Lecture 8

Inheritance

In previous lectures, we investigated how to define classes by writing all the implementations for the class.

In this lecture, we will examine another alternative: to add capabilities to already-existing classes.

References:

- Textbook chapter 5, pages 171–198.
- Textbook chapter 2, pages 67–69.

Motivation

- Suppose we want a **set of classes** to represent people of a university.
- "People" means many things: Students, Teachers, Graduate Students, Teaching Assistants, Administration staff, etc.
- Clearly, the information we store and operations we need for different type of people are different. So we need **one class for each of them**.
- We will show the prototypes of many functions but the implementation of very few. We purposely don't show the implementation because (1) they are trivial, and (2) they are not needed in understanding the concepts.
- We will also assume the presence of some types, like EStudentStatus, EDepartment, etc., which are enumerated types. E.g.:

```
enum EStudentStatus { eFullTime, ePartTime, eExchange };
```

- Let's look at the interface of the student and teacher classes.

The student class

```
class TStudent {
public:
    TStudent(string name, string hkid, string birthDate,
        string address, EStudentStatus status, EDepartment dept);
    TStudent(const TStudent& student);
    ~TStudent();
    void SetName(const string &name);
    string GetName() const;
    void SetAddress(const string &address);
    string GetAddress() const;
    void SetDepartment(EDepartment dept);
    EDepartment GetDepartment() const;
    bool EnrollForCourse(const TCourse & course);
    bool DropFromCourse(const TCourse &course);
private:
    string _name, _address, _hkid;
    EDepartment _dept;
    vector<TCourse *> _enrolled;
    ...
};
```


The teacher class

```
class TTeacher {
public:
    TTeacher(string name, string hkid, string birthDate,
        string address, ERank rank, EDepartment dept);
    TTeacher(const TTeacher& teacher);
    ~TTeacher();
    void SetName(const string &name);
    string GetName() const;
    void SetAddress(const string &address);
    string GetAddress() const;
    void SetDepartment(EDepartment dept);
    EDepartment GetDepartment() const;
    bool OfferCourse(const TCourse &course);
    bool DropCourse(const TCourse &course);
private:
    string _name, _address, _hkid;
    EDepartment _dept;
    vector<TCourse *> _coursesOffered;
    ...
};
```


The problem

- When we look at the *TStudent* and *TTeacher* interfaces, we find that there are **many similarities**.
  E.g., both have *SetName*, *GetName*, *SetAddress*, *GetAddress*, …
- If we continue to write the implementations for these member functions, **the code will also be almost the same**.
- Why this is a problem?
  - Our efforts to create these functions must be repeated.
    Why care? We can copy and paste! But…
  - The compiled program will contain **multiple functions** for **exactly the same purpose**, with exactly the same **implementation**.
    This seems more like a problem: we don't want wastage.
  - Whenever we find a bug of *GetName*, *SetName*, etc., we need to **modify all our classes** (since all classes probably have the bug).
    This is a real problem! It is very complicated and error-prone to copy, paste, search and replace all the different versions.

Inheritance: structure sharing

- To solve the problem, we have to go back to our concepts and ask **why there can be so much duplications**.
- Why both a student and a teacher have a name, a HKID, a department, an address, a birthdate, …?
- Of course… they are both person in the university. **All people in the university will have all these characteristics**.
- Why don't we **consider them as persons first**, and **teacher / students later**? Can we then move all the common parts into a "person class"?
- The basic idea of **inheritance**: we want common things to be extracted out for **easy reuse of part of the interface and implementations**.
- Note: we are not creating a class with no meaning. **TPerson has its own abstraction**: it represents any person related to the university.
  We can always dream up an abstraction. But its existence is important: it tells us what the class should look like, and how it relates with other classes.

