

Motivation

Lecture 10

Multiple Inheritance and Aggregation

We have learnt that inheritance can be used to allow classes to reuse the interface and implementation of another class.

But what if we want to reuse the interface and implementation of **two** other classes? We will see how **multiple inheritance** can be used in these cases. We will also see some **alternatives** to multiple inheritance.

References:

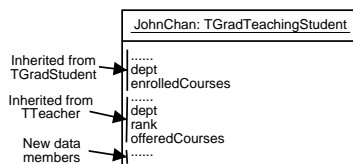
- Textbook Chapter 6 up to page 316 (ignore all references to Eiffel and Smalltalk if you don't have time to read all of them) except RTTI (pp303–306).

PMOOP(0396A)

PMOOP(0396A)-10.1

The same old idea: inheritance means “contain one”

- The basic idea of inheritance: **the object contains a base-class part**.
- E.g., if *TGradStudent* inherits from *TStudent*, then *TGradStudent* contains a part which is a *TStudent*.
- The basic idea of multiple inheritance (MI): it's just the same!
- E.g., if *TGradTeachingStudent* inherits from *TGradStudent* and *TTeacher*, then *TGradTeachingStudent* contains a part which is a *TGradStudent*, and a part which is a *TTeacher*.



PMOOP(0396A)-10.2

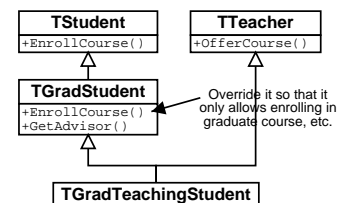
Code and class diagram

Code:

```
class TGradStudent
: public TStudent
{
    ...
};

class TGradTeachingStudent
: public TGradStudent,
  public TTeacher
{
    TGradTeachingStudent()
    : TGradStudent(...),
      TTeacher(...)
    {
        ...
    }
};
```

Class diagram:



PMOOP(0396A)-10.3

What if interface clash??

But things are not that simple... the two classes have things that clash.

- Both TStudent and TTeacher probably provide a Print function**, with exactly the same signature. What will happen?
- This won't be a problem when the class is defined.** But when you use the function, you will need to disambiguate the calls, e.g., *myGradStudent.TGradStudent::Print()*.
- More likely, you would like to **define a new Print function** for the *TGradTeachingStudent* class. The function will override the methods of **both** base classes.
- But **what if the functions are actually different things** (i.e., have different contracts)? Then the two classes **really** clash and can't be combined using MI.

PMOOP(0396A)-10.4

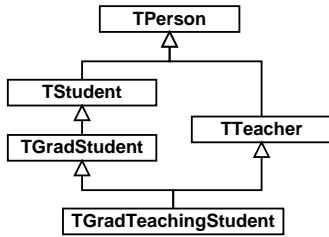
Problem of Diamond-shaped inheritance

But there is a much more serious problem waiting for us...

- TGradStudent* and *TTeacher* are both derived from *TPerson*.
- That means the *TGradStudent* part and the *TTeacher* part will **both** contain all *TPerson* data like name, address, ...
- Whenever we update, we must update **both!** So we have to **override** the *GetName*, *SetName*, ... functions to update both. And...
- We **can't use the object as a TPerson!!** We must treat it as two. E.g., to call a function that modify *TPerson*, we must call it once for each part.
- This is a major problem: **it is the same person**, why the hell we need multiple places storing it?! Is it possible to store only once?
- But... to use the object both as a grad student and a teacher, **we must have some part looking exactly the same** as the a grad student and some part as a teacher!

PMOOP(0396A)-10.5

Our real class hierarchy



- We call such types of MI a “diamond-shaped” MI.
- The **real reason** why we have multiple *Print* function is that it is inherited via **different paths**.
- Note also that **it is difficult to write Print**. We can't just call the TGradStudent::Print and TTeacher::Print—each will print the TPerson once.

