) to a pointer object of type `Derived*`, but only the first one is successful. Notice their respective initializations:

```
Base * pba = new Derived;
Base * pbb = new Base;
```

Even though both are pointers of type `Base*`, `pba` actually points to an object of type `Derived`, while `pbb` points to an object of type `Base`. Therefore, when their respective type-casts are performed using `dynamic_cast`, `pba` is pointing to a full object of class `Derived`, whereas `pbb` is pointing to an object of class `Base`, which is an incomplete object of class `Derived`.

When `dynamic_cast` cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a *null pointer* to indicate the failure. If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead.

`dynamic_cast` can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a `void*` pointer.

static_cast

`static_cast` can perform conversions between pointers to related classes, not only *upcasts* (from pointer-to-derived to pointer-to-base), but also *downcasts* (from pointer-to-base to pointer-to-derived). No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, it does not incur the overhead of the type-safety checks of `dynamic_cast`.

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

This would be valid code, although `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

Therefore, `static_cast` is able to perform with pointers to classes not only the conversions allowed implicitly, but also their opposite conversions.

`static_cast` is also able to perform all conversions allowed implicitly (not only those with pointers to classes), and is also able to perform the opposite of these. It can:

- Convert from `void*` to any pointer type. In this case, it guarantees that if the `void*` value was obtained by converting from that same pointer type, the resulting pointer value is the same.
- Convert integers, floating-point values and enum types to enum types.

Additionally, `static_cast` can also perform the following:

- Explicitly call a single-argument constructor or a conversion operator.
- Convert to *rvalue references*.
- Convert enum class values into integers or floating-point values.
- Convert any type to `void`, evaluating and discarding the value.

`reinterpret_cast`

`reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as `intptr_t`), is guaranteed to be able to be cast back to a valid pointer.

The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable.

For example:

```
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This code compiles, although it does not make much sense, since now `b` points to an object of a totally unrelated and likely incompatible class. Dereferencing `b` is unsafe.

const_cast

This type of casting manipulates the constness of the object pointed by a pointer, either to be set or to be removed. For example, in order to pass a const pointer to a function that expects a non-const argument:

```
// const_cast
#include <iostream.h>
void print (char * str)
{
    cout << str << '\n';
}
int main () {
    const char * c = "sample text";
    print ( const_cast<char *> (c) );
    return 0;
}
```

sample text [Edit & Run](#)

The example above is guaranteed to work because function print does not write to the pointed object. Note though, that removing the constness of a pointed object to actually write to it causes undefined behavior.

typeid allows to check the type of an expression:

typeid (expression)

This operator returns a reference to a constant object of type type_info that is defined in the standard header <typeinfo>. A value returned by typeid can be compared with another value returned by typeid using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

```
// typeid
#include <iostream.h>
#include <typeinfo.h>
void main ()
{
    int * a,b;
    a=0; b=0;
    if (typeid(a) != typeid(b))
    {
        cout << "a and b are of different
types:\n";
        cout << "a is: " << typeid(a).name() <<
'\n';
    }
}
```

a and b are of different
types:
a is: int *
b is: int

[Edit](#) &
[Run](#)

```
    cout << "b is: " << typeid(b).name() <<
'\n';
}
}
```

When typeid is applied to classes, typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

// typeid, polymorphic class

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

```
#include <typeinfo.h>
```

```
#include <exception.h>
```

```
class Base { virtual void f(){} };
```

```
class Derived : public Base {};
```

```
void main () {
```

```
    try {
```

```
        Base* a = new Base;
```

```
        Base* b = new Derived;
```

```
        cout << "a is: " << typeid(a).name() << '\n';
```

```
        cout << "b is: " << typeid(b).name() << '\n';
```

```
        cout << "*a is: " << typeid(*a).name() << '\n';
```

```
        cout << "*b is: " << typeid(*b).name() << '\n';
```

```
    } catch (exception& e) { cout << "Exception: " <<
```

```
e.what() << '\n'; }
```

```
}
```

a is: class Base *

b is: class Base *

*a is: class Base Edit &

*b is: class Run

Derived

Note: The string returned by member name of type_info depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.

Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class Base *). However, when typeid is applied to objects (like *a and *b) typeid yields their dynamic type (i.e. the type of their most derived complete object).

If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeid throws a bad_typeid exception.

FUNCTION PROTOTYPE/FUNCTION DECLARATION

The declaration of function in the program is known as Function Prototype.
The general syntax of function prototype is given below:

Return-type Function_ Name (Argument_List);
--

Example:

```
int sum (int a, int b);
```

Function Prototype provides following information to the compiler:

- (1) Name of the Function
- (2) Return Type of the Function
- (3) Number of arguments and their Data Type

Whenever compiler finds any function call statement, first it will check function prototype to ensure following things:

- (1) Whether the function that is called is declared or not.
- (2) Whether proper number of arguments is passed while calling the function or not.
- (3) Whether the data type of the arguments that are passed corresponds to the arguments specified in the function prototype or not.
- (4) If the function returns any value then it is corresponding to the return type specified in the function prototype or not.

If all the above mentioned criteria are satisfied, then control of the program is transferred to the function definition otherwise compiler will generate error.

While declaring the function you should keep following points in the mind:

- (1) You must specify data types for each argument separately.

Example:

```
int sum (int a, int b); // valid  
int sum (int a, b); // Not Valid
```

- (2) It is not compulsory to specify name of the arguments in the function declaration.

Example:

```
int sum (int, int); // valid
```

- (3) If function does not accept any argument then you can leave the parenthesis empty.

Example:

```
int sum (); // valid
```

Methods of Calling Function in C++

In C++ you can call the function using two methods:

(1) Call By Value:

In this method when the function is called it will first check function prototype to see whether the specified function is declared or not. If the function is declared then it will match number of arguments, data type and return type. Now compiler will create new variables and copy the value of the arguments in to newly created variable. Thus function works with the newly created variables. So it will not affect the original variables in the calling program. Using this method, we cannot alter the value of the original variable.

(2) Call By Reference:

In this method when the function is called it will first check function prototype to see whether the specified function is declared or not. If the function is declared then it will match number of arguments, data type and return type. Now instead of passing value of the variable reference of the variable is passed to the function using the concept of reference variable. Thus function works with the original variable. Using this method, we can alter the value of the original variable.

Example:

```
#include <iostream.h>
void main()
{
    int a,b;
    void swap(int &x, int &y);
    cout<<"Enter Two Numbers:";
    cin>>a>>b;
    cout<<"Before Swap\n";
    cout<<"A="<<a<<endl<<"B="<<b<<endl;
    swap(a,b);
    cout<<"After Swap\n";
    cout<<"A="<<a<<endl<<"B="<<b;
    return 0;
}
void swap (int &x, int &y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}
```

Output:

Enter Two Numbers: 10 20
Before Swap
A=10
B=20
After Swap
A=20
B=10

Return by reference

As we can pass reference of the variable to the function, function can also return a reference of the variable to the calling program.

Example:

```
#include<iostream.h>
void main()
{
    int a,b;
    int& min(int &x, int &y);
    cout<<"Enter Two Numbers";
    cin>>a>>b;
    min(a,b)=0;
    if(a==0)
    cout<<"Value of a is less then b so it is set to 0";
    else
    cout<<"Value of b is less then a so it is set to 0";
}
int & min(int &x, int &y)
{
    if (x < y)
    return x;
    else
    return y;
}
```

Output:

Enter Two Numbers: 10 20
Value of a is less then b so it is set to 0

In the above program function min returns the reference of the variable to the calling program. It compares two numbers and returns the reference of the variable whose value is minimum. In the main function the function call min (a, b) receives the reference of the variable whose value is minimum and set its value to 0.

INLINE FUNCTION

An inline function is a function that expands in **one line**. When it is invoked i.e., the compiler replaces the function call with the corresponding functions code when it is called again. It is similar like macros in C. C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

```
inline function header(argument_list)
{
Function body;
}
```

Inline- keyword

Function header- Which includes Name & return type of the function.

Argument_list- Arguments passing values from main () function.

Function body - which includes Local declarations & Statements.

Example:

```
inline double cube (double a)
{
return (a*a*a);
}
```

One of the objectives of the inline function in a program is to save memory space. All inline functions must be defined before they are invoked. Inline functions are much common superior than macros.

Macros are not real functions. The drawbacks of macros are usually error checking does not occur during compilation.

To eliminate the cost of calls to small functions. C++ provides inline functions may some of the situations where inline functions may not work are as follows:

- ✗ The function returning values, if a loop or a switch or goto exists.
- ✗ For functions not returning values, if a return statement exists.
- ✗ If the function contains static variables.
- ✗ Functions are recursive then inline functions cannot work.

Note:1) The inline keyword sends a request not a command. The compiler may ignore its request if a function definition is too long explanation & compiler compiles it normally.

2) Inline function should be used for frequently used small functions.

3) Inline functions make a program to run faster, to make a function call & return is eliminated.

//Program to implement inline functions

```
#include<iostream.h>
#include<conio.h>
inline int add(int x, int y)
{
return x+y;
}
void main()
{
int a,b,c;
clrscr();
cout<<"Enter the value of a and b \n";
cin>>a>>b;
c=add(a,b);
cout<<" Addition of a and b is "<<c;
getch();
}
/* *****OUTPUT*****
Enter the value of a and b
3
2
Addition of a and b is 5
```

ADVANTAGES OF INLINE FUNCTION: -

- 1)It does not require function calling overhead.
- 2)It also save overhead of variables push/pop on the stack, while function calling.
- 3)It also save overhead of return call from a function.
- 4)It increases locality of reference by utilizing instruction cache.
- 5)After in-lining compiler can also apply intraprocedural optimization if specified.

This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

DISADVANTAGES OF INLINE FUNCTION: -

- 1) May increase function size so that it may not fit on the cache, causing lots of cache miss.
- 2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.
- 3) It may cause compilation overhead as if somebody changes code inside inline function than all calling location will also be compiled.
- 4) If used in header file, it will make your header file size large and may also make it unreadable.

- 5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.
- 6) It's not useful for embedded system where large binary size is not preferred at all due to memory size constraints.

DEFAULT ARGUMENTS

C++ allows us to call a function without specifying all of its arguments. In such cases the function assigns a default value to the parameter which does not have matching argument in the function call. A default parameter is a function parameter that has a default value provided to it. Default values are specified when the function is declared (at the time of function prototyping). The compiler looks at the prototype to see how many arguments the function uses & alters the program for possible default values. If the user does not supply a value for this parameter, the default value will be used. If the user does supply a value for the default parameter, the user-supplied value is used.

- ✓ The function can specify as a default argument for one or more parameters using initialization.

Syntax:

```
return_type function_name(type var_name=value, type var_name=value);  
return_type function_name(type var_name, type var_name=value);
```

Example:

```
float rectarea (float l=1.0, float b=2.0); // function prototyping with more default arguments
```

```
float div (float a, float b=2); // function prototyping with one default arguments
```

- ✓ A function providing default argument to a parameter can be called with or without arguments for that parameter.

Example:

```
rectarea (); // function calling without arguments
```

```
a = rectarea (l,b); // function calling with arguments
```

- ✓ Default arguments must be right most arguments in the parameter list of a function.
- ✓ If the default argument values can be constant, global variables or function calls.
- ✓ Default arguments for a parameter can be specified only once in a program.
- ✓ It can also be used with inline function.

Example:

```
inline float amt(float p, float t, float 0.15); //inline function prototyping with default argument
```

A subsequent function call like `float value = amt (5000,7);` //one is missing.

It passes the values 5000 to p & 7 to t & function uses default value of 0.15 to r.
The function call like value = amt (5000, 7, 0.30); // no argument missing
It passes the values 5000 to p & 7 to t & 0.30 to r.

Examples of function declaration (prototyping) with default arguments are:

- 1) int mul (int i, int y=5, int k=10); //legal
- 2) int mul (int i=5, int y); //illegal
- 3) int mul (int i=5, int y, int k=10); //illegal
- 4) int mul (int i=2, int y=5, int k=10); //legal

Advantages of default arguments

- ✓ We can use default arguments to add new parameters to existing function.
- ✓ Default arguments can be used to combine similar function into one.
- ✓ We can use this when we are passing constant values to a parameter.

//Program to illustrate default argument.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    float value;
    float amt(float p,int t,float r=0.15); //function prototyping with one default
    argument
    float prin,time,rate;
    clrscr();
    cout<<"Enter the principle amt & time\n";
    cin>>prin>>time;
    value=amt(prin,time); // function calling
    cout<<"Final value="<<value<<endl;
    getch();
}
float amt(float p, int t, float r)
{
    int year=1;
    float sum=p;
    while(year<=t)
    {
        sum*=(1+r);
        year++;
    }
    return(sum);
}
```

OUTPUT

Enter the principle amt & time 2000 3

Final value=3041.75

FUNCTION OVERLOADING

Overloading refers to the use of something for different purpose.

C++ permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is also known as function polymorphism in OOP's.

We can design family of function with one function name but with different argument list. The function would perform different operation depending on argument list in the function call. The compiler will invoke correct function by checking the number of arguments & its data type but not on the function return type.

A function call first matches the prototype having the same number & the type to the arguments then call the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

- ✓ The compiler first tries to find an exact match in which the type of actual argument are the same & use that function.
- ✓ If an exact match is not found the compiler uses the integral promotion to the actual argument such as
 - 1) Char to int.
 - 2) Float to double. To find a match.
- ✓ When either of them fails the compiler tries to use the built in conversion to the actual arguments & then use the function whose match is unique. If the conversion is possible to have multiple matches then the compiler will generate an error message.

- **For Example:** long square (double n);
double square (double x);

A function calling such as square (10); will cause an error because the argument can be converted to either long or double. There by creating an ambiguity situation for the compiler to which the square function to be used.

- ✓ If all the steps fail then the compiler will try the user defined combination with integral conventions & built in conventions for unique match.

Why to overload a function name or Advantages or Benefits?

1. The main objective behind overloading function is to make programs more readable & understandable if they perform closely related task.

Example: if we want to display an int, float & string then we would like to write their separate functions to indicate same task although with different data types.

