**Unit – I / Introduction:**

**Introducing Object Oriented Approach**

**Object**-**Oriented** Programming (OOP) is the term used to describe a programming **approach** based on **objects** and classes. The **object**-**oriented** paradigm allows us to organise software as a collection of **objects** that consist of both data and behaviour.

In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into small modules that combines both data and process. The main aim of Object Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable.

In analysis phase, OO models are used to fill the gap between problem and solution. It performs well in situation where systems are undergoing continuous design, adaption, and maintenance. It identifies the objects in problem domain, classifying them in terms of data and behavior.

***The OO model is beneficial in the following ways −***

* It facilitates changes in the system at low cost.
* It promotes the reuse of components.
* It simplifies the problem of integrating components to configure large system.
* It simplifies the design of distributed systems.

## Elements of Object-Oriented System

**Let us go through the characteristics of OO System −**

* **Objects** − An object is something that is exists within problem domain and can be identified by data (attribute) or behavior. All tangible entities (student, patient) and some intangible entities (bank account) are modeled as object.
* **Attributes** − They describe information about the object.
* **Behavior** − It specifies what the object can do. It defines the operation performed on objects.
* **Class** − A class encapsulates the data and its behavior. Objects with similar meaning and purpose grouped together as class.
* **Methods** − Methods determine the behavior of a class. They are nothing more than an action that an object can perform.
* **Message** − A message is a function or procedure call from one object to another. They are information sent to objects to trigger methods. Essentially, a message is a function or procedure call from one object to another.

## Features of Object-Oriented System

An object-oriented system comes with several great features which are discussed below.

### *Encapsulation*

Encapsulation is a process of information hiding. It is simply the combination of process and data into a single entity. Data of an object is hidden from the rest of the system and available only through the services of the class. It allows improvement or modification of methods used by objects without affecting other parts of a system.

### *Abstraction*

It is a process of taking or selecting necessary method and attributes to specify the object. It focuses on essential characteristics of an object relative to perspective of user.

### *Relationships*

All the classes in the system are related with each other. The objects do not exist in isolation, they exist in relationship with other objects.

***There are three types of object relationships −***

* **Aggregation** − It indicates relationship between a whole and its parts.
* **Association** − In this, two classes are related or connected in some way such as one class works with another to perform a task or one class acts upon other class.
* **Generalization** − The child class is based on parent class. It indicates that two classes are similar but have some differences.

### *Inheritance*

Inheritance is a great feature that allows to create sub-classes from an existing class by inheriting the attributes and/or operations of existing classes.

### *Polymorphism and Dynamic Binding*

Polymorphism is the ability to take on many different forms. It applies to both objects and operations. A polymorphic object is one who true type hides within a super or parent class.

In polymorphic operation, the operation may be carried out differently by different classes of objects. It allows us to manipulate objects of different classes by knowing only their common properties.

## Structured Approach Vs. Object-Oriented Approach

The following table explains how the object-oriented approach differs from the traditional structured approach −

|  |  |
| --- | --- |
| **Structured Approach** | **Object Oriented Approach** |
| It works with Top-down approach. | It works with Bottom-up approach. |
| Program is divided into number of submodules or functions. | Program is organized by having number of classes and objects. |
| Function call is used. | Message passing is used. |
| Software reuse is not possible. | Reusability is possible. |
| Structured design programming usually left until end phases. | Object oriented design programming done concurrently with other phases. |
| Structured Design is more suitable for offshoring. | It is suitable for in-house development. |
| It shows clear transition from design to implementation. | Not so clear transition from design to implementation. |
| It is suitable for real time system, embedded system and projects where objects are not the most useful level of abstraction. | It is suitable for most business applications, game development projects, which are expected to customize or extended. |
| DFD & E-R diagram model the data. | Class diagram, sequence diagram, state chart diagram, and use cases all contribute. |
| In this, projects can be managed easily due to clearly identifiable phases. | In this approach, projects can be difficult to manage due to uncertain transitions between phase. |

**Relating to other paradigms (functional, data decomposition)**

Functional decomposition corresponds to the various functional relationships as how the original complex business function was developed. It mainly focusses on how the overall functionality is developed and its interaction between various components.

Large or complex functionalities are more easily understood when broken down into pieces using functional decomposition.

