Institute for Advanced Computing And Software Development (IACSD) Akurdi, Pune

C++ Programming

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,

Nigdi Pradhikaran, Akurdi, Pune - 411044.

Introduction to c++

-C++ is a statically typed, compiled, general-purpose, case-sensitive,free-form programming language that supports procedural, object-oriented, and generic programming.
-C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
-C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.
-C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Features of C++:

1)Object-oriented:
C++ is an object-oriented programming language. This means that the focus is on "objects" and manipulations around these objects. Information about how these manipulations work is abstracted out from the consumer of the object.

2)Rich library support:
Through C++ Standard Template Library (STL) many functions are available that help in quickly writing code. For instance, there are standard libraries for various containers like sets, maps, hash tables, etc.

3)Speed: C++ is the preferred choice when latency is a critical metric. The compilation, as well as the execution time of a C++ program, is much faster than most other general purpose programming languages.

4)Compiled: A C++ code has to be first compiled into low-level code and then executed, unlike interpreted programming languages where no compilation is needed.

5)Pointer Support: C++ also supports pointers which are widely used in programming and are often not available in several programming languages.

6)Memory Management
C++ allows us to allocate the memory of a variable or an array in run time. This is known as Dynamic Memory Allocation.

In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.

In C++, the memory must be de-allocated dynamically allocated memory manually after it is of no use.
The allocation and deallocation of the memory can be done using the new and delete operators respectively.

Writing Cpp program:

```
// C++ program to display "Hello World"
// Header file for input output functions
#include <iostream>
using namespace std;
// Main() function: where the execution of  program begins
int main(){
   // Prints hello world
   cout << "Hello World";
   return 0;
}
```

note:
"using namespace std" means we use the namespace named std.
"std" is an abbreviation for standard.
So that means we use all the things with in "std" namespace.
If we don't want to use this line of code, we can use the things in this namespace like this. std::cout, std::endl.

cin and cout:
-cin is an object of the input stream and is used to take input from input streams like files, console, etc.
-cout is an object of the output stream that is used to show output.
-cin is an input statement while cout is an output statement.
-cerr-error message

```
cin>>var_name;
cout<<"message"<<var_name;
```

Four pillars of object-oriented development –
-Encapsulation-binding of data and function within a class
-Data hiding/Abstraction-hiding implementation details
-Inheritance:-reusability of code
-Polymorphism:-Many forms

What is a Token in C++?
What is a token?
A C++ program is composed of tokens which are the smallest individual unit. Tokens can be one of several things, including keywords, identifiers, constants, operators, or punctuation marks.

There are 6 types of tokens in c++:
Keyword
Identifiers
Constants
Strings
Special symbols
Operators

***1. Keywords:**
In C++, keywords are reserved words that have a specific meaning in the language and cannot be used as identifiers.

***2. Identifiers:**
In C++, an identifier is a name given to a variable, function, or another object in the code.
Rules-
-The first character must be a letter or an underscore.
-Identifiers cannot be the same as a keyword.
-Identifiers cannot contain any spaces or special characters except for the underscore.
-Identifiers are case-sensitive, meaning that a variable named "myVariable" is different from a variable named "myvariable".

***3. Constants:**
- In C++, a constant is a value that cannot be changed during the execution of the program.
- Constants often represent fixed values used frequently in the code, such as the value of pi or the maximum size of an array.
example:


```
const double PI = 3.14159;
```

Q.How to initialize the constant data members of class?

-**const Data Members**. **Data members** of a **class** may be dec lared as **const** .

- Such a **data member** must be **initialized** by the constructor using an **initialization** list.

-Once **initialized,** a **const data member** may never be modified, not even in the constructor or destructor.

A const data member cannot be initialized at the time of declaration or within the member function definition.

Declaration

const *data_type constant_member_name*;

Initialization

class_name(): *constant_member_name(value)* {

}

Program-

```
#include <iostream>
using namespace std;
class Number
{       private:
                const int x;
        public:
                //const initialization
                Number():x(36){}
                //print function
                void display()
                {cout<<"x="<<x<<endl;}
};

int main()

{       Number NUM;

        NUM.display();

        return 0;
}
```

***4. String:**
- In C++, a string is a sequence of characters that represents text.
- Strings are commonly used in C++ programs to store and manipulate text data.
-To use strings in C++, you must include the <string> header file at the beginning of your program.

***5. Special symbols:**
1: ampersand (&) is used to represent the address of a variable
2. tilde (~) is used to represent the bitwise NOT operator.
3. asterisk (*), used as the multiplication and pointer operators.
4: pound sign (#) is used to represent the preprocessor directive, a special type of instruction processed by the preprocessor before the code is compiled.
5. (%) is used as the modulus operator, which is used to find the remainder when one number is divided by another.
6. vertical bar (|) is used to represent the bitwise OR operator,
7. caret (^) is used to represent the bitwise XOR operator.
8. exclamation point (!) is used to represent the logical NOT operator, which is used to negate a Boolean value.

***6. Operators:**
-Arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, and division. Some examples of arithmetic operators include + (used for addition), - (used for subtraction), * (used for multiplication), and / (used for division).

-Relational operators
relational operators include == (used to check for equality), != (used to check for inequality), > (used to check if a value is greater than another), and < (used to check if a value is less than another).

-Logical operators
logical operators include && (used for the logical AND operation), || (used for the logical OR operation), and ! (used for the logical NOT operation).

Q.Variable initialization in C++

There are two ways to initialize the variable. One is static initialization in which the variable is assigned a value in the program and another is dynamic initialization in which the variables is assigned a value at the run time.

syntax of variable initialization.-

datatype variable_name = value;

Here,
datatype – The datatype of variable like int, char, float etc.
variable_name – This is the name of variable given by user.

Example

```
#include <iostream>
using namespace std;
int main() {
  int a = 20;
  int b;
  cout << "The value of variable a : "<< a; // static initialization
  cout << "\nEnter the value of variable b : "; // dynamic initialization
  cin >> b;
  cout << "\nThe value of variable b : "<< b;
  return 0;
}
```

Output
The value of variable a : 20
Enter the value of variable b : 28
The value of variable b : 28

C++ Variables, Literals and Constants

C++ Variables
In programming, a variable is a container (storage area) to hold data.

e.g.
int age = 14;

Rules for naming a variable
A variable name can only have alphabets, numbers, and the underscore _.
A variable name cannot begin with a number.
It is a preferred practice to begin variable names with a lowercase character. For example, name is preferable to Name.
A variable name cannot be a keyword. For example, int is a keyword that is used to denote integers.
A variable name can start with an underscore. However, it's not considered a good practice.

C++ Fundamental Data Types

Data Type	Meaning	Size (in Bytes)
int	Integer	2 or 4
float	Floating-point	4
double	Double Floating-point	8
char	Character	1
wchar_t	Wide Character	2
bool	Boolean	1
void	Empty	0

Data types in C++ are mainly Divided into 3 Types:

1. Primitive Data Types: These data types are built-in or predefined data types and can be used directly by the user to declare variables. Example: int, char, float, bool, etc. Primitive data types available in C++ are:
Integer
Character
Boolean
Floating Point
Double Floating Point
Valueless or Void
Wide Character

2. Derived Data Types: Derived data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
Function
Array
Pointer
Reference

3. Abstract or User-Defined Data Types: Abstract or User-Defined data types are defined by the user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
Class
Structure
Union
Enumeration

Q.Wide char data type-
Wide Character: Wide character data type is also a character data type but this data type has a size greater than the normal 8-bit data type. Represented by wchar_t. It is generally 2 or 4 bytes long.

sizeof()
-The size of variables might be different , depending on the compiler and the computer you are using.
e.g
use sizeof() operator to get size of various data types.