Probably these functions may be

Display_int (); // for displaying integers

Display_float (); // for displaying floating numbers.

Display_string (); // for displaying string.

Display_char (); // for displaying character.

2. It is difficult task to remember all function names while implementing them for very large set of data types instead of writing three different function names. We can write two or more functions having same name declared in same scope with different parameters are called overloaded functions.
3. Overloading functions/methods will allow you to have more functions/methods available for any scenario. The first function is more general and is easier to call due to the number of parameters. The second function requires one more extra parameter but allows for more precision in what you want done.
4. Also note that you can easily swap between the overloaded methods being called just by changing the number of parameters. The parameter types still apply.

When not to overload a function name?

There are certain situations where a set of functions will operate on same data types but do not perform the same operations. In such cases functions overloading is not advisable.

Example:

1. Stack:

In implementing stack operator, we may define & declare the overloaded functions stackops () which is intended to perform push & pop operations. Here the stackops () may operate on stack class but does not perform the same operation. So writing push () & pop () exclusively is better.

2. Linked list:

We can insert a set of functions to insert a new node into the linked list.

NODE insert_front (NODE *list, int item);

NODE insert_end (NODE *list, int item);

NODE insert_atpos (NODE *list, int pos, int item);

Although all these functions are intended to perform insertion operation only. It is not good idea to overload them with the name insert (). So writing these functions with different names make the program more readable & understandable.

/*Program to illustrate function overloading to swap two integers, two floating numbers and two characters*/

```
#include<iostream.h>
#include<conio.h>
void swap(int &ix,int &iy);
void swap(float &fx,float &fy);
void swap(char &cx,char &cy);
void main()
{
int ix,iy;
float fx,fy;
char cx,cy;
clrscr();
cout<<"Enter 2 integers:";
cin>>ix>>iy;
cout<<"Enter 2 floating point numbers:";
cin>>fx>>fy;
cout<<"Enter 2 characters:";
cin>>cx>>cy;
cout<<"\n Integers:";
cout<<"\n ix="<<ix<<"\n iy="<<iy;
swap(ix,iy);
cout<<"\n After swapping";
cout<<"\n ix="<<ix<<"\n iy="<<iy;
cout<<"\n Floating point numbers:";
cout<<"\n fx="<<fx<<"\n fy="<<fy;
swap(fx,fy);
cout<<"\n After swapping";
cout<<"\n fx="<<fx<<"\n fy="<<fy;
cout<<"\n Characters";
cout<<"\n cx="<<cx<<"\n cy="<<cy;
swap(cx,cy);
cout<<"\n After swapping";
cout<<"\n cx="<<cx<<"\n cy="<<cy;
getch();
}
void swap(int &a,int &b)
{
int temp;
temp=a;
a=b;
```



```
b=temp;
}
void swap(float &a,float &b)
{
float temp;
temp=a;
a=b;
b=temp;
}
void swap(char &a,char &b)
{
char temp;
temp=a;
a=b;
b=temp;
}
```

/*-----OUTPUT-----*/

Enter 2 integers:20 30

Enter 2 floating point numbers: 3.4 5.6

Enter 2 characters: A B

Integers:

ix=20

iy=30

After swapping

ix=30

iy=20

Floating point numbers:

fx=3.4

fy=5.6

After swapping

fx=5.6

fy=3.4

Characters

cx=A

cy=B

After swapping

cx=B

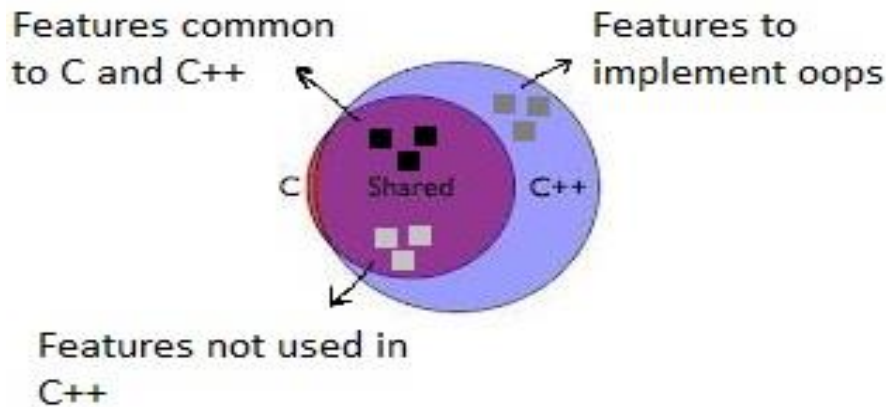
cy=A*/

नहि ज्ञानेन सदृशं

SHREE MEDHA COLLEGE

UNIT-II

CLASSES AND OBJECTS



Basic Introduction:

- C++ is derived from C Language. It is a Superset of C.
- Earlier C++ was known as C with classes.
- In C++, the major change was the addition of classes and a mechanism for inheriting class objects into other classes.
- Most C Programs can be compiled in C++ compiler.
- C++ expressions are the same as C expressions.
- All C operators are valid in C++.

Following are the differences Between C and C++ :

C	C++
1. C is Procedural Language.	1. C++ is non-Procedural i.e Object oriented Language.
2. No virtual Functions are present in C	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	3. The concept of polymorphism is used in C++. Polymorphism is the most Important Feature of OOPS.
4. Operator overloading is not possible in C.	4. Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.

7. Multiple Declaration of global variables are allowed.	7. Multiple Declaration of global variables are not allowed.
8. In C <ul style="list-style-type: none">scanf() Function used for Input.printf() Function used for output.	8. In C++ <ul style="list-style-type: none">Cin>> Function used for Input.Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	9. Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other Functions	10. In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	11. C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
12. No inheritance is possible in C.	12. Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	13. In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	14. It support both built-in and user define data types.
15. In C, Exception Handling is not present.	15. In C++, Exception Handling is done with Try and Catch block.

INTRODUCTION TO CLASSES AND OBJECTS

The classes are the most important feature of C++ that leads to Object Oriented programming. Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.

The variables inside class definition are called as data members and the functions are called member functions. It is also known as **ABSTRACT DATA TYPE**.

For example: Class of birds, all birds can fly and they all have wings and beaks. So here flying is a behavior and wings and beaks are part of their characteristics. And there are many different birds in this class with different names but they all possess this behavior and characteristics.

Similarly, class is just a blue print, which declares and defines characteristics and behavior, namely data members and member functions respectively. And all objects of this class will share these characteristics and behavior.

More about Classes

1. Class name must start with an uppercase letter. If class name is made of more than one word, then first letter of each word must be in uppercase. *Example*,
class Study, class StudyTonight etc
2. Classes contain data members and member functions, and the access of these data members and variable depends on the access specifiers (discussed in next section).
3. Class's member functions can be defined inside the class definition or outside the class definition.
4. Classes in C++ are similar to structures in C, the only difference being, class defaults to private access control, whereas structure defaults to public.
5. All the features of OOPS, revolve around classes in C++. Inheritance, Encapsulation, Abstraction etc.
6. Objects of class holds separate copies of data members. We can create as many objects of a class as we need.
7. Classes do possess more characteristics, like we can create abstract classes, immutable classes, all this we will study later.

OBJECTS

Class is mere a blueprint or a template. No storage is assigned when we define a class. Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.

Each object has different data variables. Objects are initialized using special class functions called **Constructors**. We will study about constructors later. And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object. Destructor performs this task.

```
class Abc
{
    int x;
    void display(){} //empty function
};

void main()
{
    Abc obj; // Object of class Abc created
}
```

Access Control or Specifiers in Classes

Now before studying how to define class and its objects, let's first quickly learn what access specifiers are.

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

These access specifiers are used to set boundaries for availability of members of class be it data members or member functions

Access specifiers in the program, are followed by a colon. You can use either one, two or all 3 specifiers in the same class to set different boundaries for different class members. They change the boundary for all the declarations that follow them.

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class PublicAccess
{
    public: // public access specifier
    int x; // Data Member Declaration
    void display(); // Member Function declaration
}
```

Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default, class variables and member functions are private.

```
class PrivateAccess
{
    private: // private access specifier
    int x; // Data Member Declaration
    void display(); // Member Function declaration
}
```

Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A. We will learn this later.)


```
class ProtectedAccess
{
protected: // protected access specifier
int x;      // Data Member Declaration
void display(); // Member Function declaration
}
```

DEFINING CLASS AND DECLARING OBJECTS

When we define any class, we are not defining any data; we just define a structure or a blueprint, as to what the object of that class type will contain and what operations can be performed on that object.

Below is the syntax of class definition,

```
class ClassName
{
Access specifier:
Data members;
Access specifier:
Member Functions(){}
};
```

Member function: A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Here is an example, we have made a simple class named Student with appropriate members,

```
class Student
{
public:
int rollno;
string name;
};
```

So its clear from the syntax and example, class definition starts with the keyword "class" followed by the class name. Then inside the curly braces comes the class body, that is data members and member functions, whose access is bounded by access specifier. A class definition ends with a semicolon, or with a list of object declarations.

Example:

```
class Student
{
public:
int rollno;
string name;
}A,B;
```

Here A and B are the objects of class Student, declared with the class definition. We can also declare objects separately, like we declare variable of primitive data types. In this case the data type is the class name, and variable is the object.

```
void main()
{
Student A;
Student B;
}
```

Both A and B will have their own copies of data members.

DEFINING MEMBER FUNCTIONS

Member functions can be defined in two locations

1. Inside the class definition.
2. Outside the class definition.

INSIDE THE CLASS DEFINITION:

A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

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

Example:

/*Program to create a class called Item where member functions are defined inside the class. */

```
#include<iostream.h>
#include<conio.h>
class Item
{
int num;
float cost;
public:
```

```
void getdata(int a,float b)
{
    num=a;
    cost=b;
}
void putdata()
{
    cout<<"\nNumber:"<<num;
    cout<<"\nCost : "<<cost;
}
};
void main()
{
    clrscr();
    Item x;
    cout<<"\nObject x\n";
    x.getdata(101,300.99);
    x.putdata();
    Item y;
    cout<<"\n\nObject y\n";
    y.getdata(102,700.50);
    y.putdata();
    getch();
}
```

OUTPUT

Object x
Number:101
Cost :300.98999

Object y
Number:102
Cost :700.5

Member function Outside the class definition: A member function that declare inside the class have to be defined separately outside the class. Class should contain the function header means the function prototype. Function body can be defined outside the class definition using scope resolution operator (::).

Syntax:

<pre>Return_type class_name :: function name(arg_list) { Function body; }</pre>

The member functions incorporated identity label in the header. It tells the compiler that which function belongs to which class in the program. Indirectly private data members can be accessed outside the class using member function of that class.

Example:

/*Program to create a class called employee using array of objects where member functions are defined outside the class.*/

```
#include<iostream.h>
#include<conio.h>
class Emp
{
    char name[20];
    float age;
    public:
    void getdata();
    void putdata();
};
void Emp::getdata()
{
    cout<<"\nEnter name:";
    cin>>name;
    cout<<"\nEnter age:";
    cin>>age;
}
void Emp::putdata()
{
    cout<<"Name:"<<name<<"\n";
    cout<<"Age : "<<age<<"\n";
}
const size=3;
void main()
{
    clrscr();
    Emp manager[size];
    for(int i=0;i<size;i++)
    {
        cout<<"\nDetails of manager" <<i+1<<"\n";
        manager[i].getdata();
    }
    cout<<"\n";
    for(i=0;i<size;i++)
    {
        cout<<"\nManager"<<i+1<<"\n";
        manager[i].putdata();
    }
    getch();
}
```

-----OUTPUT-----

Details of manager1

Enter name:Varun

Enter age:28

Details of manager2

Enter name:Vikas

Enter age:32

Details of manager3

Enter name:Vani

Enter age:29

Manager1

Name:Varun

Age :28

Manager2

Name:Vikas

Age :32

Manager3

Name:Vani

Age :29

MAKING AN OUTSIDE MEMBER FUNCTION INLINE:

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of the function definition.

Example:

```
class item
{
int a, b;
public:
inline void getdata(int a,float b);
};
```



```
inline void item :: getdata(int a,float b) // outside member function inline
{
number=a;
cost=b;
}
```

Nesting of member functions: Member functions can be called using its name inside another member function of same class. This known as ***nesting of member function***. A user defined function can be called another user defined function; the control will transfer internally from one function to another function.

Example:

// Program to illustrate nesting of member functions //

```
#include <iostream.h>
#include <conio.h>
class nest_func
{
int m,n;
public:
void input();
void output();
int largest();
};
void nest_func :: input()
{
cout<<"Enter two numbers to find largest"<<endl;
cin<<m<<n;
}
int nest_func :: largest()
{
if ( m > n )
return m;
else
return n;
}
void nest_func :: output()
{
cout<<"Largest number="<<largest(); // calling another member function
}
void main()
{
nest_func obj;
obj.input();
obj.output();
getch();}
```

Private member function: A private member function can only be called by another function that is a member of its class. Even an object can invoke a private member function using dot operator. Consider a class as defined below

```
class sample
{
    int m;
    void read(void);           // private member function
public:
    void update(void);
    void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();           // won't work; objects cannot access
                    // private members
```

is illegal. However, the **function read()** can be called by the **function update()** to update the value of **m**.

```
void sample :: update(void)
{
    read();           // simple call; no object used
}
```

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;           // provides value for array size

class array
{
    int a[size];             // 'a' is int type array
public:
    void setval(void);
    void display(void);
};
```

The array variable **a[]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Program 5.3 shows how these operations are implemented using a class with arrays as data members.