**Functional decomposition** is applied by **computer** engineers to illustrate the steps to be taken in breaking down the **function** of a system,process, and device into its basic components. Basically, **functional decomposition** helps one to focus and simplify the software **programming** procedure.

The individual functions and sub-functions of a process or system can be graphically displayed in a hierarchical order as a **functional decomposition diagram**, which shows how the various functions are organized.

**Data decomposition** is a highly effective technique for breaking work into small tasks that can be parallelized.

**Abstraction**

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

**Abstraction using Classes:** We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

**Abstraction in Header files:**One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

**Abstraction using access specifiers**

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

* Members declared as **public**in a class, can be accessed from anywhere in the program.
* Members declared as **private**in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

|  |
| --- |
| #include <iostream>  using namespace std;    class implementAbstraction  {      private:          int a, b;        public:            // method to set values of          // private members          void set(int x, int y)          {              a = x;              b = y;          }            void display()          {              cout<<"a = " <<a << endl;              cout<<"b = " << b << endl;          }  };    int main()  {      implementAbstraction obj;      obj.set(10, 20);      obj.display();      return 0;  } |

Output:

a = 10

b = 20

You can see in the above program we are not allowed to access the variables a and b directly, however one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

**Advantages of Data Abstraction**:

* Helps the user to avoid writing the low level code
* Avoids code duplication and increases reusability.
* Can change internal implementation of class independently without affecting the user.
* Helps to increase security of an application or program as only important details are provided to the user.

**Encapsulation**

In normal terms **Encapsulation**is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.  
Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keep records of all the data related to finance. Similarly the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

In C++ encapsulation can be implemented using Class and [access modifiers](https://www.geeksforgeeks.org/access-modifiers-in-c/). Look at the below program:

|  |
| --- |
| #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;  } |

output:

5

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are binded together which is nothing but encapsulation.

**Role of access specifiers in encapsulation**

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

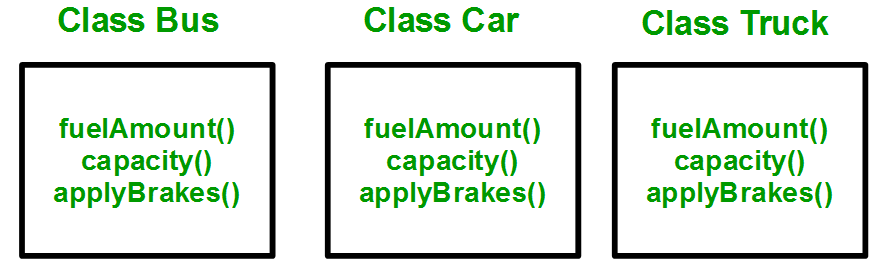
1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public**access specifier

**Inheritance**

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.  
**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.

**Why and when to use inheritance?**

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



|  |
| --- |
| #include <bits/stdc++.h>  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;     } |

Output:

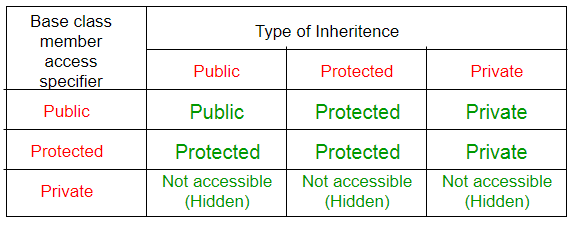
Child id is 7

Parent id is 91

In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

**Modes of Inheritance**

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

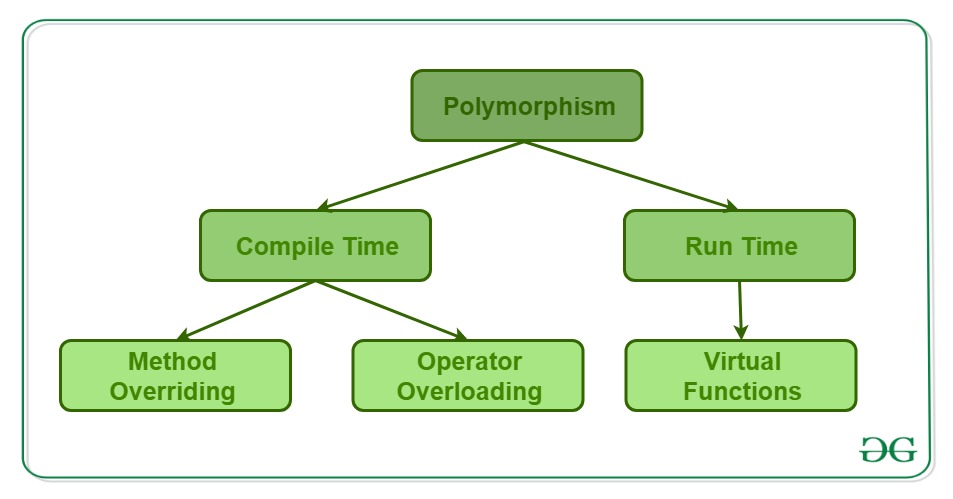
The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:  


