ECE 575/
Object Oriented Modeling and Programming (in C++)

Part 1: System Modeling

Moon Ho Hwang
mhhwang@ece.arizona.edu

2006 Fall
ECE Department,
University of Arizona
Categorization of Program Languages

- Structured Programming Language
  - FORTURAN, COBOL,
  - C, Pascal

- Object-Oriented Programming Language
  - C++
  - Java, C#
  - Python

Modeling: OOM/P
Object in OOP

- An object class consists of
  - Data: Storing Properties
    - int age; string name; float height;
  - Functions: Action
    - read_data, write_data;
    - {construct/destruct}_class_instance;
Object Class

- Ex: Class Student
  - Data:
    - Name
    - Student_ID
  - Function:
    - Do_Take_Class()

```cpp
class Student {
    string Name;
    string ST_id;

    void Do_Take_Class() const {
        cout << "take a class." << endl;
    }
};
```

Figure1. C++ code for Student class.
Object Class

- Ex: Class Professor
  - Data:
    - Name
    - Employee_ID
  - Function:
    - Do_Teach_Class()

```cpp
class Professor
{
    string Name;
    string EP_id;

    void Do_Teach_Class() const
    {
        cout << "teach a class." << endl;
    }
};
```

Figure 2. C++ code for Professor class.
Class Student and Professor have common features:
- Name
- Do_something

Diagram:
- Student
  - ST_ID
- Professor
  - Name
  - Do_???
  - EP_ID
Base Class

- If we make a class Person as a base class of Student and Professor, it can inherit its common features to them.

- Common features:
  - Data: string Name;
  - Function: Do();
A base class *inherits* its feature to the derived class.

- **Person**
  - String Name;
  - void Do();

- **Student**
  - String ST_ID;
  - void Take_Class();
  - void Do();

- **Professor**
  - String EP_ID;
  - void Teach_Class();
  - void Do();
Abstract vs Concrete

- See Person_Student_Professor Example.
- An instance of a class is a physical useable memory in a computer.
- A concrete class can be an instance.
- An abstract class cannot be an instance, but it provides a conceptual foundation of its derived classes.
- In general, if there is an abstract class, there is supposed to be a derived class which is concrete.
Making Abstract/Concrete classes

- Member Encapsulation:
  - public, protect, private

- Using encapsulation of Constructor
  - Non-public constructor => Abstract class
  - Public constructor => Concrete class

```cpp
class Person
{
protected:
  string Name;
  Person(string name): Name(name){}
};
```

```cpp
class Student: public Person
{
public:
  Student(string name):Person(name){}
};
```
Making an Abstract/Concrete class (cont.)

- Using *pure virtual* functions
  - If a class having non-implemented pure virtual function => Abstract class
  - If not => Can be a Concrete class

```cpp
class Person
{
...
public:
    virtual void Do() = 0;
};
```
A constructor of a class is a function making an instance of the class.

The function name of constructor is equal to *the class name*.

For class `Person`, `Person (string name);`

More than one constructor with different arguments are possible

`Person (string name);`

`Person (const Person& ob); /* copy constructor */`
Destructor (in C++)

- The destructor of a class is the function deleting an instance of the class.
- All destructor functions start with “~” and have the same name as the class name.
  - For class Person, ~Person();
- No arguments allowed, so only one destructor can be defined for each class.
- If we don’t need a special function of destructor, we don’t have to define the destructor of a class.
- Usually, destructor is defined as a **virtual** function. (Why?)
virtual vs non-virtual

A virtual function is a function which performs different ways according to its specific instance.

class Student: public Person
{
    /*virtual*/ void Do()
    {
        cout << Name << " takes ECE575." << endl;
    }
};

class Professor: public Person
{
    public:
    /*virtual*/ void Do()
    {
        cout << Name << " teaches ECE575." << endl;
    }
};
virtual vs non-virtual

- The virtual function supports *polymorphism* which means multiple appearances, because it works different depending on associated instances.
virtual vs non-virtual (con.)

- A function with its body of a derived class is called a *overriding* function.
  