TPerson

```
class TPerson {
public:
    TPerson(string name, string hkid, string birthDate,
        string address);
    TPerson(const TPerson& person);
    ~TPerson();
    void SetName(const string &name);
    string GetName() const;
    void SetAddress(const string &address);
    string GetAddress() const;
    string GetId() const;
    void Print() const;
private:
    string _name, _address, _birthDate;
    string _hkid;
};
```

It is a simple class by itself: you know how to implement it.

It is perfectly legal to create a TPerson object.

Revised TStudent

```
class TStudent: public TPerson {
public:
    TStudent(string name, string hkid, string birthDate,
        string address, EStudentStatus status, EDepartment dept);
    TStudent(const TStudent& student);
    ~TStudent();
    void SetDepartment(EDepartment dept);
    EDepartment GetDepartment() const;
    bool EnrollForCourse(const TCourse & course);
    bool DropFromCourse(const TCourse &course);
private:
    EDepartment _dept;
    ...
};
```

- Note the first line. "**class** *TStudent*: **public** *TPerson*" says the *TStudent* class will inherit **all properties** of TPerson.
- Then it adds a few members of its own.

Revised TTeacher

```
class TTeacher: public TPerson {
public:
    TTeacher(string name, string hkid, string birthDate,
        string address, ERank rank, EDepartment dept);
    TTeacher(const TTeacher& teacher);
    ~TTeacher();
    void SetDepartment(EDepartment dept);
    EDepartment GetDepartment() const;
    bool OfferCourse(const TCourse &course);
    bool DropCourse(const TCourse &course);
private:
    EDepartment _dept;
    vector<TCourse *> _coursesOffered;
    ...
};
```

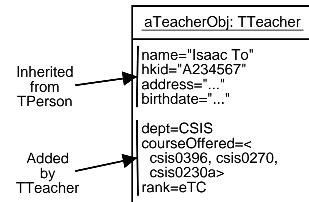
Note how inheritance **removed most of the code duplications**.

Why not let TPerson has a department? That reduces the usefulness of TPerson: some employee might not have a department.

Think: what if people are allowed to have 2 departments?

What an object looks like?

We draw objects like this:



Such diagrams are called UML (Unified Modelling Language) Object diagrams.

This shows that TPerson is actually part of the TTeacher object, and has all the fields. So **TPerson functions can be used unchanged** in TTeacher.

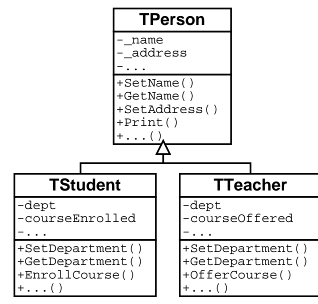
Note: the object **does not contain member functions**: Member functions are not per-object information. Instead, they are functions that can be used to manipulate all objects of the same class.

Graphically showing class relations

We usually want to show relations between classes graphically: **writing out all the code is too tedious and unreadable**.

We represent our class design like the diagram on the right.

... called UML class diagrams.



An **empty arrow** denotes the inheritance relationship between the classes.

The - and + shows the "visibility", meaning **private** and **public** resp.

The two regions are for data members and member functions. Either can be omitted if not important.

Terminology in Inheritance

- If a class D inherit from another class B, **any object of D must have all the property of B**, and **can be used as a B**.
- In other words, **all the contracts between B and the user also apply** between D and the user.
- We call B the **Base class**, and D the **Derived class**.
- The relationship is usually said to be an "**is-a**" relation: **every instance of the derived class is also an instance of the base class** (Every student is a person).

  What is meant by "is also an instance of ..."? It just means the same thing as the above definition: "can also be used like an instance of ...".
- We say "**D is-a B**", although what we really mean is "**Every instance of D can be used in exactly the same way as an instance of B**".

  It is very important to understand what this "twist of words" actually means. Otherwise you get wrong class hierarchy easily.