**Polymorphism**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.  
Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.  
Polymorphism is considered as one of the important features of Object Oriented Programming.

**In C++ polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism

****

**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**.

***// C++ program for function overloading***

#include <bits/stdc++.h>

using namespace std;

class Geeks

{

    public:

    // function with 1 int parameter

    void func(int x)

    {

        cout << "value of x is " << x << endl;

    }

    // function with same name but 1 double parameter

    void func(double x)

    {

        cout << "value of x is " << x << endl;

    }

    // function with same name and 2 int parameters

    void func(int x, int y)

    {

        cout << "value of x and y is " << x << ", " << y << endl;

    }

};

int main() {

    Geeks obj1;

    // Which function is called will depend on the parameters passed

    // The first 'func' is called

    obj1.func(7);

    // The second 'func' is called

    obj1.func(9.132);

    // The third 'func' is called

    obj1.func(85,64);

    return 0;

}

Output:

value of x is 7

value of x is 9.132

value of x and y is 85, 64

In the above example, a single function named func acts differently in three different situations which is the property of polymorphism.

[**Operator Overloading**](https://www.geeksforgeeks.org/operator-overloading-c/): C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.

***// Operator Overloading***

|  |
| --- |
| #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();  } |

Output:

12 + i9

In the above example the operator ‘+’ is overloaded. The operator ‘+’ is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit [this](https://www.geeksforgeeks.org/operator-overloading-c/) link.

[**Runtime polymorphism**](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/): 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**.

***// C++ program for function overriding***

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 () //print () is already virtual function in derived class, we could also declared as virtual void print () explicitly

    { cout<< "print derived class" <<endl; }

    void show ()

    { cout<< "show derived class" <<endl; }

};

//main function

int main()

{

    base \*bptr;

    derived d;

    bptr = &d;

    //virtual function, binded at runtime (Runtime polymorphism)

    bptr->print();

    // Non-virtual function, binded at compile time

    bptr->show();

    return 0;

}

Output:

print derived class

show base class

**Basic programming of C++**

## Example: Check Prime Number

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int n, i;
6. bool isPrime = true;
7. cout << "Enter a positive integer: ";
8. cin >> n;
9. for(i = 2; i <= n / 2; ++i)
10. {
11. if(n % i == 0)
12. {
13. isPrime = false;
14. break;
15. }
16. }
17. if (isPrime)
18. cout << "This is a prime number";
19. else
20. cout << "This is not a prime number";
21. return 0;
22. }

**Output**

Enter a positive integer: 29

This is a prime number.

## Example: Multiply two matrices without using functions

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;
6. cout << "Enter rows and columns for first matrix: ";
7. cin >> r1 >> c1;
8. cout << "Enter rows and columns for second matrix: ";
9. cin >> r2 >> c2;
10. // If column of first matrix in not equal to row of second matrix,
11. // ask the user to enter the size of matrix again.
12. while (c1!=r2)
13. {
14. cout << "Error! column of first matrix not equal to row of second.";
15. cout << "Enter rows and columns for first matrix: ";
16. cin >> r1 >> c1;
17. cout << "Enter rows and columns for second matrix: ";
18. cin >> r2 >> c2;
19. }
20. // Storing elements of first matrix.
21. cout << endl << "Enter elements of matrix 1:" << endl;
22. for(i = 0; i < r1; ++i)
23. for(j = 0; j < c1; ++j)
24. {
25. cout << "Enter element a" << i + 1 << j + 1 << " : ";
26. cin >> a[i][j];
27. }
28. // Storing elements of second matrix.
29. cout << endl << "Enter elements of matrix 2:" << endl;
30. for(i = 0; i < r2; ++i)
31. for(j = 0; j < c2; ++j)
32. {
33. cout << "Enter element b" << i + 1 << j + 1 << " : ";
34. cin >> b[i][j];
35. }
36. // Initializing elements of matrix mult to 0.
37. for(i = 0; i < r1; ++i)
38. for(j = 0; j < c2; ++j)
39. {
40. mult[i][j]=0;
41. }
42. // Multiplying matrix a and b and storing in array mult.
43. for(i = 0; i < r1; ++i)
44. for(j = 0; j < c2; ++j)
45. for(k = 0; k < c1; ++k)
46. {
47. mult[i][j] += a[i][k] \* b[k][j];
48. }
49. // Displaying the multiplication of two matrix.
50. cout << endl << "Output Matrix: " << endl;
51. for(i = 0; i < r1; ++i)
52. for(j = 0; j < c2; ++j)
53. {
54. cout << " " << mult[i][j];
55. if(j == c2-1)
56. cout << endl;
57. }
58. return 0;
59. }

