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. 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

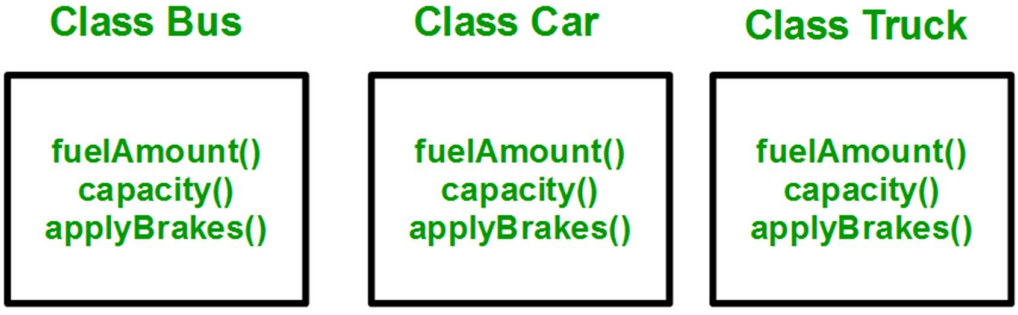
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

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:

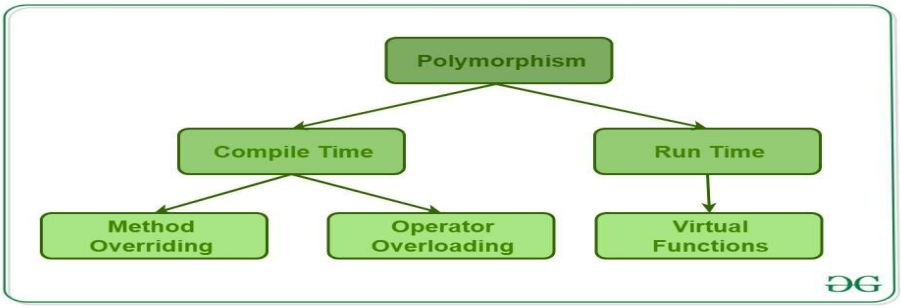
Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

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: 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 link.

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**.

// 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.
4.  int main()
5.  {
6.    int n, i;
7.    bool isPrime = true;
8.
9.    cout << "Enter a positive integer: ";
10.   cin >> n;
11.
12.   for(i = 2; i <= n / 2; ++i)
13.   {
14.      if(n % i == 0)
```

```
    {
      isPrime = false;
      break;
    }
  }
  if (isPrime)
    cout << "This is a prime number";
  else
    cout << "This is not a prime number";

  return 0;
}
```

Output

```
Enter a positive integer: 29
This is a prime number.
```

Example: Multiply two matrices without using functions

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

int main()
{
   int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;

   cout << "Enter rows and columns for first matrix: ";
   cin >> r1 >> c1;
   cout << "Enter rows and columns for second matrix: ";
   cin >> r2 >> c2;

   // If column of first matrix in not equal to row of second matrix,
   // ask the user to enter the size of matrix again.
   while (c1!=r2)
   {
      cout << "Error! column of first matrix not equal to row of second.";

      cout << "Enter rows and columns for first matrix: ";
      cin >> r1 >> c1;

      cout << "Enter rows and columns for second matrix: ";
      cin >> r2 >> c2;
   }

   // Storing elements of first matrix.
   cout << endl << "Enter elements of matrix 1:" << endl;
   for(i = 0; i < r1; ++i)
      for(j = 0; j < c1; ++j)
      {
         cout << "Enter element a" << i + 1 << j + 1 << " : ";
```

```
            cin >> a[i][j];
        }

    // Storing elements of second matrix.
    cout << endl << "Enter elements of matrix 2:" << endl;
    for(i = 0; i < r2; ++i)
        for(j = 0; j < c2; ++j)
        {
            cout << "Enter element b" << i + 1 << j + 1 << " : ";
            cin >> b[i][j];
        }

    // Initializing elements of matrix mult to 0.
    for(i = 0; i < r1; ++i)
        for(j = 0; j < c2; ++j)
        {
            mult[i][j]=0;
        }

    // Multiplying matrix a and b and storing in array mult.
    for(i = 0; i < r1; ++i)
        for(j = 0; j < c2; ++j)
            for(k = 0; k < c1; ++k)
            {
                mult[i][j] += a[i][k] * b[k][j];
            }

    // Displaying the multiplication of two matrix.
    cout << endl << "Output Matrix: " << endl;
    for(i = 0; i < r1; ++i)
    for(j = 0; j < c2; ++j)
    {
        cout << " " << mult[i][j];
        if(j == c2-1)
            cout << endl;
    }

    return 0;
}
```