PMOOP(0396A)-10.6

Virtual base

- Is there really **no solution**? Well... there is.
- The intermediate classes like *TStudent* and *TTeacher* can derive from the base classes in a special way: **virtually**.
The class that directly inherit the base class must make this decision.
- Just like virtual functions, **virtual base class** adds some run-time flexibility to the object.
- In particular, the *TStudent* and *TTeacher* no longer **expect a copy** of the base class at a fixed-place. Instead, they contain **pointers to base-class parts** at fixed place.
- Each object of derived classes will have **one base-class part**.
- Whenever we have a pointer of TTeacher and **need to find the TStudent or its data**, the pointer is used.

PMOOP(0396A)-10.7

Code

```

class TPerson
{
    ...
};

class TStudent
: virtual public TPerson
{
    ...
};

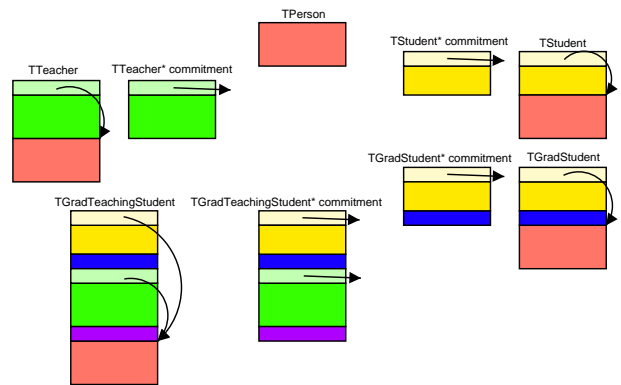
class TTeacher
: virtual public TPerson
{
    ...
};

...// No change for TGradStudent
...// and TGradTeachingStudent
  
```

- Note that **virtual is defined in the intermediate class**, not in the class doing MI, and not in the ultimate base class.
- It is impossible to do it in the class doing MI: the object structure would be fixed by then.
- It is nice if we can define it in the **base**, but C++ doesn't allow that.
- Why no need to modify grad student class to inherit the student class virtually? We **do not perceive** the possibility of a class inheriting Student via **two different paths**.

PMOOP(0396A)-10.8

Object structure with Virtual Base



So virtual means “commit pointer only, implementation in final class”.

PMOOP(0396A)-10.9

Conceptual model

- It is important to see that **the only TPerson object** is shared by the TStudent base class and the TTeacher base class.
- In other words, a class function of the TStudent (or TTeacher, or any other) class **must not consider itself as the sole owner** of a TPerson.
- There is a more interesting consequence: the TPerson object **must be considered to be owned by the final object**, not the intermediate class.
- Consequently, **only the final class** may initialize the object, **not any intermediate class!** E.g., constructor of TTeacher must not initialize TPerson if it is called as part of TGradTeachingStudent construction.
- This needs compiler support: we have **no way to ask some code not to be executed** if it is executed when initializing a derived class.
Initialization of virtual base (TPerson) is skipped when initializing intermediate class (TTeacher) part of derived class (TGradTeachingStudent).

PMOOP(0396A)-10.10

Initialization of virtual base: example

```

class TStudent: virtual public TPerson {
    TStudent(string name, string hkid, string birthdate, ...)
    : TPerson(name, hkid, birthdate, ...), // Not used by derived classes
      _uNumber(nNumber), ... {}
    ...
};
// Similar for TTeacher
class TGradStudent: public TStudent {
    TGradStudent(string name, string hkid, string birthdate, ...)
    : TPerson(name, hkid, birthdate, ...), // Must have this!!
      TStudent(name, hkid, birthdate, ...), // Student part still need init
      _advicer(advicer), ... {}
}
class TGradTeachingStudent: public TGradStudent, public TTeacher {
    TGradTeachingStudent(string name, string hkid, string birthdate, ...)
    : TPerson(name, hkid, birthdate, ...), // Must have this!!
      TGradStudent(name, hkid, birthdate, ...),
      TTeacher(name, hkid, birthdate, ...) {}
}
  
```

PMOOP(0396A)-10.11

Overridden member functions with virtual base

What about **member functions** of the base class?

