

Object Oriented Programming using C++

CSE-III
CSE206-OOP

Dr. Priyanka Singh

UNIT II: FEATURES OF OBJECT-ORIENTED PROGRAMMING

Introduction to Classes and Objects, Making sense of core object concepts (Encapsulation, Abstraction, Polymorphism, Classes, Messages Association, Interfaces). Constructors and its types, Destructors - Passing Objects as Function arguments and Returning Objects from Functions.

Example of Shallow Copy

```
#include <iostream>
using namespace std;
class room {
private: int length;          int breadth; int height;
public:
    void get_data(int len, int brea, int heig)
    {
        length = len;
        breadth = brea;
        height = heig;
    }
    void show_data()
    {
        cout << " Length = " << length<< "\nBreadth = " << breadth<< "\nHeight = " << height<<endl;
    }
};

int main()
{
    room r1,r3;
    r1.get_data(10,20,30);
    r1.show_data();
    room r2 = r1;
    r2.show_data();
    r3=r2;
    r3.show_data();

    return 0;
}
```

Example of Deep Copy

```
#include <iostream>
using namespace std;
class room {
public:    int length; int breadth; int *height;
public:    room()
        {height = new int;}
        void get_data(int len, int brea, int heig)
        {length = len; breadth = brea; *height = heig;
        }
        void show_data()
        { cout << " Length = " << length << "\n Breadth = " << breadth << "\n Height = " << *height << endl; }
        room(room &r)
        {
            length = r.length;    breadth = r.breadth;
            height = new int;
            *height = *(r.height); //    *height =56;
        }
};

int main()
{
    room r1;
    r1.get_data(23,14,26);
    r1.show_data();
    room r2 = r1;
    r2.show_data();
    return 0;
}
```

	Shallow Copy	Deep copy
1.	When we create a copy of object by copying data of all member variables as it is, then it is called shallow copy	When we create an object by copying data of another object along with the values of memory resources that reside outside the object, then it is called a deep copy
2.	A shallow copy of an object copies all of the member field values.	Deep copy is performed by implementing our own copy constructor.
3.	In shallow copy, the two objects are not independent	It copies all fields, and makes copies of dynamically allocated memory pointed to by the fields
4.	It also creates a copy of the dynamically allocated objects	If we do not create the deep copy in a rightful way then the copy will point to the original, with disastrous consequences.

Destructors

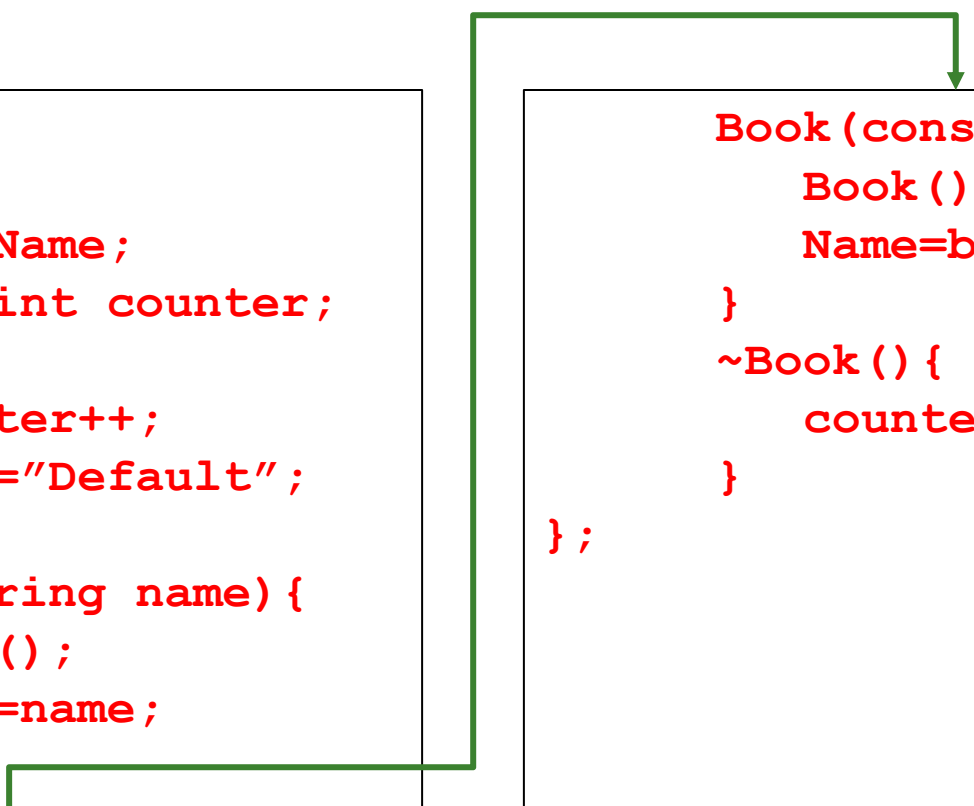
- Destructor is a special class function which destroys the object as soon as the scope of object ends.
- Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

```
class Book
{
    public:
    String Name;
    Book()
    {Name="Default";}
    ~Book()
    {Name=" Hey";}
};
```

