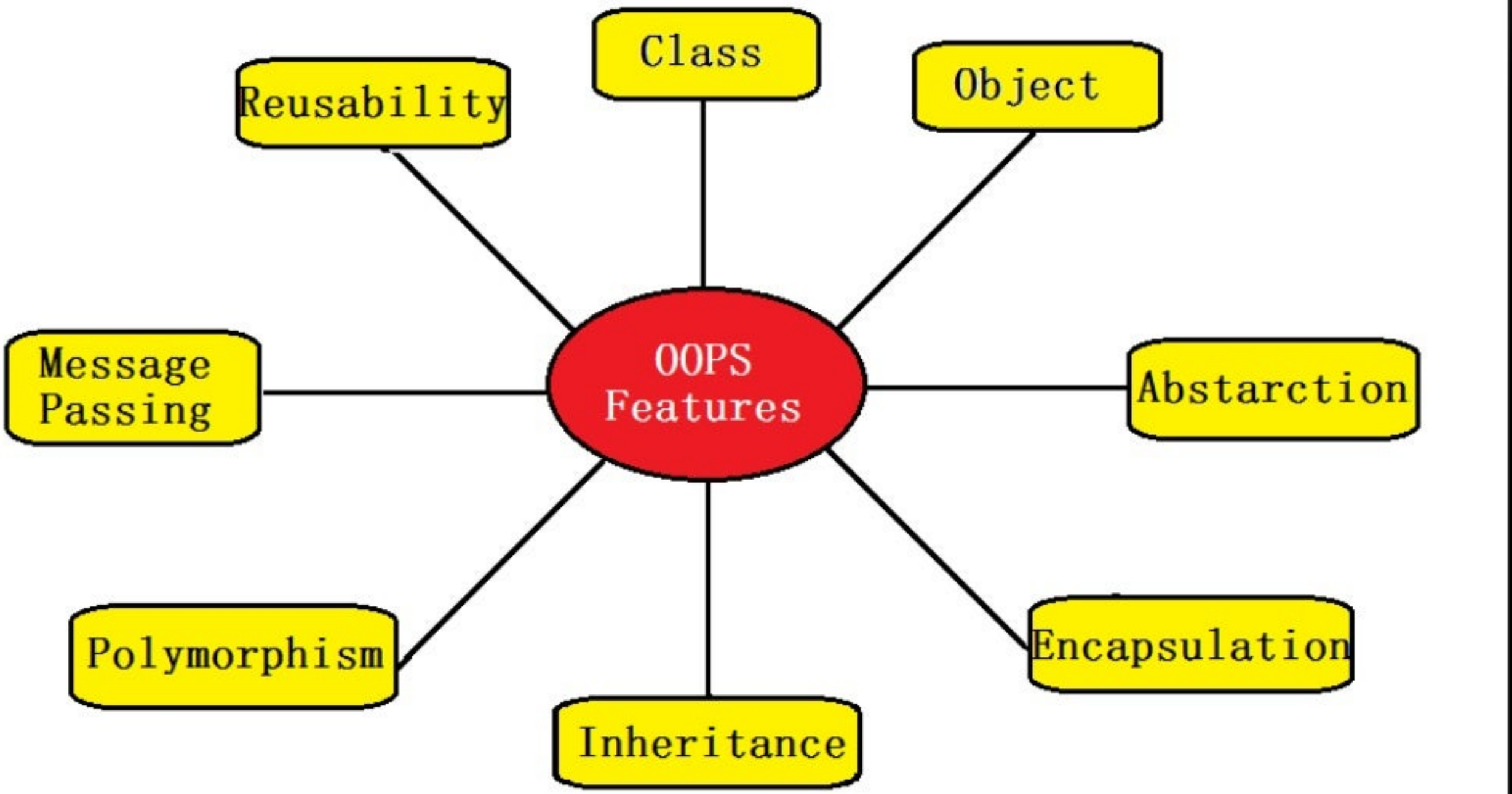
INTRODUCTION
TO
OOPs

FEATURES OF OBJECT ORIENTED PROGRAMMING

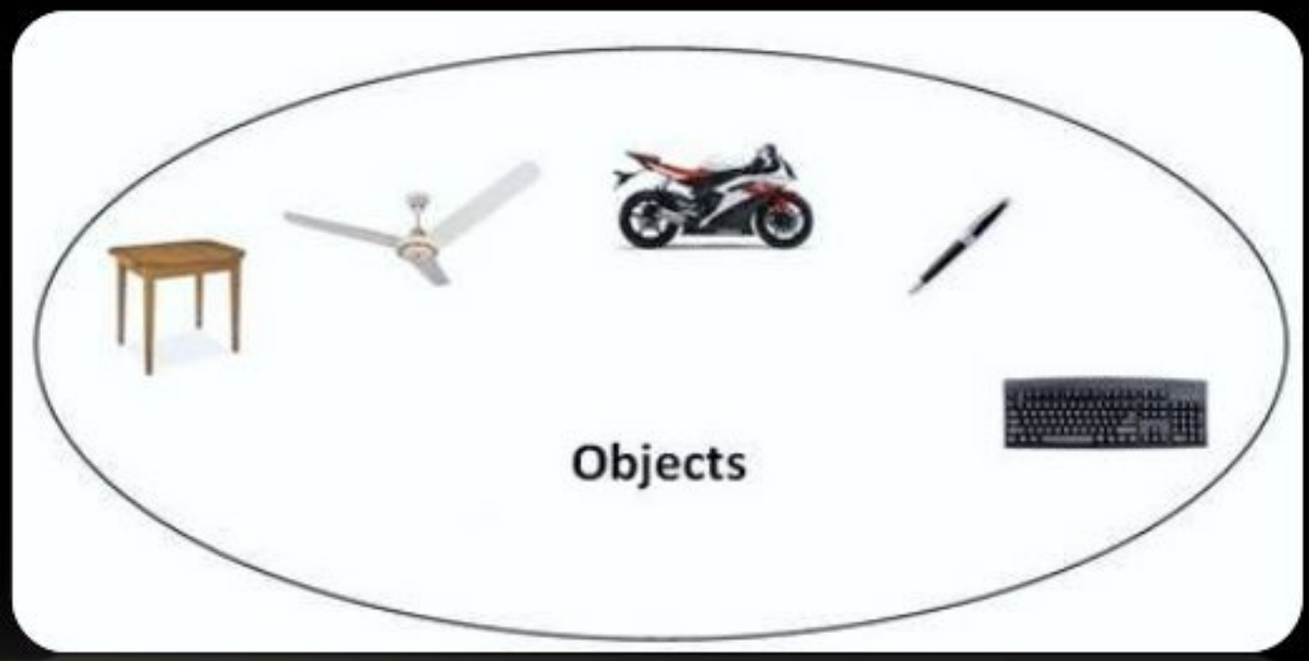
- Objects
- Classes
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Overloading
- Exception Handling
- Constructor & Destructor