/* Program to create a class called employee using array of objects where member functions are defined outside the class. */

```
#include<iostream.h>
#include<conio.h>
class Emp
{
    char name[20];
    float age;
public:
    void getdata();
    void putdata();
};
void Emp::getdata()
{
    cout<<"\nEnter name:";
    cin>>name;
    cout<<"\nEnter age:";
    cin>>age;
}
void Emp::putdata()
{
    cout<<"Name:"<<name<<"\n";
    cout<<"Age :"<<age<<"\n";}
const size=3;
void main()
{
    clrscr();
    Emp manager[size];
    for(int i=0;i<size;i++)
    {
        cout<<"\nDetails of manager" <<i+1<<"\n";
        manager[i].getdata();
    }
    cout<<"\n";
    for(i=0;i<size;i++)
    {
        cout<<"\nManager"<<i+1<<"\n";
        manager[i].putdata();
    }
    getch();
}
```

OUTPUT:

Details of manager1

Enter name:Varun

Enter age:28

Details of manager2

Enter name:Vikas

Enter age:32

Details of manager3

Enter name:Vani

Enter age:29

Manager1

Name:Varun

Age :28

Manager2

Name:Vikas

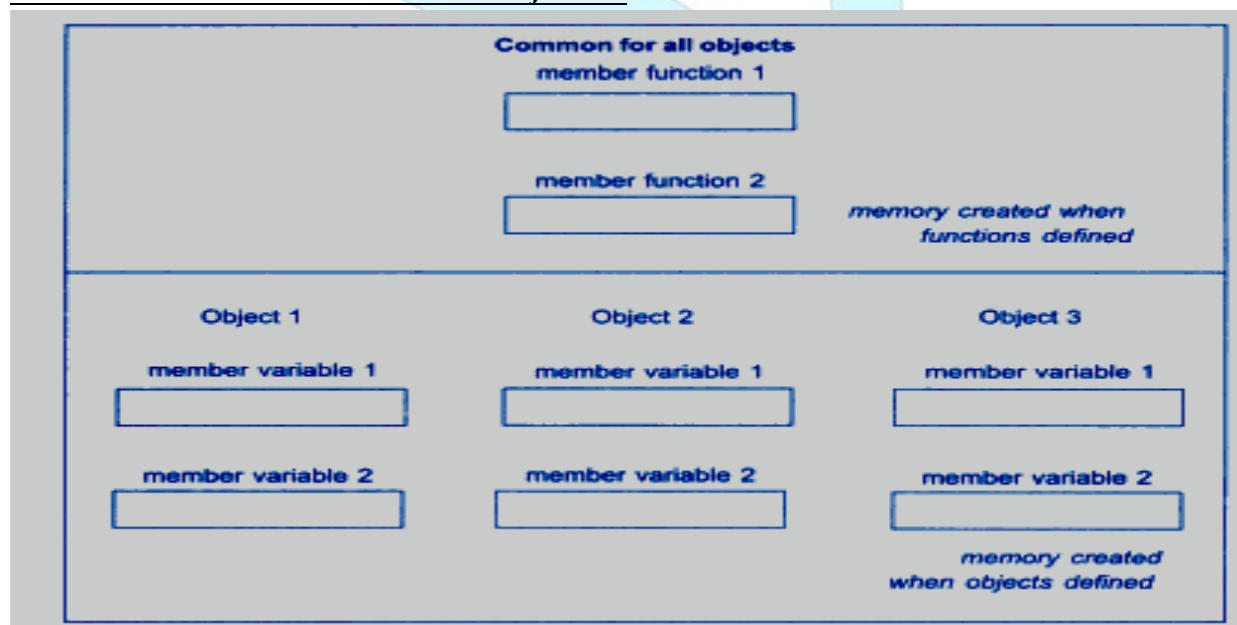
Age :32

Manager3

Name:Vani

Age :29

MEMORY ALLOCATION FOR OBJECTS

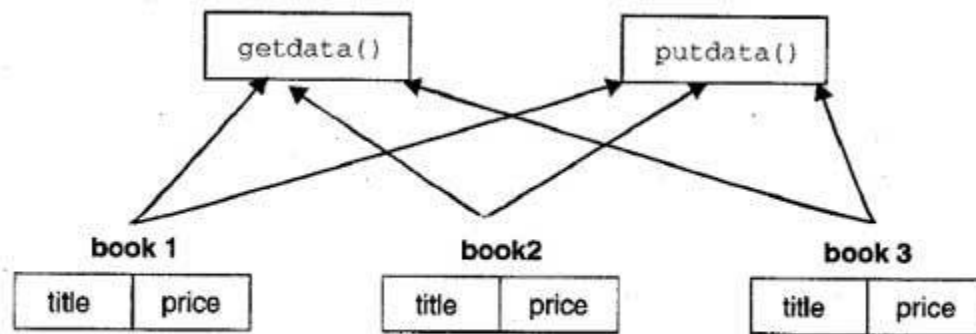


Before using a member of a class, it is necessary to allocate the required memory space to that member. The way the memory space for data members and member functions is allocated is different regardless of the fact that both data members and member functions belong to the same class.

The memory space is allocated to the data members of a class only when an object of the class is declared, and not when the data members are declared inside the class. Since a single data member can have different values for different objects at the same time, every object declared for the class has an individual copy of all the data members.

On the other hand, the memory space for the member functions is allocated only once when the class is defined. In other words, there is only a single copy of

each member function, which is shared among all the objects. For instance, the three objects, namely, book1, book2 and book3 of the class book have individual copies of the data members title and price. However, there is only one copy of the member functions getdata () and putdata () that is shared by all the three objects



Memory Allocation for the Objects of the Class book

STATIC DATA MEMBER OF CLASS

In C++ memory is allocated separately for each data member for different objects.

So if you change the value of data member of class using one object will not affect the value of the same data member for other object of the same class. However sometimes it is required that all the objects of the same class **share some common variables**. This can be accomplished using the concept of **static data member**.

We can declare static data members using **static keyword**.

Static data member having several characteristics which are given below:

- (1) Hence all the object of the same class share static data member memory is allocated only once to the static data member. It remains common for all the objects of the same class.
- (2) Static data member is initialized to 0 when first object of the class is created.

Static data member is declared in the class but it must be defined outside the class using class name and scope resolution operator (::) because memory allocation for static data member of the class is performed different then normal data member of the class and it is not the part of class object.

Example:

```
#include<iostream.h>
#include<conio.h>
class item
{
static int count;
public:
void DisplayCounter()
{
```



```
count++;  
cout<<"count:"<<count<<endl;  
}  
};  
int item::count;  
int main()  
{  
item a,b,c;  
a.DisplayCounter();  
b.DisplayCounter();  
c.DisplayCounter();  
return 0;  
}
```

Output:

Count: 1
Count: 2
Count: 3

STATIC MEMBER FUNCTION OF CLASS

Like data member of the class, member function of the class can also be declared as a static. Static member functions are designed to work with static data members. In order to make a member function as static we need to precede the function declaration with **static** keyword.

Static member function having several characteristics which are given below:

- (1) Static member function can access only static member of the class in which it is declared.
- (2) Static member function is not a part of class object so it can not be called using object of the class.

Static member function can be called using class name and scope resolution operator as shown below:

Class_Name :: Function_Name ();
--

Example:

```
#include <iostream.h>  
#include <conio.h>  
class test  
{  
private:  
static int count;  
public:  
void setCount(void);
```

```
static void DisplayCounter(void);
};
void test :: setCount(void)
{
    count++;
}
void test :: DisplayCounter(void)
{
    cout<<"Count:"<< count << endl;
}
int test::count;
void main( )
{
    test t1, t2;
    clrscr();
    t1.setCount();
    test :: DisplayCounter();
    t2.setCount();
    test :: DisplayCounter();
    getch();
}
```

Output:

Count: 1
Count: 2

PASS OBJECT AS FUNCTION ARGUMENT AND RETURN OBJECT

In C++ we can pass object as a function argument and also return an object from the function.

An object can be passed to the function using two methods:

- (1)**Call By value:** In which copy of the object is passed to the function.
- (2)**Call By reference:** In which an address of the object is passed to the function.

Example:

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

```
class Demo
```

```
{
    int a,b;
```

```
public:
```

```
Demo()
```

```
{
    a=0;
    b=0;
}
```

```
Demo(int x, int y)
```

```
{
    a=x;
```

```
b=y;
}
void display()
{
cout<<"A="<<a<<endl<<"B="<<b<<endl;
}
Demo sum(Demo D2);
};
Demo Demo::sum(Demo D2)
{
Demo D3;
D3.a = a + D2.a;
D3.b = b + D2.b;
return D3;
}
void main()
{
Demo B1(1,2);
Demo B2(3,4);
Demo B3;
B3 =B1.sum(B2);
B3.display();
}
```

Output:

A=4
B=6

Above program performs the addition of data member of two objects and store it in the third object.

Consider the function defined inside class:

Demo sum(Demo D2);

It accepts one object of the class Demo as an argument and returns an object to the calling function.

When we call the function using following statement:

B3 =B1.sum(B2);

Data member of object B1 is passed to the function implicitly while data member of object B2 is passed to the function explicitly. After performing addition of data member of two objects B1 and B2 it will store it in the data member of third object and return that object.

CONSTRUCTORS

Constructors are *special class functions which performs initialization of every object*. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object. Classes can have complicated internal structures, so object initialization and clean-up of a class is much more complicated than for any other data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. The construction can be for example: initialization for objects or memory allocation. The destruction may involve deallocation of memory or other clean-up for objects.

Constructors and destructors are declared within a class declaration (as like any other member function). A constructor or a destructor can be defined in-line or external to the class declaration. We may declare some default arguments when we make a constructor. There are some restrictions that apply to constructors and destructors:

Characteristics or Features of Constructor & Destructor

- A constructor is a method that has the same name as its class.
- A destructor is a method that has as its name the class name prefixed by a tilde, ~.
- Neither constructors nor destructors return values. They have no return type specified.
- Constructors can have arguments.
- Constructors can be overloaded.
- If any constructor is written for the class, the compiler will not generate a default constructor.
- The default constructor is a constructor with no arguments, or a constructor that provides defaults for all arguments.
- The container classes such as vector require default constructors to be available for the classes they hold. Dynamically allocated class arrays also require a default constructor. If any constructors are defined, you should always define a default constructor as well.
- Destructors have no arguments and thus cannot be overloaded.
- Pointers and references cannot be used on constructors and destructor's (It is not possible to get their address).
- Constructors and destructors cannot be declared static, const or volatile.
- Constructors cannot be declared with the keyword virtual.

Note: the same *access rules* apply to constructors and destructors as with any other member function.

Constructors are called automatically by the compiler when defining class objects. The destructors are called when a class object goes out of scope.

Syntax:

```

class classname
{
    ..... ;
    public :
        classname( );
};
classname :: classname( )
{
    //Body of constructor
}

```

Diagram labels with arrows pointing to the code:

- Private members (points to the first line of the class body)
- Constructor declaration (points to the public constructor declaration inside the class)
- Constructor definition (points to the constructor definition outside the class)
- //Body of constructor (points to the body of the constructor definition)

```

class A
{
    int x;
    public:
    A(); //Constructor
};

```

While defining a constructor you must remember that the name of constructor will be same as the name of the class, and constructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution: `::` operator.

```

class A
{
    int i;
    public:
    A(); //Constructor declared
};

```

```

A::A() // Constructor definition
{
    i=1;
}

```

The various types of Constructor are as follows:-

1. Default Constructor
2. Parameterized Constructor.
3. Copy Constructor.

1. DEFAULT CONSTRUCTOR: - Default constructor is the constructor which doesn't take any argument. It has no parameter. Default Constructor is also called as Empty Constructor which has no arguments and It is Automatically called when we create the object of class but Remember name of Constructor is same as name of class

and Constructor never declared with the help of Return Type. Means we can't declare a Constructor with the help of void Return Type., if we never Pass or declare any Arguments then this called as the Default Constructors.

Syntax :

<pre>class_name () { Constructor Definition }</pre>

//Program to implement to Default constructor

```
#include<iostream.h>  
#include<conio.h>  
class Point  
{  
    int x,y;  
public:  
    Point()    //Default Constructor  
    {  
        x=y=0;  
    }  
    Point (int xp, int yp)    //Parameterized Constructor  
    {  
        x=xp;  
        y=yp;  
    }  
    Point (Point &p)    //Copy Constructor  
    {  
        x=p.x;  
        y=p.y;  
    }  
    ~Point()    //Destructor  
    {  
        cout<<"\nDestructor called automatically for all the created objects";  
        getch();  
        void display()  
        {  
            cout<<"\n"<<"X="<<x<<"\n"<<"Y="<<y<<endl;  
        }  
    };  
    void main()  
    {  
        clrscr();  
        Point p1;  
        Point p2(20,30);
```

```
Point p3(p2);
cout<<"\nDefault constructor called \n";
p1.display();
cout<<"\nParameterized constructor called \n";
p2.display();
cout<<"\nCopy constructor copying parameterized values \n";
p3.display();
getch();
}
/*-----OUTPUT-----
Default constructor called
X=0
Y=0

Parameterized constructor called
X=20
Y=30

Copy constructor copying parameterized values
X=20
Y=30

Destructor called automatically for all the created objects
Destructor called automatically for all the created objects
Destructor called automatically for all the created objects*/
```

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
    int side;
};

void main()
{
    Cube c;
    cout << c.side;
}
```

Output : 0

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

2. PARAMETERIZED CONSTRUCTOR: - This is another type Constructor which has some Arguments and same name as class name but it uses some Arguments So For this We have to create object of Class by passing some Arguments at the time of creating object with the name of class. When we pass some Arguments to the Constructor then this will automatically pass the Arguments to the Constructor and the values will retrieve by the Respective Data Members of the Class.

Syntax:

```
class_name (argument_list)
{
    Constructor Definition
}
```

//Program to implement to Parameterised constructor

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

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