Counting Objects with Constructors & Destructors

```
class Book{  
    public:  
        String Name;  
        static int counter;  
        Book() {  
            counter++;  
            Name="Default";  
        }  
        Book(String name){  
            Book();  
            Name=name;  
        }  
};
```

```
Book(const Book& b) {  
    Book();  
    Name=b.Name;  
}  
~Book() {  
    counter--;  
}  
};
```



Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

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

Access Specifier

```
class MyClass {  
    protected:  
        // Attributes  
        // Methods  
};
```

```
class MyClass { // The class  
    public:      // Access specifier  
        // class members goes here  
};
```

```
class MyClass {  
    public:    // Public access specifier  
        int x;    // Public attribute  
    private:  // Private access specifier  
        int y;    // Private attribute  
};
```

```
int main() {  
    MyClass myObj;  
    myObj.x = 25;    // Allowed (public)  
    myObj.y = 50;    // Not allowed (private)  
    return 0;  
}
```

Encapsulations

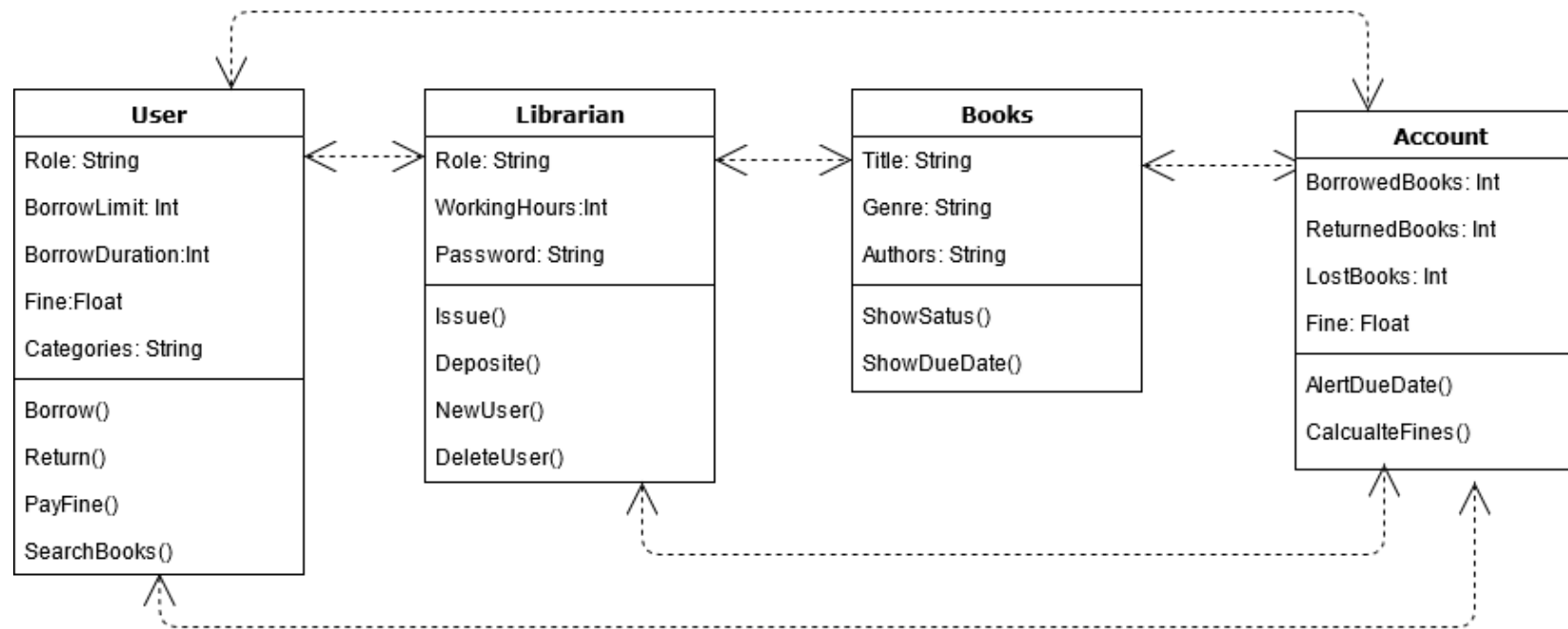
“Encapsulation”, is to make sure that "sensitive" data is hidden from users.

```
class Employee {  
    private:  
        // Private attribute  
        int salary;  
  
    public:  
        // Setter  
        void setSalary(int s) {  
            salary = s;  
        }  
}
```

```
// Getter  
        int getSalary() {  
            return salary;  
        }  
};  
  
int main() {  
    Employee myObj;  
    myObj.setSalary(50000);  
    cout << myObj.getSalary();  
    return 0;  
}
```