OBJECT ORIENTED PROGRAMMING



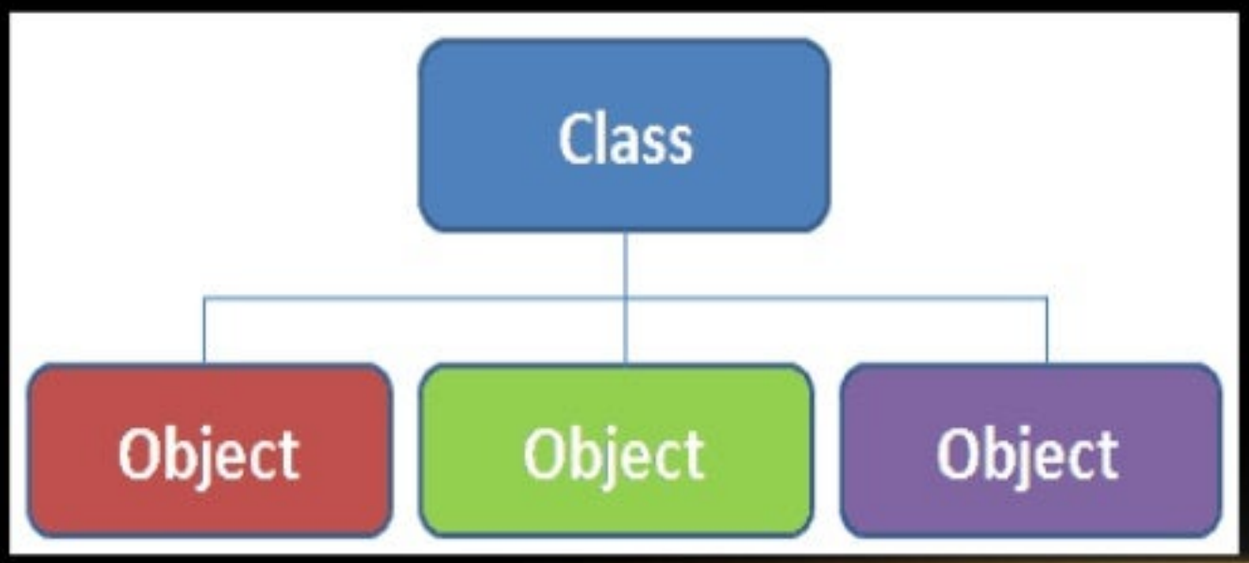
OBJECTS

- Objects are basic unit of OOP.
- They are instance of a class.
- Consists of various data members and member functions.
- These data types and member functions are bundled together as a unit is called objects.



CLASSES

- It is similar to Structure in C.
- Class is user defined data type.
- It holds own data members and member functions.
- Class can be accessed and used only by instance of that class. It is basically blueprint for object.



```
// Header Files
#include <iostream.h>
#include<conio.h>
// Class Declaration
class person
{
//Access - Specifier
public:
// Variable Declaration
  string name;
  int number;
};
//Main Function
int main()
{
// Object Creation For Class
  person obj;
//Get Input Values For Object Variables
  cout<<"Enter the Name :";
  cin>>obj.name;
  cout<<"Enter the Number :";
  cin>>obj.number;
//Show the Output
  cout << obj.name << ": " << obj.number <<
                                        endl;

  getch();
  return 0;
}
```

ACCESS SPECIFIERS

- **Private:** Private member defines that the members or methods can be accessed within the same class only.
- **Public:** Public member defines that the variable or methods can be accessed at any where within the project.
- **Protected:** Protected member can be accessed to the class which is inherited by other class.
- By default, all members and function of a class is private i.e if no access specifier is specified.

SYNTAX OF DECLARING ACCESS MODIFIERS IN C++

```
class
{
    private:
        // private members and function
    public:
        // public members and function
    protected:
        // protected members and function
};
```

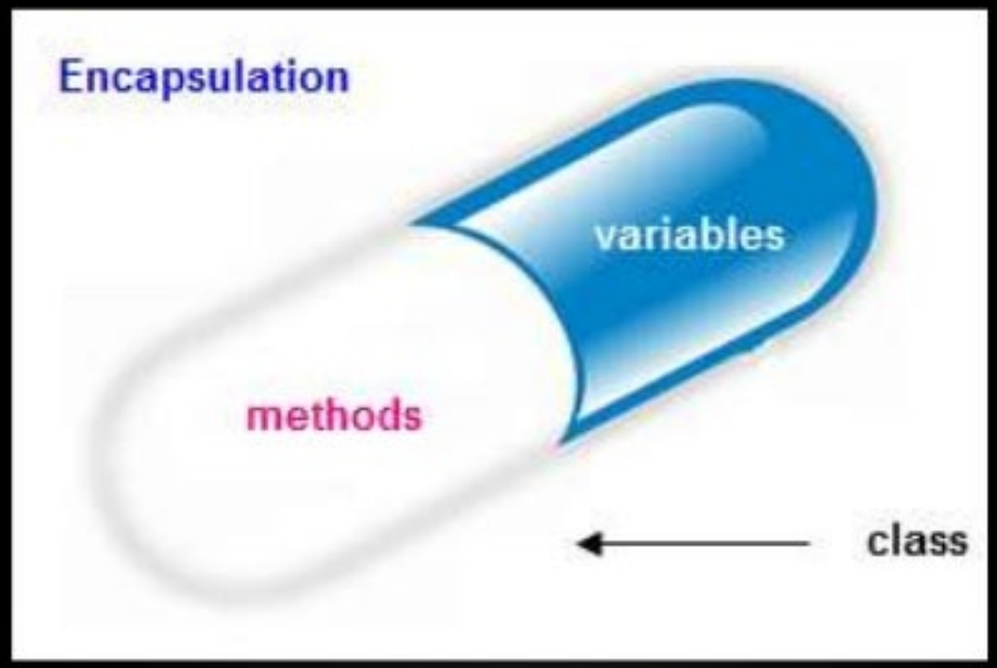
ABSTRACTION

- A model complex of a system that includes only the details essential to perspective of the viewer of the system.
- An Abstraction is a model of a complex system that includes only the essential details.
- Abstractions are the fundamental way that we manage complexity. Helps to manage complexity of a large system.
- Support our quality goals of modifiability and reusability.

```
#include<iostream.h>
 #include<conio.h>
class sum
{
// hidden data from outside
world
private:
int a,b,c;
public:
void add()
{
 cout<<"Enter any two numbers: ";
cin>>a>>b; c=a+b;
cout<<"Sum: "<<c;
}
 };
void main()
 {
sum s;
s.add();
 getch();
 }
```

