Lecture 2

Basic Concepts of OOPs

Basic Concepts of OOPs with C++

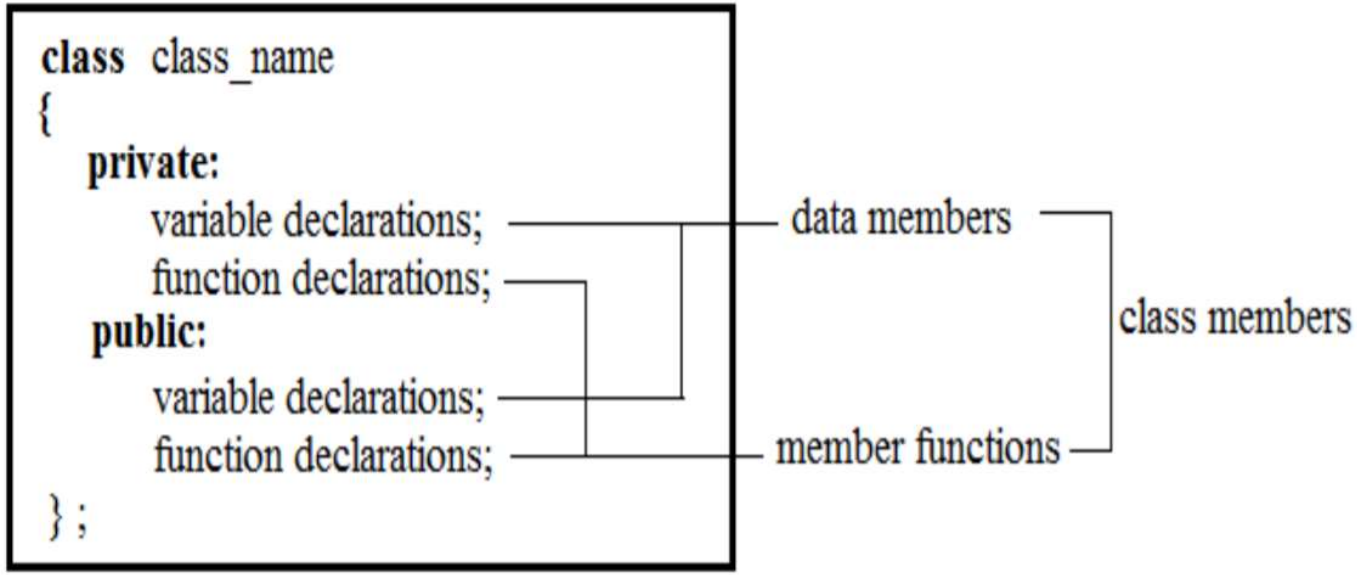
- Object Oriented Programming Language

- Basic Concepts:
  - Classes
  - Objects
  - Data Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Dynamic Binding
  - Message Passing

Classes

- Class is a collection of objects of similar type.
- Class is a way to bind the data and its associated functions together.
- Objects are basically variable of the class or an object is an instance of a class.
- *Examples*:
  - Shape is a class and Rectangle, Square, Triangle are its objects.
- A class specification has 2 parts:
  - Class declaration
  - Class function definitions
- *Class declaration:* describes type and scope of its members.
- *Class function definitions:* describes how the class functions are implemented.

Classes



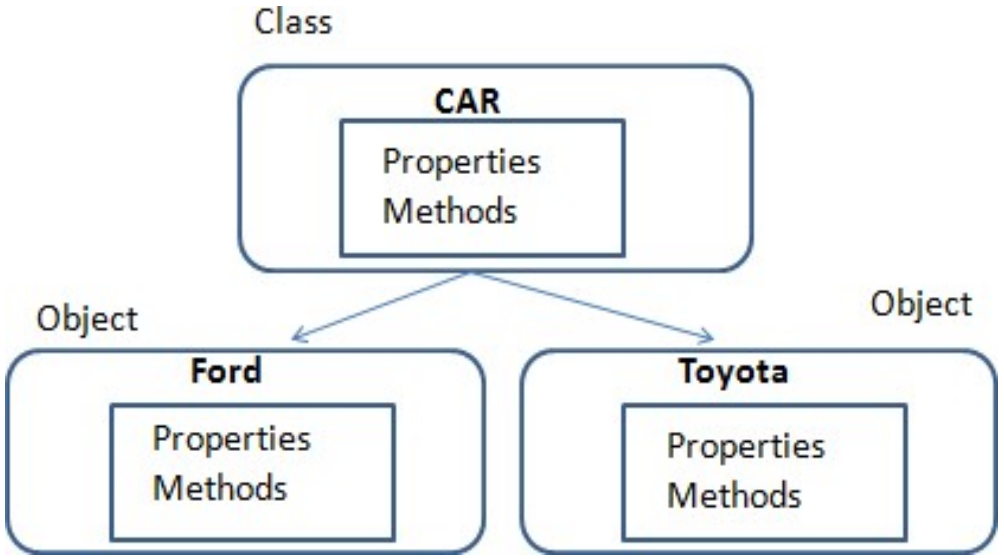
Example:

```
class person
{
        char name[20];
        int id;

public:
        void getdetails(){}
};
```