**Output**

Enter rows and column for first matrix: 3

2

Enter rows and column for second matrix: 3

2

Error! column of first matrix not equal to row of second.

Enter rows and column for first matrix: 2

3

Enter rows and column for second matrix: 3

2

Enter elements of matrix 1:

Enter elements a11: 3

Enter elements a12: -2

Enter elements a13: 5

Enter elements a21: 3

Enter elements a22: 0

Enter elements a23: 4

Enter elements of matrix 2:

Enter elements b11: 2

Enter elements b12: 3

Enter elements b21: -9

Enter elements b22: 0

Enter elements b31: 0

Enter elements b32: 4

Output Matrix:

24 29

6 25

## Example 1: Check Whether Number is Even or Odd using if else

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int n;
6. cout << "Enter an integer: ";
7. cin >> n;
8. if ( n % 2 == 0)
9. cout << n << " is even.";
10. else
11. cout << n << " is odd.";
12. return 0;
13. }
14. **Output**
15. Enter an integer: 23
16. 23 is odd.

## Programs to print triangles using \*, numbers and characters

#### Example 1: Program to print half pyramid using \*

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int rows;
6. cout << "Enter number of rows: ";
7. cin >> rows;
8. for(int i = 1; i <= rows; ++i)
9. {
10. for(int j = 1; j <= i; ++j)
11. {
12. cout << "\* ";
13. }
14. cout << "\n";
15. }
16. return 0;
17. }

#### Example 2: Program to print half pyramid a using numbers

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int rows;
6. cout << "Enter number of rows: ";
7. cin >> rows;
8. for(int i = 1; i <= rows; ++i)
9. {
10. for(int j = 1; j <= i; ++j)
11. {
12. cout << j << " ";
13. }
14. cout << "\n";
15. }
16. return 0;
17. }

#### Example 3: Program to print half pyramid using alphabets

A

B B

C C C

D D D D

E E E E E

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. char input, alphabet = 'A';
6. cout << "Enter the uppercase character you want to print in the last row: ";
7. cin >> input;
8. for(int i = 1; i <= (input-'A'+1); ++i)
9. {
10. for(int j = 1; j <= i; ++j)
11. {
12. cout << alphabet << " ";
13. }
14. ++alphabet;
15. cout << endl;
16. }
17. return 0;
18. }

## Programs to print inverted half pyramid using \* and numbers

#### Example 4: Inverted half pyramid using \*

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int rows;
6. cout << "Enter number of rows: ";
7. cin >> rows;
8. for(int i = rows; i >= 1; --i)
9. {
10. for(int j = 1; j <= i; ++j)
11. {
12. cout << "\* ";
13. }
14. cout << endl;
15. }
17. return 0;
18. }

#### Example 5: Inverted half pyramid using numbers

1 2 3 4 5

1 2 3 4

1 2 3

1 2

1

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int rows;
6. cout << "Enter number of rows: ";
7. cin >> rows;
8. for(int i = rows; i >= 1; --i)
9. {
10. for(int j = 1; j <= i; ++j)
11. {
12. cout << j << " ";
13. }
14. cout << endl;
15. }
16. return 0;
17. }

## Programs to display pyramid and inverted pyramid using \* and digits

#### Example 6: Program to print full pyramid using \*

\*

\* \* \*

\* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \*

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int space, rows;
6. cout <<"Enter number of rows: ";
7. cin >> rows;
8. for(int i = 1, k = 0; i <= rows; ++i, k = 0)
9. {
10. for(space = 1; space <= rows-i; ++space)
11. {
12. cout <<" ";
13. }
14. while(k != 2\*i-1)
15. {
16. cout << "\* ";
17. ++k;
18. }
19. cout << endl;
20. }
21. return 0;
22. }