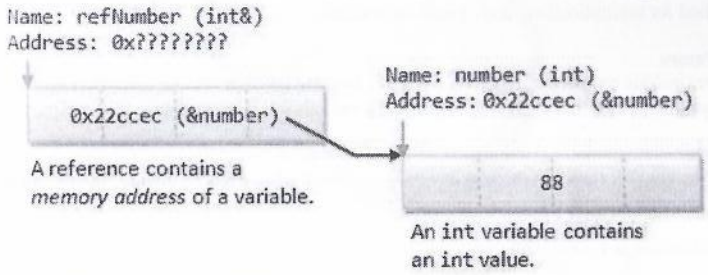
```
cout << "Size of char : " << sizeof(char) << endl;
cout << "Size of int : " << sizeof(int) << endl;
cout << "Size of short int : " << sizeof(short int) << endl;
cout << "Size of long int : " << sizeof(long int) << endl;
cout << "Size of float : " << sizeof(float) << endl;
cout << "Size of double : " << sizeof(double) << endl;
```


Reference Variable

- A reference variable is an alias, that is, another name for an already existing variable.
- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- The main use of references is acting as function formal parameters to support pass-by-reference.
- In an reference variable,it is passed into a function,the function works on the original copy (instead of a clone copy in pass-by-value).
- Changes inside the function are reflected outside the function.
- A reference is similar to a pointer. In many cases, a reference can be used as an alternative to pointer, in particular, for the function parameter.
- While using references you should know
- References have to be initialized.
- No memory is allocated to references.

Q.How References Work?

- A reference works as a pointer. A reference is declared as an alias of a variable. It stores the address of the variable,as illustrated:



Q.Diffrence between Pointer and reference variable-

Pointers	Reference
It's a separate variable that stores an address of another variable.	It's an alternative name given to the variable.
It has its own separate block of memory.	It doesn't have a separate block of memory.
It's a flexible connection i.e. a pointer declared can point to any variable, provided it's a non- const.	It's a rigid connection. i.e. A reference associated with a variable while initialization can't be assigned to another variable.

It needs an indirection operator for dereferencing.	It doesn't need any operator for dereferencing.

Function:

Function protyping-

-A function is a group of statements that together perform a task.
-Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.
-A function declaration tells the compiler about a function's name, return type, and parameters.
-A function definition provides the actual body of the function.

function declartion-

return_type function_name(parameter);

C++ function definition is as follows –

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

function call-

function_name(actual paramter);

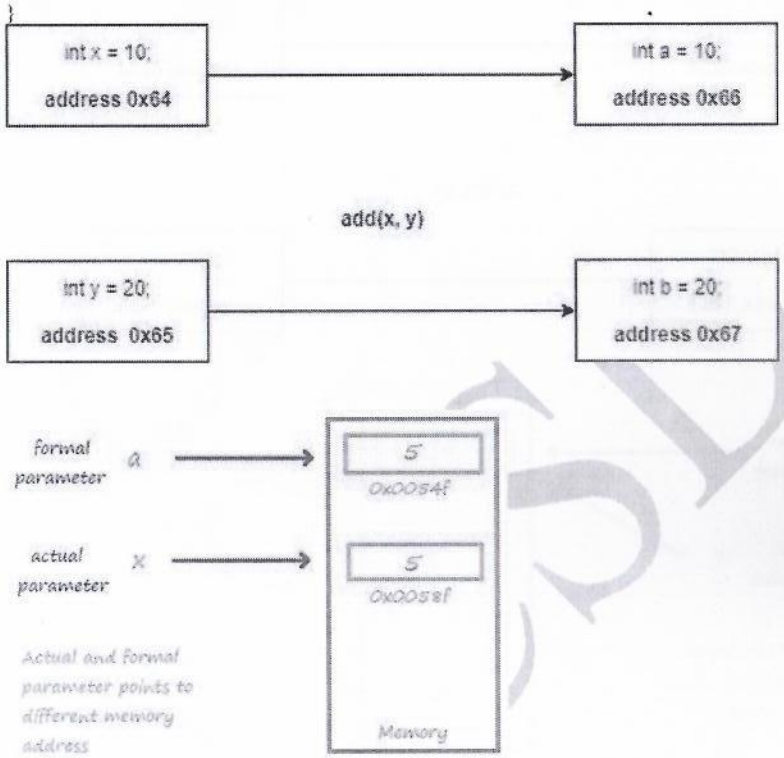
Types of function in C++-

- call by value
- call by reference
- call by address
- default argument/parameter in function
- inline function
- friend function

Call by value

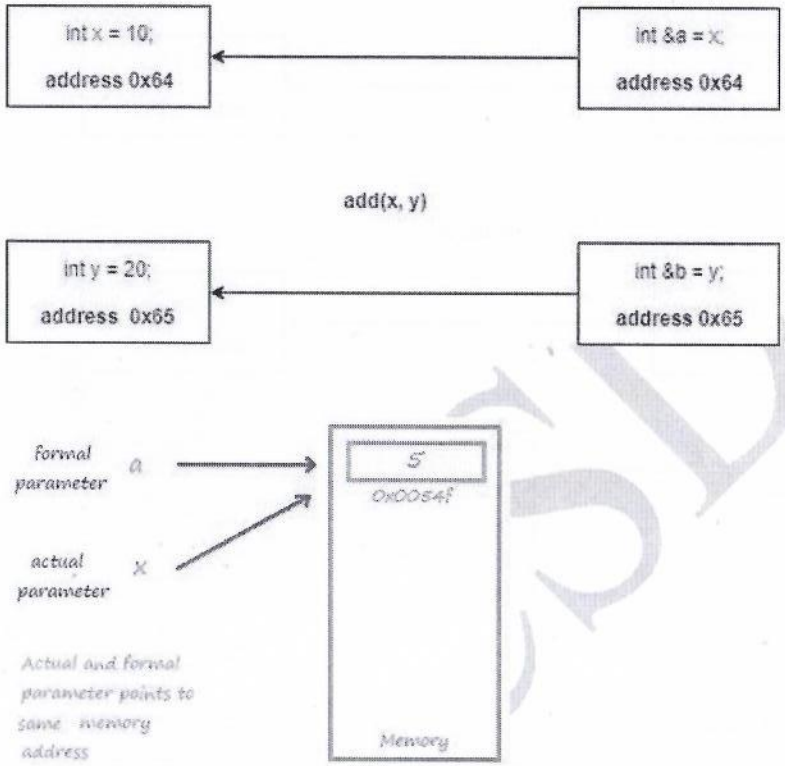
-widely used method.
-you don't want your original values of the variables to be changed.
-only the values of the variables are passed.
-achieved by creating dummy variables in memory.

```
void main(){
int x=8, y=6;
 int s=add(x,y); cout<<s;
}
int add(int a, int b)
{ return a+b
```

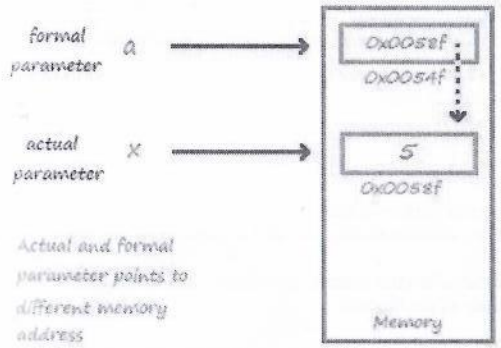
Call by Reference-

-Dummy variables are not created,
-A reference of an already existing variable is passed to the method.
-This reference points to the same memory location
-Hence separate copies are not made in the memory.
-The important point to note - changes made in the reference variables are reflected in the actual variable.



Call by address / Pass by address -

-The function arguments are passed as address.
-The caller function passes the address of the parameters.
-Pointer variables are used in the function definition.
-With the help of the Call by address method, the function can access the actual parameters and modify them. pointer variable holds the address of the actual parameter, hence the changes done by the formal parameter is also reflected in the actual parameter.



Default argument in function-

-A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument.
-In case any value is passed, the default value is overridden.

```
int sum(int x, int y, int z = 0, int w = 0)  //assigning default values to z,w as 0
{
   return (x + y + z + w);
}
int main()
{
   cout << sum(10, 15) << endl;              //25
   cout << sum(10, 15, 25) << endl;     //50
   cout << sum(10, 15, 25, 30) << endl;         //80
   return 0;
}
```

inline function -

-C++ provides an inline functions to reduce the function call overhead.
-Inline function is a function that is expanded in line when it is called.
-When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.
-This substitution is performed by the C++ compiler at compile time.
-Inline function may increase efficiency if it is small.
-Inline functions to reduce the function call overhead.