Objects

- Basic runtime entity in an object – oriented system.
- Often termed as "*instance*" of a class.
- Example 1:
  - a person, a place, a car, a bank account *etc.*
- They can be of any type based on its declaration.



Objects

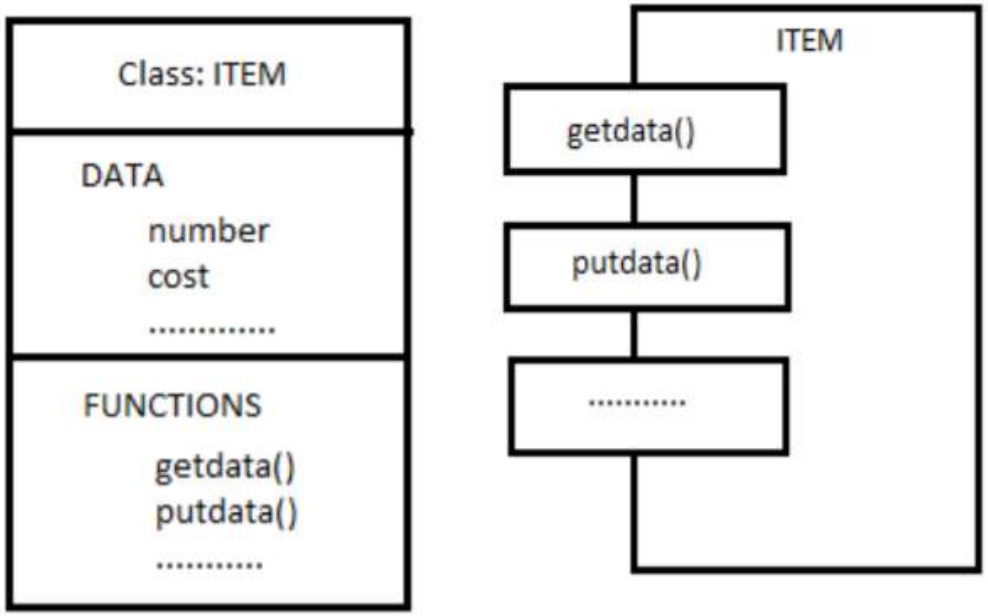
- When program is executed, objects interact by sending messages at runtime.
- Example 2:
  - Two objects namely, "*customer*", "*account*" . Customer object may send a message to the account object requesting for bank balance.

Example:

```
class person
{
        char name[20];
        int id;
public:
        void getdetails(){}
};
int main()
{
        person p1; // p1 is a object
}
```

Example: Class and Object

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



Creating Objects:

item x;

Accessing Class Members:

Object_name.function-name (actual-arguments);

x.getdata(100, 75.5); x.putdata( );