ENCAPSULATION

- It can be said **Data binding.**
- **Encapsulation** is the method of combining the data and functions inside a class. This hides the data from being accessed from outside a class directly, only through the functions inside the class is able to access the information.



```
#include <iostream.h> class Add
{
private:
int x,y,r;
public:
int Addition(int x, int y)
{
r= x+y;
return r;
}
void show( )
{ cout << "The sum is::" << r << "\n";}
}s;
void main()
{
Add s;
s.Addition(10, 4);
s.show();
}
```

integer values "x,y,r" of the class "Add" can be accessed only through the function "Addition". These integer values are encapsulated inside the class "Add".

➡ **ENCAPSULATION**

OUTPUT

The sum is:: 14

INHERITANCE

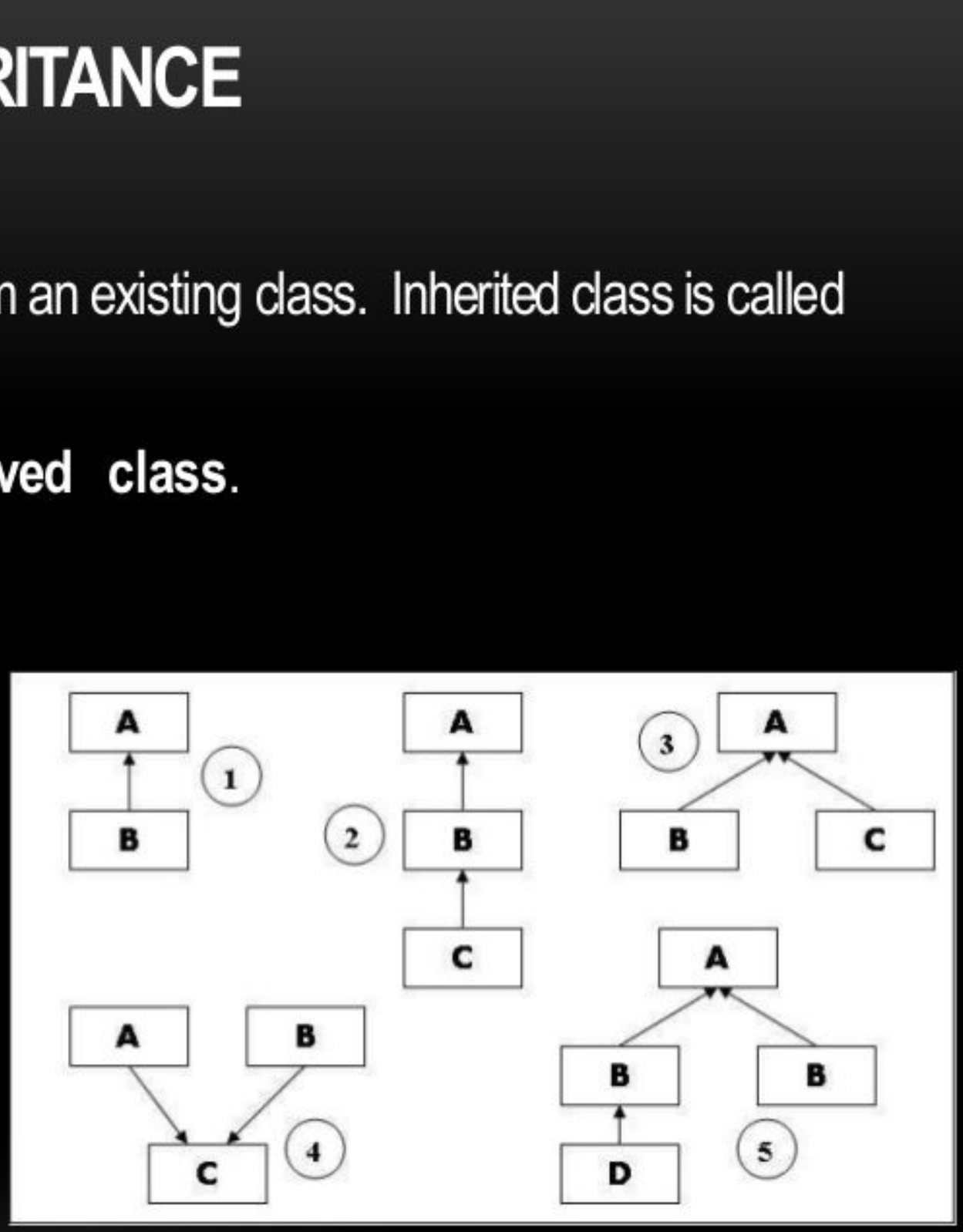
- Code reusability.
- Process of forming new class from an existing class. Inherited class is called **Base class**.
- class which inherits is called **Derived class**.