  ```
  void Student::Do() { /* ... */ }
  ```

- Notice that *overloading* functions are a set of functions whose function names are same but having different arguments.
  
  ```
  void Student::Do(int n) { /* ... */ }
  ```
virtual vs non-virtual (con.)

- In Java, all functions are virtual.
- In C++ and C#, only functions having modifier virtual are virtual.
- If we define all functions as virtual (or overriding) functions, what can be advantage or disadvantage?
  - Advantage: Easy to use
  - Disadvantage: Slower and need bigger memory storage.
- See VirtualFuncs Example.
Static vs Instance

- *Static* is a modifier indicating associated data or functions of a *class* are not depending on its instance, but the class itself.

- Ex.: Current time

```cpp
class Person
{
    protected:
        static double current_Time;
        string Name;
};
```

- See *Static Example*. 
Static vs Instance

- Since static function is independent from a class instance, it cannot be a virtual function which is related to the instance.

```cpp
class Person {
    public:
        virtual static Do();
};
```
constant function (in C++)

- A constant function is a function in which data field of an instance cannot change. We can access data for reading through the function.

```cpp
class Person
{
    string Name;

    string get_name() const { return Name; }
    void set_name(string n) const { Name = n; }  //<-wrong
}
```
constant function (in C++)

- A constant function is related to an instance of class, but a static function is not, so a static function cannot be a constant function at the same time.

```cpp
class Person
{
public:
    static string get_name() const;
};
```
Value, pointer, and reference (in C++)

- A Value-type data exists in a stack area of code.
- To access a field of a value-type class instance, we indicate by “.” mark.

```c++
Student A("a");
A.Name;
```

- A pointer-type data exists either in a stack area of code or out of the stack, called the heap area (when we allocate the instance by calling “new” operator).
- To access the memory address, we use “->”.

```c++
Student* A = new Student("a");
A->Name;
```
A reference-type data exists either in a stack area of code or out of the stack, called the heap area (when we allocate the instance by calling “new” operator).

To access the memory address, we use “.” mark.

```cpp
Student& A = *new Student("a");
A.Name;
```
Value of Memory Address

- A value of a pointer can be assessed by adding operator "*".

```cpp
Student* A = new Student("a");
assert(A->Name == (*A).Name);
```
Address of Value or Pointer

- A address of a value or pointer can be assessed by adding operator "&".
- See ValuePointerReference Example.

```
Student A("a");
assert(A.Name == (&A)->Name);
```
Passing Value Argument

- If we are passing an value-type (VT) argument to a function, the whole class of the argument is copied at the stack area.

```cpp
Void A::set_data(HugeClass d)
{
    data = d;
}
```
If we are passing a pointer-type (PT) argument or reference-type (RF) argument to a function, the memory address is copied, not the whole class.

```cpp
Void A::set_data(HugeClass* d)
{
    data = *d;
}

Void A::set_data(HugeClass& d)
{
    data = d;
}
```
Safety vs Efficiency

Since the *whole class* is copied when using a VT argument, the calling speed of VT can be slower than that of PT or RT.

However, when using PT or RT, the argument can be changed inside the function, so it can be unsafe.

```c++
Void A::set_data(HugeClass& d) {
    data = d;
    d.reset(); /* HugeClass::reset() is not const function */
}
```
Passing constant Reference Argument

- To guarantee that the PT or RT argument doesn’t change, we can define it *constant* argument.
- See CallBy Example.

```cpp
Void A::set_data(const HugeClass& d)
{
    data = d;
    d.reset(); // ← compile error!
}
```
Chapter Summary

- Abstract vs Concrete Class: Person_Student_Professor Example
- Constructor and Destructor: Constructor_Destructor Example
- Virtual and function overriding: VirtualFuncs Example
- Static vs Non-static members: Static Example
- Constant vs non-constant members:
  - Function: Person_Student_Professor Example
  - Argument: ValuePointerReference Example
- Value, Pointer, and Reference:
  - ValuePointerReference Example
- Call by Value, Pointer, Reference: CallBy Example