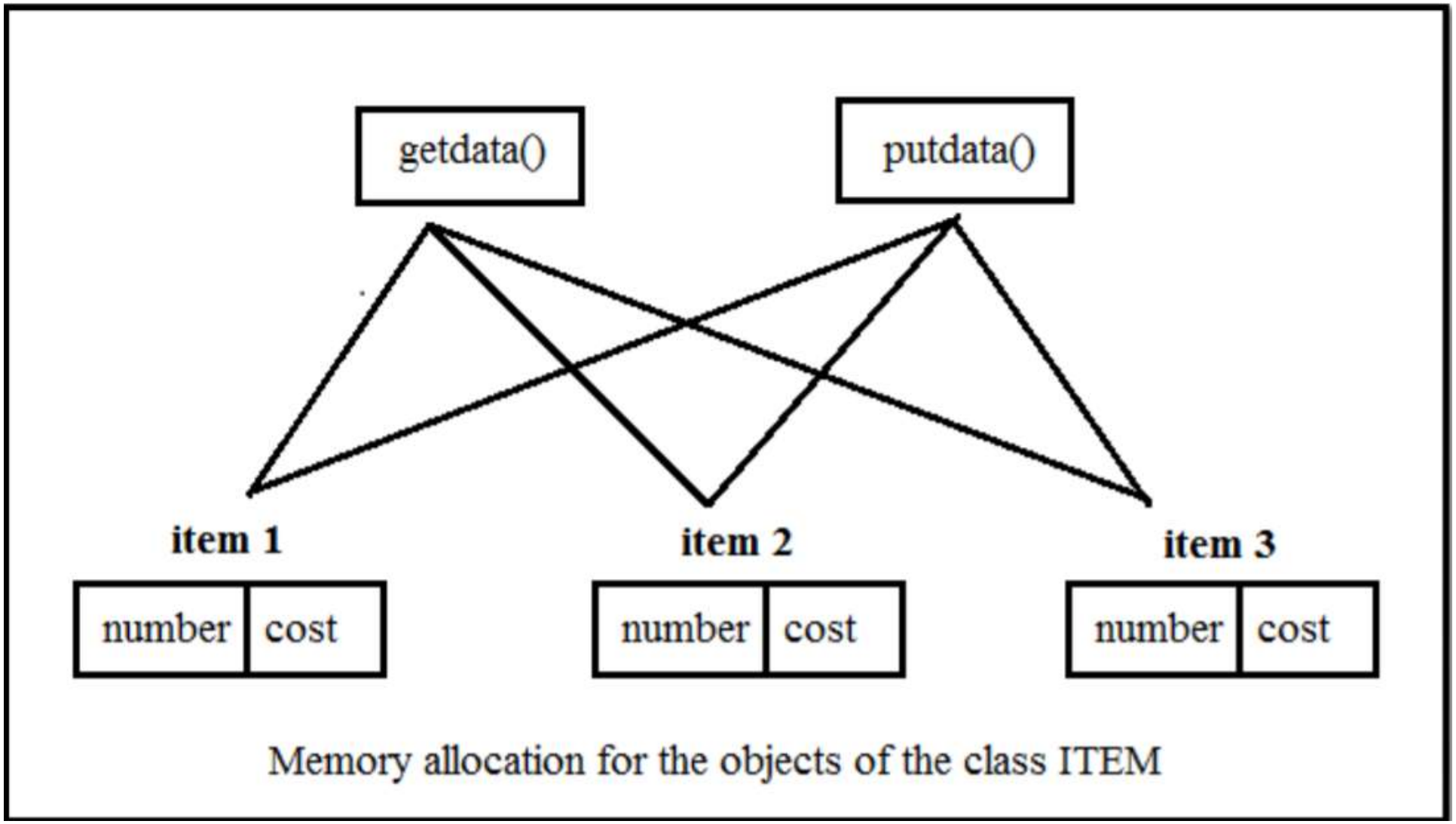
Memory Allocation of Objects

- Memory space for objects is allocated when they are declared and not when the class is specified : **partially true**.

- Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.

- Only space for member variables is allocated separately for each object. Because, member variables will hold different data values for different objects.

Example



Memory allocation for the objects of the class ITEM

Abstraction

- *Providing only essential information* to the outside world i.e. to represent the needed information in program without presenting the details.
- Data abstraction is a programming/ design technique that relies on the **separation of interface and implementation**.
- In C++, they provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

- Eg: While using an object (that is an instance of a class) the built in data types and the members in the class are ignored. This is known as **data abstraction**.

Abstraction Example

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

class implementAbstraction
{
    public:
        int a, b;