PROS

- Helps to reduce code size.

TYPES OF INHERITANCE IN C++

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



SINGLE INHERITANCE

```
#include<iostream>
using namespace std;
class single_base
{
protected:
int a,b;
public:
void get()
{
cout<<"Enter a & b";
cin>>a>>b;
} };
```

```
class single_derived :
public single_base
{
int c;
public:
void output()
{
c=a+b;
cout<<"Sum:"<<c;
}
};
```

```
int main()
{
single_derived obj;
obj.get();
obj.output();
return 0;
}
```

OUTPUT

```
Enter a & b   5 5
Sum:10Press any key to continue . . .
```

MULTILEVEL INHERITANCE

```
#include <iostream>
 using namespace std;
class A
{
 public:
 void display()
 { cout<<"Base class content."; }
 };
 class B : public A
{
};
 class C : public B
{
 };
int main()
{
 C obj;
 obj.display();
return 0;
}
```

HIERARCHICAL INHERITANCE

```
#include<iostream>
using namespace std;
class father
// Base class derivation
{
  int age;
  char name [20];
   public:
   void get()
   {
     cout << "\nEnter father's
name:";
     cin >> name;
     cout << "Enter father's age:";
     cin >> age;
   }
   void show()
   {
     cout << "\n\nFather's name is
" << name;
     cout << "\nFather's age is "
<< age;
   }
};
```

```
class son : public father
// First derived class derived
from father class
{
  int age;
  char name [20];
public:
  void get()
  {
    father :: get();
    cout << "Enter son's name:";
    cin >> name;
    cout << "Enter son's age:";
    cin >> age;
  }
  void show()
  {
    father::show();
        cout << "\nSon's name is " <<
name;
        cout << "\nSon's age is " <<
age;
     }
};
```

```
class daughter : public father
// Second derived class derived
from the father class
{
  int age;
  char name [20];
 public:
   void get()
   {
     father :: get();
     cout << "Enter daughter's
name:";
     cin >> name;
     cout << "Enter daughter's
age:";
     cin >> age;
   }
   void show()
   {
     father::show();
     cout << "\nDaughter's name is "
<< name;
     cout << "\nDaughter's age is "
<< age;
   }
};
```

```
int main ()
{
  son s1;
  daughter d1;
  s1.get();
  d1.get();
  s1.show();
  d1.show();
  return 0;
}
```

OUTPUT

```
Enter father's name:AjithKumar
Enter father's age:45
Enter son's name:Aadvik
Enter son's age:3

Enter father's name:AjithKumar
Enter father's age:45
Enter daughter's name:Anoushka
Enter daughter's age:10


Father's name is AjithKumar
Father's age is 45
Son's name is Aadvik
Son's age is 3

Father's name is AjithKumar
Father's age is 45
Daughter's name is Anoushka
Daughter's age is 10Press any key to continue . . .
```

MULTIPLE INHERITANCE

```
#include<iostream>
using namespace std;
class base1
{       public:
        int a;
        void firstdata()
        {
        cout<<"Enter a:";
                cin>>a;
        }
};
class base2
{       public:
        int b;
        void seconddata()
        {
        cout<<"Enter b:";
        cin>>b;
                }
        };
class deriverd : public base1,public base2
        {       public:
                int c;
                void addition()
                {
                        c=a+b;
                cout<<"Output"<<c;
                }
};
int main()
{
deriverd obj;
obj.firstdata();
obj.seconddata();
obj.addition();
return 0;
}
```