#### Example 7: Program to print pyramid using numbers

1

2 3 2

3 4 5 4 3

4 5 6 7 6 5 4

5 6 7 8 9 8 7 6 5

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int rows, count = 0, count1 = 0, k = 0;
6. cout << "Enter number of rows: ";
7. cin >> rows;
8. for(int i = 1; i <= rows; ++i)
9. {
10. for(int space = 1; space <= rows-i; ++space)
11. {
12. cout << " ";
13. ++count;
14. }
15. while(k != 2\*i-1)
16. {
17. if (count <= rows-1)
18. {
19. cout << i+k << " ";
20. ++count;
21. }
22. else
23. {
24. ++count1;
25. cout << i+k-2\*count1 << " ";
26. }
27. ++k;
28. }
29. count1 = count = k = 0;
30. cout << endl;
31. }
32. return 0;
33. }

#### Example 8: Inverted full pyramid using \*

\* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \*

\* \* \*

\*

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int rows;
6. cout << "Enter number of rows: ";
7. cin >> rows;
8. for(int i = rows; i >= 1; --i)
9. {
10. for(int space = 0; space < rows-i; ++space)
11. cout << " ";
12. for(int j = i; j <= 2\*i-1; ++j)
13. cout << "\* ";
14. for(int j = 0; j < i-1; ++j)
15. cout << "\* ";
16. cout << endl;
17. }
18. return 0;
19. }

### Example 9: Print Pascal's triangle

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

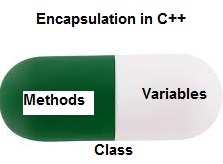
1 5 10 10 5 1

**Source Code**

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int rows, coef = 1;
6. cout << "Enter number of rows: ";
7. cin >> rows;
8. for(int i = 0; i < rows; i++)
9. {
10. for(int space = 1; space <= rows-i; space++)
11. cout <<" ";
12. for(int j = 0; j <= i; j++)
13. {
14. if (j == 0 || i == 0)
15. coef = 1;
16. else
17. coef = coef\*(i-j+1)/j;
18. cout << coef << " ";
19. }
20. cout << endl;
21. }
22. return 0;
23. }

**Unit – II / Classes and Objects:**

**Encapsulation**



Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

In C++ encapsulation can be implemented using Class and [access modifiers](https://www.geeksforgeeks.org/access-modifiers-in-c/). Look at the below program:

|  |
| --- |
| // c++ program to explain  // Encapsulation    #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;  } |

output:

5

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are binded together which is nothing but encapsulation.

**Role of access specifiers in encapsulation**

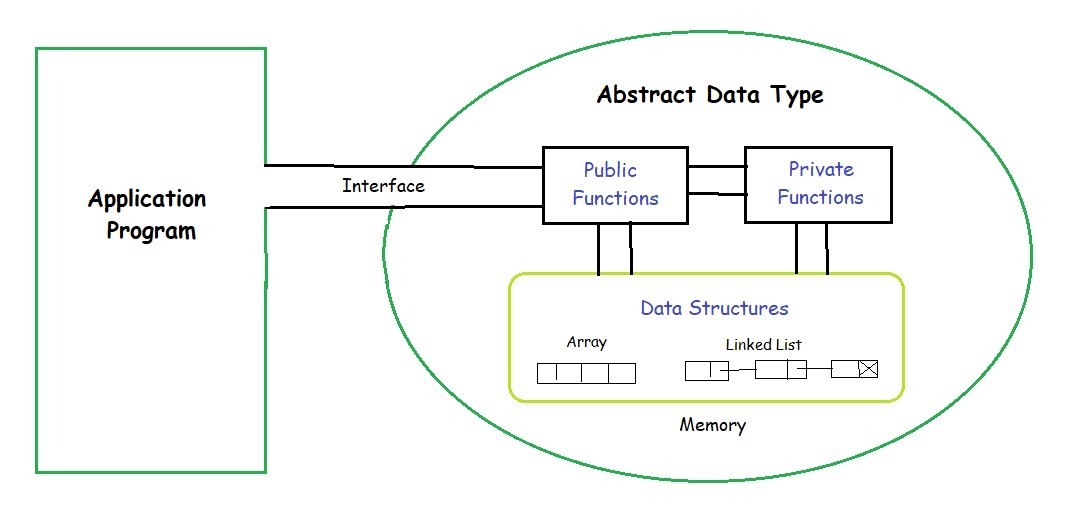
As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public**access specifier

