# Object Oriented Analysis and Design

CS-291

# Abstraction

- Humans manage complexity through abstraction.
- A good abstraction is achieved by having
  - meaningful name reflecting the function
  - **minimum** and at the same time **complete** features
  - coherent features (relatedness)

# Kinds of Abstraction

- **Entity Abstraction**

  –An object that represents a useful model of a problem domain or solution domain entity

- **Action Abstraction**

  –An object that provides a generalized set of operations, all of which perform the same kind of function

  – Example: *scrambler, which embodies a general set of encryption functions, each of some level of complexity.*

- **Virtual Machine Abstraction**

–An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations

- Example: Operating System

- **Coincidental Abstraction**

–An object that packages a set of operations that have no relation to each other

- This form of packaging is not considered a good thing.

# The Three OOP Principles

- All object-oriented programming languages provide mechanisms to implement the object-oriented model
  - Encapsulation
  - Inheritance, and
  - Polymorphism.

# Encapsulation

- *Encapsulation* binds together code and the data it manipulates
- **Keeps both safe** from outside interference and misuse
  - Access to the code and data inside the wrapper is tightly controlled through a well-defined interface
- Example: Driving a car
- Everyone knows how to access it and thus can use it regardless of the implementation details—**and without fear of unexpected side effects**
- In most OO programming languages the basis of encapsulation is the class

# Encapsulation

- A *class* defines the structure and behavior (data and code) that will be shared by a set of objects
- Member variables and member functions
- Interface of a class
- Each object of a given class contains the structure and behavior defined by the class
- Objects are referred to as *instances of a class*
- A class is a logical construct
- An object has physical reality

- Class provides mechanisms for hiding the complexity of the implementation inside the class
- Each function or variable in a class may be marked private or public (or protected)
- ***public*** interface of a class represents everything that external users of the class need to know
- The ***private*** methods and data can only be accessed by code that is a member of the class.
- Since the private members of a class may only be accessed through the class' public methods, we can ensure that no improper actions take place.
- Of course, this means that the public interface should be carefully designed

## The general form of a class declaration in C++

```
class class-name {
    private data and functions
access-specifier:
    data and functions
access-specifier:
    data and functions
// ...
access-specifier:
    data and functions
} object-list;
```

```
class stack {
    int stck[SIZE];
    int tos;
public:
    void init();
    void push(int i);
    int pop();
};
void stack::push(int i){
// code
}
stack    mystack;
mystack.push(1);
```

```
mystack.tos = 0; // Error
```

## Constructors and Destructors:

Object requires initialization before it can be used

```
class stack {
    int stck[SIZE];
    int tos;
public:
    stack();
    void push(int i);
    int pop();
};
```

- An object's constructor is automatically called when the object is created.
- An object‘s constructor is called once for global or static local objects.
- For local objects, the constructor is called each time the object declaration is encountered.

- 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
- Abstraction "helps people to think about what they are doing,"
- Encapsulation "allows program changes to be reliably made with limited effort

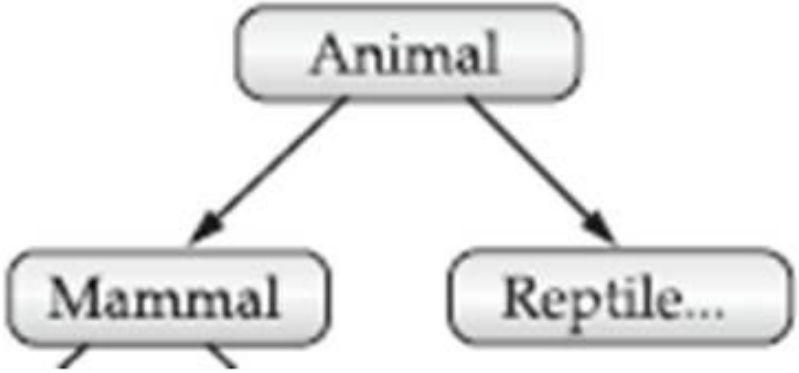
- Encapsulation: Low coupling (minimize the number of cases where changes to one module necessitates a change in other module)
- Abstraction: High cohesion
- Abstraction is when a client of a module doesn't need to know more than is in the interface
- Encapsulation is when a client of a module isn't able to know more than is in the interface

# “is a” hierarchy

- Discovery of **common** abstractions and mechanisms enables us to understand complex systems.
- For example, with just a few minutes of orientation, an experienced pilot can step into a multiengine jet aircraft he or she has never flown before and safely fly it
- A turbofan engine is a specific kind of jet engine
- A jet engine represents a generalization of the properties common to every kind of jet engine
- a turbofan engine is simply a specialized kind of jet engine

# Inheritance

- *Inheritance* is the process by which one object acquires the properties of another object
- Supports the concept of hierarchical classification
- Without this, each object would need to define all of its characteristics explicitly.
- By use of inheritance, an object need only define those qualities that make it unique within its class
- It can inherit its general attributes from its parent



**The benefits of inheritance**

1. Reusability of software components and code Sharing
2. Increased reliability
3. Consistency of interface

# *Class hierarchy in C++*

```
class building {
   int rooms;

public:
   void set_rooms(int num){
      rooms = num;
   }
   int get_rooms(){
      return rooms;
   }
};
```

```
class house : public building {
   int bedrooms;
public:
   void set_bedrooms(int num);
   int get_bedrooms();
};
class school : public building {
   int classrooms;
   int offices;
public:
   void set_classrooms(int num);
   int get_classrooms();
   void set_offices(int num);
   int get_offices();
};
```

```
int main()
{
    house h;
    school s;
    h.set_rooms(12);
    h.set_bedrooms(5);
    cout << "house has " << h.get_bedrooms();
    cout << " bedrooms\n";
    s.set_rooms(200);
    s.set_classrooms(180);
    cout << "school has " << s.get_classrooms();
    cout << " classrooms\n";
    return 0;
}
```

```
class base {
        int i, j;
public:
        void set(int a, int b) { i=a; j=b; }
        void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
        int k;
public:
        derived(int x) { k=x; }
        void show() { cout << k << "\n"; }
};
int main(){
        derived ob(3);
        ob.set(1, 2); // access member of base
        ob.show(); // uses member of derived class ...
```