```
class Point
```

```
{
```

```
int x,y;
```

```
public:
```

```
Point()    //Default Constructor
```

```
{
```

```
x=y=0;
```

```
}
```

```
Point (int xp, int yp)    //Parameterized Constructor
```

```
{
```

```
x=xp;
```

```
y=yp;
```

```
}
```

```
Point (Point &p)    //Copy Constructor
```

```
{
```

```
x=p.x;
```

```
y=p.y;
```

```
}
```

```
~Point()    //Destructor
```

```
{
```

```
cout<<"\nDestructor called automatically for all the created objects";
```

```
getch();}
```

```
void display()
```

```
{
```

```
cout<<"\n"<<"X="<<x<<"\n"<<"Y="<<y<<endl;
```

```
}
```

```
};
```

```
void main()
{
clrscr();
Point p1;
Point p2(20,30);
Point p3(p2);
cout<<"\nDefault constructor called \n";
p1.display();
cout<<"\nParameterized constructor called \n";
p2.display();
cout<<"\nCopy constructor copying parameterized values \n";
p3.display();
getch();
}
/*-----OUTPUT-----*/
```

Default constructor called

X=0

Y=0

Parameterized constructor called

X=20

Y=30

Copy constructor copying parameterized values

X=20

Y=30

Destructor called automatically for all the created objects

Destructor called automatically for all the created objects

Destructor called automatically for all the created objects*/

By using parameterized constructor in above case, we have initialized 1 objects with user defined values. We can have any number of parameters in a constructor.

3. COPY CONSTRUCTOR: - This is also Another type of Constructor. In this Constructor we pass the object of class into the Another Object of Same Class. As name Suggests You Copy, means Copy the values of one Object into the another Object of Class. This is used for Copying the values of class object into an another object of class So we call them as Copy Constructor and for Copying the values We have to pass the name of object whose values we want to Copying and When we are using or passing an Object to a Constructor then we must have to use the & Ampersand or Address Operator. *Example Program same as parameterised constructor*

CONSTRUCTOR OVERLOADING

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

```
class Student
{
    int rollno;
    string name;
public:
    Student(int x)
    {
        rollno=x;
        name="None";
    }
    Student(int x, string str)
    {
        rollno=x ;
        name=str ;
    }
};
```

```
int main()
{
    Student A(10);
    Student B(11,"Ram");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

DESTRUCTOR: As we know that Constructor is that which is used for Assigning Some Values to data Members and for Assigning Some Values This May also used Some Memory so that to free up the Memory which is Allocated by Constructor, Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope. Destructor is used which gets Automatically Called at the End of Program and we don't have to Explicitly Call a Destructor and Destructor Can't be

Parameterized or a Copy This can be only one Means Default Destructor which Have no Arguments. For Declaring a Destructor, we have to use ~tiled Symbol in front of Destructor.

Syntax

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor; with a **tilde** ~ sign as prefix to it.

```
class A
{
public:
~A(); // Destructor
};
```

Destructors will never have any arguments.

Example program same as parameterised constructor.

SINGLE DEFINITION FOR BOTH DEFAULT AND PARAMETERIZED CONSTRUCTOR

In this example we will use **default argument** to have a single definition for both default and parameterized constructor.

```
class Dual
{
int a;
public:

Dual(int x=0)
{
a=x;
};

void main()
{
Dual obj1;
Dual obj2(10);
}
```

Here, in this program, a single Constructor definition will take care for both these object initializations. We don't need separate default and parameterized constructors.

Another way of Member initialization in constructors

The constructor for this class could be defined, as usual, as:

```
Circle :: Circle(double r)
{
radius = r;
}
```

It could also be defined using member initialization as:

```
Circle :: Circle(double r) : radius(r)
{ }
```

DYNAMIC CONSTRUCTOR

This constructor is used to allocate the memory to the objects at the run time. The memory allocation to objects is allocated with the help of 'new' operator. **By using this constructor, we can dynamically initialize the objects.**

Example :

```
#include <iostream.h>
#include <conio.h>
class Account
{
private:
int account_no;
int balance;
public :
Account(int a,int b)
{
account_no=a;
balance=b;
}
void display()
{
cout<< "\nAccount number is : "<< account_no;
cout<< "\nBalance is : " << balance;
}
};
void main()
{clrscr();
int an,bal;
cout<< "Enter account no : ";
cin >> an;
cout<< "\nEnter balance : ";
cin >> bal;
Account *acc=new Account(an,bal); //dynamic constructor
acc->display(); //'->' operator is used to access the method
getch();}
```

UNIT - III

FRIEND FUNCTIONS

One of the important concepts of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data. But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a friend function or/and a friend class.

FRIEND FUNCTION IN C++

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

The compiler knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Declaration of friend function in C++

```
class class_name
{
    ... ..
    friend return_type function_name(argument/s);
    ... ..
}
```

Now, you can define the friend function as a normal function to access the data of the class. No friend keyword is used in the definition.

```
class className
{
    ... ..
    friend return_type functionName(argument/s);
    ... ..
}

return_type functionName(argument/s)
{
    ... ..
    // Private and protected data of className can be accessed from
    // this function because it is a friend function of className.
    ... ..
}
```

/*program to create a class complex to hold a complex number. Write friend functions add 2 complex numbers. */

```
#include<iostream.h>
#include<conio.h>
class complex
{
float real,img;
public:
void get_complex();
void show_complex();
friend complex add_complex(complex c1,complex c2);
void complex::get_complex()
{
cout<<"Enter real and imaginary number for complex number \n";
cin>>real>>img;
}
void complex::show_complex()
{
cout<<real<<"+"i"<<img<<endl;
}
complex add_complex(complex c1,complex c2)
{
complex c;
c.real=c1.real+c2.real;
c.img=c1.img+c2.img;
return(c);
}
void main()
{
clrscr();
complex c1,c2,c3;
c1.get_complex();
c2.get_complex();
c3=add_complex(c1,c2);
cout<<"\n Complex number 1: ";
c1.show_complex();
cout<<"\n Complex number 2: ";
c2.show_complex();
cout<<"-----\n";
cout<<"Sum of two complex numbers =";
c3.show_complex();
getch();}
/*-----OUTPUT-----
Enter real and imaginary number for complex number 3 4.5
Enter real and imaginary number for complex number 2 3.5
Complex number 1: 3+i4.5
Complex number 2: 2+i3.5
Sum of two complex numbers =5+i8*/
```

OPERATOR OVERLOADING

An operator is a symbol that tells the compiler to perform specific task. Every operator have their own functionality to work with built-in data types. Class is user-defined data type and compiler doesn't understand, how to use operators with user-defined data types. To use operators with user-defined data types, they need to be overload according to a programmer's requirement.

Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.

An operator can be overloaded by defining a function to it. The function for operator is declared by using the operator keyword followed by the operator.

There are two types of operator overloading in C++

- Binary Operator Overloading
- Unary Operator Overloading

Overloading Binary Operator

Binary operator is an operator that takes two operand(variable). Binary operator overloading is similar to unary operator overloading except that a binary operator overloading requires an additional parameter.

Binary Operators

- Arithmetic operators (+, -, *, /, %)
- Arithmetic assignment operators (+=, -=, *=, /=, %=)
- Relational operators (>, <, >=, <=, !=, ==)

Overloading Unary Operator

Unary operator is an operator that takes single operand(variable). Both increment (++) and decrement (--) operators are unary operators. In C++, it's possible to change the way operator works (for user-defined types). In this article, you will learn to implement operator overloading feature.

Why is operator overloading used?

You can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive. For example,

You can replace the code like:

calculation = add(multiply(a, b), divide(a, b));

to

calculation = (a*b) + (a/b);

How to overload operators in C++ programming?

To overload an operator, a special operator function is defined inside the class as:

```
class className
{
    ... ..
    public
```



```
returnType operator symbol (arguments)
```

```
{
    ... ..
}
... ..};
```

- Here, returnType is the return type of the function.
- The returnType of the function is followed by operator keyword.
- Symbol is the operator symbol you want to overload. Like: +, <, -, ++
- You can pass arguments to the operator function in similar way as functions.

// Program to illustrate operator overloading.

```
#include<iostream.h>
#include<conio.h>
class operator_overload
{
int x,y,z;
public:
void getdata(int a, int b);
void display();
void operator -(); //Overloading Unary - operator
void operator +(); //Overloading Binary + operator
operator_overload()
{
x=y=z=0;
}
};
void operator_overload::getdata(int a, int b)
{
x=a;
y=b;
}
void operator_overload::display()
{
cout<<"\n"<<"X="<<x<<"\n"<<"Y="<<y<<"\n"<<"Z="<<z<<endl;
}
void operator_overload::operator -()
{
x= -x;
y= -y;
}
void operator_overload::operator +()
{
z=x+y;
}
```

```
void main()
{
clrscr();
operator_overload obj;
int x,y;
cout<<"Enter x and y values \n";
cin>>x>>y;
obj.getdata(x,y);
obj.display();
cout<<"Calling operator -() and +()"<<endl;
-obj;
+obj;
obj.display();
getch();
}
/*-----OUTPUT-----
Enter x,y values
3
2
X=3
Y=2
Z=0
Calling operator -() and +()
X=-3
Y=-2
Z=-5 */
```

Operator Overloading Rules

There are some rules or restrictions for overloading operators. These rules are as given below:

1. You can overload only existing operator. New operator cannot be created.
2. you cannot overload following operators:

Sizeof operator

Conditional operator (? :)

Scope resolution operator (::)

Class member access operator (., *)

3. By overloading operator you cannot change the syntax rules of operator.

4. You cannot use friend function to overload following operators:

= Assignment operator.

() Function call operator.

{ } Subscripting operator.

-> Class member access operator.

5. When you overload unary operator using member function it will take no explicit argument. But if you overload unary operator using friend function then it will take one explicit argument.
6. When you overload binary operator using member function it will take one explicit argument. But if you overload binary operator using friend function then it will take two explicit arguments.
7. Overloaded operators cannot have default arguments.

OPERATING OVERLOADING USING FRIEND FUNCTIONS

A C++ friend functions are special functions which can access the private members of a class. They are considered to be a loophole in the Object Oriented Programming concepts, but logical use of them can make them useful in certain cases. For instance: when it is not possible to implement some function, without making private members accessible in them. This situation arises mostly in case of operator overloading.

/*Program to create a class complex to hold a complex number. Write friend functions add 2 complex numbers. */

```
#include<iostream.h>
#include<conio.h>
class complex
{
float real,img;
public:
void get_complex();
void show_complex();
friend complex add_complex(complex c1,complex c2);
};
void complex::get_complex()
{
cout<<"Enter real and imaginary number for complex number \n";
cin>>real>>img;
}
void complex::show_complex()
{
cout<<real<<"i"<<img<<endl;
}
complex add_complex(complex c1,complex c2)
{
complex c;
c.real=c1.real+c2.real;
c.img=c1.img+c2.img;
return(c);
}
```

```
void main()
{
clrscr();
complex c1,c2,c3;
c1.get_complex();
c2.get_complex();
c3=add_complex(c1,c2);
cout<<"\n Complex number 1: ";
c1.show_complex();
cout<<"\n Complex number 2: ";
c2.show_complex();
cout<<"-----\n";
cout<<"Sum of two complex numbers =";
c3.show_complex();
getch();
}
/*-----OUTPUT-----
Enter real and imaginary number for complex number
3 4.5
Enter real and imaginary number for complex number
2 3.5

Complex number 1: 3+i4.5

Complex number 2: 2+i3.5
-----
Sum of two complex numbers =5+i8 */
```

Friend functions have the following properties:

- 1) Friend of the class can be member of some other class.
- 2) Friend of one class can be friend of another class or all the classes in one program, such a friend is known as GLOBAL FRIEND.
- 3) Friend can access the private or protected members of the class in which they are declared to be friend, but they can use the members for a specific object.
- 4) Friends are non-members hence do not get "this" pointer.
- 5) Friends, can be friend of more than one class, hence they can be used for message passing between the classes.
- 6) Friend can be declared anywhere (in public, protected or private section) in the class.

STRING MANIPULATION USING OPERATOR OVERLOADING

C++ allows us the facility of manipulate strings using the concept of operator overloading.

For example we can overload + operator to **concat** two strings. We can overload == operator to **compare** two strings. Consider Following Example in which we overload + operator to concat two strings.

INHERITANCE

This mechanism of deriving a new class from existing/old class is called "inheritance". „ The old class is known as "base" class, "super" class or "parent" class"; and the new class is known as "sub" class, "derived" class, or "child" class.

or

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Inheritance is very similar to a parent-child relationship. When a class is inherited all the functions and data member are inherited, although not all of them will be accessible by the member functions of the derived class. But there are some exceptions to it too.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

NOTE: All members of a class except Private, are inherited

PURPOSE OR ADVANTAGES: -

One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units. Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type of a method is superclass

Reusability -- Inheritance allows to add more features to an existing class without modifying it , and in this way provides reusability of attributes defined resulting into faster development time, easier maintenance and easier control

Extensibility -- extending the base class logic as per business logic of the derived class

Data hiding -- base class can decide to keep some data private so that it cannot be altered by the derived class

Overriding--With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

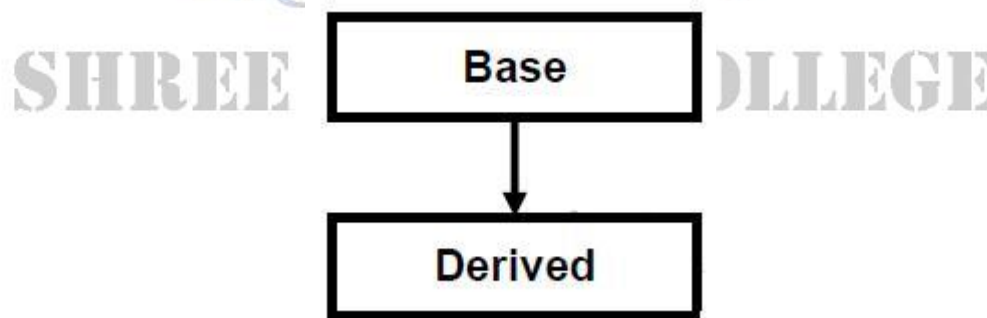
DISADVANTAGES: -

1. One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes
2. Main disadvantage of using inheritance is that the two classes (base and inherited class) get tightly coupled. This means one cannot be used independent of each other.
3. Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)
4. If a method is deleted in the "super class" or aggregate, then we will have to re-factor in case of using that method. Here things can get a bit complicated in case of inheritance because our programs will still compile, but the methods of the subclass will no longer be overriding superclass methods. These methods will become independent methods in their own right.

TYPES OF INHERITANCE

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance.