**Abstract Data Types**

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

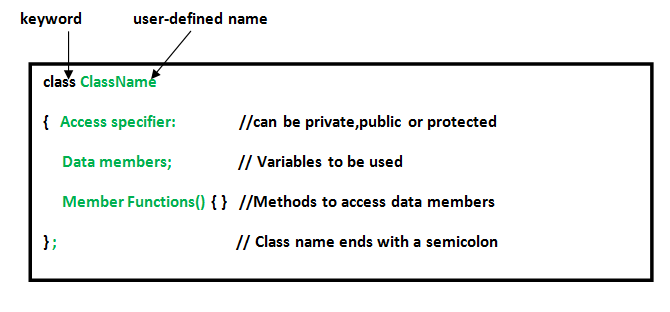
[](https://media.geeksforgeeks.org/wp-content/uploads/20190828194629/ADT.jpg)

**Object & Classes**

**Class:** A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.  
For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

* A Class is a user defined data-type which has data members and member 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.
* In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

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

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

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

**Syntax:**

**ClassName ObjectName;**

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

**Accessing Data Members**

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.  
This access control is given by [Access modifiers in C++](https://www.geeksforgeeks.org/access-modifiers-in-c/). There are three access modifiers : **public, private and protected**.

|  |
| --- |
| // C++ program to demonstrate  // accessing of data members    #include <bits/stdc++.h>  using namespace std;  class Geeks  {      // Access specifier      public:        // Data Members      string geekname;        // Member Functions()      void printname()      {         cout << "Geekname is: " << geekname;      }  };    int main() {        // Declare an object of class geeks      Geeks obj1;        // accessing data member      obj1.geekname = "Abhi";        // accessing member function      obj1.printname();      return 0;  } |

**Output:**

Geekname is: Abhi

**Attributes**

**Attributes** are one of the key features of modern C++ which allows the programmer to **specify additional information to the compiler to enforce constraints(conditions), optimise certain pieces of code or do some specific code generation**. In simple terms, an attribute acts as an annotation or a note to the compiler which provides additional information about the code for optimization purposes and enforcing certain conditions on it. Introduced in C++11, they have remained one of the best features of C++ and are constantly being evolved with each new version of C++.

### Purpose of attributes in C++

* **To enforce constraints on the code:**

Here constraint refers to a condition, that the arguments of a particular function must meet for its execution (precondition). In previous versions of C++, the code for specifying constraints was written in this manner

|  |
| --- |
| int f(int i)  {      if (i > 0)          return i;      else          return -1;        // Code  } |

It increases the readability of your code and avoids the clutter that is written inside the function for argument checking.

|  |
| --- |
| int f(int i)[[expects:i > 0]]  {      // Code  } |

* **To give additional information to the compiler for optimisation purposes:**

Compilers are very good at optimization but compared to humans they still lag at some places and propose generalized code which is not very efficient. This mainly happens due to the lack of additional information about the “problem” which humans have. To reduce this problem to an extent C++ standard has introduced some new attributes that allow specifying a little more to the compiler rather than the code statement itself. Once such example is that of likely.

|  |
| --- |
| int f(int i)  {      switch (i) {      case 1:          [[fallthrough]];          [[likely]] case 2 : return 1;      }      return -1;  } |

When the statement is preceded by likely compiler makes special optimizations with respect to that statement which improves the overall performance of the code. Some examples of such attributes are **[carries\_dependency], [likely], [unlikely]**

* **Suppressing certain warnings and errors that programmer intended to have in his code:**

It happens rarely but sometimes the programmer intentionally tries to write a faulty code which gets detected by the compiler and is reported as an error or a warning. One such example is that of an unused variable which has been left in that state for a specific reason or of a switch statement where the break statements are not put after some cases to give rise to fall-through conditions. In order to circumvent errors and warnings on such conditions, C++ provides attributes such as **[maybe\_unused]** and **[fallthrough]** that prevent the compiler from generating warnings or errors.

|  |
| --- |
| #include <iostream>  #include <string>    int main()  {        // Set debug mode in compiler or 'R'      [[maybe\_unused]] char mg\_brk = 'D';        // Compiler does not emit any warnings      // or error on this unused variable  } |

**Methods**

Methods are **functions** that belongs to the class.