        // method to set values
        // of a and b
        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
```

Encapsulation

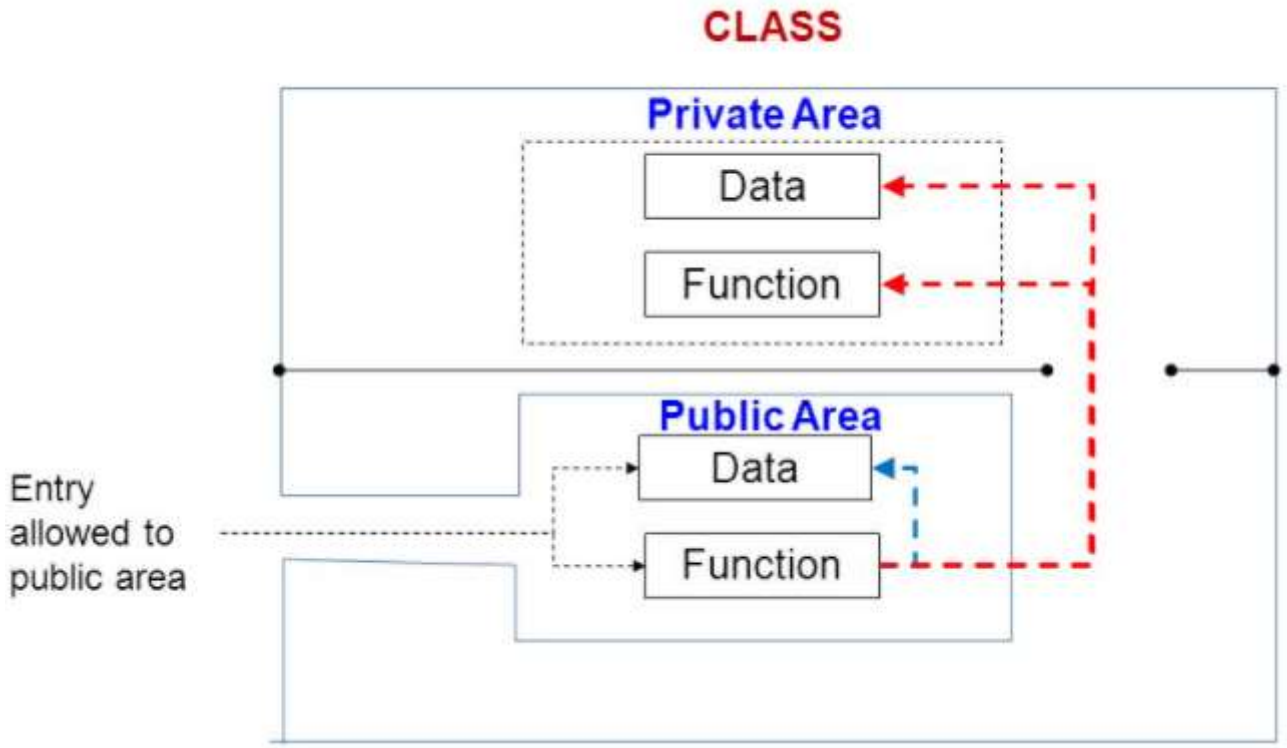
- All C++ programs are composed of the following two fundamental elements:
  - **Program statements (code):** This is the part of a program that performs actions and they are called functions.
  - **Program data:** The data is the information of the program which affected by the program functions.
- Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.
- Data encapsulation led to the important OOP concept of **data hiding.**
- C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes.**

Encapsulation

- Example: a class can contain following *Access Specifiers*

  - private, protected, public members

- By default, members of class are private, while, by default, members of structure are public.



Encapsulation Example

```
#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;
        }
};
```

```
int main()
{
    Encapsulation obj;

    obj.set(5);

    cout<<obj.get();