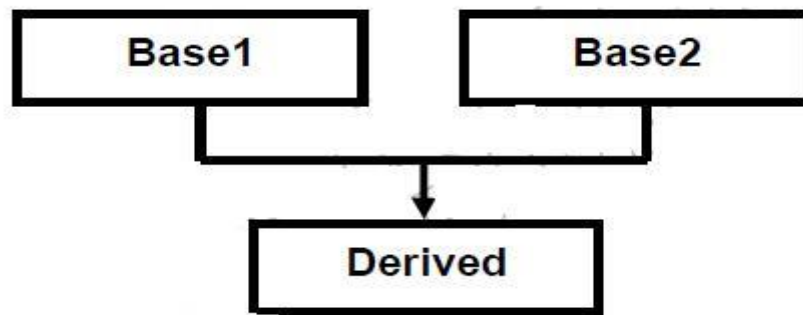
1. Single Inheritance: When a single derived class is created from a single base class then the inheritance is called as single inheritance.



Single Inheritance Example Program

2. Multiple Inheritance : When a derived class is created from more than one base class then that inheritance is called as multiple inheritance. But

multiple inheritance is not supported by .net using classes and can be done using interfaces.

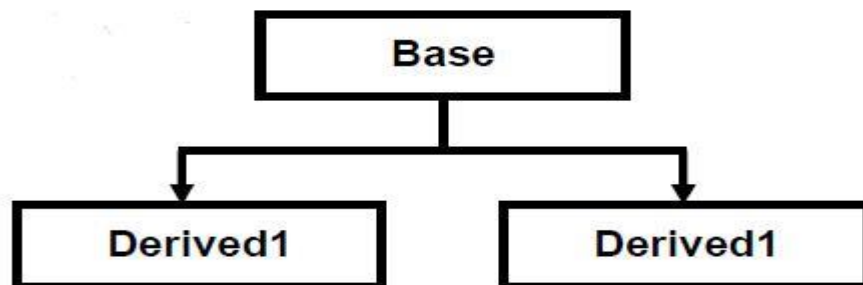


Multiple Inheritance Example

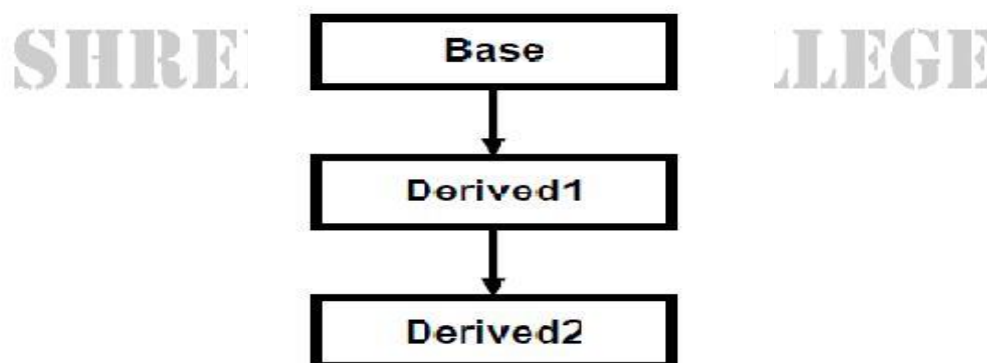
Handling the complexity that causes due to multiple inheritance is very complex. Hence it was not supported in dotnet with class and it can be done with interfaces.

In Inheritance Upper Class whose code we are actually inheriting is known as the Base or Super Class and Class which uses the Code are known as Derived or Sub Class.

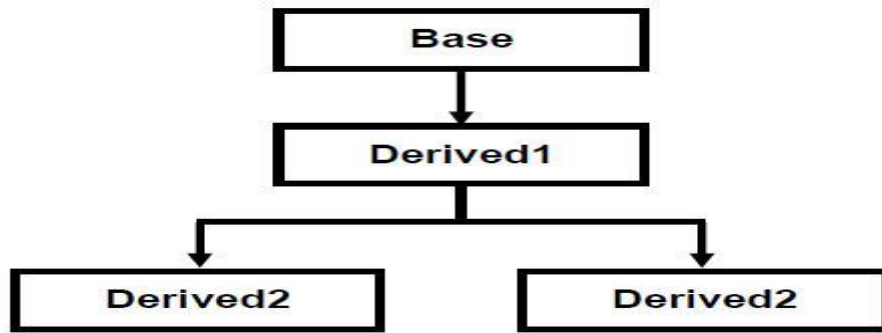
3. Hierarchical Inheritance : When more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



4. Multi-Level Inheritance: When a derived class is created from another derived class, then that inheritance is called as multi-level inheritance.



5. Hybrid(Virtual) Inheritance: Any combination of single, hierarchical and multi-level inheritances is called as hybrid inheritance.



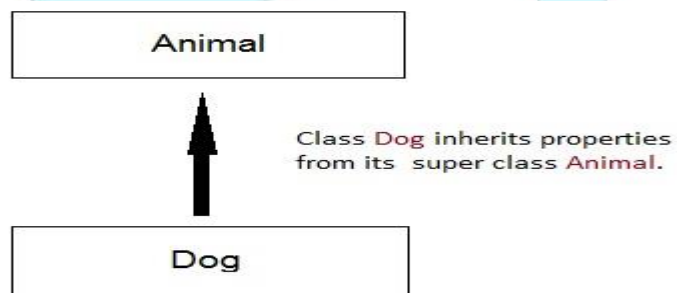
Basic Syntax of Inheritance

```
class Subclass_name : access_mode Superclass_name
```

While defining a subclass like this, the super class must be already defined or atleast declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, private or protected.

Example of Inheritance



```

class Animal
{
public:
int legs = 4;
};
class Dog : public Animal
{
public:
int tail = 1;
};
  
```

```

void main()
{
    Dog d;
    cout << d.legs;
    cout << d.tail;
}
  
```

Output : 4 1

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : **public** Superclass

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // By default its private inheritance

3) Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : **protected** Superclass

Base class	Derived Class		
	Public Mode	Private Mode	Protected Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Table showing all the Visibility Modes

Significance of visibility or access specifiers modes

Public Access mode

- Derived class can access the public and protected members of the base class, but not the private members of the base class.
- The public members of the base class become public members of the derived class and protected members of the base class become protected members of the base class.
- In public derivation access specifiers for inherited members in the derived class.
- Derive class publicly when the situation wants the derived class to have all the attributes of the base class, plus some extra features.

Protected Access Mode

- Derived class can access the public and protected members of the base class protected, but not the private members of the base class.
- The public and protected members of the base class become protected members of the derived class.
- Derive protected when the features are required to be hidden from the outside world and at the same time required to be inheritable.

Private Access Mode

- Derived class can access the public and protected members of the base class privately, but not the private members of the base class.
- The public and protected members of the base class become private members of the derived class.
- Derive class privately when the derived class requires to use some attributes of the base class and these inherited features cannot be inherited further.

Inheritance and the Base class

- Members intended to be inherited and the same time intended to be available to every function, even to non-members, should be declared as public members in the base class.
- Members intended to be Inherited but at the same time hidden from outside world, should be declared under protected section.
- Members which are not to be inherited further should be declared under private section.
- Public members of base class can be accessed from its own class, derived class and from objects outside class.
- Protected members of base class can be accessed from its own class, derived class but not from objects outside class.
- Private members of base class are only accessible within its class only.

SINGLE INHERITANCE

Inheritance is the process of inheriting properties of objects of one class by objects of another class. The class which inherits the properties of another class is called Derived or Child or Sub class and the class whose properties are inherited is called Base or Parent or Super class. *When a single class is derived from a single parent class, it is called **Single inheritance**. It is the simplest of all inheritance.*

For example,

- Animal is derived from living things
- Car is derived from vehicle
- Typist is derived from staff

Syntax of Single Inheritance

```
class base_classname
{
    properties;
    methods;
};

class derived_classname : visibility_mode base_classname
{
    properties;
    methods;
};
```

//Program to illustrate single inheritance

```
#include<iostream.h>
#include<conio.h>
class Base
{
protected:
int x;
public:
void accept()
{
cout<<"Enter the value of x"<<endl;
cin>>x;
}
};

class Derived:public Base
{
int y;
public:
void getdata()
{
cout<<"Enter the value of y\n";
cin>>y;
}
void display()
{
cout<<"x="<<x<<"\n"<<"y="<<y<<endl;
}
};
```

```
void main()
{
    Derived d;
    clrscr();
    d.accept();
    d.getdata();
    d.display();
    getch();
}
/*-----OUTPUT-----
Enter the value of x
88
Enter the value of y
44
x=88
y=44 */
```

MULTIPLE INHERITANCE

Inheritance is the process of inheriting properties of objects of one class by objects of another class. The class which inherits the properties of another class is called Derived or Child or Sub class and the class whose properties are inherited is called Base or Parent or Super class. *When a class is derived from two or more base classes, such inheritance is called **Multiple Inheritance**. It allows us to combine the features of several existing classes into a single class.*

For example,

- Petrol is derived from both liquid and fuel.
- A child has character of both his/her father and mother, etc

Syntax of Multiple Inheritance

```
class base_class1
{
    properties;
    methods;
};
class base_class2
{
    properties;
    methods;
};
... ..
... ..
class base_classN
{
    properties;
    methods;
};
```

```
class derived_classname : visibility_mode base_class1, visibility_mode
base_class2,... ,visibility_mode base_classN
{
    properties;
    methods;
};
```

//Program to illustrate multiple inheritance.

```
#include<iostream.h>
#include<conio.h>
class Area
{
public:
float area_calc(float l,float b)
{
return l*b;
}
};
class Perimeter
{
public:
float peri_calc(float l,float b)
{
return 2*(l+b);
}
};
class Rectangle:private Area,private Perimeter
{
private:
float length,breadth;
public:
Rectangle():length(0.0),breadth(0.0){}
void get_data()
{
cout<<"Enter length:";
cin>>length;
cout<<"Enter breadth:";
cin>>breadth;
}

float area_calc()
{
return Area::area_calc(length,breadth);
}
float peri_calc()
{
return Perimeter::peri_calc(length,breadth);
}
};
```

```
void main()
{
    Rectangle r;
    clrscr();
    r.get_data();
    cout<<"Area="<<r.area_calc();
    cout<<"\nPerimeter="<<r.peri_calc();
    getch();
}
/*-----OUTPUT-----
Enter length:
5
Enter breadth:9
Area=45
Perimeter=28 */
```

AMBIGUITY IN MULTIPLE INHERITANCE

In multiple inheritance, a single class is derived from two or more parent classes. So, there may be a possibility that two or more parents have same named member function. If the object of child class needs to access one of the same named member function, then it results in ambiguity. The compiler is confused as method of which class to call on executing the call statement.

For example,

```
#include <iostream.h>
#include <conio.h>
class A
{
    public:
    void display()
    {
        cout <<"This is method of A";
    }
};
```

```
class B
{
    public:
    void display()
    {
        cout <<"This is method of B";
    }
};
```

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

```
void main()
{
    C sample;
    sample.display(); /*causes ambiguity*/
    getch();
}
```

AMBIGUITY RESOLUTION OF MULTIPLE INHERITANCE IN C++

This problem can be resolved by class name and using scope resolution operator to specify the class whose method is called.

Syntax

<code>derived_objectname.parent_classname::same_named_function([parameter]);</code>

In the above example, if we want to call the method of class A then we can call it as below,

sample.A::display();

Similarly, if we need to call the method of class B then,

sample.B::display();

//program to display petrol's data using Multiple Inheritance from fuel and liquid

```
#include <iostream>
#include <conio.h>
using namespace std;
class liquid
{
    float specific_gravity;
public:
    void input()
    {
        cout<<"Specific gravity: ";
        cin>>specific_gravity;
    }
    void output()
    {
        cout<<"Specific gravity: "<<specific_gravity<<endl;
    }
};
class fuel
{
    float rate;
public:
    void input()
    {
        cout<<"Rate(per liter): $";
        cin>>rate;
    }
}
```



```
void output()
{
    cout<<"Rate(per liter): $"<<rate<<endl;
}
};
class petrol: public liquid, public fuel
{
public:
    void input()
    {
        liquid::input();
        fuel::input();
    }
    void output()
    {
        liquid::output();
        fuel::output();
    }
};
void main()
{
    petrol p;
    cout<<"Enter data"<<endl;
    p.input();
    cout<<endl<<"Displaying data"<<endl;
    p.output();
    getch();
}
```

Output

Enter data
Specific gravity: 0.7
Rate(per liter): \$0.99

Displaying data
Specific gravity: 0.7
Rate(per liter): \$0.99

In this program, *petrol* is derived from *fuel* having attribute *rate* and *liquid* having attribute *specific gravity*. So the public features of both *fuel* and *petrol* are inherited to *petrol*. Every class has a method named *input()* for providing input and another method named *output()* to display the data.

HIERARCHICAL INHERITANCE

If more than one class is inherited from the base class, it's known as hierarchical inheritance. In hierarchical inheritance, all features that are common in child classes are included in the base class.

For example: Physics, Chemistry, Biology are derived from Science class.