output

```
Enter a:5
Enter b:5
Output10Press any key to continue . . .
```

HYBRID INHERITANCE

```
#include<iostream.h>
#include<conio.h>
class arithmetic
{
protected:
int num1, num2;
public:
void getdata()
{
cout<<"For Addition:";
cout<<"\n Enter the first number: ";
cin>>num1;
cout<<"\n Enter the second
number: ";
cin>>num2;
}
};
class plus: public arithmetic
{
protected:
int sum;
public:
void add()
{
sum=num1+num2;
}
};
class minus
{
protected:
int n1,n2,diff;
public:
void sub()
{
cout<<"\n For Subtraction:";
cout<<"\n Enter the first number: ";
cin>>n1;
cout<<"\n Enter the second
number: ";
cin>>n2;
diff=n1-n2;
}
};
class result:public plus, public
minus
{
public:
void display()
{
cout<<"\n Sum of "<<num1<<" and
"<<num2<<"= "<<sum;
cout<<"\n Difference of "<<n1<<"
and "<<n2<<"= "<<diff;
}
};
 void main()
{
result z;
z.getdata();
z.add();
z.sub();
z.display();
getch();
}
```

OUTPUT

```
For Addition:
Enter the first number: 5

Enter the second number: 5

For Subtraction:
Enter the first number: 6

Enter the second number: 5

Sum of 5 and 5= 10
Difference of 6 and 5= 1
```

POLYMORPHISM

- It makes the code More readable.
- Function with same name but different arguments. Functioning is different.
- Poly refers to many.

Types of Polymorphism

- Compile time Polymorphism.
- Run time polymorphism.

REAL LIFE EXAMPLE OF POLYMORPHISM IN C++

In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son

COMPILE TIME POLYMORPHISM

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

- **Method overloading**
- **Method overriding**

DIFFERENT WAYS TO OVERLOAD THE METHOD

There are two ways to overload the method in C++

- **By changing number of arguments or parameters**
- **By changing the data type**

METHOD OVERLOADING IN C++ BY CHANGING NUMBER OF ARGUMENTS

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

OUTPUT

```
30
60
Press any key to continue . . .
```

METHOD OVERLOADING IN C++ BY CHANGING THE DATA TYPE

```
#include<iostream.h>
#include<conio.h>
class Addition
{
public:
void sum(int a, int b)
{
cout<<a+b;
}
void sum(float a, float b,float c)
{
cout<<a+b+c;
}
};
void main()
{
Addition obj;
obj.sum(10, 20);
cout<<endl;
obj.sum(10.5,20.5,30.0);
}
```

OUTPUT

```
30
61
Press any key to continue . . .
```

METHOD OVERRIDING IN C++

```
#include<iostream.h>
#include<conio.h>
class Base
{
 public:
 void show()
 {
  cout<<"Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout<<"Derived Class";
 }
};
void main()
{
 Base b;       //Base class object
 Derived d;  //Derived class object
 b.show();    //Early Binding Occurs
 d.show();
getch();
}
```

OUTPUT

```
Base class
Derived Class
```

RUN TIME POLYMORPHISM(VIRTUAL FUNCTION)

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
 virtual void show()
 {
  cout<<"Hello base class";
 }
};
class B : public A
{
 public:
  void show()
  {
   cout<<"Hello derive class";
  }
};
void main()
{
A obj1;
B obj2;
obj1.show();
obj2.show();
getch();
}
```

VIRTUAL FUNCTION

OUTPUT

```
Hello base class
Hello derive class
```

FRIEND FUNCTION IN C++

```
#include<iostream.h>
#include<conio.h>
class employee
{
private:
      friend void sal();
};
void sal()
{
int salary=4000;
cout<<"Salary: "<<salary;
}
void main()
{
employee e;
sal();
getch();
}
```

OUTPUT

```
Salary: 4000
```

CONSTRUCTOR IN C++

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

FEATURES OF CONSTRUCTOR

- **The same name as the class itself.**
- **no return type.**

SYNTAX

```
classname()
{
 ....
}
```

Why use constructor ?

- The main use of constructor is placing user defined values in place of default values.

How Constructor eliminate default values ?

- Constructor are mainly used for eliminate default values by user defined values, whenever we create an object of any class then its allocate memory for all the data members and initialize there default values. To eliminate these default values by user defined values we use constructor.