    return 0;
}
```

Output:
5

Inheritance

- Inheritance works on the basis of **re-usability**.

- This provides us an opportunity to reuse the code functionality and fast implementation time.

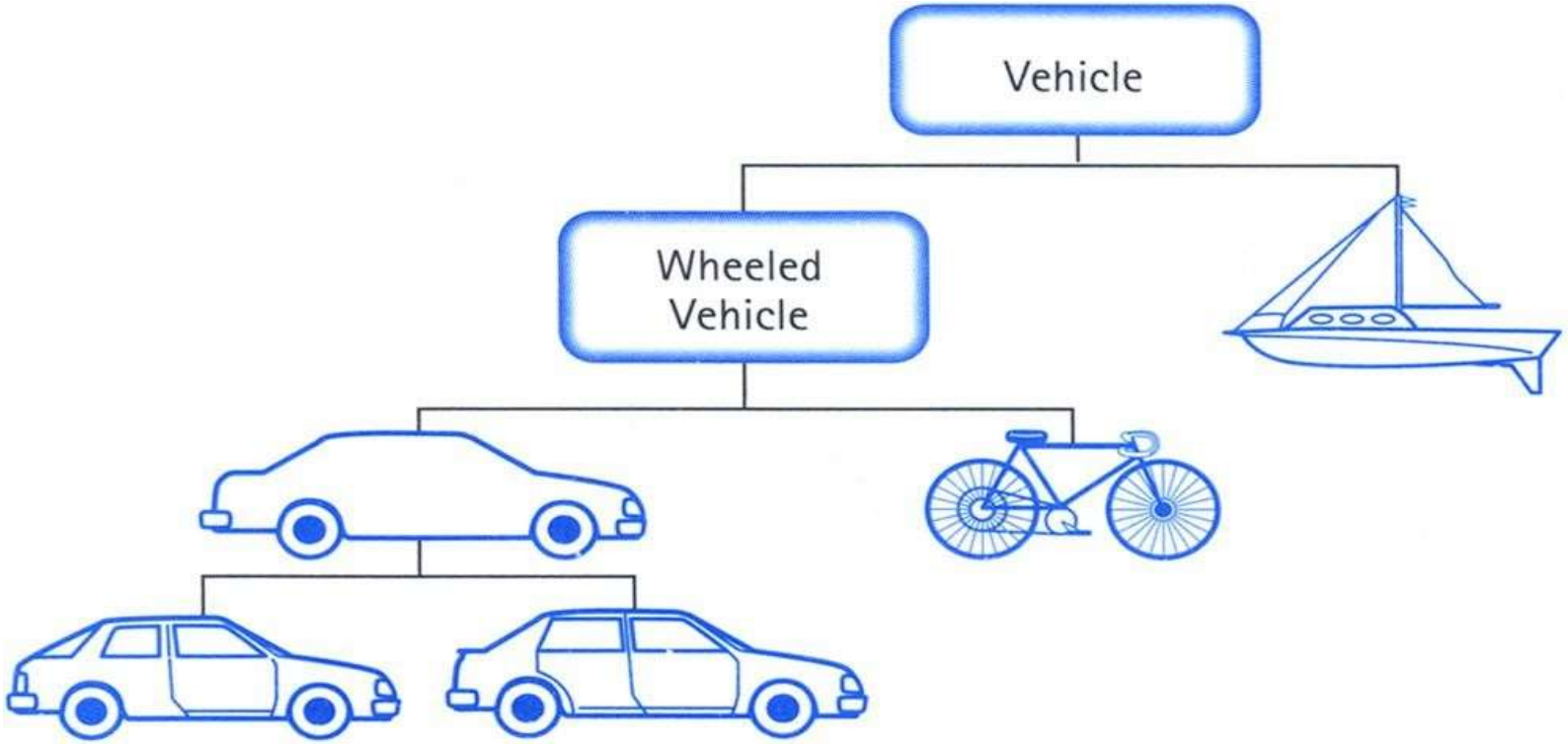
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

- The idea of inheritance implements the **is a** *relationship*.

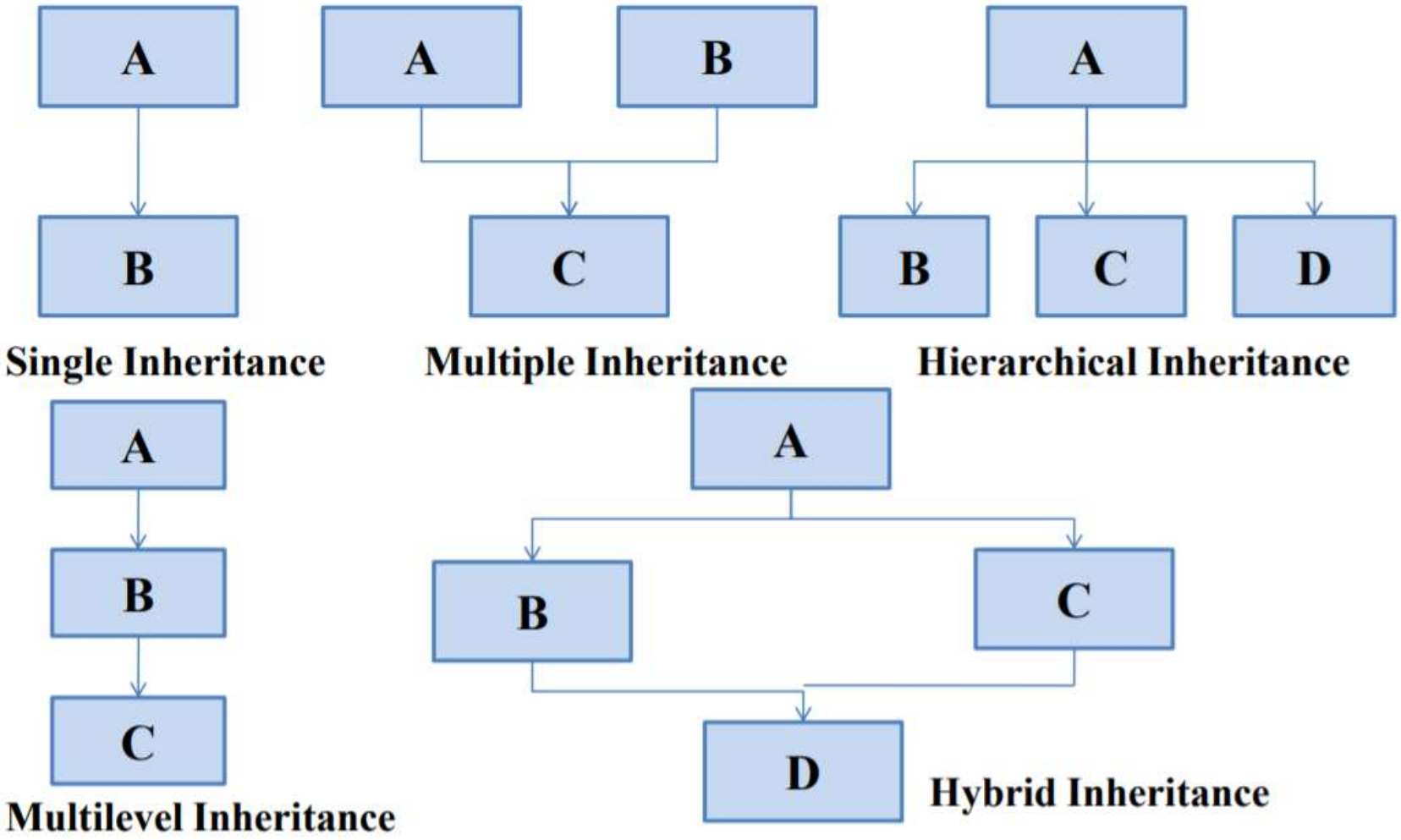
Inheritance

"The mechanism of deriving a new class from an old one is called inheritance."

- The old class is known as *Base Class*.
- New one is called the *Derived Class*.



Forms of Inheritance



Single Inheritance Example

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

//Base class
class Parent
{
    public:
      int id_p;
};

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