Syntax of Hierarchical Inheritance

```

class base_class
{
    ... ..
};

class first_derived_class: public base_class
{
    ... ..
};

class second_derived_class: public base_class
{
    ... ..
};

class third_derived_class: public base_class
{
    ... ..
};

```

//Program to illustrate hierarchical inheritance.

```

#include<iostream.h>
#include<conio.h>
class Person
{
protected:
int id;
char *name;
};
class Student:public Person
{
float percentage;
public:
void readstud()
{
cout<<"Enter the id,name & percentage\n";
cin>>id>>name>>percentage;
}
void displaystud()
{
cout<<"Student id:"<<id;
cout<<"\nName:"<<name;
cout<<"\nPercentage:"<<percentage;
}
};
class Faculty:public Person
{
char *dept;

```

```
float salary;
public:
void readfac()
{
cout<<"\nEnter the id,name,department & salary\n";
cin>>id>>name>>dept>>salary;
}

void displayfac()
{
cout<<"Faculty id:"<<id;
cout<<"\nName:"<<name;
cout<<"\nDepartment:"<<dept;
cout<<"\nSalary:"<<salary;
}
};

void main()
{
clrscr();
Student s;
Faculty f;
s.readstud();
cout<<"\n*****STUDENT DETAILS*****\n";
s.displaystud();
cout<<"\n";
f.readfac();
cout<<"\n*****FACULTY DETAILS*****\n";
f.displayfac();
getch();
}
/*-----OUTPUT-----
Enter the id,name & percentage 111 Akhil 90
```

*****STUDENT DETAILS*****

Student id:111

Name:Akhil

Percentage:90

Enter the id,name,department & salary 121 Ashwin Computers 10000

*****FACULTY DETAILS*****

Faculty id:121

Name:Ashwin

Department:Computers

Salary:10000 */

MULTILEVEL INHERITANCE

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

Syntax for Multilevel Inheritance

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

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

//Program to illustrate multilevel inheritance

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Student
{
protected:
int roll_no;
char*name;
public:
void getnumber(int a,char n[]);
void putnumber();
};
void Student::getnumber(int a,char n[])
{
roll_no=a;
strcpy(name,n);
}
void Student::putnumber()
{
cout<<"Roll_no:"<<roll_no<<endl;
cout<<"Name:"<<name;
}
class Test:public Student
{
protected:
int s1,s2;
```

```
public:
void getmarks(int,int);
void putmarks();
};
void Test::getmarks(int x,int y)
{
s1=x;
s2=y;
}
void Test::putmarks()
{
cout<<"\ns1:"<<s1<<endl<<"s2:"<<s2<<endl;
}
class Result:public Test
{
int total;
public:
void display();
};
```

```
void Result::display()
{
total=s1+s2;
putnumber();
putmarks();
cout<<"Total="<<total<<endl;
}
```

```
void main()
{
Result r;
int roll_no,s1,s2;
char*name;
clrscr();
cout<<"Enter the roll_no,name,s1,s2";
cin>>roll_no>>name>>s1>>s2;
r.getnumber(roll_no,name);
r.getmarks(s1,s2);
r.display();
getch();
}
```

```
/*-----OUTPUT-----
```

```
Enter the roll_no,name,s1,s2 121 Anusha 80 90
```

```
Roll_no:121
```

```
Name:Anusha
```

```
s1:80
```

```
s2:90
```

```
Total=170*/
```


HYBRID (VIRTUAL) INHERITANCE

Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance. A class is derived from two classes as in multiple inheritance. However, one of the parent classes is not a base class. It is a derived class.

//Program to illustrate hybrid inheritance.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student
{
protected:
int roll_no;
char *name;
public:
void get_no(int a,char n[])
{
roll_no=a;
strcpy(name,n);
}
void put_no()
{
cout<<"roll_no="<<roll_no<<endl;
cout<<"name="<<name<<endl;
}
};
```

class test:public student

```
{
protected:
float p1,p2;
public:
void get_marks(float x,float y)
{
p1=x;
p2=y;}
void put_marks()
{
cout<<"marks obtained:"<<endl<<"sub1:"<<p1<<endl<<"sub2:"<<p2<<endl;
}};
```

class sports

```
{
protected:
float score;
```

```
public:
void get_score(float s)
```

```
{
score=s;
}

void put_score()
{
cout<<"sports marks:"<<score<<endl;
}
};
class result:public test,public sports
{
float total;
public:
void display()
{
total=p1+p2+score;
put_no();
put_marks();
put_score();
cout<<"total score:"<<total<<endl;
}
};

void main()
{
clrscr();
int n,roll_no,s1,s2,s;
char *name;
result st1;
cout<<"Enter the roll_no,name,s1,s2,s";
cin>>roll_no>>name>>s1>>s2>>s;
st1.get_no(roll_no,name);
st1.get_marks(s1,s2);
st1.get_score(s);
st1.display();
getch();
}
/*-----OUTPUT-----*/
```

Enter the roll_no, name, s1, s2, s 11132 Sreedevi 85 98 85

Roll No=11132
Name : Sreedevi
Marks Obtained:
Sub1:85
Sub2:98
Sports marks: 85
Total Score: 260 */

UNIT IVPOLYMORPHISM

Polymorphism means more than one function with same name, with different working. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and morphs means forms. So polymorphism means many forms. Polymorphism can be static or dynamic. In static polymorphism memory will be allocated at compile-time. In dynamic polymorphism memory will be allocated at run-time. Both function overloading and operator overloading are an examples of static polymorphism. *Virtual function is an example of dynamic polymorphism.*

Static polymorphism is also known as early binding and compile-time polymorphism.

Dynamic polymorphism is also known as late binding and run-time polymorphism.

Real life example of Polymorphism in C++

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, here one person has different-different behaviors.



In Shopping malls behave like Customer

In Bus behave like Passenger

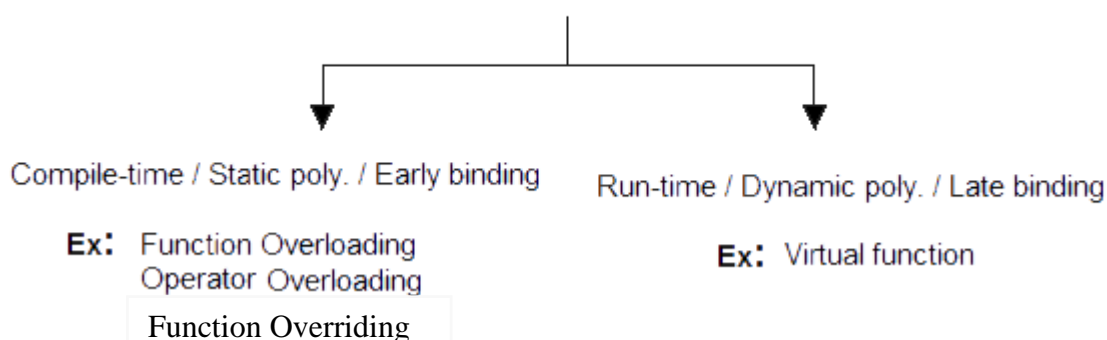
In School behave like Student

At Home behave like Son Sitesbay.com

Type of polymorphism

- Compile time polymorphism
- Run time polymorphism

Types of Polymorphism



Static Binding or Compile time polymorphism

In C++ programming you can achieve compile time polymorphism in two way, which is given below;

- Function or Method overloading
- Function or Method overriding
- Operator Overloading

Function or Method Overloading in C++

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as method overloading. In below example method "sum()" is present in Addition class with same name but with different signature or arguments.

Example of Method Overloading in C++

```
#include<iostream.h>
#include<conio.h>
class Addition
{
public:
void sum(int a, int b)
{
cout<<a+b;
}
void sum(int a, int b, int c)
{
cout<<a+b+c;
}
};
void main()
{
clrscr();
Addition obj;
obj.sum(10, 20);
cout<<endl;
obj.sum(10, 20, 30);
}
```

Output

30
60

नहि ज्ञानेन सदृशं

SHREE MEDHA COLLEGE

Function or Method Overriding in C++

Define any method in both base class and derived class with same name, same parameters or signature, this concept is known as **method overriding**. In below example same method "show()" is present in both base and derived class with same name and signature.

Function Overriding

Giving new implementation of base class method into derived class is called function overriding.

Signature of base class method and derived class must be same. Signature involves:

- ✓ Number of arguments
- ✓ Type of arguments
- ✓ Sequence of arguments

Example of Method Overriding in C++

```
#include<iostream.h>
#include<conio.h>
class Base
{
public:
void show()
{
cout<<"Base class";
}
};

class Derived:public Base
{
public:
void show()
{
cout<<"Derived Class";
}
};

void mian()
{
Base b;    //Base class object
Derived d; //Derived class object
b.show();  //Early Binding Occurs
d.show();
getch();
}
```

Output

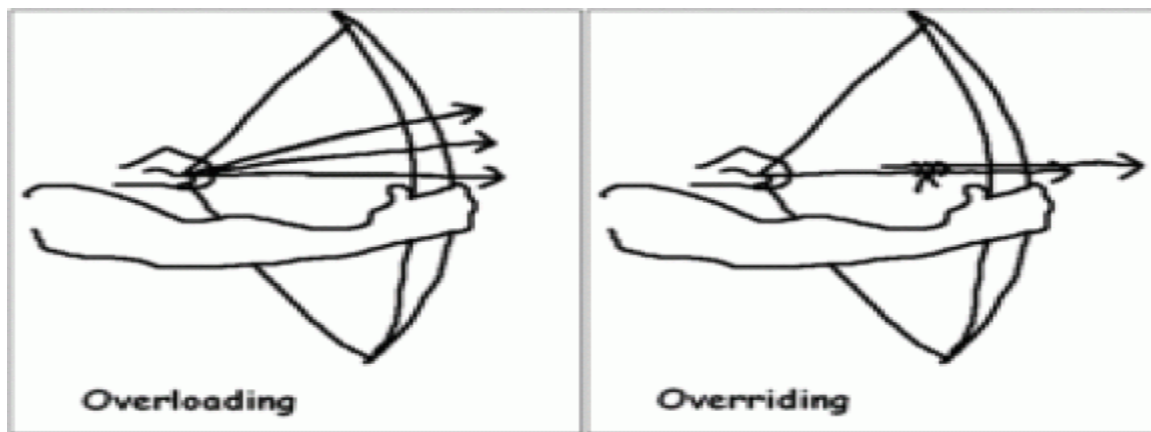
Base class
Derived Class

METHOD OVERLOADING VS METHOD OVERRIDING

- Overloading can occur without inheritance. Overriding of functions occurs when one class is inherited from another class.
- Overloaded functions must differ in function signature i.e. either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
- Overloaded functions are in same scope; whereas Overridden functions are in different scopes.

	Method Overloading	Method Overriding
Definition	Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	Sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behaviour	Method Overloading is to “add” or “extend” more to method’s behaviour.	Method Overriding is to “Change” existing behaviour of method.
Polymorphism	It is a compile time polymorphism .	It is a run time polymorphism .
Inheritance	It may or may not need inheritance in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have different signature .	In Method Overriding, methods must have same signature .
Relationship of Methods	Relationship is there between methods of same class.	Relationship is there between methods of super class and sub class.
Criteria	Methods have same name different signatures but in the same class.	Methods have same name and same signature but in the different class.
No. of Classes	Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.

Diagram to understand exact difference: -



POINTER TO OBJECTS

C++ allows you to have pointers to objects. The pointers pointing to objects are referred to as Object Pointers.

C++ Declaration and Use of Object Pointers

Just like other pointers, the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :

class-name * object-pointer;

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type. For example, to declare optr as an object pointer of Sample class type, we shall write

Sample *optr ;

where Sample is already defined class. When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

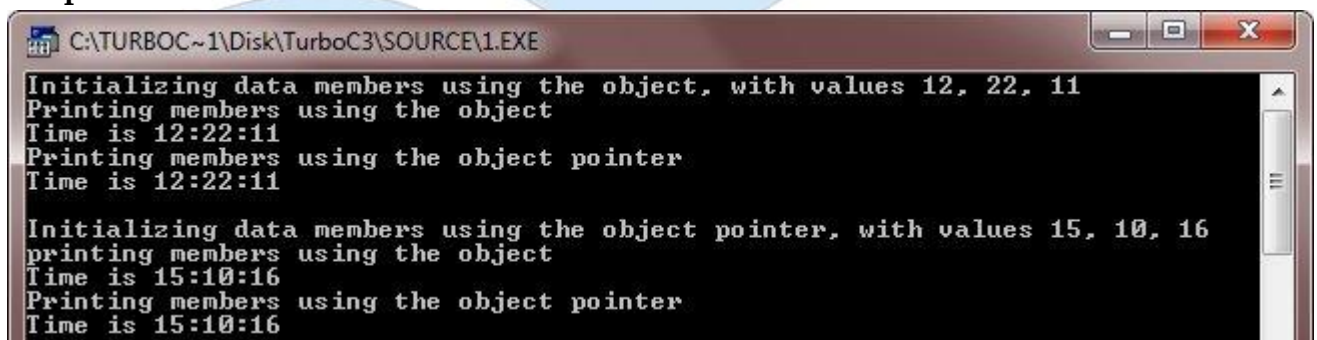
//Program to implement pointer to objects

```
#include<iostream.h>
#include<conio.h>
class Time
{
    short int hh, mm, ss;
public:
    Time()
    {
        hh = mm = ss = 0;
    }
    void getdata(int i, int j, int k)
    {
        hh = i;
```

```
        mm = j;
        ss = k;
    }
    void prndata(void)
    {
        cout<<"\nTime is "<<hh<<":"<<mm<<":"<<ss<<"\n";
    }
};

void main()
{
    clrscr();
    Time T1, *tptr;
    cout<<"Initializing data members using the object, with values 12, 22, 11\n";
    T1.getdata(12,22,11);
    cout<<"Printing members using the object ";
    T1.prndata();
    tptr = &T1;
    cout<<"Printing members using the object pointer ";
    tptr->prndata();
    cout<<"\nInitializing data members using the object pointer, with values
15, 10, 16\n";
    tptr->getdata(15, 10, 16);
    cout<<"printing members using the object ";
    T1.prndata();
    cout<<"Printing members using the object pointer ";
    tptr->prndata();
    getch();
}
```

Output



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1.EXE
Initializing data members using the object, with values 12, 22, 11
Printing members using the object
Time is 12:22:11
Printing members using the object pointer
Time is 12:22:11