Without virtual base class, such functions must be overridden to call the function for all paths to the base. Sometimes this is indeed impossible.

- If the function is not overridden in either intermediate class: no modification needed, since there is only one ultimate-base part.
- What if intermediate class has overridden the function?
- If the ultimate-base function is not called in the intermediate class: simple—just call each of the intermediate class implementation.
- If the ultimate-base function is called by both intermediate classes: impossible. If we call each intermediate class function, the ultimate-base function is called twice.
- What to do then? The solution involves **turning the function into either easy cases**. Let's see how to implement *Print()* correctly.

PMOOP(0396A)-10.12

Overriding member functions: code

```
class TPerson {
public:
    void Print() { // not virtual
        ... // print itself
        PrintPostHook();
    }
protected:
    virtual void PrintPostHook() {}
};
class TStudent
: virtual public TPerson
{
protected:
    virtual void PrintPostHook() {
        ... // print student part
    }
};
// Similar for TTeacher

class TGradStudent
: public TStudent
{
protected:
    virtual void PrintPostHook() {
        TStudent::PrintPostHook();
        ... // print grad student part
    }
};
class TGradTeachingStudent
: public TGradStudent, public TTeacher
{
protected:
    virtual void PrintPostHook() {
        TGradStudent::PrintPostHook();
        TTeacher::PrintPostHook();
    }
};
```

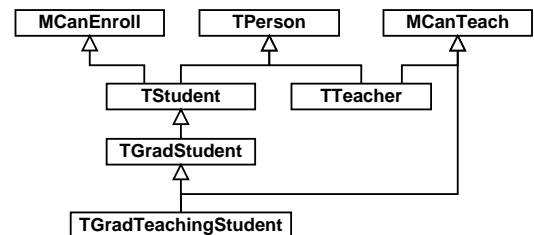
PMOOP(0396A)-10.13

A better abstraction: Mix-in

- It is clear that **MI adds considerable complexity** when inheritance is diamond shape. Is there a simpler way that avoid it?
- No, if the requirement is that a *TGradTeachingStudent* can be used both as a *TGradStudent* and a *TTeacher*.
There are other solutions, but they are even more complicated. Every language feature is there for some reason.
- On the other hand, **many times we don't really need that**. For example, we might just need *TGradTeachingStudent* to be able to teach.
- What is "able to teach"? There is a class (e.g., *MCanTeach*) **which provides functions** like *OfferCourse*, and **data** like *courseOffered*.
- We call it a **Mixin** class: we won't be directly creating such a class. Instead, it is always used to mix with another class.
- If this is the case, **we can avoid the diamond-shape MI** and simplify our design.

PMOOP(0396A)-10.14

New class diagram



- All teaching functionalities are in *MCanTeach*. All course enrollment functionalities are in *MCanEnroll*.
- Note that **no diamond-shape inheritance** is needed.
- But now *TGradTeachingStudent* cannot be used as *TTeacher*.
- There are some more problems, we will resolve it next time.

PMOOP(0396A)-10.15

Limitations to inheritance

- Inheritance is a **static**, i.e., **compile-time**, relation.
- When an object is **created**, its capability is **fixed**.
- That means such an object cannot take new functionality **during run-time**. E.g., a *TStudent* cannot become a *TGradStudent* later.
The alternative is to destroy the *TStudent*, create a *TGradStudent*, and get back to the original state. But that "get back to original state" is usually very complicated, e.g., many objects may have pointers pointing to the original object.
- Inheriting to a class also **commit to that exact class**.
- E.g., if you inherit the *TStudent* class and modify it to *TGradStudent*, the **same change cannot be applied on another derived class** of *TPerson* or *TStudent*.
It seems really unlikely that we need that. Consider another example: if we derive *TSquare* to *THollowSquare*, the same change cannot be applied on another class derived from *TShape*, or even *TSquare*!