Q. What are inline function?

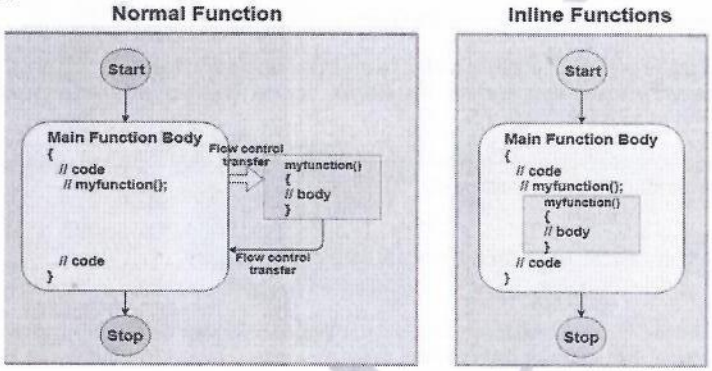
-Inline function is a function which when invoked requests the compiler to replace the calling statement with its body.

-A keyword **inline** is added before the function name to make it inline. It is an optimization technique used by the compilers as it saves time in switching between the functions otherwise.

- Member functions of a class are inline by default even if the keyword **inline** is not used.

Syntax of Inline Function

```
inline return_type function_name ([argument list])
{
  body of function
}
```



Q. Is it possible to have recursive inline function?

-An **inline function** cannot be **recursive** because in case of **inline function** the code is merely placed into the position from where it is called and does not maintain an information on stack which is necessary for **recursion**.

-Inline is just a request to the compiler to treat the call to a function like a macro and substitute it rather than treating it as a function and making a call.

-This request may or may not be considered by a compiler where in the second case it treats it as a normal function rather than inline.

-So even though you do try to make a recursive function inline the compiler will ignore it and consider it as a normal function and work in the normal fashion to execute it.

Friend function

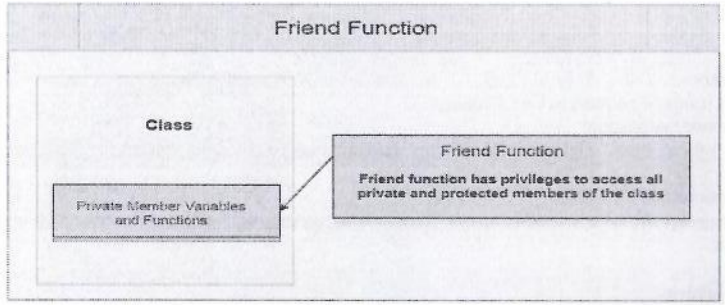
Q. Need of friend function in CPP.

Friend Function In C++

-A friend function in C++ is a function that is preceded by the keyword "friend". When the function is declared as a friend, then it can access the private and protected data members of the class.

-A friend function is declared inside the class with a friend keyword preceding as shown below.


```
class className{

 .....

 friend returnType functionName(arg list);
 };
```



In object-oriented programming, a **friend function**, that is a "friend" of a given class, is a function that is given the same access as methods to private and protected data.

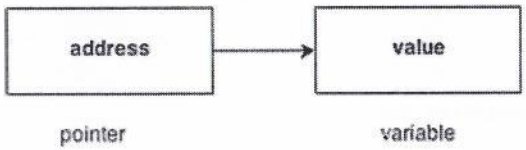
Friend function characterstics

- It is not in scope of class.
- It cannot be called using object of that class.
- It can be invoked like a normal function.
- It should use a dot operator for accessing members.
- It can be public or private.
- It has objects as arguments.
- Perhaps the most common use of friend functions is **overloading << and >>** for I/O.

Memory Management and Pointers-

C++ pointers-

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Usage of pointer-
There are many usage of pointers in C++ language.
1) Dynamic memory allocation
In C language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.
2) Arrays, Functions and Structures
Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Array of Pointers-

Array and pointers are closely related to each other.
In C++, the name of an array is considered as a pointer, i.e., the name of an array contains the address of an element.
C++ considers the array name as the address of the first element.

```
int *ptr[5];     // array of 5 integer pointer.
```

we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.

The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.

```
int a; // variable declaration.
  ptr[2] = &a;
```

Memory Management in c++

Q.What is Memory Management?

Memory management is a process of managing computer memory and assigning the memory space to the programs to improve the overall system performance.

C++ also supports malloc() and calloc() functions, but C++ also defines unary operators such as new and delete to perform the same tasks, i.e., allocating and freeing the memory dynamically.

New operator-

A new operator is used to create the object
while a delete operator is used to delete the object.

Advantages of the new operator-
-It does not use the sizeof() operator as it automatically computes the size of the data object.

-It automatically returns the correct data type pointer, so it does not need to use the typecasting.
-Like other operators, the new and delete operator can also be overloaded.
-It also allows you to initialize the data object while creating the memory space for the object.

Q.Differences between the malloc() and new-

-The new operator constructs an object, i.e., it calls the constructor to initialize an object while malloc() function does not call the constructor.

-The new operator invokes the constructor, and the delete operator invokes the destructor to destroy the object.
This is the biggest difference between the malloc() and new.

-The new is an operator, while malloc() is a predefined function in the stdlib header file.

-The operator new can be overloaded while the malloc() function cannot be overloaded.
If the sufficient memory is not available in a heap, then the new operator will throw an exception while the malloc() function returns a NULL pointer.

-In the new operator, we need to specify the number of objects to be allocated while in malloc() function, we need to specify the number of bytes to be allocated.

-In the case of a new operator, we have to use the delete operator to deallocate the memory. But in the case of malloc() function, we have to use the free() function to deallocate the memory.

Delete operator-

-It is an operator used in C++ programming language, and it is used to de-allocate the memory dynamically.
-This operator is mainly used either for those pointers which are allocated using a new operator or NULL pointer.

Syntax-
delete pointer_name;

Points to remember-
-It is either used to delete the array or non-array objects which are allocated by using the new keyword.
-To delete the array or non-array object, we use delete[] and delete operator, respectively.
-The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap.
-It does not destroy the pointer, but the value or the memory block,which is pointed by the pointer is destroyed.

C++ OOP Concepts:
-The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.
-Object Oriented Programming is a paradigm that provides many concepts such as inheritance, data binding, polymorphism etc.
-The programming paradigm where everything is represented as an object is known as truly object-oriented programming language.
-Smalltalk is considered as the first truly object-oriented programming language.
Object means a real word entity such as pen, chair, table etc.
Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects.

It simplifies the software development and maintenance by providing some concepts:
Object
Class
Inheritance
Polymorphism
Abstraction
Encapsulation

Class:
-A template for creating similar objects.
-Maps real world entities into classes through data members and member functions.
-A user defined type.
-An object is an instance of a class.
-By writing a class and creating objects of that class, one can map two concepts of object model, abstraction and encapsulation.
-A class in C++ is an encapsulation of data members and member functions that manipulate the data.

Class Component:
-A class declaration consists of following components
-Access specifiers: restrict access of class members
    private
    protected
    public
-Data members
-Member functions
-Constructors
-Destructors

Q. What are c++ access specifiers?
Access specifiers define how the members (attributes and methods) of a class can be accessed.

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

Access Specifiers in C++

Class Member Access Specifiers	Accessible from own class	Accessible from derived class	Accessible from object
Private Member	Yes	No	No
Protected Member	Yes	Yes	No
Public Member	Yes	Yes	Yes

DataMembers And Members Functions:

-Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

Following are the types of member functions of a class:
Mutator - Changes contents of instance members
Accessor - Accesses instance members
Facilitator - Helps to view the values of attributes of object
Helper - A private function, accessed from public member functions of same class to help in their implementation

Constructor:
-Special member function of the class with same name as its class name.
-Used to initialize attributes of an object
-Implicitly called when objects are created.
-Without any input parameter is no-argument constructor.
-Rules for implementing constructor:
1.No return type for constructor. Not even void.
2.Multiple constructors can be written - different number, types and order of parameters.
3.Must have same name as that of the class.