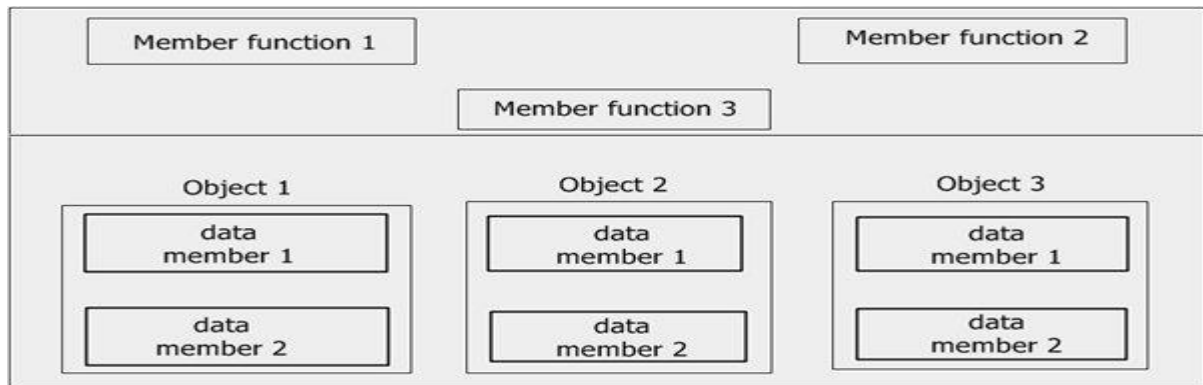
Initializing data members using the object pointer, with values 15, 10, 16
printing members using the object
Time is 15:10:16
Printing members using the object pointer
Time is 15:10:16
```

The above program is self-explanatory. To access public members using an object the dot operator is used and to access public members using an object pointer the arrow operator is used.

As you know, when a pointer is incremented, it points to the next element of its type. The same is true of pointer to object.

C++ this Pointer

As soon as you define a class, the member functions are created and placed in the memory space only once. That is, only one copy of member functions is maintained that is shared by all the objects of the class. Only space for data members is allocated separately for each object (See the following figure).



This has an associated problem. If only one instance of a member function exists, how does it come to know which object's data member is to be manipulated? For example, if member function 3 is capable of changing the value of data member2 and we want to change the value of data member2 of object1. How could the member function3 come to know which object's data member2 is to be changed?


The answer to this problem is this pointer. When a member function is called, it is automatically passed an implicit (in-built) argument that is a pointer to the object that invoked the function. This pointer is called this. That is if object1 is invoking member function3, then an implicit argument is passed to member function3 that points to object1 i.e., this pointer now points to object1. The this pointer can be thought of analogous to the ATM card. For instance, in a bank there are many accounts. The account holders can withdraw amount or view their bank-statements through Automatic-Teller-Machines. Now, these ATMs can withdraw from any account in the bank, but which account are they supposed to work upon? This is resolved by the ATM card, which gives the identification of user and his accounts, from where the amount is withdrawn.

Similarly, the this pointer is the ATM cards for objects, which identifies the currently-calling object. The this pointer stores the address of currently-calling object. To understand this, consider the following example program (following program illustrates the functioning of this pointer) :

// Program demonstrates about the this pointer.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Salesman
{
    char name[1200];
    float total_sales;
public:
    Salesman(char *s, float f)
    {
        strcpy(name, "");
        strcpy(name, s);
        total_sales = f;
    }
    void prnobject(void)
    {
        cout.write(this->name, 26);    // use of this pointer
        cout<<" has invoked prnobject().\n";
    }
};
void main()
{
    clrscr();
    Salesman  Rajat("Rajat", 21450),  Ravi("Ravi", 23190),  Vikrant("Vikrant",
19142);
    /* above statement creates three objects */
    Rajat.prnobject();
    Vikrant.prnobject();
    Ravi.prnobject();
    getch();
}
```

output:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\2.EXE
Rajat has invoked prnobject().
Vikrant has invoked prnobject().
Ravi has invoked prnobject().
```

It is obvious from the above output that when the object Rajat invokes the member function prnobject(), this points to Rajat and thus prints the name data member of Rajat. Similarly, this points to Vikrant and Ravi objects when they invoke prnobject().

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus inside prnobject(), the statement : cout.write(this->name, 26) ; is same as cout.write(name,26);

Remembers, the this pointer points to the object invoked prnobject(). Thus, this->name refers to that object's copy of name. The (this) refers to the object itself. With (*this), dot(.) operator should be used in order to access the member elements of the object as this is not a pointer to object, it is the object itself.

Virtual Function

*Giving new implementation of base class method into derived class and the calling of this new implemented function with derived class's object is called **function overriding**.*

*Giving new implementation of derived class method into base class and the calling of this new implemented function with base class's object is done by making base class function as **virtual function**.*

Virtual function is used in situation, when we need to invoke derived class function using base class pointer. We must declare base class function as virtual using **virtual** keyword preceding its normal declaration. The base class object must be of pointer type so that we can dynamically replace the address of base class function with derived class function. This is how we can achieve "**Runtime Polymorphism**".

If we don't use virtual keyword in base class, base class pointer will always execute function defined in base class.

Describe the virtual function and virtual function table.

A virtual function in C++ is :

- A simple member function of a class which is declared with "virtual" keyword
- It usually performs different functionality in its derived classes.
- The resolving of the function call is done at run-time.

Virtual Table:

- A virtual table is a mechanism to perform dynamic polymorphism i.e., run time binding. Virtual table is used to resolve the function calls at runtime. Every class that uses virtual functions is provided with its own virtual functions.

- Every entry in the virtual table is a pointer that points to the derived function that is accessible by that class. A hidden pointer is added by a compiler to the base class which in turn calls *_vptr which is automatically set when an instance of the class is created and it points to the virtual table for that class.

//Program to implement Virtual Functions

#include<iostream.h>

#include<conio.h>

class A

{

int a;

public:

A()

{

a = 1;

}

virtual void show()

{

cout <<a;

}

};

Class B: public A

{

int b;

public:

B()

{

b = 2;

}

virtual void show()

{

cout <<b;

}

};

void main()

{

A *pA;

B oB;

pA = &oB;

pA->show();

}

OUTPUT 2

since pA points to object of B and show() is virtual in base class A.

Pure Virtual Function in C++

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. Pure virtual function doesn't have body or implementation. We must implement all pure virtual functions in derived class. Pure virtual function is also known as **abstract function**.

A class with at least one pure virtual function or abstract function is called **abstract class**. We can't create an object of abstract class. Member functions of abstract class will be invoked by derived class object.

//Program to create geometrical figure & find the area using pure virtual functions.

```
#include<iostream.h>
#include<conio.h>
class Geometrical_figure
{
public:
float a,b,area;
void getdata()
{
cin>>a>>b;
}
virtual void display()
{
}
};

class Triangle:public Geometrical_figure
{
public:
void display()
{
area=(a*b)/2;
cout<<"Area of the triangle="<<area<<endl;
}
};

class Rectangle:public Geometrical_figure
{
public:
void display()
{
area=a*b;
cout<<"Area of the rectangle="<<area<<endl;
}
};
```

```
void main()
{
Geometrical_figure *p1;
Triangle t1;
Rectangle r1;
char ch;
clrscr();
cout<<"Enter the type of the geometricalfigure(type t/r)"<<endl;
```

```
cin>>ch;
p1=new Geometrical_figure;
if(ch=='t' | | ch=='T')
{
cout<<"\nyour choice is for triangle"<<endl;
cout<<"Enter the values for base & height"<<endl;
p1=&t1;
p1->getdata();
p1->display();
}
else if(ch=='r' | | ch=='R')
{
cout<<"your choice is for rectangle"<<endl;
cout<<"Enter the values for length & breadth"<<endl;
p1=&r1;
p1->getdata();
p1->display();
}
else
{
cout<<"Not an valid entry for geometrical_figure"<<endl;
}
getch();
}
```

/*-----OUTPUT-----*/

Run 1:

Enter the type of the geometricalfigure(type t/r)

t

your choice is for triangle

Enter the values for base & height

2

4

Area of the triangle=4

Run 2:

Enter the type of the geometricalfigure(type t/r)

r

your choice is for rectangle

Enter the values for length & breadth

2

4

Area of the rectangle=8 */

ABSTRACT CLASS

Abstract class is used in situation, when we have partial set of implementation of methods in a class. For example, consider a class have four methods. Out of four methods, we have an implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use abstract class.

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.

A class with at least one **pure virtual function** or **abstract function** is called abstract class.

Pure virtual function is also known as abstract function.

- We can't create an object of abstract class b'coz it has partial implementation of methods.
- Abstract function doesn't have body

We must implement all abstract functions in derived class.

C++ File Handling or Stream classes

As we know, at the time of execution, every program comes in main memory to execute. Main memory is volatile and the data would be lost once the program is terminated. If we need the same data again, we have to store the data in a file on the disk. A file is sequential stream of bytes ending with an end-of-file marker.

Types of file supported by C++:

- Text Files
- Binary Files

Difference between text file and binary file

Text file is human readable because everything is stored in terms of text. In binary file everything is written in terms of 0 and 1, therefore binary file is not human readable.

A newline(\n) character is converted into the carriage return-linefeed combination before being written to the disk. In binary file, these conversions will not take place.

In text file, a special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file. There is no such special character present in the binary mode files to mark the end of file.

In text file, the text and characters are stored one character per byte. For example, the integer value 23718 will occupy 2 bytes in memory but it will occupy 5 bytes in text file. In binary file, the integer value 23718 will occupy 2 bytes in memory as well as in file.

The fstream.h Header File

C++'s standard library called fstream, defines the following classes to support file handling.

ofstream class : Provides methods for writing data into file. Such as, open(), put(), write(), seekp(), tellp(), close(), etc.

ifstream class : Provides methods for reading data from file. Such as, open(), get(), read(), seekg(), tellg(), close(), etc.

fstream class : Provides methods for both writing and reading data from file. The fstream class includes all the methods of ifstream and ofstream class.

Opening a file using open() member function

The open() function takes file-name argument. The purpose of opening the file i.e, whether for reading or writing, depends on the object associated with open() function.

Example of opening file

```
ofstream fout;
fout.open("filename"); // Open file for writing
ifstream fin;
fin.open("filename"); // Open file for reading
fstream f;
f.open("filename",mode);
// Using fstream, we can do both read and write, therefore,
// mode specifies the purpose for which the file is opened.
```

File Opening Modes

Mode	Purpose
ios::in	Open a text file for reading.
ios::out	Open a text file for writing. If it doesn't exist, it will be created.
ios::ate	Open a file and move the pointer at the end-of-file.
ios::app	Open a text file for appending. Data will be added at the end of the existing file. If file doesn't exist, it will be created.
ios::binary	Open file in a binary mode.
ios::nocreate	The file must already exist. If file doesn't exist, it will not create new file.
ios::trunc	If the file already exists, all the data will be lost.

Reading and Writing into File

We can read data from file and write data to file in four ways.

- Reading or writing characters using **get()** and **put()** member functions. More info
- Reading or writing formatted I/O using insertion operator (<<) and extraction operator (>>).
- Reading or writing object using read() and write() member functions.

TEMPLATE

C++ template is used in situation where we need to write the same function for different data types. For example, if we need a function to add two variables. The variable can be integer, float or double. For this purpose, we have to write one function for each data type. To avoid writing the same function for different data types we use **template**.

There are two types of templates in C++ :

- Function template
- Class template

Function Template or Generic Functions

A generic function that represents several functions performing same task but on different data types is called function template. Function templates are those functions which can handle different data types without separate code for each of them.

For example, a function to add two integer and float numbers requires two functions. One function accepts integer types and the other accept float types as parameters even though the functionality is the same. Using a function template, a single function can be used to perform both additions. It avoids unnecessary repetition of code for doing same task on various data types.

Syntax for function template

```
template < class T1, class T2, ... >
returntype function_name (arguments of type T1, T2, ...)
{
    statement(s);
    ... ..
}
```

/Program to find largest of 2 integers & floating numbers using

Function templates. */

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

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

```
template<class T>
```

```
void Large(T a,T b)
```

```
{
```

```
T c;
```

```
if(a>b)
```

```
cout<<a<<" is larger than "<<b;
```

```
else
```

```
cout<<b<<" is larger than "<<a;
```

```
}
```

```
void main()
```

```
{
```

```
int i1,i2;
```

```
float f1,f2;
```

```
clrscr();
```

```
cout<<"\nEnter any two integer numbers"<<endl;
```

```
cin>>i1>>i2;
```

```
Large(i1,i2);
```

```
cout<<"\nEnter any two floating numbers"<<endl;
```

```
cin>>f1>>f2;
```

```
Large(f1,f2);
```

```
getch();
```

```
}
```

```
/*-----OUTPUT-----
```

```
Enter any two integer numbers 92 82
```

```
92 is larger than 82
```

```
Enter any two floating numbers 2.5 5.9
```

```
5.9 is larger than 2.5 */
```

नहि ज्ञानेन सदृशं

CLASS TEMPLATE OR GENERIC CLASS

The relationship between a class template and an individual class is like the relationship between a class and an individual object. *An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.*

Note the distinction between the terms class template and template class:

Class template is a template used to generate template classes. You cannot declare an object of a class template.

Template class is an instance of a class template. A template definition is identical to any valid class definition that the template might generate, except for the following:

Syntax for Class Template

```
template <class T1, class T2, ...>
class classname
{
    attributes;
    methods;
};
```

- The class template definition is preceded by

template< *template-parameter-list* >

where *template-parameter-list* is a comma-separated list of one or more of the following kinds of template parameters:

- type
- non-type
- template
- Types, variables, constants and objects within the class template can be declared using the template parameters as well as explicit types (for example, int or char).

//Program to illustrate for class templates

```
#include<iostream.h>
#include<conio.h>
template<class T>
class Addition
{
    T a,b,total;
public:
    void getdata();
    void sum();
};
template<class T>
void Addition<T>::getdata()
{
    cout<<"Enter two numbers\n"<<endl;
    cin>>a>>b;
}
template<class T>
void Addition<T>::sum()
{
    T total;
    total=a+b;
    cout<<a<<"+"<<b<<"="<<total<<endl;
}
```