UML: Class Diagram

Remember the library example?



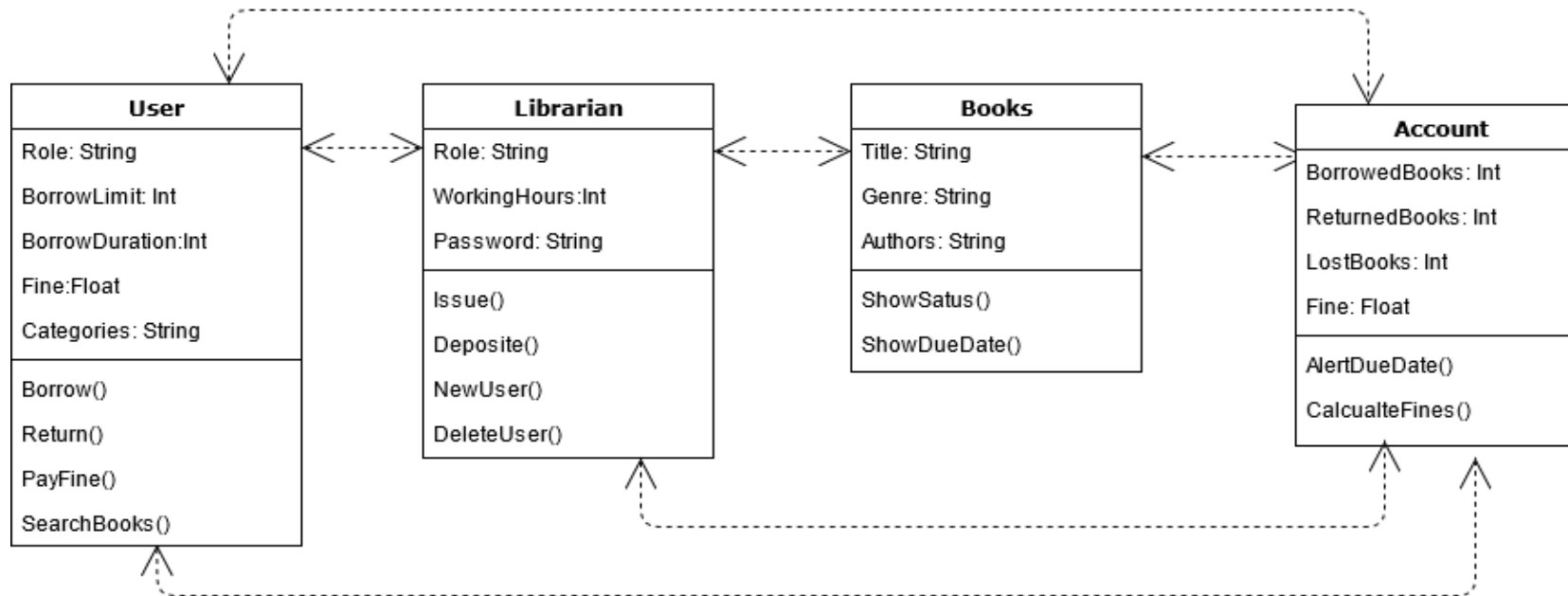
Class Identification: Exercise

■ Can you identify the classes here?