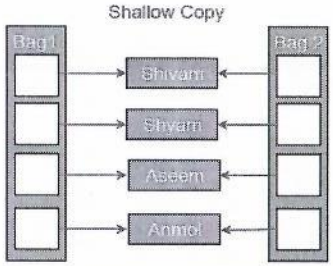
Q.What is copy constructor? Its need.
-A copy constructor is a member function which initializes an object using another object of the same class.
-A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

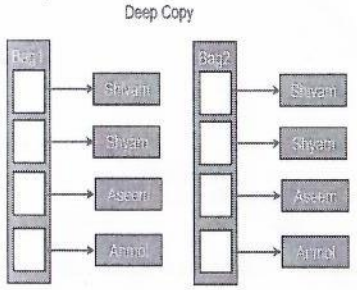
```
#include<iostream>
using namespace std;

class Point
{
private:
   int x, y;
public:
   Point(int x1, int y1) { x = x1; y = y1; }

   // Copy constructor
   Point(const Point &p2) {x = p2.x; y = p2.y; }

   int getX()      { return x; }
   int getY()      { return y; }
};

int main()
{
   Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1; // Copy constructor is called here

   // Let us access values assigned by constructors
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

   cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

   return 0;
}
```

In C++, a Copy Constructor may be called in following cases:
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When the compiler generates a temporary object.
   - If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects.
   - The compiler created copy constructor works fine in general.
   - We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection..etc.

Default constructor does only shallow copy.



Deep copy is possible only with user defined copy constructor.
In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



Q. In c++ can constructor and destructor be inline functions.

-Yes. Any function can be declared inline, and putting the function body in the class definition is one way of doing that.

-Putting the function definition in the class body equivelent to marking a function with the inline keyword. That means the function may or may not be inlined by the compiler.

Q. Can we have virtual constructor? Justify.

The virtual mechanism works only when we have a base class pointer to a derived class object.

In C++, the **constructor cannot be virtual**, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.

Virtual functions are used in order to invoke functions based on the type of object pointed to by the pointer, and not the type of pointer itself. But a **constructor** is not "invoked".

It is called only once when an object is declared. So, a **constructor** cannot be made **virtual** in C++.

Destructor:
-Destructor is a special member function of the class that is invoked implicitly to release the resources held by the object.
Characteristics:
-Has same name as that of class.
-Does not have a return type or parameters.
-Cannot be overloaded. Therefore a class can have only one destructor.
Implicitly called whenever an object ceases to exist.
Syntax:
~Date( );

Q. Can we have virtual destructor? Justify.
Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the **base class should be defined with a virtual destructor**.

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.

```
// A program with virtual destructor
#include<iostream>

using namespace std;

class base {
public:
        base()
        { cout<<"Constructing base \n"; }
        virtual ~base()
        { cout<<"Destructing base \n"; }
};


class derived: public base {
public:
        derived()
        { cout<<"Constructing derived \n"; }
        ~derived()
        { cout<<"Destructing derived \n"; }
};

int main(void)
{
derived *d = new derived();
base *b = d;
delete b;
getchar();
return 0;
}
```

Abstraction:

-Abstraction is the process of identifying the key aspects of an entity and ignoring the rest.
-Only those aspects are selected that are important to the current problem scenario.
Example : Abstraction of a person object
-Enumerate attributes of a "person object" that need to be created for developing a database
useful for social survey
useful for health care industry
useful for payroll system

Encapsulation:

-Encapsulation is a mechanism used to hide the data, internal structure, and implementation details of an object.
-All interaction with the object is through a public interface of operations.
-The user knows only about the interface; any changes to the implementation does not affect the user.

Inheritance:

-Classification helps in handling complexity.
-Inheritance is the process by which one object can acquire the properties of another object.

-Broad category is formed and then sub-categories are formed.
"is – a" a kind of hierarchy.

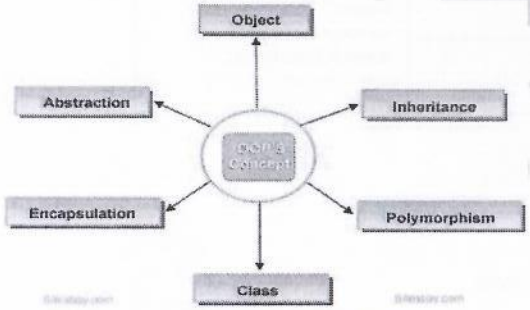
Polymorphism:

-The ability of different types of objects to respond to the same message in different ways is called polymorphism.
-Polymorphism helps to :
Design extensible software; as new objects can be added to the design without rewriting existing procedures.

Q. What is Object Oriented Programming?

-Object-Oriented programming is about creating objects that contain both data and functions.
-OOP is faster and easier to execute
-OOP provides a clear structure for the programs
-In oop, everything is represented as an object and when programs are executed, the objects interact with each other by passing messages.

-An object need not know the implementation details of another object for communicating.



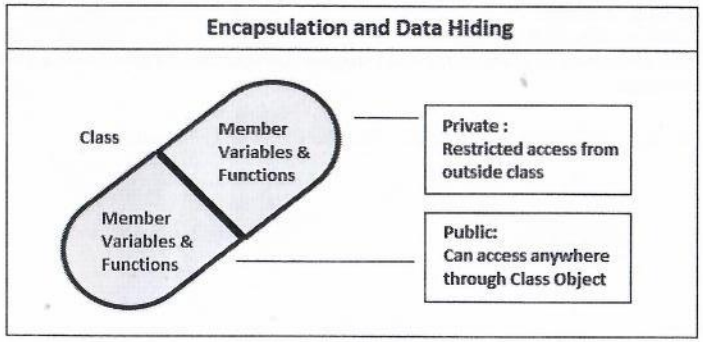
Q. What are main features of OOP?

- **Classes**
  Class is a user defined data type, It work as an object constructor
- **Encapsulation**
  It is the process of binding and hiding the state and behavior.
- **Abstraction**
  Selective ignorance.
- **Inheritance**
  It is the way of reusing already define classes with is-a relationship.
  Generalization
  Specialization
- **Polymorphism**
  Same message is implemented differently.

Q. What is Encapsulation?

-Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

-Encapsulation is achieve using two ways
1)Make the data members private.
2)Provide getters and setters for getting and setting value respectively.




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

class Encapsulation
{
        private:
                // data hidden from outside world
                int x;

        public:
                // function to set value of
                // variable x
                void set(int a)
                {
                        x =a;
                }

                // function to return value of
                // variable x
                int get()
                {
                        return x;
                }
};

// main function
int main()
```



```
{
        Encapsulation obj;

        obj.set(5);

        cout<<obj.get();
        return 0;
}
```

Q.What is containment-
Containment:
One object may contain another as a part of its attribute
Document contains sentences which contain words.
Computer system has a hard disk, processor, RAM, mouse, monitor, etc.
Containment need not be physical
E.g.1. Computer system has a warranty.
2.Employee has a DOB

Program-

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

class Date
{
        int dd, mm, yy;
public:
        Date()
        {
                dd = 0; mm = 0; yy = 0;
        }

        Date(int d, int m, int y)
        {
                dd = d;
                mm = m;
                yy = y;
        }
        void disp()
        {
                cout << "\ndate is:" << dd << "-" << mm << "-" << yy;
        }

};//end of class Date

class Emp
{       int eid, salary;
        Date dob;  //containment
public:
        Emp()
        {       eid = 0;
                salary = 0;
```

```
    }
    Emp(int i, int s, int day, int mon, int yr)
    {       eid = i;
            salary = s;
            dob = Date(day, mon, yr);

    }

    void disp1()
    {
            cout << "emplyee info:" << "eid:"<<eid << "salary:" << salary;
            dob.disp();
    }
};
int main()
{
    Emp e1(101, 2000, 12, 5, 1985);
    e1.disp1();

    return 0;
}
```

Containment VS Inheritance
Containment is used:
-When the features of an existing class are wanted inside a new class, but not its interface.
-e.g.Computer system has a hard disk.
-Car has an engine, chassis, steering wheel.

Inheritance is used:
-When it is necessary that the new type has to be the same type as the base class.
-Computer system is an electronic device.
-e.g.Car is a vehicle.

Q.What is sequence of constructor and destructors invocation in containment?

1.First the constructor of contained objects
2.Then constructor of container object

3.First the destructor of container objects
4.Then destructor of contained object

Q.Explain this keyword
this always holds a reference of an object which invokes the member function.
this points to an individual object.
It is a hidden parameter that is passed to every class member function.

Static Variables:
Some characteristics or behaviors belong to the class rather than a specific instance
e.g.interestRate, CalculateInterest method for a SavingsAccount class
e.g.count variable in Employee to automatically generate employee id

Such data members are static for all instances.
Change in static variable value affects all instances.
Also known as class variable.

Application-
To keep track how many objects created
Data to be shared by all objects is stored in static data members.
Only a single copy exists.
Class scope and lifetime is for entire program.