There are two ways to define functions that belongs to a class:

* Inside class definition
* Outside class definition

In the following example, we define a function inside the class, and we name it "myMethod".

**Note:** You access methods just like you access attributes; by creating an object of the class and by using the dot syntax (.):

### *Inside Example*

class MyClass {        // The class  
  public:              // Access specifier  
    void myMethod() {  // Method/function defined inside the class  
      cout << "Hello World!";  
    }  
};  
  
int main() {  
  MyClass myObj;     // Create an object of MyClass  
  myObj.myMethod();  // Call the method  
  return 0;  
}

To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifiying the name of the class, followed the scope resolution :: operator, followed by the name of the function:

### *Outside Example*

class MyClass {        // The class  
  public:              // Access specifier  
    void myMethod();   // Method/function declaration  
};  
  
// Method/function definition outside the class  
void **MyClass::myMethod()** {  
  cout << "Hello World!";  
}  
  
int main() {  
  MyClass myObj;     // Create an object of MyClass  
  myObj.myMethod();  // Call the method  
  return 0;  
}

**C++ class declaration**

##### Class Declaration & Structure in C++

Classes are a blueprint for creating individual objects that contain the general characteristics of a defined object type.

##### Syntax

**class** myClass { *//class header*

**public**:

*//fields, constructors*

*//public method declarations*

**private**:

*//private fields*

*//private method declarations*

};

##### Notes

C++ classes themselves do not have a modifier.

There are two access modifiers within the class, which control the access for its corresponding members. Members with 'public' access are directly accessible through an instance of the object. Members with 'private' access can only be accessed within the class itself.

**Example**

**class** Bike {

**public**:

*//public fields:*

**int** speed;

**int** gear;

*//Constructor:*

Bike(**int** startSpeed, **int** startGear) {

gear = startGear;

speed = startSpeed;

}

*//Methods:*

**void** **setGear** (**int** newValue) {

gear = newValue;

}

**void** **applyBrake**(**int** decrement) {

speed -= decrement;

}

**void** **speedUp**(**int** increment) {

speed += increment;

}

}

**Static Identity and behavior of an Object**

**state** and behaviour are the basic properties of an **Object**. **state** tells us about the type or the value of that **object** where as behaviour tells us about the operations or things that the **object** can perform.

An **object's** attributes are usually static, and the values of the attributes are usually dynamic. The term "**behavior**" refers to how **objects** interact with each other, and it is defined by the operations an **object** can perform. ... It retains its **identity** regardless of changes to its **state** or **behavior**.

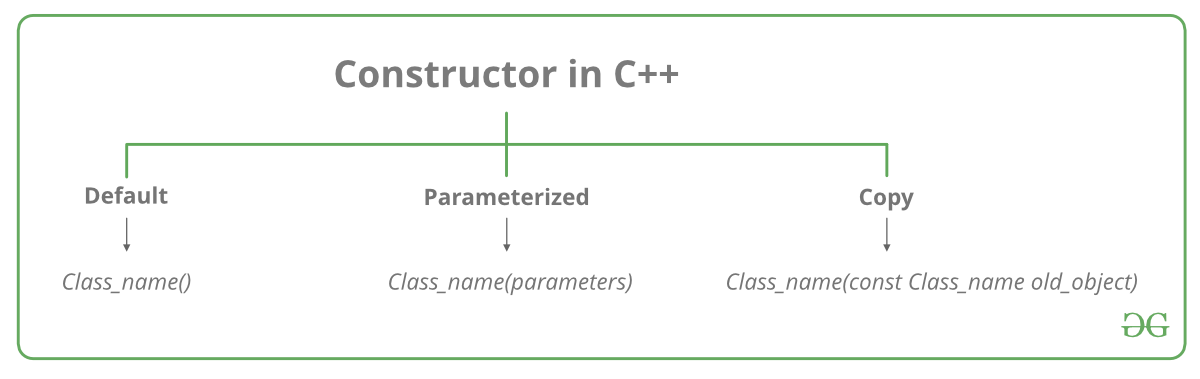
**Constructors and Destructors**

**What is constructor?**

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

A constructor is different from normal functions in following ways:

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



**Types of Constructors**

1. [**Default Constructors:**](https://www.geeksforgeeks.org/c-internals-default-constructors-set-1/) Default constructor is the constructor which doesn’t take any argument. It has no parameters.