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.
4.  int main()
5.  {
6.     int n;
7.
8.     cout << "Enter an integer: ";
9.     cin >> n;
10.
11.    if ( n % 2 == 0)
12.        cout << n << " is even.";
13.    else
14.        cout << n << " is odd.";
15.
16.    return 0;
17. }
```

18. **Output**

```
19. Enter an integer: 23
20. 23 is odd.
```

Programs to print triangles using *, numbers and characters

Example 1: Program to print half pyramid using *

```
*
* *
* * *
* * * *
* * * * *
```

Source Code

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = 1; i <= rows; ++i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << "* ";
        }
        cout << "\n";
    }
    return 0;
}
```

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

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = 1; i <= rows; ++i)
```

```
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << j << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

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

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

int main()
{
    char input, alphabet = 'A';

    cout << "Enter the uppercase character you want to print in the last row: ";
    cin >> input;

    for(int i = 1; i <= (input-'A'+1); ++i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << alphabet << " ";
        }
        ++alphabet;

        cout << endl;
    }
    return 0;
}
```

Programs to print inverted half pyramid using * and numbers

Example 4: Inverted half pyramid using *

```
* * * * *
* * * *
* * *
* *
```

```
*
```

Source Code

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = rows; i >= 1; --i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << "* ";
        }
        cout << endl;
    }

    return 0;
}
```

Example 5: Inverted half pyramid using numbers

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

Source Code

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = rows; i >= 1; --i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << j << " ";
```

```
16.         }
17.         cout << endl;
18.     }
19.
20.     return 0;
21. }
```

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.
4.  int main()
5.  {
6.     int space, rows;
7.
8.     cout <<"Enter number of rows: ";
9.     cin >> rows;
10.
11.    for(int i = 1, k = 0; i <= rows; ++i, k = 0)
12.    {
13.        for(space = 1; space <= rows-i; ++space)
14.        {
15.            cout <<"  ";
16.        }
17.
18.        while(k != 2*i-1)
19.        {
20.            cout << "* ";
21.            ++k;
22.        }
23.        cout << endl;
24.    }
25.    return 0;
26. }
```

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

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

int main()
{
    int rows, count = 0, count1 = 0, k = 0;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = 1; i <= rows; ++i)
    {
        for(int space = 1; space <= rows-i; ++space)
        {
            cout << "  ";
            ++count;
        }

        while(k != 2*i-1)
        {
            if (count <= rows-1)
            {
                cout << i+k << " ";
                ++count;
            }
            else
            {
                ++count1;
                cout << i+k-2*count1 << " ";
            }
            ++k;
        }
        count1 = count = k = 0;

        cout << endl;
    }
    return 0;
}
```

Example 8: Inverted full pyramid using *

```
* * * * * * * * *
 * * * * * * *
  * * * * *
   * * *
```

```
    *
```

Source Code

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = rows; i >= 1; --i)
    {
        for(int space = 0; space < rows-i; ++space)
            cout << "  ";

        for(int j = i; j <= 2*i-1; ++j)
            cout << "* ";

        for(int j = 0; j < i-1; ++j)
            cout << "* ";

        cout << endl;
    }

    return 0;
}
```

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

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

int main()
{
    int rows, coef = 1;

    cout << "Enter number of rows: ";
    cin >> rows;
```

```
10.
11.     for(int i = 0; i < rows; i++)
12.     {
13.         for(int space = 1; space <= rows-i; space++)
14.             cout <<"  ";
15.
16.         for(int j = 0; j <= i; j++)
17.         {
18.             if (j == 0 || i == 0)
19.                 coef = 1;
20.             else
21.                 coef = coef*(i-j+1)/j;
22.
23.             cout << coef << "    ";
24.         }
25.         cout << endl;
26.     }
27.
28.     return 0;
29. }
```