Q.What is static member?

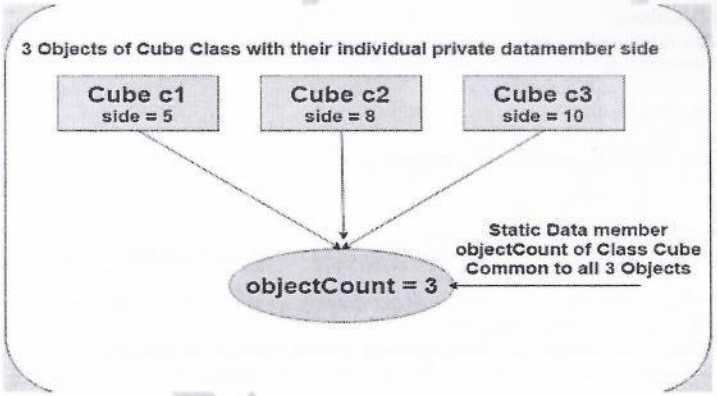
-We can define class members static using **static** keyword.

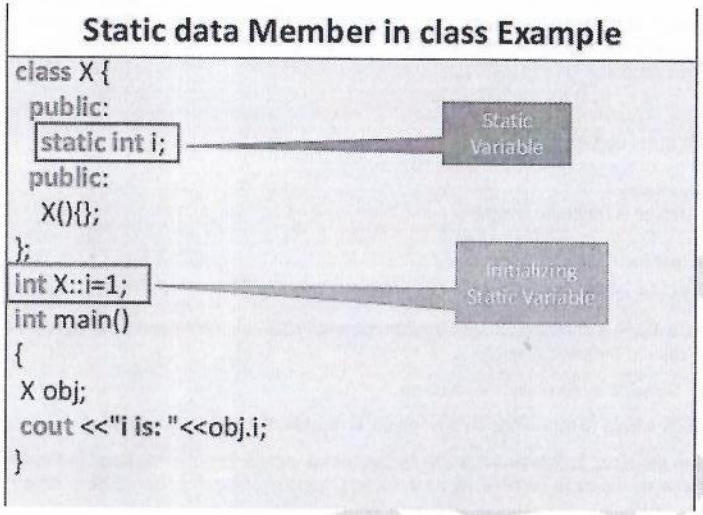
-When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

-A static member is shared by all objects of the class.

-All static data is initialized to zero when the first object is created, if no other initialization is present.

-We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.





Static Member Functions:
-It Can access static data members only.
-Invoked using class name as:

Class_Name::Static_Function();

-this pointer is never passed to a static member function.

Function Overloading:
-Using functions with same name but different signatures in the same program is called function overloading.
-Function overloading is a feature of object oriented programming where two or more functions can have the same name but different parameters.
-When a function name is overloaded with different jobs it is called Function Overloading.
-In Function Overloading "Function" name should be the same and the arguments should be different.

Inheritance:
-In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.
- In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.
-In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class.
- The derived class is the specialized class for the base class.

Advantage of C++ Inheritance
Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.
Types Of Inheritance
1.Single inheritance
2.Multiple inheritance
3.Hierarchical inheritance
4.Multilevel inheritance
5.Hybrid inheritance

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name : visibility-mode base_class_name
{
  // body of the derived class.
}
```

Where,
derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

In C++, the default mode of visibility is private.

The private members of the base class are never inherited.

Q.How to make a Private Member Inheritable?
-The private member is not inheritable.
-If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.
C++ introduces a third visibility modifier, i.e., protected.
-The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Public: When the member is declared as public, it is accessible to all the functions of the program.
Private: When the member is declared as private, it is accessible within the class only.
Protected: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Mode of Inheritance-
Private Mode Of Inheritance:
Class B:private A{}

The private members of base class A are not accessible in the derived class B:
All the protected members of the base class A are treated as private members of Derived class B;
All the public members of the base class A are treated as private members of Derived class B;

Protected Mode Of Inheritance:
Class B:protected A{}

The private members of base class A are not accessible in the derived class B:
All the protected members of the base class A are treated as protected members of Derived class B;
All the public members of the base class A are treated as protected members of Derived class B;

Public Mode Of Inheritance:
Class B:pubilc A{}

The private members of base class A are not accessible in the derived class B:
All the protected members of the base class A are treated as protected members of Derived class B;
All the public members of the base class A are treated as public members of Derived class B;

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance-
Single Inheritance:
One parent can have only one child. And that child have only one parent.

```
class A
{
 ---
};

class B :public A
{
  ----
};
```

C++ Multi Level Inheritance :

```
class A
{
 ---
};
class B:public A
{
 ---
};
```



```
class C: public B
{
 -----
};
```

When a class is derived from another derived class it is called as multilevel inheritance.
When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

C++ Multiple Inheritance

```
class A              //base class1
{
 ---
};

class B              //base class2
{
  ----
};


class C:public B , public  A      //derived class
{
 ---
};
```

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

C++ Hybrid Inheritance
Hybrid inheritance is a combination of more than one type of inheritance.

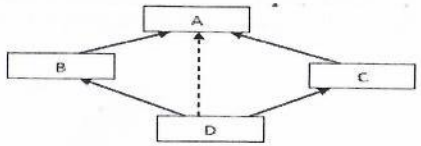
C++ Hierarchical Inheritance
Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

```
class A  //base class
{
 ---
};

class B :public A   //derived class1
{
  ----
};
class C :public A   //derived class2
{
 --
};
```

Diamond Inheritance: A class is derived from two classes and both these classes are derived from a common base class, then it is called as diamond inheritance.

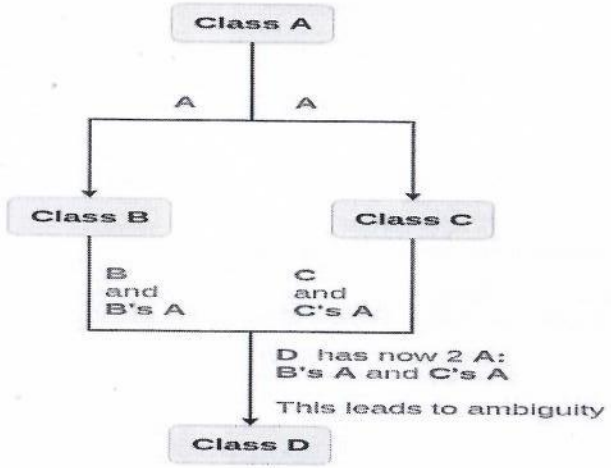
Q. When you should use the virtual inheritance?

To avoid Diamond inheritance problem we use virtual inheritance.



Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:
Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**.
One through class **B** and second through class **C**.
When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**.

Syntax 1:
```
class B : virtual public A
{
};
```

Syntax 2:
```
class C : public virtual A
{
};
```

Problem: Ambiguities in Diamond Inheritance-
Consider Employee as base class, SalesPerson and Manger are derived class and SalesManager derived from SalesPerson and Manger class(diamond Inheritance)

Case 1: what happens when two base classes contain a function with same name?
For example, SalesPerson as well as Manager class contain getName() function.

Case 2 : when derived class has multiple copies of the same base class.
For,example, SalesManager has multiple copies of data members of Employee class.

Duplicate data member ambiguity can be resolved by declaring a virtual base class.
Derive SalesPerson and Manager using virtual keyword and then derive SalesManager from these two.

By declaring base class as virtual, multiple copies of base class data member are not created.
There is only one copy of common base class data members in memory and its pointer reference is there in the derived class object.

The meaning of virtual keyword is overloaded.
The virtual keyword appears in the base lists of the derived classes.