```
#include<iostream.h>
#include<conio.h>
class sum
{
int a,b,c;
sum()
{
a=10;
b=20;
c=a+b;
cout<<"Sum: "<<c;
}
};
void main()
{
sum s;
getch();
}
```

CONSTRUCTOR

OUTPUT

Sum: 30

DESTRUCTOR IN C++

- **Destructor** is a member function which deletes an object. A destructor function is called automatically when the object goes out of scope:

When destructor call

- when program ends
- when a block containing temporary variables ends
- when a delete operator is called

Features of destructor

- The same name as the class but is preceded by a tilde (~)
- no arguments and return no values

Syntax

~classname() { ..... }

```
#include<iostream.h>
#include<conio.h>
class sum
{
int a,b,c;
sum()
{
a=10;
b=20;
c=a+b;
cout<<"Sum: "<<c;
}
~sum()
{
cout<<endl<<"call destructor";
}
delay(500);
};
void main()
{
sum s;
cout<<endl<<"call main";
getch();
}
```

OUTPUT

```
Sum: 30
call main
call destructor
```

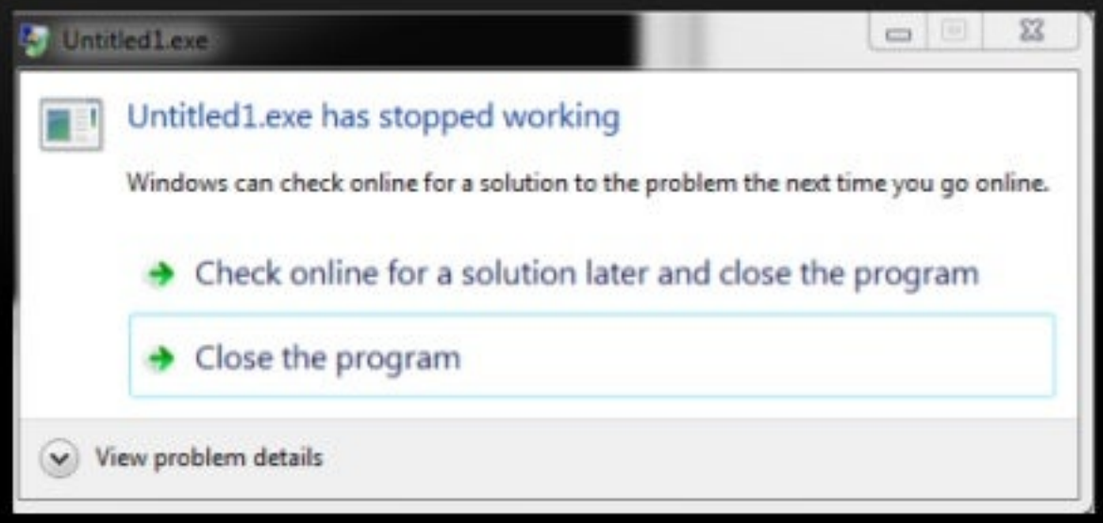
EXCEPTION HANDLING

- Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch, and throw**.
- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

```
Try
 {
 // protected code
}
catch( ExceptionName e1 )
 {
 // catch block
}
catch( ExceptionName e2 )
 {
 // catch block }
catch( ExceptionName eN )
 {
 // catch block
}
```

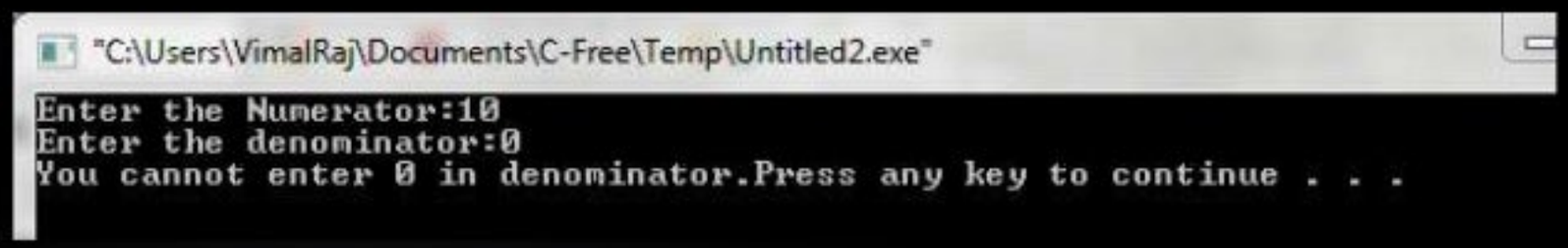
WITHOUT EXCEPTION