- ❑ A student can renew his borrowed books
- ❑ A square is a polygon
- ❑ Shyam is a student
- ❑ Every student has a name
- ❑ 100 paisa is one rupee
- ❑ Students live in hostels
- ❑ Every student is a member of library
- ❑ The department has many students

UML Class Diagram: Relationship

- So far we talked about the nodes/objects mostly.
- We avoided the the edges to retain the simplicity level.



UML Class Diagram: Relationship

- So far we talked about the nodes/objects mostly.
- We avoided the the edges to retain the simplicity level.

- Association: two peer classes
- Dependency: Student & Books
- Aggregation: C2 is part of C1
- Composition: C2 is not stand alone
- Inheritance: Between parent and child
- Realization: between Blueprint and realizer

Association

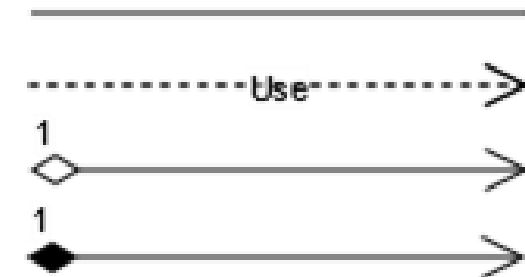
Dependency

Aggregation

Composition

Inheritance

Realization



Relationship Identification: Exercise

- Can you identify the classes here?
 - ❑ A student can renew his borrowed books
 - ❑ A square is a polygon
 - ❑ Shyam is a student
 - ❑ Every student has a name
 - ❑ 100 paisa is one rupee
 - ❑ Students live in hostels
 - ❑ Every student is a member of library
 - ❑ The department has many students

Relationship Identification: Exercise

■ Can you identify the relationships here?

- A **student** can renew his borrowed **books**
- A **square** is a **polygon**
- **Shyam** is a **student**
- Every **student** has a name
- 100 **paisa** is one **rupee**
- **Students** **live in** **hostels**
- Every **student** **is a member of** **library**
- The **department** **has many** **students**

Class

Object

Attribute

Association

Dependency

Aggregation

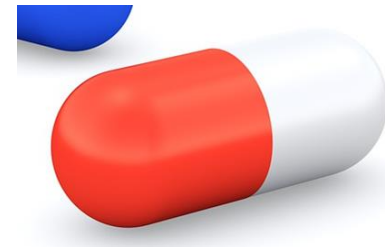
Composition

Inheritance

Realization

OOPS Principle: Encapsulation

Encapsulations



- **Encapsulation:** -- refers to the bundling of data, along with the methods that operate on that data, into a single unit.
- mechanism of restricting the direct access to some components of an object
- The term came from **capsule**.
- “Techniques used to enclose medicines—in a relatively stable shell known as a capsule” – Wiki
- **Capsules are used**
 - ☐ -- When tablets can not formed
 - ☐ -- The release of the medicine can be controlled

Encapsulations

- Encapsulation:
 - -- Bundling is achieved by Class
 - -- Information hiding is controlled using two techniques
- Access Specifiers:
- Getter/Setter Methods:

Encapsulations

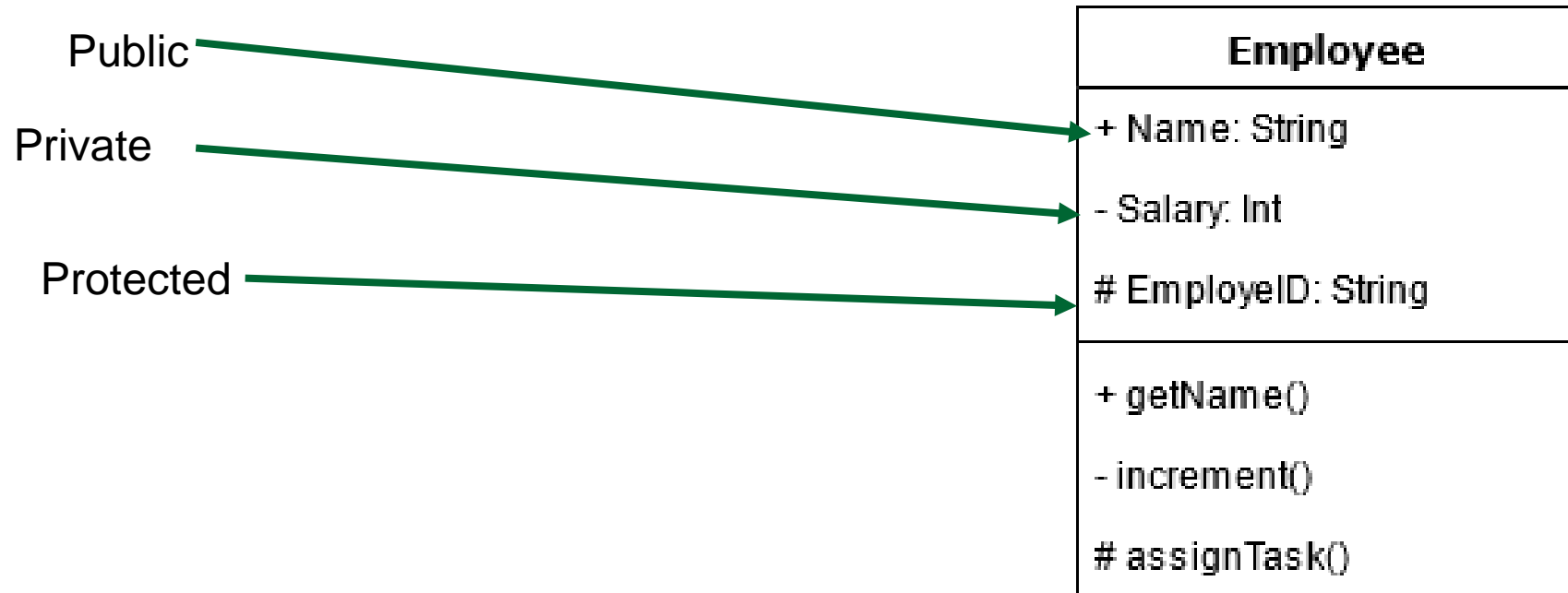
- “Encapsulation”, is to make sure that "sensitive" data is hidden from users.

```
class Employee {  
    private:  
        // Private attribute  
        int salary;  
    public:  
        // Setter  
        void setSalary(int s) {  
            salary = s;  
        }  
}
```

```
// Getter  
        int getSalary() {  
            return salary;  
        }  
};  
  
int main() {  
    Employee myObj;  
    myObj.setSalary(50000);  
    cout << myObj.getSalary();  
    return 0;  
}
```

UML: Data Hiding

Content Here



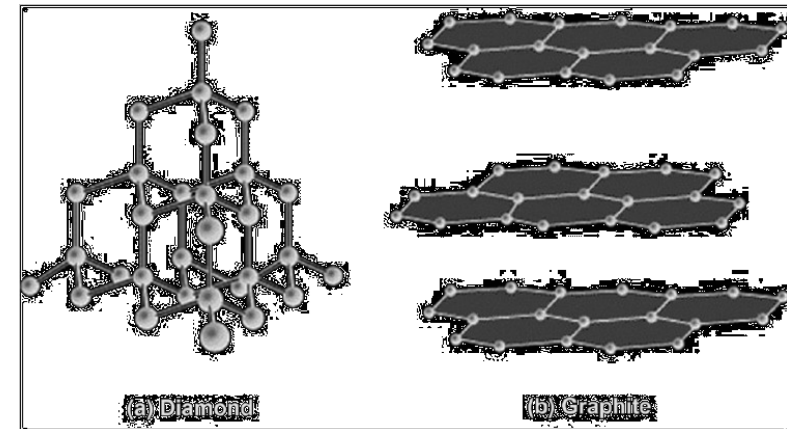
Encapsulations

“Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behavior of an object...encapsulation focuses on the implementation that gives rise to this behavior” -- Grady Booch