```
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

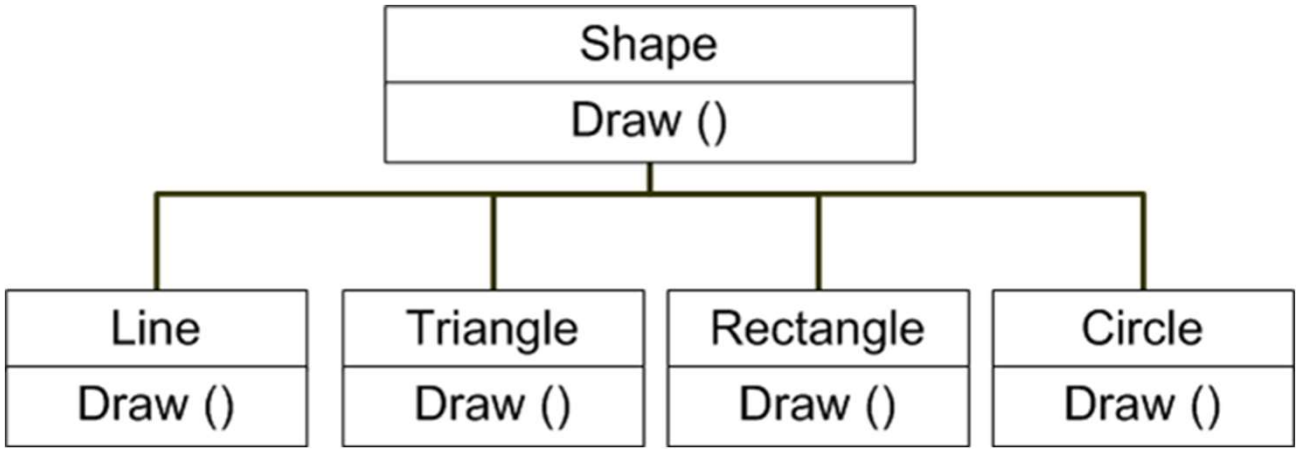
Child id is 7

Parent id is 91

Polymorphism

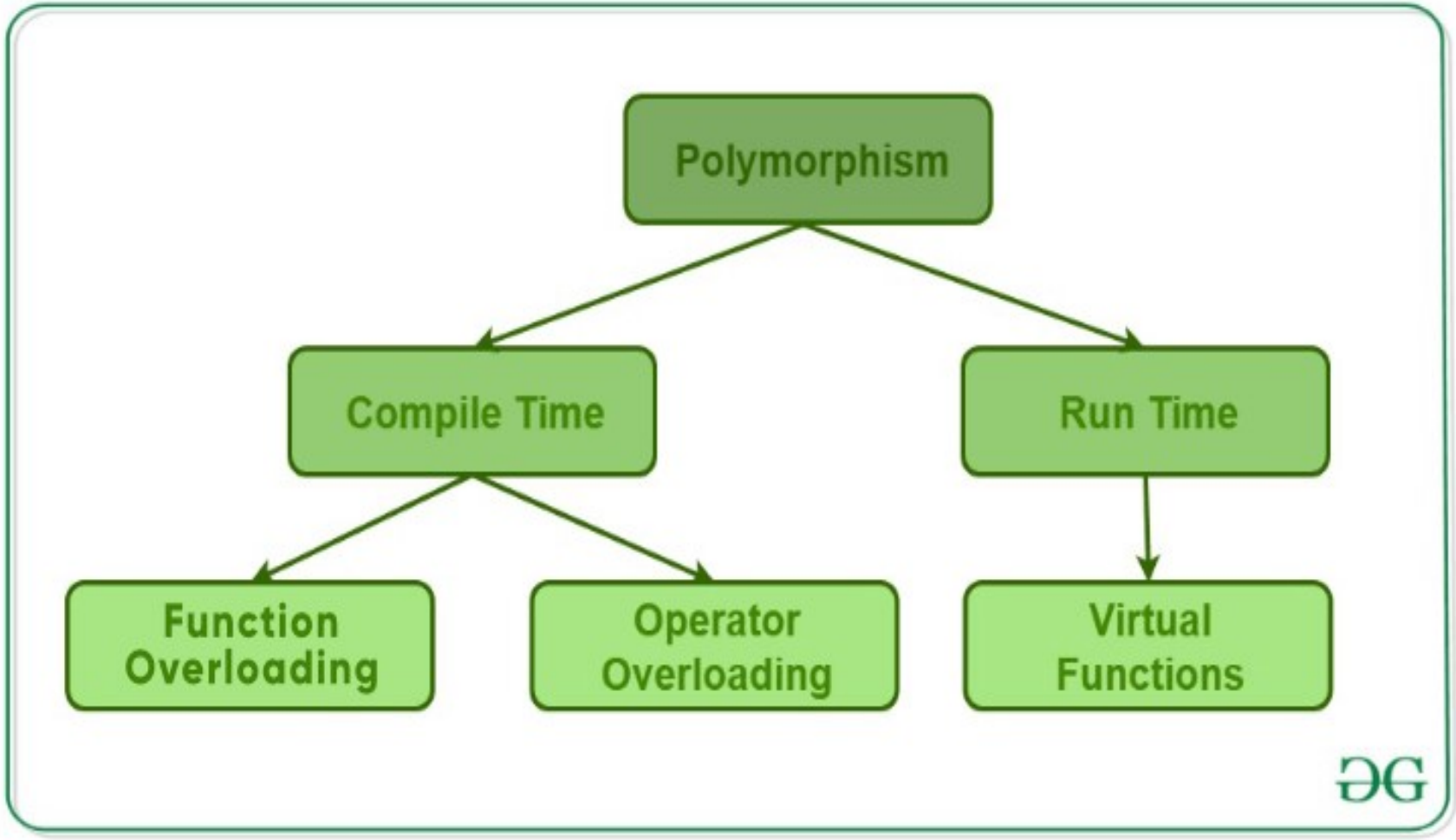
- *Poly* means **many** and *morphism* means **changing or alterable**.
- The word **polymorphism** means having *many forms*.

Polymorphism



- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Polymorphism



Function Overloading Example

```
#include<iostream>
using namespace std;
class ovld
{
    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() {

    ovld obj1;

    // 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;
}
```