Example wrong interpretation

A famous example of wrong "is-a": deriving Square from Rectangle.

```
class TRectangle {
public:
    TRectangle(double x = 0,
        double y = 0);
    void SetXY(double x, double y);
    void ScaleXY(double sx, double sy);
private:
    double _x, _y;
};
```

```
class TSquare: public TRectangle {
public:
    TSquare(double x = 0):
        TRectangle(x, x);
    // SetXY?!
    // ScaleXY?!
}
```

So **a square is not a (mutable) rectangle**!

i.e., the class hierarchy is wrong.

Reason: **a square cannot support all the operations possible for rectangle**, i.e., cannot adhere to **the same contract**.

What about the reverse, i.e., deriving Rectangle from Square? Does it break contract?

Constructing derived class

So much for the interface. Now let's see how to make the **definitions for the constructor**.

```
TPerson::TPerson(string name, string hkid, string birthDate, string address):
    _name(name), _hkid(hkid), _birthDate(birthDate), _address(address) {}

TStudent::TStudent(string name, string hkid, string birthDate,
    string address, EStudentStatus status, EDepartment dept):
    TPerson(name, hkid, birthDate, address),
    _status(status), _dept(dept) {}
```

- We **never initialize fields in base class**. Their initialization is the responsibility of a constructor of the base class.
- We can **specify the constructor to use**, in the same way that we initialize a data member (we use a class name in place of the field name).
- If no base class constructor is specified, the default one is used.
  In this case it would be an error: TPerson has no default constructor.

Destructing derived class

- How about destructors? Do we need to **call the destructor** of the base class?
- Answer: **no**. **Destructor calls are automatic**.
- What is the **ordering**?
- Answer: in **exactly the reverse order as constructors** are executed.
- For constructors, the base classes are constructed before the main object. For destructors, the main object is destroyed before the subparts.
- So if both TPerson and TStudent have destructors, that of TStudent is called before that of TPerson.
- Constructors of data members are called in exactly the same order as the data fields appears in the struct. Destructors of data members are called in the reverse order.

What can be done by derived class

- Member functions of the derived class **can access the public members** (usually, member functions) of its base class.
- Member functions of the derived class **cannot access the private members** of its base class.
- Apart from **private** and **public**, the base class may has **protected** sections.
- Member functions of the derived class **can access protected members of objects of the same type** defined in its base class.
- Member functions of the derived class **cannot access** protected members of objects of other type—even if it is a base type of the object.
- Just like **public**, **protected** data members should be avoided: they make it difficult to change the implementations.
  In other words, all data members of a "proper" class should be private.

Overloading member functions

- **Question**: Can I define a member function in the derived class **with the same name as a member function of the base class**?
- **Answer**: Yes.
- **Question**: What are their relationship?
- **Answer**: Not much. The functions of the derived class will "shadow" the function of the based class so that it is more difficult to call the base class function. But that's it: **the functions are basically unrelated**.
  We call this overloading. This happens even if all arguments are the same.
- In particular, the **base-class functions will never call the derived-class function**.
  Later we will learn another kind of member functions, called virtual member functions, which allows base class functions to call that of the derived class.
- It is similar to have a local variable shadowing another variable. It can be confusing, and is **not usually useful**.

Examples

```
class TB {
public:
    void MyMethod1() { MyMethod2(); }
    void MyMethod2() { cout << "TB" << endl; }
};

class TD: public TB {
public:
    void MyMethod2() { cout << "TD" << endl; }
};

int main() {
    TD d;
    d.MyMethod2(); // Call TD::MyMethod2
    d.TB::MyMethod2(); // Call TB::MyMethod2 explicitly
    d.MyMethod1(); // Will call TB::MyMethod2 indirectly!
}
```

Understand it this way: *d.MyMethod1*() calls MyMethod1 for the TB part of d, so it always calls the methods of TB.