OOPS Principle: Polymorphism

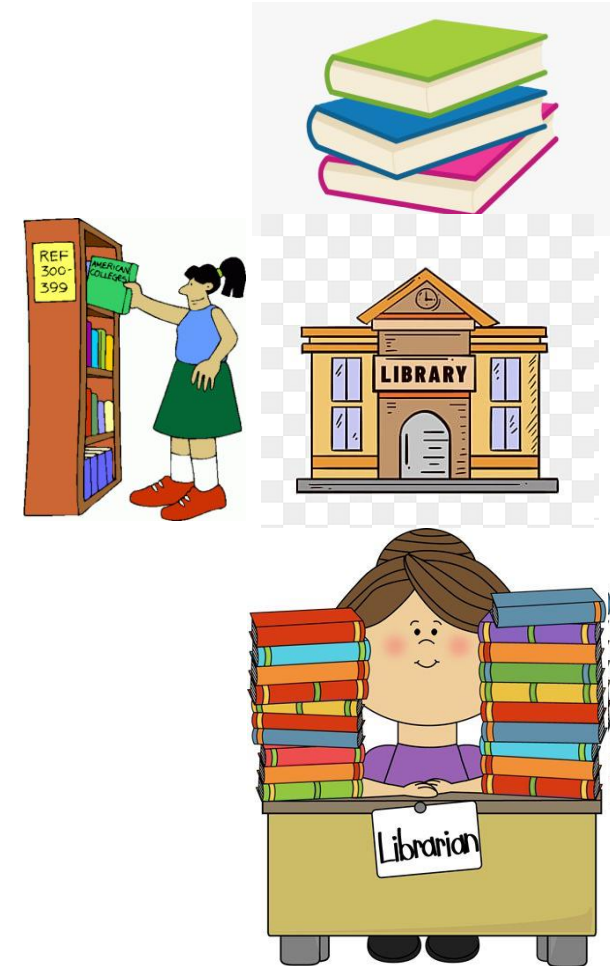
Polymorphism

- **Poly= Multi ; morphe= Form**
- Allows us to perform a single action in different ways.
- Recap organic chemistry
- Carbon has this property
- The crystal structures are different
- Diamond vs Graphite



Polymorphism

- Let's go back to the Library example
- Issue book ()
 - Can be used by both Students
 - Can be used by Faculties
- The tasks are different
- Similarly
 - “+” with integer
 - “+” with complex number
- The characteristics/processing are different.



UML: Polymorphism

Librarian
+ Role: String + WorkingHrs: Int + Password: String
+ Issue(Book,Student):void + Issue(Book, Faculty): void + Deposit(Book, Faculty): void + Deposit(Book, Student): void

Librarian
+ Role: String + WorkingHrs: Int + Password: String
+ operator + (Book,Book): Book + operator + (User, User): User + operator + (Account, Account): Account

C++: Overloading

C++: Overloading

- In C++ the “Overloading” is the way to realize polymorphism.
- Two types of overloading:
 - Function/ Method overloading
 - Operator overloading

C++ Syntax: Function Overloading

```
void print(int i) {  
    cout << " Here is int " << i << endl;  
}  
void print(double f) {  
    cout << " Here is float " << f << endl;  
}  
  
int main() {  
    print(10);  
    print(10.10);  
    return 0;  
}
```