```
class Employee              //base class
{
------
};

class Manager: virtual public Employee   //derived class
{
      .....
};
class SalesPerson: virtual public Employee    //derived class
 {
      .....
};

class SalesManger : public Manger , Public SalesPerson
{

     -----
}
```

Problems of Multiple Inheritance
class B : public A, public C { ... }

If multiple base classes contain a function with same name then resolve by any of the following ways:

Use scope resultion operator –
bobj.A::func() or
bobj.C::func()

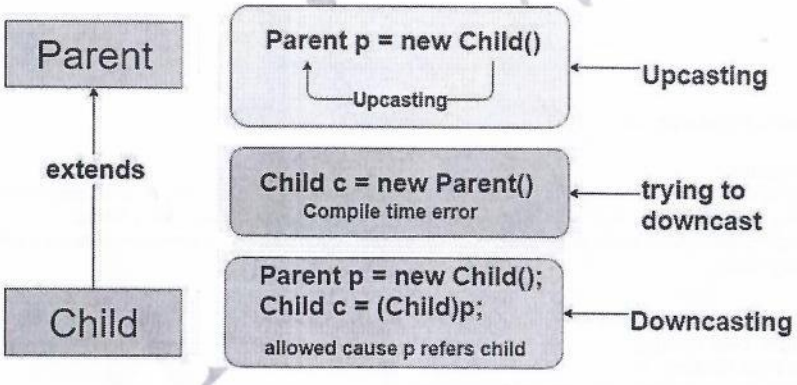
Override func() in B class.

Q.What is the need of down casting?

Downcasting is a process, which consists in converting base class pointer (or reference) to derived class pointer.

Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the **is-a** relationship is not, in most of the cases, symmetric.

A derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.



Generic Pointers:

e.g.
```
class Employee   //base class
{
---
---

};
class Trainer:public Employee  //derived class
{
---
---
};
```

In is-a relationship ,a base class pointer can point to a derived object without typecasting
i.e Employee *ptrEmp; //base class pointer
Trainer tObj; //derived class object
ptrEmp=&tObj;//no need of type casting

Base class reference can also refer to a derived object without cast;This is called as upCasting.

Derived object can be assigned to base object without any cast .
tObj=(Trainer)e; //casting is required

Q.What is generic pointer and its need?

-When a variable is declared as being a pointer to type **void** it is known as a ***generic pointer***.
-Since we cannot have a variable of type **void**, the pointer will not point to any data and therefore cannot be dereferenced.
-It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term *Generic pointer*.
-This is very useful when you want a pointer to point to data of different types at different times.

Generic Pointers

void *: a "pointer to anything"

```
void   *p;
int     i;
char    c;
p = &i;
p = &c;
putchar(*(char *)p);
```

type cast: tells the compiler to "change" an object's type (for type checking purposes - does not modify the object in any way)

Dangerous! Sometimes necessary...

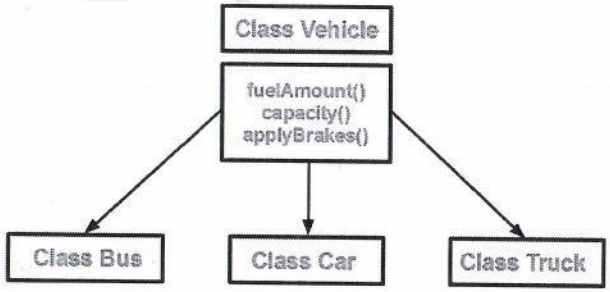
Lose all information about what type of thing is pointed to

- Reduces effectiveness of compiler's type-checking
- Can't use pointer arithmetic

Q5] What is inheritance? What is the purpose of Inheritance?

-One object takes the form of another object
-i.e one object acquires all properties and and functionality of another object .
-Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods. Inheritance allows us to reuse of code, it improves reusability in your application .
-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.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.




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

//Base class
class Parent
{
        public:
        int id_p;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
        public:
        int id_c;
};

//main function
int main()
{

                Child obj1;

                // An object of class child has all data members
                // and member functions of class parent
                obj1.id_c = 7;
                obj1.id_p = 91;
                cout << "Child id is " << obj1.id_c << endl;
                cout << "Parent id is " << obj1.id_p << endl;

                return 0;
}
```

Q.Explain Polymorphism.

Polymorphism:
-Ability of different related objects to respond to the same message in different ways is called polymorphism. The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.
- It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Types of Polymorphism-

1.Compile Time Binding
2. Run Time Binding.
Binding is an association of function call to an object.

Compile-time binding
The binding of a member function call with an object at compile-time.
Also called static type or early binding.
achived using
-function overloading
-operator overloading

Q. Explain function overloading and operator overloading.

-Function overloading reduces the investment of **different function** names and used to perform similar functionality by more than one **function**.

- Operator overloading : A feature in C++ that enables the redefinition of **operators**. This feature operates on user defined objects.

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

Function overloading can be considered as an example of polymorphism feature in C++.

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

void print(int i) {
 cout << " Here is int " << i << endl;
}
void print(double  f) {
 cout << " Here is float " << f << endl;
}
void print(char const *c) {
 cout << " Here is char* " << c << endl;
}

int main() {
 print(10);
 print(10.10);
 print("ten");
 return 0;
}
```

Operator Overloading in C++

In C++, we can make operators to work for user defined classes.

This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.
Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

```
#include<iostream>

using namespace std;

 class Complex {

private:

   int real, imag;
```



```
public:

   Complex(int r = 0, int i =0)  {real = r;   imag = i;}

   // This is automatically called when '+' is used with

   // between two Complex objects

   Complex operator + (Complex const &obj) {

      Complex res;

      res.real = real + obj.real;

      res.imag = imag + obj.imag;

      return res;

   }

   void print() { cout << real << " + i" << imag << endl; }

};

int main()

{    Complex c1(10, 5), c2(2, 4);

   Complex c3 = c1 + c2; // An example call to "operator+"

    c3.print();

}
```

Run-time Polymorphism (Dynamic binding)-
The binding of the function call to an object at run time.
Also called dynamic binding or late binding.
Achieved using virtual functions and inheritance.

Virtual Function
To implement late binding, the function is declared with the keyword virtual in the base class.
Points to note:
-Virtual function is a member function of a class.
-Virtual functions can be redefined in the derived class as per the design of the class.
-Also considered virtual by the compiler
-It is used to tell the compiler to perform dynamic linkage or late binding on the function.
-There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
-A 'virtual' is a keyword preceding the normal declaration of a function.

When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

syntax-within base class

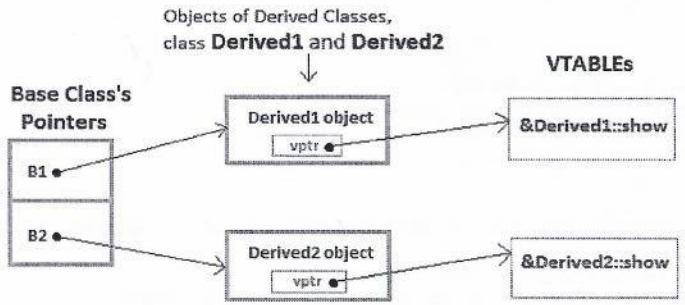
```
virtual ret_type fun_name()
{
//
}
```

in derived class-

```
ret_type fun_name()
{
//
}
```

Q7] What are virtual functions? Write an example.

-Virtual functions allow us to create a list of **base** class pointers and call methods of any of the derived classes without even knowing kind of derived class object.
-Virtual function is the member function of base class that u redefine in aderived class.
-It is declared using **virtual** keyword.
-It is used to tell the compiler to perform a dynamic leakage or late binding on the function.
-Syntax-----virtual int sum();
-When function is made virtual , c++ determines which function is to be invoked at the runtime based on the type of the object pointed to by base class pointer.
-Virtual function must be the member function of same class
-Virtual function can not be a static member.
-We can not have virtual constructor, but we can have virtual destructor.



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.


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

class Base {
public:
   virtual void show()
   {
      cout << " In Base \n";
   }
};

class Derived : public Base {
public:
   void show()
   {
      cout << "In Derived \n";
   }
};

int main(void)
{
   Base* bp = new Derived;

   // RUN-TIME POLYMORPHISM
   bp->show();

   return 0;
}
```

Q9] What are the VTABLE and VPTR?

-**Vtable** is a table of pointers to functions, used by a class instance for run-time method overrides. This is made possible by making methods virtual, using the virtual keyword. **Vptr** is the pointer within the class instance to the **vtable**.
- To implement virtual functions, C++ uses a special form of late binding known as the virtual table or vTable. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

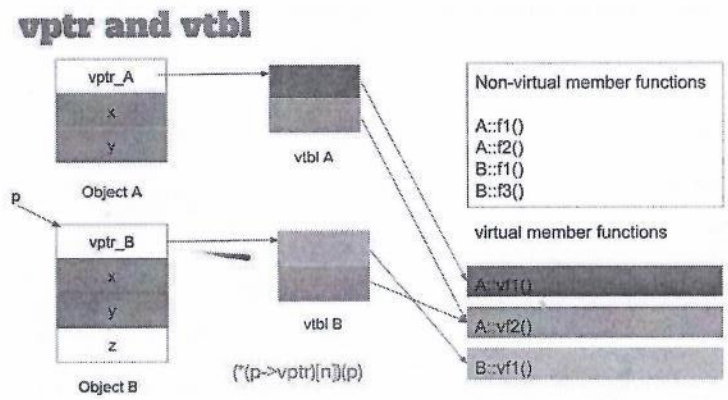
Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table.