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

Output:

a: 10

b: 20

**Note:**Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

1. **Parameterized Constructors:**It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

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

Output:

p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

Example e = Example(0, 50); // Explicit call

Example e(0, 50); // Implicit call

1. **Uses of Parameterized constructor:**
   1. It is used to initialize the various data elements of different objects with different values when they are created.
   2. It is used to overload constructors.

**Can we have more than one constructors in a class?**  
Yes, It is called [Constructor Overloading](https://www.geeksforgeeks.org/constructor-overloading-c/).

**3. Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on [Copy Constructor](https://www.geeksforgeeks.org/copy-constructor-in-cpp/).

Whenever we define one or more non-default constructors( with parameters ) for a class, a default constructor( without parameters ) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it’s considered to be the best practice to always define a default constructor.

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

Output:

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

**What is destructor?**  
Destructor is a member function which destructs or deletes an object.

**When is destructor called?**  
A destructor function is called automatically when the object goes out of scope:  
(1) the function ends  
(2) the program ends  
(3) a block containing local variables ends  
(4) a delete operator is called   
**How destructors are different from a normal member function?**  
Destructors have same name as the class preceded by a tilde (~)  
Destructors don’t take any argument and don’t return anything

class String

{

private:

    char \*s;

    int size;

public:

    String(char \*); // constructor

    ~String();      // destructor

};

String::String(char \*c)

{

    size = strlen(c);

    s = new char[size+1];

    strcpy(s,c);

}

String::~String()

{

    delete []s;

}

**Can there be more than one destructor in a class?**  
No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

**When do we need to write a user-defined destructor?**  
If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

**Can a destructor be virtual?**  
Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function.

**Object Types**

Use runtime type identification (commonly referred to as RTTI) to query the address of the object for the type of object it points to.

#include <iostream>

#include <typeinfo>

using namespace std;

class Base {};

class Derived : public Base {};

int main() {

Base b, bb;

Derived d;

// Use typeid to test type equality

if (typeid(b) == typeid(d)) { // No

cout << "b and d are of the same type.\n";

}

if (typeid(b) == typeid(bb)) { // Yes

cout << "b and bb are of the same type.\n";

}

if (typeid(d) == typeid(Derived)) { // Yes

cout << "d is of type Derived.\n";

}

}

## Discussion

Example shows you how to use the operator typeid to determine and compare the type of an object. typeid takes an expression or a type and returns a reference to an object of type\_info or a subclass of it (which is implementation defined). You can use what is returned to test for equality or retrieve a string representation of the type’s name. For example, you can compare the types of two objects like this:

if (typeid(b) == typeid(d)) {

This will return true if the type\_info objects returned by both of these are equal. This is because typeid returns a reference to a static object, so if you call it on two objects that are the same type, you will get two references to the same thing, which is why the equality test.

**Meta Class**

A **meta class** is a **class** of a **class** i.e. the objects of this **class** can themselves act as **classes**. So a user can add or remove attributes at run time.

In [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming) [programming](https://en.wikipedia.org/wiki/Computer_programming), a **metaclass** is a [class](https://en.wikipedia.org/wiki/Class_(computer_science)) whose instances are classes. Just as an ordinary class defines the behavior of certain objects, a metaclass defines the behavior of certain classes and their instances. Not all object-oriented programming [languages](https://en.wikipedia.org/wiki/Programming_languages) support metaclasses. Among those that do, the extent to which metaclasses can override any given aspect of class behavior varies. Metaclasses can be implemented by having classes be [first-class citizen](https://en.wikipedia.org/wiki/First-class_citizen), in which case a metaclass is simply an object that constructs classes. Each language has its own [metaobject protocol](https://en.wikipedia.org/wiki/Metaobject_protocol" \o "Metaobject protocol), a set of rules that govern how objects, classes, and metaclasses interact.

**Abstract classes**

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

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

**Example**

**class** **AbstractClass** {

**public**:

**virtual** void AbstractMemberFunction() = 0; *// Pure virtual function makes*

*// this class Abstract class.*

**virtual** void NonAbstractMemberFunction1(); *// Virtual function.*

void NonAbstractMemberFunction2();

};

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