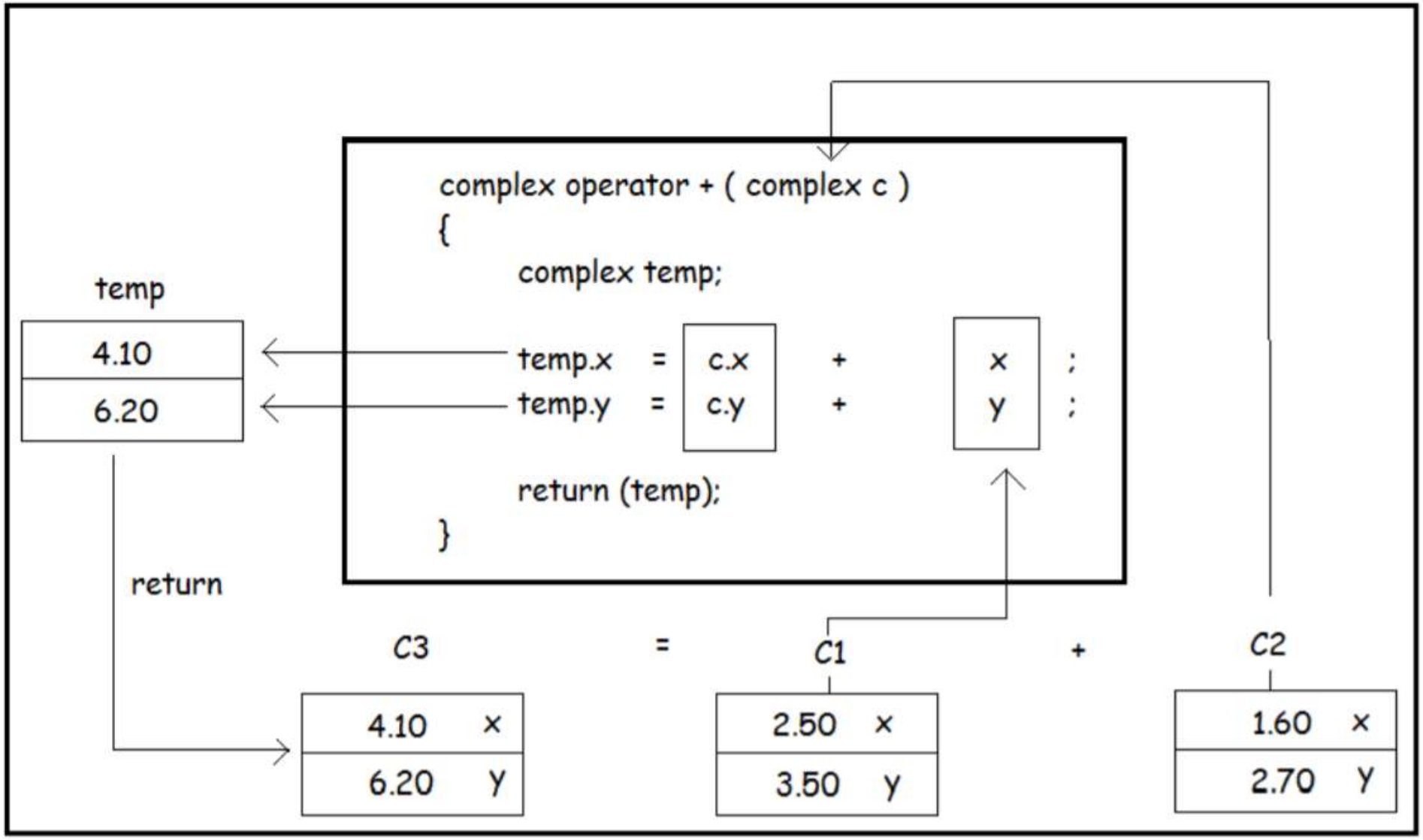
Output:
value of x is 7
value of x is 9.132
value of x and y is 85, 64

Operator Overloading Example

```
class complex
{
        float x;
        float y;
    public:
        complex(){}
        complex(float real, float imag)
        {    x=real;    y=imag;    }
        complex operator+ (complex);
};
int main ()
{
     complex C1, C2, C3;
    C1 = complex(2.5, 3.5);
    C2 = complex(1.6, 2.7);
    C3 = C1 + C2;
}
```

```
complex complex :: operator+(complex c)
{
    complex temp;
    temp.x = x + c.x;
    temp.y = y + c.y;
    return(temp);
}
```

Operator Overloading Example



Complete Example

```
#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
```

Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.

- Dynamic binding (*late binding*) means that the code associated with a given procedure call is **not** known until the **time of the call at run-time**.

- Associated with *polymorphism* and *inheritance*.

Message Passing

- OOPs consists of a set of objects that communicate with each other.

- This involves following steps:
  - *Creating classes* that define objects and their behavior
  - *Creating objects* from class definitions.
  - Establishing *communication* among *objects*

- A message for an object is a **request for execution of a procedure**, and therefore will **invoke a function** (procedure) in the receiving object that generates the desired result.

Message Passing

- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.