```
void main()
{
clrscr();
Addition<int> iobj;
Addition<float> fobj;
iobj.getdata();
cout<<"Addition of two integer numbers is "<<endl;
iobj.sum();
fobj.getdata();
cout<<"\nAddition of two floating point numbers is "<<endl;
fobj.sum();
getch();
}
/*-----OUTPUT-----*/
Enter two numbers 2 4
Addition of two integer numbers is
2+4=6
Enter two numbers 2.4 5.6
Addition of two floating point numbers is
2.4+5.6=8 */
```

ADVANTAGES & DIS ADVANTAGES OF TEMPLATES

Advantages:

- It provides us **type-safe, efficient** generic containers and generic algorithms
- The main reason for using C++ and templates is the trade-offs in performance and maintainability outweigh the bigger size of the resulting code and longer compile times.
- The drawbacks of not using them are likely to be much greater.

Disadvantages:

- Templates can lead to slower compile-times and possibly larger executable.
- Compilers often produce incomprehensible poor error diagnostics and poor error messages.
- The design of the STL collections tends to lead to a lot of copying of objects. The original smart pointer, `std::auto_ptr`, wasn't suitable for use in most collections. Things could be improved if we use other smart pointers from boost or C11.

Exception Handling - try catch

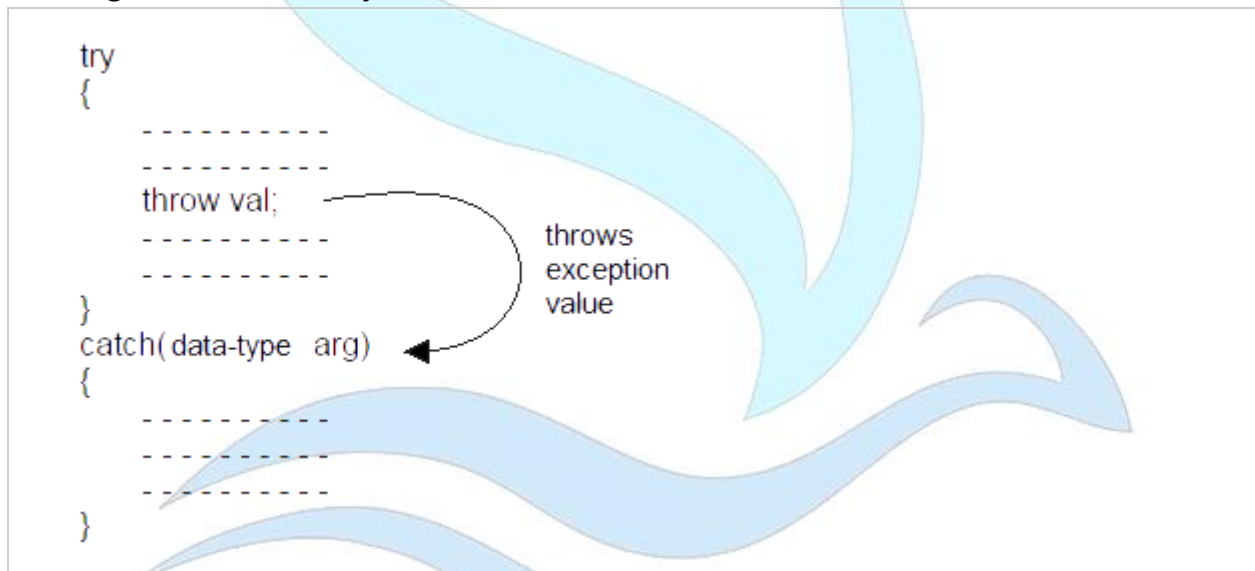
An exception is a situation, which occurred by the runtime error. In other words, an exception is a runtime error. An exception may result in loss of data or an abnormal execution of program.

Exception handling is a mechanism that allows you to take appropriate action to avoid runtime errors.

C++ provides three keywords to support exception handling.

- **Try** : The try block contain statements which may generate exceptions.
- **Throw** : When an exception occur in try block, it is thrown to the catch block using throw keyword.
- **Catch** : The catch block defines the action to be taken, when an exception occur.

The general form of try-catch block in c++.



//Program to implement simple try-throw-catch

```

#include<iostream.h>
#include<conio.h>
void main()
{
    int n1,n2,result;
    cout<<"\nEnter 1st number : ";
    cin>>n1;
    cout<<"\nEnter 2nd number : ";
    cin>>n2;
    try
    {
        if(n2==0)
  
```

```
throw n2;    //Statement 1
else
{
result = n1 / n2;
cout<<"\nThe result is : "<<result;
}
}
catch(int x)
{
cout<<"\nCan't divide by : "<<x;
}
cout<<"\nEnd of program.";
}
```

Output :

```
Enter 1st number : 45
Enter 2nd number : 0
Can't divide by : 0
End of program
```

The catch block contains the code to handle exception. The catch block is similar to function definition.

```
catch(data-type arg)
{
-----
-----
-----};
```

Data-type specifies the type of exception that catch block will handle, Catch block will receive value, send by throw keyword in try block.

Multiple Catch Statements

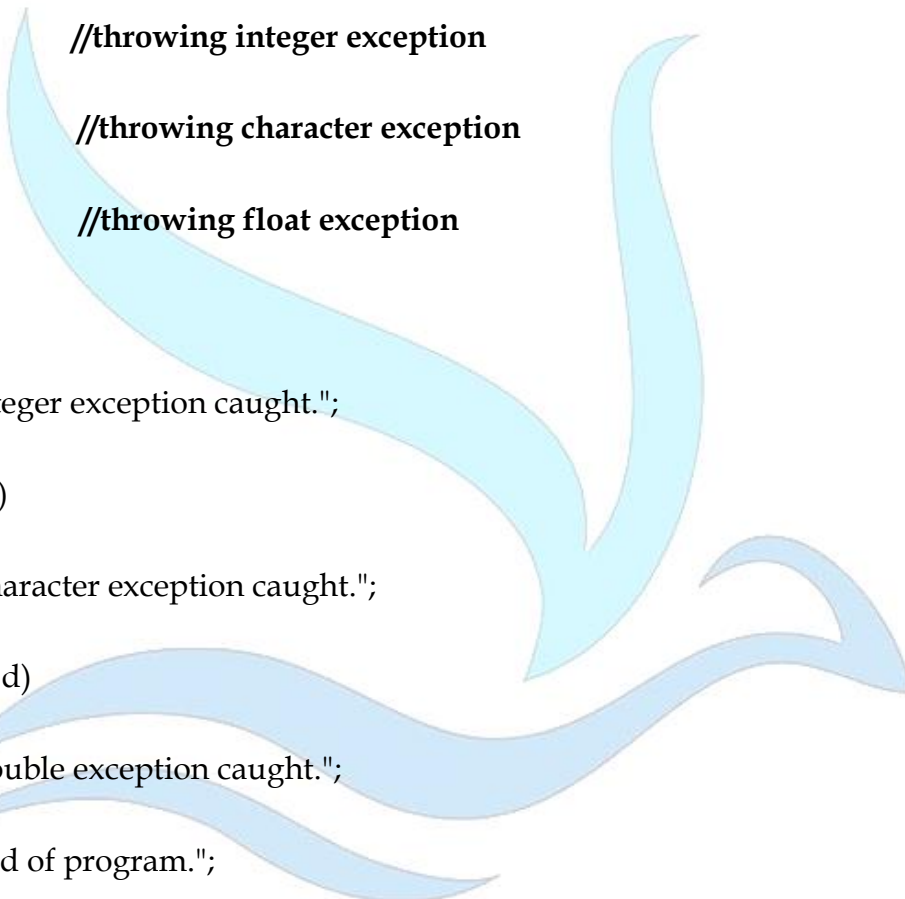
A single **try** statement can have multiple catch statements. Execution of particular catch block depends on the type of exception thrown by the throw keyword. If throw keyword send exception of integer type, catch block with integer parameter will get execute.

//Program to implement multiple catch blocks

```

#include<iostream.h>
#include<conio.h>
void main()
{
int a=2;
try
{
if(a==1)
throw a;           //throwing integer exception
else if(a==2)
throw 'A';         //throwing character exception
else if(a==3)
throw 4.5;         //throwing float exception
}
catch(int a)
{
cout<<"\nInteger exception caught.";
}
catch(char ch)
{
cout<<"\nCharacter exception caught.";
}
catch(double d)
{
cout<<"\nDouble exception caught.";
}
cout<<"\nEnd of program.";
}

```


Output :

Character exception caught.
End of program.

Catch All Exceptions

The above example will have caught only three types of exceptions that are integer, character and double. If an exception occurs of long type, no catch block will get execute and abnormal program termination will occur. To avoid this, we can use the catch statement with three dots as parameter (...) so that it can handle all types of exceptions.

//Program to implement catch all exceptions

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a=1;
try
{
if(a==1)
throw a;           //throwing integer exception
else if(a==2)
throw 'A';         //throwing character exception
else if(a==3)
throw 4.5;         //throwing float exception
}
catch(...)
{
cout<<"\nException occur.";
}
cout<<"\nEnd of program.";
}
```

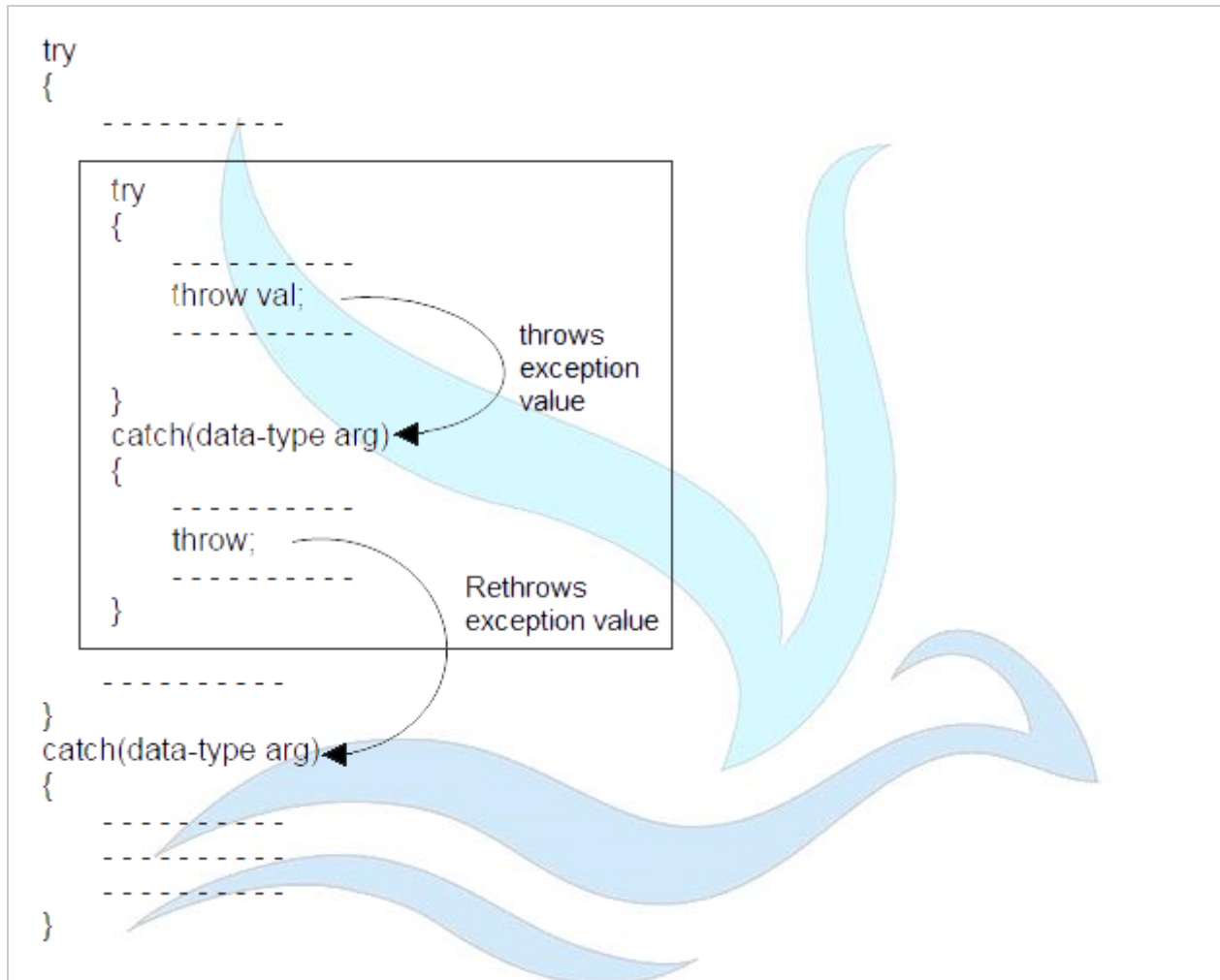
Output :
Exception occur.
End of program.

नहि ज्ञानेन सद्रशं
SHREE MEDHA COLLEGE

Rethrowing Exceptions

Rethrowing exception is possible, where we have an inner and outer try-catch statements (Nested try-catch). An exception to be thrown from inner catch block to outer catch block is called rethrowing exception.

Syntax of rethrowing exceptions



नहि ज्ञानेन सद्रशं
SHREE MEDHA COLLEGE

//Program to implement rethrowing exceptions

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a=1;
try
{
try
{
throw a;
}
catch(int x)
{
cout<<"\nException in inner try-catch block.";
throw x;
}
}
catch(int n)
{
cout<<"\nException in outer try-catch block.";
}
cout<<"\nEnd of program.";
}
```

Output :

Exception in inner try-catch block.
Exception in outer try-catch block.
End of program.

नहि ज्ञानेन सद्रशं
SHREE MEDHA COLLEGE

Restricting Exceptions

We can restrict the type of exception to be thrown, from a function to its calling statement, by adding throw keyword to a function definition.

//Program to implement restricting exceptions

```
#include<iostream.h>
#include<conio.h>
void Demo() throw(int ,double)
{
    int a=2;
    if(a==1)
        throw a;           //throwing integer exception
    else if(a==2)
        throw 'A';         //throwing character exception
    else if(a==3)
        throw 4.5;         //throwing float exception
}
void main()
{
    try
    {
        Demo();
    }
    catch(int n)
    {
        cout<<"\nException caught.";
    }
    cout<<"\nEnd of program.";
}
```

The above program will abort because we have restricted the Demo() function to throw only integer and double type exceptions and Demo() is throwing character type exception.