C++ Syntax : 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;}
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();
}
```

Inheritance

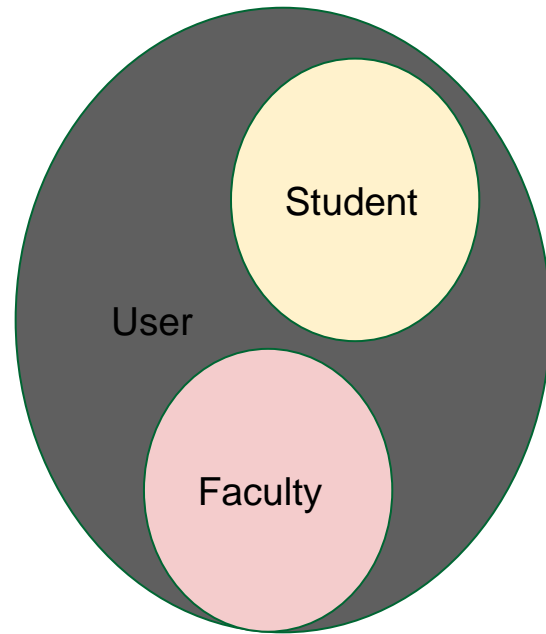
Inheritance

■ Cambridge Dictionary

- ~~Money, land, or possessions received from someone after the person has died~~
- Inheritance means the particular characteristics received from parents through the genes.

Inheritance

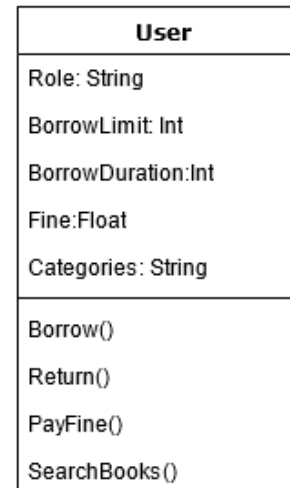
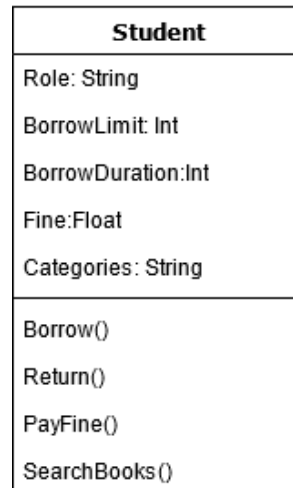
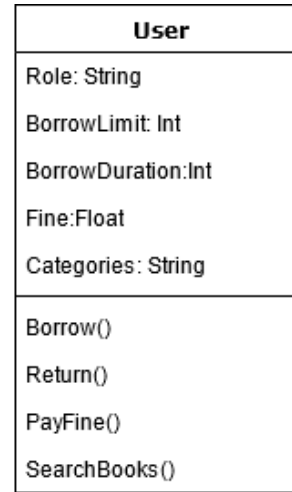
- Let's go back to the Library example
- Segregate the User into different categories



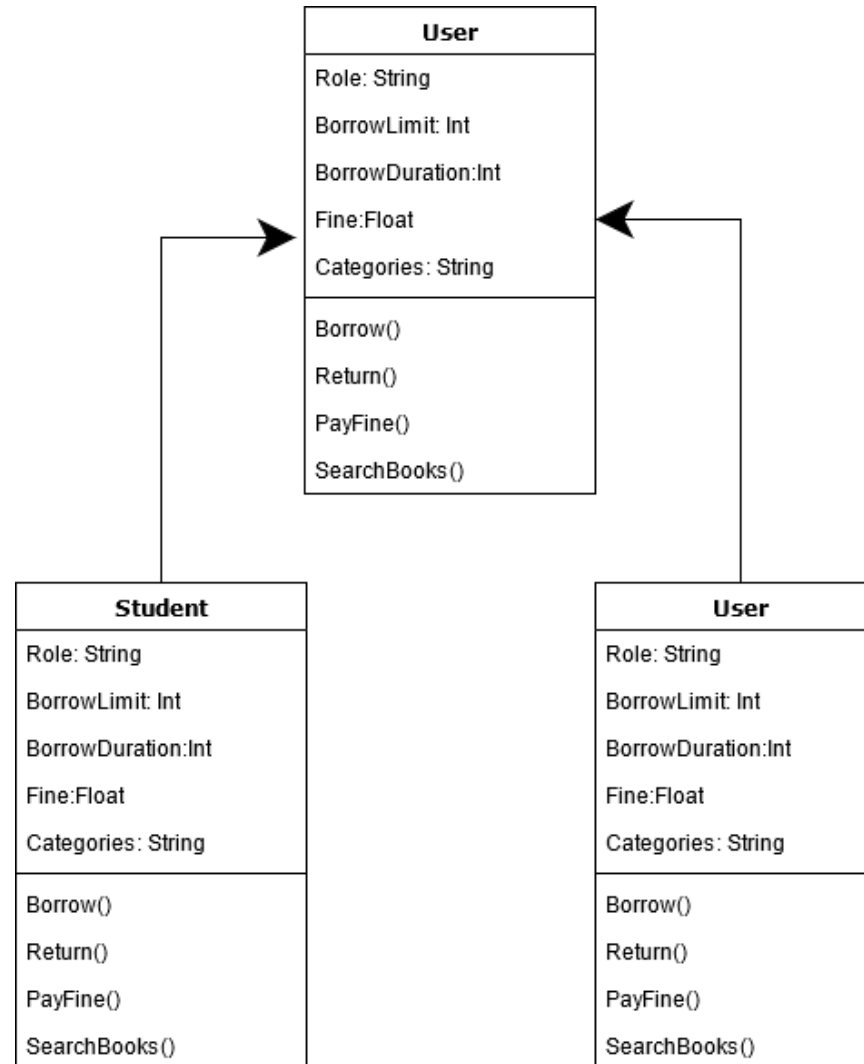
Properties are inherited
Actions are similar



UML: Class Diagram & Inheritance



UML: Class Diagram & Inheritance



Inheritance