Using Base and Derived class together

- We stated that if “**D is-a B**”, then “**Every instance of D can be used in exactly the same way as an instance of B**”.
- It means that at any place of the program, **if an object of class B is expected, we can also supply an object of class D**.
- We call this **interface consistency**: B and D has the same interface.
- **The reverse is not true!** If you have a place that requires a D, you cannot give a B instead.

```
class TB { ... }
class TD: public class TB { ... }
int f(TB);
int g(TD);
```

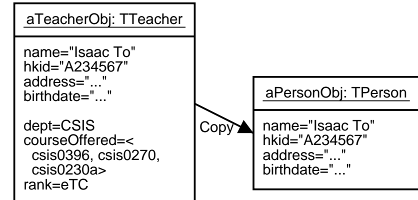
```
void someFunc() {
    TB b = TD();      // Valid, do slicing
    TD d = b;         // Invalid
    TB *pb = new TD;  // Valid
    TD *pd = new TB;  // Invalid
    f(d);             // Valid, do slicing
    g(b);             // Invalid
}
```


Slicing

What will happen when a **derived class object is copied to a base class object**? (Like *TB b* = *d*;)

The **copy constructor** (or assignment operator) of the **base class** (not the derived class!) is invoked to perform the copying.

The result is usually **a copy of a “slice” (part) of the object**, so we call the copying operation “**slicing**”.



What happens on pointers?

- What happens if we have **a pointer to TD** (say, pd) and we **cast it to a pointer to TB** (say, pb)?
- The pointer will then **point to the TB part** of the object.
  Note that now it is no longer pointing at a TD-type object. Instead, it points to a TB-type sub-object of the TD object.
- Technically, this happens only if the cast is a *static_cast* or is a coercion (automatic cast). If you do a *reinterpret_cast* or use the C casting operator, the **address of TD will be used as a pointer to TB.**
  This works as long as you are not doing multiple inheritance. The address of the first base class of an object always has the same address as the object itself.
- Since pb is pointing to a TB object and not a TD object, **we cannot call a TD operation** from pb.
- And since pb is pointing to a TB object, *pb*→*method*() **calls the method of TB** even if TD overloads it.

Example

```
class TB {
public:
    void MyMethod1() { MyMethod2(); }
    void MyMethod2() { cout << "TB" << endl; }
};
class TD {
public:
    void MyMethod2() { cout << "TD" << endl; }
    void MyMethod3() { cout << "TD" << endl; }
};
int main() {
    TD *d = new TD;
    TB *b = d;// Coerce to a pointer to TB
    d→MyMethod2(); // Call TD::MyMethod2
    b→MyMethod2(); // Call TB::MyMethod2!
    b→MyMethod3(); // Compile error!
}
```

This is strange until you understand b is pointing to a TB subpart of TD.

Polymorphic Substitution Principle

- If we have a derived type object d, then we have another base type object b, which behaviour on B’s member functions is exactly the same as d.
- In other words, **the behaviour of the same function applied to d and to b is the same**—at least conceptually.
- This is called the **Polymorphic Substitution Principle (PSP)**.
- Currently this is **trivially true**: when d is interpreted as a B typed object, it is really one—all the functions are those defined by B.
- But later when we learn how to **override** member functions, it is important to **keep this behaviour**. I.e., we have to be careful that the new implementation means the same thing—at least conceptually.
- PSP is the **contract between the base-class and the derived class**.
  No guarantee that a base class will work if PSP is violated.

Conclusion

- Now we know that we can define derived class D from a base class B.
- It is very **similar** to that **D contains a type-B data member**, except…
  - a type D object can **directly use all the public members** of B.
  - a type D object has the interface of type B. **Casting** (in fact, extracting the class B subpart) is **automatically done**.
- But this is usually not enough. Sometimes we want a **pointer to type B** to behave differently if it is pointing to an object of type D.
- Technically, **we can do that**: we can have a data member in B which is a function pointer, and modify that in the constructor of D.
- But it is **tedious** (have to write function pointers ourselves), **error-prone** (someone can accidentally modify the function pointer) and **unclean** (it breaks data abstraction). We will learn better way to achieve it.