This table is simply a static array that the compiler creates at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class.

Each entry in this vTable is simply a Function Pointer that points to the most-derived function accessible by that class ie the most Base Class.

The compiler also adds a hidden pointer to the base class, which we will call *__vPtr.

*__vPtr is set (automatically) when a class instance is created so that it points to the virtual table for that class. *__vPtr is inherited by derived classes.




```
#include<iostream.h>

class Base
{  public:
   virtual void function1() {cout<<"Base :: function1()\n";};
   virtual void function2() {cout<<"Base :: function2()\n";};
   virtual ~Base(){};
};

class D1: public Base
{  public:
   ~D1(){};
   virtual void function1() { cout<<"D1 :: function1()\n";};
};

class D2: public Base
{  public:
   ~D2(){};
   virtual void function2() { cout<< "D2 :: function2\n";};
};

int main()
{
  D1 *d = new D1;;
  Base *b = d;

  b->function1();
  b->function2();

  delete (b);

  return (0);
}
```


Pure Virtual Function:
A virtual function without any executable code
Declared by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.
For example, in class cEmployee
virtual float computeSalary() = 0;
A class containing at least one pure virtual function is termed as abstract class.

Q. What is object slicing?
In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

```
class Base
 {
 int x, y;
 };
class Derived : public Base
 { ssss
int z, w;
};
int main()
{
   Derived d;
   Base b = d; // Object Slicing,  z and w of d are sliced off
}
```

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

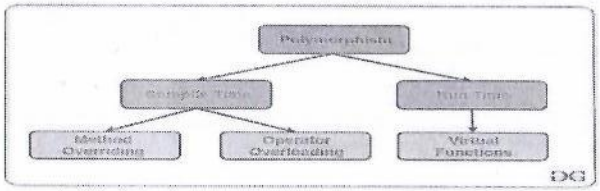
```
#include <iostream>
using namespace std;
class Base
{
protected:
   int i;
public:
   Base(int a)    { i = a; }
   virtual void display()
   { cout << "I am Base class object, i = " << i << endl; }
};
class Derived : public Base
{
   int j;
public:
   Derived(int a, int b) : Base(a) { j = b; }
   virtual void display()
   { cout << "I am Derived class object, i = "
        << i << ", j = " << j << endl;  }
};
void somefunc (Base obj)
{
   obj.display();
}
```

```
int main()
{
   Base b(33);
   Derived d(45, 54);
   somefunc(b);
   somefunc(d);  // Object Slicing, the member j of d is sliced off
   return 0;
}
```

object slicing occurs when an object of a subclass type is copied to an object of superclass type: the superclass copy will not have any of the member variables defined in the subclass. These variables have, in effect, been "sliced off".

Q. What is Polymorphism? How it is supported by C++?

-Polymorphism is same message given to generalize things for same behavior but implemented differently.
-Two types of **polymorphism** are **supported by C++**. They are, Function Overloading and overriding.
-Function overloading is also known as static polymorphism and Overriding is also known as dynamic polymorphism. Two or more method having same name but different argument in same class.



Compile time polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments.**

```
.#include <iostream>

using namespace std;
class Demo
{
        public:

        void func(int x)
        {
                cout << "value of x is " << x << endl;
        }

        void func(double x)
        {
                cout << "value of x is " << x << endl;
        }
```



```
        void func(int x, int y)
        {
                cout << "value of x and y is " << x << ", " << y << endl;
        }
};

int main() {

        Demo obj1;

        obj1.func(7);

        obj1.func(9.132);

        obj1.func(85,64);
        return 0;
}
```

Runtime polymorphism: This type of polymorphism is achieved by Function Overriding.

- **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

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

class base
{
public:
   virtual void print ()
   {
        cout<< "print base class" <<endl;
   }

   void show ()
   {
        cout<< "show base class" <<endl;
   }
};

class derived:public base
{
public:
   void print ()
   {
        cout<< "print deriveed class" <<endl;
   }

   void show ()
   {
         cout<< "show derived class" <<endl;
   }
};
```

```
//main function
int main()
{
   base *bptr;
   derived d;
   bptr = &d;
   bptr->print();
   bptr->show();
  return 0;
}
```

Type of Classes:

Concrete class

A class which describes the functionality of the objects

Abstract class

A class which contains generic or common features that multiple derived classes can share.
-Cannot be instantiated...(can not create object)

Pure abstract class

All the member functions of a class are pure virtual functions.
It is just an interface and cannot be instantiated.(can not create object)

Abstract Class:

An object of an abstract class cannot be created.
However, pointer or reference to abstract class can be created.
Therefore, abstract classes support run-time polymorphism.
Pure virtual functions must be overridden in derived classes; otherwise derived classes are treated as also abstract.

Q. Difference in concrete and polymorphic class

-A concrete class can be instantiated because it provides (or inherits) the implementation for all of its methods.
-Concrete class is ordinary class that contains relevant data members and functions.
-This class can be instantiated

-A **class** that declares or inherits a virtual function is called a **polymorphic class.**
-Polymorphic class contains generalized data members and member functions among which atleast One function is virtual.

Q. Difference in polymorphic and abstract class.

-A **class** that declares or inherits a virtual function is called a **polymorphic class**
-Polymorphic class contains generalized data members and member functions among which atleast One function is virtual.

-An abstract class cannot be instantiated because at least one method has not been implemented.
-Abstract classes are meant to be extended.

- If they provide any implementation detail, it can be reused by all child classes.

-A special case is the *pure abstract class*, which provides no implementation at all.

-These classes do not help with code reuse, which makes them fundamentally different

Q. What is RTTI? How does type id operator works?

Run-time type information (RTTI) is a mechanism that allows the type of an object to be determined during program execution.

```
#include<iostream>
using namespace std;
class B { virtual void fun() {} };
class D: public B { };

int main()
{
   B *b = new D;
   D *d = dynamic_cast<D*>(b);
   if(d != NULL)
      cout << "works";
   else
      cout << "cannot cast B* to D*";
   getchar();
   return 0;
}
```

The typeid **operator** provides a program with the ability to retrieve the *actual* derived type of the object referred to by a pointer or reference.

The **typeid** operator allows the type of an object to be determined at run time.

The result of **typeid** is a const type_info&. The value is a reference to a type_info object that represents either the *type-id* or the type of the *expression*, depending on which form of **typeid** is used. For more information, see type_info Class..

The **typeid** operator doesn't work with managed types (abstract declarators or instances). For information on getting the Type of a specified type, see typeid.

```
#include <iostream>      // std::cout
#include <typeinfo>      // operator typeid
int main() {
 int i;
 int * pi;
 std::cout << "int is: " << typeid(int).name() << '\n';
 std::cout << "  i is: " << typeid(i).name() << '\n';
 std::cout << " pi is: " << typeid(pi).name() << '\n';
 std::cout << "*pi is: " << typeid(*pi).name() << '\n';
 return 0;
}
```

Q.Can we downcast object using dynamic_cast operator? Justify.

A cast is an operator that forces one data type to be converted into another data type. In C++, **dynamic casting** is, primarily, used to *safely* downcast; i.e., cast a base class pointer (or reference) to a derived class pointer (or reference). It can also be used for upcasting; i.e., casting a derived class pointer (or reference) to a base class pointer (or reference).

Dynamic casting checks consistency at runtime; hence, it is slower than static cast.

To use dynamic_cast<new_type>(ptr) the base class should contain at least one virtual function.
RTTI is short for **Run-time Type Identification**. RTTI is to provide a standard way for a program to determine the type of object during runtime.

RTTI allows programs that use pointers or references to base classes to retrieve the actual derived types of the objects to which these pointers or references refer.
RTTI is provided through following operators:

1.The **typeid** operator, which returns the actual type of the object referred to by a pointer (or a reference).

2.The **dynamic_cast** operator, which safely converts from a pointer (or reference) to a base type to a pointer (or reference) to a derived type.

3. The **reinterpret_cast**: this operator is used for converting one pointer to another pointer of any type. It doesn't check/match the type of pointer. The use of this casting is avoided.

Dynamic Cast

- dynamic_cast is an operator that converts safely one type to another type. In the case, the conversation is possible and safe, it returns the address of the object that is converted. Otherwise, it returns nullptr.
- dynamic_cast has the following syntax