PMOOP(0396A)-10.16

Aggregation

Another simple, basic OOP mechanism can be used to create solutions for all these problems: **containment** or **aggregation**.

- An object can **contain one or more pointers** that point to objects of some other classes.
- Since it is a **pointer**, **polymorphism works**. We don't commit to a single class, but can use any derived class instead.
- And it is a **data field**, you can **change** it to some other values **at run time**. We say that the **lifetimes** of the objects are independent.
- But if you use containment, **the interface is not reused**.
E.g., if our *TGradTeachingStudent* class contains two pointers one to a *TGradStudent* and the other to a *TTeacher*, you cannot use a *TGradTeachingStudent* as a *TTeacher* or *TGradStudent*.
- But as illustrated in the Mixin example, **sometimes it is not needed**.

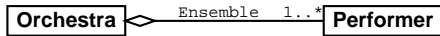
PMOOP(0396A)-10.17

Class diagram for aggregates

Aggregations are “has-a” relationships: an object is solely in the control of a holder object, and is considered **part of** the holder.

In class diagrams, aggregates are shown using **diamonds**. A **hollow diamond** means the **lifetime** relation is **weak** (i.e., can change to another). A **filled diamond** means the lifetime relation is **strong**.

- **Aggregate:** Orchestra owns any number of performers. Orchestra can change performers.



- **Composition:** An airplane owns the 2 to 4 engines, and any number of seats. A plane can't change seats or engines.



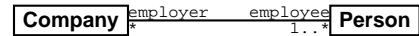
PMOOP(0396A)-10.18

Association: Non-ownership relations

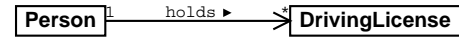
Sometimes it is useful for an object to keep pointers of other objects, even though **something else** (e.g., other objects, functions, etc) is **owning** it.

We call such relationships **associations**. In class diagrams, we use a line **without diamond**. Navigability (which object contains the pointers) is shown by arrows. We can make it clearer using role/association names.

- **Example 1:** A person can work as employee of any number of companies. A company can employ one or more persons.



- **Example 2:** A person can hold any no. of driving licenses, while each license is held by one person. We can find all licenses held by a person.



PMOOP(0396A)-10.19

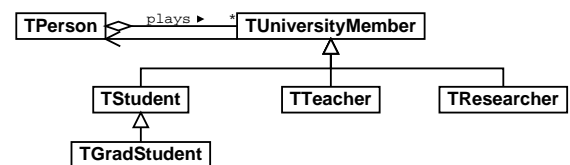
Dynamically changing roles

Let's get back to our university example.

- In the Mixin solution, each person has some **capabilities**, and we assume that the capabilities is **fixed**.
- In a more **dynamic** environment, capabilities **change** during the execution of our program.
- In such scenario, **each person has a changing set of roles**.
- So roles are run-time entities with a **different lifetime from the person object**. They **must be separate objects**.
- Each role has a different set of operations, so **they must be of different classes**.
- How a person can hold all **role objects of different type?** We use **polymorphism**, i.e., all roles have a common base class.

PMOOP(0396A)-10.20

Class diagram



- TUniversityMember now becomes the base role class.
- The role played can be students, teacher, researcher, etc.: derived classes of TUniversityMember.
- A person can play any number of roles.
- Each role is for one person only.
- But there is one problem remaining: **how to find the role we need?** Again, we will see it in the next lesson.

PMOOP(0396A)-10.21

Conclusion

- Multiple inheritance can be used to achieve a large degree of code and interface reuse.
- The most important use of multiple inheritance is to combine different interfaces from different base class.
- Inheritance can be too inflexible, especially when roles of objects change during run-time.
- Such problems can be avoided using pointers to other objects.
- Based on the lifetime relationship and ownership, such pointer containments are sub-divided into aggregates, compositions and associations.
- Such design decisions are usually not seen in the program. They are documented in class diagrams instead.

PMOOP(0396A)-10.22