

Motivation

Lecture 9

Dynamic Binding and Polymorphism

So far, our classes are very clean: once we write a class, nothing of the class can be changed, and the class can only be accessed through public member functions.

We will complicate this picture by introducing ways to modify the behaviour of the subpart when we derive a new class. We will see why we want to do such things, how they are accompanied, and when and when not to do it.

References:

- Textbook: Chapter 5

PMOOP(0396A)

PMOOP(0396A)-9.1

- Suppose we have a pointer *pperson* of type *TPerson **.
- We know that *pperson* may be pointing to an object which is a *TPerson* itself, or may be pointing to the *TPerson* **subpart** of an object which is a *TStudent*, *TTeacher*, ...
Or it might be pointing to nowhere: null pointer.
- Suppose we call *pperson*→*Print()*. This **always** end up calling *TPerson::Print*—even if *TStudent::Print* or *TTeacher::Print* is defined.
- What if we want the print operation to “**print the full object**”?
- In other words, is it possible for the **base** class *TPerson* to have a method which **behaviour depends on what type is the full object**?
- In general, such decisions **cannot be made during compilation**.
E.g., *pperson* may be passed to a function as argument, and the exact type depends on how the function is called.

Doing it the crude way

There is one way we can do it: by a **function pointer**.

```
class TPerson {
public:
    TPerson();
    void Print() { _printFn(this); };
    ...
protected: // Ugly
    void (*_printFn)(TPerson *);
private:
    ...
};
void PrintPerson(TPerson *pPerson) {
    // Print TPerson
}
TPerson::TPerson() {
    ...
    _printFn = PrintPerson;
}

class TStudent: public TPerson {
public:
    TStudent();
    ...
};
void PrintStudent(TPerson *pPerson)
{
    TStudent *pStudent =
        static_cast<TStudent *>
        (pPerson);
    // Print student
}
TStudent::TStudent() {
    ...
    _printFn = PrintStudent;
}
```

PMOOP(0396A)-9.2

PMOOP(0396A)-9.3

Why this is not good

Can we **improve the scheme**? There are some fixable problems:

- There is an ugly **protected data member**. It would be better to change it to private, and initialize it through constructor arguments to *TPerson*.
- When there are many such functions, it is not economic to have **all objects storing a copy of the list of functions**. It is better for it to be **shared** by the whole class.
- We need to write functions **external** to the class. This is because a normal function pointer cannot accept a class method. We can change it to a “member function pointer” like `void (TPerson::*)()`.

But the **primary problems** are **unfixable**:

- This is very **tedious**. It is better done by the language.
- This does not express that the whole class use the same function.

Direct compiler support: virtual member functions

- We can ask the compiler to automatically perform all the things we have just done. We call this **dynamic binding**.
Binding a function name to a function implementation only at runtime.
- All we need is to **tell in the base class definition** that we want such automatic support: **define the function to be virtual**.
- E.g., **virtual void Print() const**;
- The base class can then provide an **implementation** of the method. This implementation can be **overridden** by derive class.
- Just like our hand-coded version, some place of the object is reserved for storing a **pointer to (member) function**, which is used whenever the function *Print* is called, so it always print the full object.
- The **implementation** of the function of each class is still available, e.g., *TPerson::Print* and *TStudent::Print*.
You can use them if you don't need dynamic binding.

PMOOP(0396A)-9.4

Example

Since we have direct compiler support, our code becomes much simpler.

```
class TPerson {
public:
    virtual void Print() const {
        // Print the TPerson
        // object in "this"
    }
    ...
};

class TStudent: public TPerson {
public:
    virtual void Print() const {
        // Print the TStudent
        // object in "this"
    }
    ...
};
```

Technically, the **virtual** keyword of *TStudent::Print* is not needed. But it is usually left there to remind readers that function overriding is done.

Now, if we have a pointer *p* of type *TPerson ** pointing to the *TPerson* part of a *TStudent*:

- *p*→*Print()* calls *TStudent::Print*.
- *p*→*TPerson::Print()* calls *TPerson::Print*.