```
dynamic_cast<new_type> (object)
```

- If we want to use dynamic cast for downcasting, base class should be polymorphic - it must have at least one virtual function. Modify base class Person by adding a virtual function

```
Employee e1;
Manager* m3 = dynamic_cast<Manager*>(&e1);
```

Q.Explain about const_cast and static_cast –

const_cast-

-It is used to change the constant value of any object or we can say it is used to remove the constant nature of any object.

Syntax

const_cast<type name>(expression)

e.g.
Input:

```
const int x = 50;
const int* y = &x;
cout<<"old value is"<<*y<<"\n";
int* z=const_cast<int *>(y);
*z=100;
cout<<"new value is"<<*y;
```

Output:
old value is 50
new value is 100

- const_cast can be used to pass constant data to another function that does not accept constant data.

```
//main function
int main()
{
  base *bptr;
  derived d;
  bptr = &d;
  bptr->print();
  bptr->show();
 return 0;
}
```

Type of Classes:

Concrete class

A class which describes the functionality of the objects

Abstract class

A class which contains generic or common features that multiple derived classes can share.

-Cannot be instantiated...(can not create object)

Pure abstract class

All the member functions of a class are pure virtual functions.

It is just an interface and cannot be instantiated.(can not create object)

Abstract Class:

An object of an abstract class cannot be created.
However, pointer or reference to abstract class can be created.
Therefore, abstract classes support run-time polymorphism.
Pure virtual functions must be overridden in derived classes; otherwise derived classes are treated as also abstract.

Q. Difference in concrete and polymorphic class

-A concrete class can be instantiated because it provides (or inherits) the implementation for all of its methods.
-Concrete class is ordinary class that contains relevant data members and functions.
-This class can be instantiated

-A **class** that declares or inherits a virtual function is called a **polymorphic class.**
-Polymorphic class contains generalized data members and member functions among which atleast One function is virtual.

Q. Difference in polymorphic and abstract class.

-A **class** that declares or inherits a virtual function is called a **polymorphic class**
-Polymorphic class contains generalized data members and member functions among which atleast One function is virtual.

-An abstract class cannot be instantiated because at least one method has not been implemented.
-Abstract classes are meant to be extended.

- If they provide any implementation detail, it can be reused by all child classes.

-A special case is the *pure abstract class*, which provides no implementation at all.

-These classes do not help with code reuse, which makes them fundamentally different

Q. What is RTTI? How does type id operator works?

Run-time type information (RTTI) is a mechanism that allows the type of an object to be determined during program execution.

```
#include<iostream>
using namespace std;
class B { virtual void fun() {} };
class D: public B { };

int main()
{
  B *b = new D;
  D *d = dynamic_cast<D*>(b);
  if(d != NULL)
    cout << "works";
  else
    cout << "cannot cast B* to D*";
  getchar();
  return 0;
}
```

The typeid **operator** provides a program with the ability to retrieve the *actual* derived type of the object referred to by a pointer or reference.

The **typeid** operator allows the type of an object to be determined at run time.

The result of **typeid** is a const type_info&. The value is a reference to a type_info object that represents either the *type-id* or the type of the *expression*, depending on which form of **typeid** is used. For more information, see type_info Class..

The **typeid** operator doesn't work with managed types (abstract declarators or instances). For information on getting the Type of a specified type, see typeid.

```
#include <iostream>     // std::cout
#include <typeinfo>     // operator typeid
int main() {
  int i;
  int * pi;
  std::cout << "int is: " << typeid(int).name() << '\n';
  std::cout << "  i is: " << typeid(i).name() << '\n';
  std::cout << " pi is: " << typeid(pi).name() << '\n';
  std::cout << "*pi is: " << typeid(*pi).name() << '\n';
 return 0;
}
```

```
#include <iostream>
using namespace std;
int change(int* p2) {
  return (*p2 * 10);
}
int main() {
  const int num = 100;
  const int *p = #
  int *p1 = const_cast <int *>(p);
  cout << change(p1);
  return 0;
}
```

Output-
1000

2.Static_cast

- It is a compile-time cast.
- It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.

Syntax

static_cast <dest_type> (source);

The return value of static_cast will be of dest_type.

e.g.

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

int main()
{
   float f = 3.5;

   // Implicit type case
   // float to int
   int a = f;
   cout << "The Value of a: " << a;

   // using static_cast for float to int
   int b = static_cast<int>(f);
   cout << "\nThe Value of b: " << b;
}
```

Output

```
The Value of a: 3
The Value of b: 3
```


Q. When should static_cast, dynamic_cast, const_cast and reinterpret_cast be used in C++?

const_cast

can be used to remove or add const to a variable. This can be useful if it is necessary to add/remove constness from a variable.

static_cast

This is used for the normal/ordinary type conversion. This is also the cast responsible for implicit type coersion and can also be called explicitly. You should use it in cases like converting float to int, char to int, etc.

dynamic_cast

This cast is used for handling polymorphism. You only need to use it when you're casting to a derived class. This is exclusively to be used in inheritence when you cast from base class to derived class.

reinterpret_cast

This is the trickiest to use. It is used for reinterpreting bit patterns and is extremely low level. It's used primarily for things like turning a raw data bit stream into actual data or storing data in the low bits of an aligned pointer.

Q. How do we create user defined exceptions? Explain with example.

catch() block and inside the try block we **create** an object of MyException class and use the **throw** keyword to explicitly **throw** an **exception** using this object.

This Object is then caught in the catch block where we print out the message by accessing the what() function of this(MyException) class's object.

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

class MyException : public exception
 {
 public:
   char * what () {
     return "C++ Exception";
   }
};

int main() {
  try {
     throw MyException();
  }catch(MyException e) {
     cout << "MyException caught" <<endl;
     cout << e.what() <<endl;
```

```
  } catch(exception e) {
    //Other errors
  }

  return 0;
}
```

Q. What is iterator?

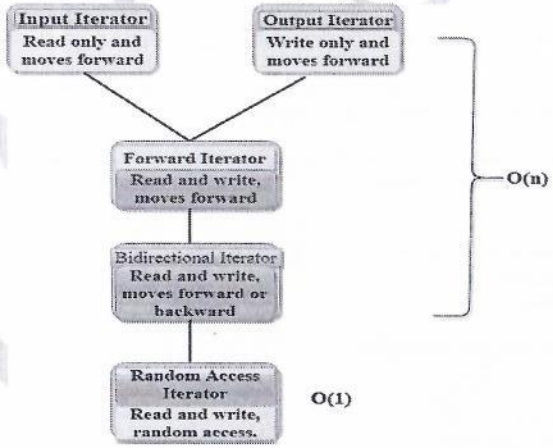
An **iterator** is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container.

They can be visualised as something similar to a pointer pointing to some location and we can access content at that particular location using them.

Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers.

The most obvious form of iterator is a pointer. A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers.

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.



Interfaces in C++ (Abstract Classes)

- An Interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
-The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
-A class is made abstract by declaring at least one of its functions as pure virtual function.

A pure virtual function is specified by placing "= 0" in its declaration as follows –

```
class Box {
  public:
    // pure virtual function
    virtual double getVolume() = 0;

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

-The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. \
-Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.
-Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.
-Classes that can be used to instantiate objects are called concrete classes.\

Abstract Class Example
Consider the following example where parent class provides an interface to the base class to implement a function called getArea() –

```
#include <iostream>

using namespace std;

// Base class
class Shape {
  public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
      width = w;
    }

    void setHeight(int h) {
      height = h;
```

```
    }


  protected:
    int width;
    int height;
};

// Derived classes
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2;
    }
};

int main(void) {
  Rectangle Rect;
  Triangle  Tri;

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

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

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

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

  return 0;
}


//Output
Total Rectangle area: 35
Total Triangle area: 17
```

In above e.g. an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.