```
#include<iostream>
using namespace std;
class User {
    public:
    int borrowLimit;
    int borrowDuration;
    double fine;
};
class Student: public User{
    public:
    String Role ="Student";
};

class Faculty: public User{
    public:
    String Role ="Faculty"
};
int main(){
    User u1; Student s1; Faculty f1;
    cout << u1.fine << s1.fine
    << f1.fine << endl;
}
```

Method Overriding

```
#include <iostream>
using namespace std;
class Base {
public:
    void print() {
        cout << "Base Function" << endl;
    }
};
class Derived : public Base {
public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

C++ Syntax : Operator Overriding

```
#include<iostream>
using namespace std;
class Complex {
private:
int real, imag;
public:
Complex(int r = 0, int i =0) {real =
r; imag = i;}
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();
}
```

Access Specifier Revisited

“public” & “private” we have seen already

What is “protected” access?

- Provides restricted access but not stringent like private
- Somewhat between private and public
- Only child can access protected members

```
class MyClass {  
    public:  
    private:  
    protected:  
};
```

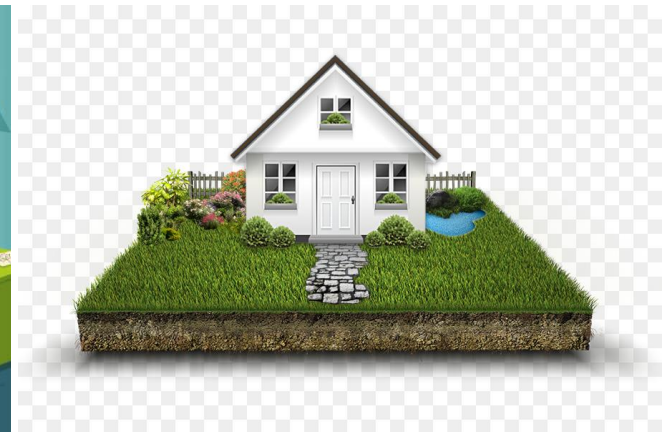
PROTECTED



PRIVATE



PUBLIC



Access Specifier Revisited

```
class Parent { // base class
protected:
    int id_protected;
};
class Child : public Parent { // derived class
public:
    void setId(int id){
        // Child class is able to access the inherited
        // protected data members of base class
        id_protected = id;
    }
    void displayId(){
        cout << "id_protected is: " << id_protected << endl;
    }
};
```

```
int main(){
    Child obj1;
    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

Passing and Returning Objects in C++

- In C++ we can pass class's objects as arguments and also return them from a function the same way we pass and return other variables. No special keyword or header file is required to do so.
- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax:

```
function_name(object_name);
```

```
#include <iostream>
using namespace std;
class Example {
public:    int a;
        void add(Example E)
        {a = a + E.a;
         };
int main()
{
    Example E1, E2;
    E1.a = 50;
    E2.a = 100;
    cout << "Initial Values \n";
    cout << "Value of object 1: " << E1.a<< "\n& object 2: " << E2.a<< "\n\n";
    E2.add(E1);
    cout << "New values \n";
    cout << "Value of object 1: " << E1.a<< "\n& object 2: " << E2.a<< "\n\n";
    return 0;
}
```

Returning Object as argument

Syntax:

object = return object_name;

Returning Object as argument

```
#include <iostream>
using namespace std;
class Example {
public:   int a;
        Example add(Example Ea, Example Eb)
        {           Example Ec;   Ec.a = Ea.a + Eb.a;
                return Ec;
        };
int main()
{Example E1, E2, E3;
E1.a = 50;E2.a = 100;E3.a = 0;
cout << "Initial Values \n";
cout << "Value of object 1: " << E1.a<< ", \nobject 2: " << E2.a<< ", \nobject 3: " << E3.a<< "\n";
E3 = E3.add(E1, E2);
cout << "New values \n";
cout << "Value of object 1: " << E1.a<< ", \nobject 2: " << E2.a<< ", \nobject 3: " << E3.a << "\n";
return 0;
}
```

Practice question

Create a class named 'Rectangle' with two data members- length and breadth and a function to calculate the area which is 'length*breadth'. The class has three constructors which are :

- having no parameter - values of both length and breadth are assigned zero.
- having two numbers as parameters - the two numbers are assigned as length and breadth respectively.
- having one number as parameter - both length and breadth are assigned that number.

Now, create objects of the 'Rectangle' class having none, one and two parameters and print their areas.