```
#include <iostream>
#include <string>
using namespace std;
int main() {
        int numerator, denominator, result;
        cout <<"Enter the Numerator:";
        cin>>numerator;
        cout<<"Enter the denominator:";
        cin>>denominator;
        result = numerator/denominator;
        cout<<"\nThe result of division is:" <<result;
}
```



WITH EXCEPTION

```
#include <iostream>
#include <string>
using namespace std;
int main() {
int numerator, denominator, result;
cout <<"Enter the Numerator:";
cin>>numerator;
cout<<"Enter the denominator:";
cin>>denominator;
try {
if(denominator == 0)
 {
throw denominator;
        }
result = numerator/denominator;
cout<<"\nThe result of division is:" <<result;
}
catch(int num) {
cout<<"You cannot enter "<<num<<" in
denominator.";
}
}
```



THIS POINTER

```
#include <iostream>
#include <conio.h>
using namespace std;
class sample
 {
 int a, b;
public:
void input(int a, int b)
 {
 this->a=a+b;
 this->b=a-b;
}
void output()
 {
 cout<<"a = "<<a<<endl<<"b = "<<b;
}
};
 int main()
{
 sample x;
x.input(5,8);
 x.output();
 getch();
return 0;
}
```

INLINE FUNCTION

```
#include<iostream.h>
#include<conio.h>
int a,b;
inline int sum(int a,int b)
{
return(a+b);
}
int main()
{
int x,y;
clrscr();
cout<<"two num";
cin>>x>>y;
cout<<"sum of two
num"<<sum(x,y);
getch();
return 0;
}
```

Inline Function

TEMPLATES

- In c++ programming allows functions or class to work on more than one data type at once without writing different codes for different data types.
- It is often uses in larger programs for the purpose of of code reusability and flexibility of program.

It can used in two ways:

1. **function Template**
2. **Class Template**

FUNCTION TEMPLATE

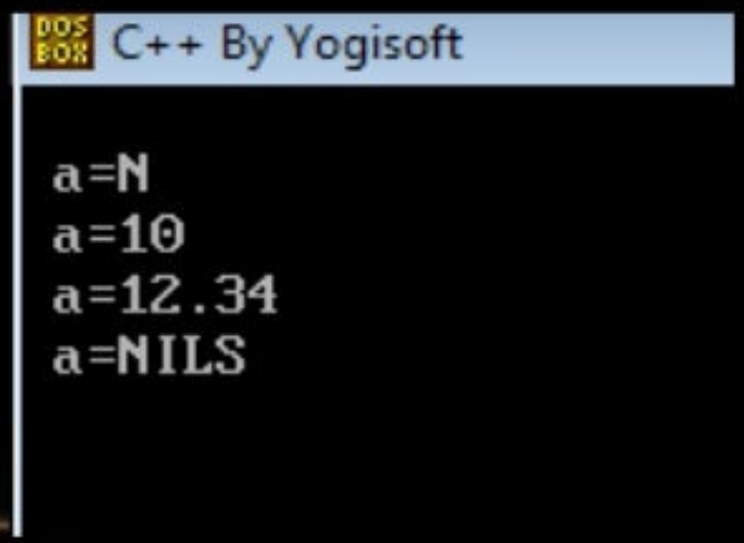
A single function template can work on different types at once but, different functions are needed to perform identical task on different data types.

Syntax:

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

```
#include<iostream.h>
#include<conio.h>
template<class T>
        void show(T a)
         {
                cout<<"\n a="<<a;
         }
void main()
 {
        clrscr();
        show('n');
        show(12.34);
        show(10);
        show("nils");
        getch();
 }
```

Output:



CLASS TEMPLATE

The general form of class templates is shown here:

Syntax:

```
template<class type>class class-name
        {
       // body  of  function
          }
```

Output:


```
#include<iostream.h>
 #include<conio.h>
template<class T>
 class a
  {
      private:
           T x;
  public:
      a()
      {}
      a(T a)
          {
              x=a;
                  }
Void show()
{
Cout<<"\n x = "<<x;  }
};
        Void  main()
         {
              clrscr();
               a<char>a1('B');
                a1.show();
                a<int>a2(10);
               a2.show();
               a<float> a3(12.34f);
               a3.show();
                a<double>a4(10.5);
               A4.show();
                getch();
                }
```