PMOOP(0396A)-9.5

Incremental overriding

At many times, we need to override a method by doing a few more things. In this case, we want to be able to call the original method. E.g.:

```
class TStudent: public TPerson {
public:
    virtual void Print() const {
        TPerson::Print(); // First print it like a normal person
        // before printing the things specific to a student
    }
    ...
};
```

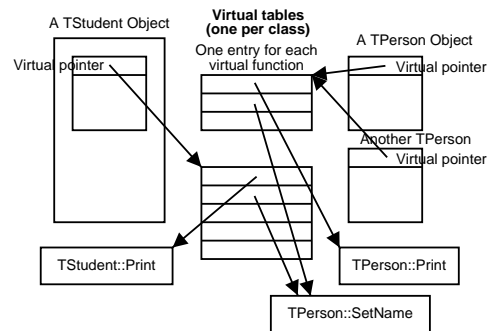
This is possible as long as *Print* is a **public** or **protected** member of *TPerson*.

NB: **private** members can still be overridden, but you cannot chain the call like this.

PMOOP(0396A)-9.6

How it is implemented?

What is the **internal representation** of an object with virtual function?



Question: What happens when we call $p \rightarrow Print()$?

PMOOP(0396A)-9.7

How costly is dynamic binding?

Dynamic binding has several costs:

- Memory for **one extra pointer in each object (vpointer)**. This pointer is needed only if the object has a part which has a virtual function.
- Whenever we call a virtual function, we need to **dereference** the vpointer twice: once to find the vtable, once to find the function itself.
- Because the called function is not known at compile time, **virtual member function calls cannot be inlined**.
- Each class with a virtual function needs some memory to store a table (**vtable**) which holds all function pointers. It is needed whether or not any method is overridden.
- This **needs to be initialized** when the program starts to execute, so **startup time is increased**, especially if you have a lot of such classes.

But most of the time the cost is **not too prohibitive**.

PMOOP(0396A)-9.8

Words of caution: avoid copy!

- In all our discussion, we are talking about a situation in which **we have a pointer to object**. What if we have an object itself? E.g.,

```
TStudent student;
TPerson person = student;
person.Print(); // Call which Print?
```

- Recall that here we do copying: the **copy constructor** of *TPerson* is used to create a new *TPerson* object. So *person* is **not a TStudent at all!**
- Now it should be clear that *TPerson::Print* will be called.
- How about references? E.g.,

```
TStudent student;
TPerson &person = student; // Just an alias
person.Print(); // Call TStudent::Print.
```

In other words, dynamic binding is done only for pointers and references.

PMOOP(0396A)-9.9

Polymorphism: consequence of dynamic binding

- Since now we can be sure that **the member function called for a pointer to TPerson * part of a TStudent** always end up calling functions of *TStudent*, we can be a bit more lousy in our wording.
- We can directly say that **the pointer of type TPerson * is pointing to a TStudent**.
- Under this new wording, **a pointer of type TPerson * can point to any object of class being a derived class of TPerson**—including *TStudent*, *TTeacher*, etc.
- We call this **polymorphism**: “multiple shape” in the same object. Or rather, multiple shape in the same pointer or reference.
- Polymorphism is nothing new in implementation (it just means that we do only overriding, no overloading). But it allows us to view the pointers in a **simpler and more flexible way**.

PMOOP(0396A)-9.10

Obligation of the derived classes

Now, **why the writer of TPerson knows to make Print virtual?**

- Of course, *TPerson* provides an implementation of *Print* to **achieve certain task**: to print out all information in the person record.
- The writer of *TPerson* must also be expecting that his class will be reused, e.g., by the writer of *TStudent* and *TTeacher*.
- More interestingly, **the writer of TPerson acknowledge that his default implementation may not be good enough** for other users of the class. So he makes it **virtual** for others to override. Yes, the writer of a base class needs to know all these things. Without such knowledge, there is every possibility for derived classes to break the assumptions of the base classes.
- The *TPerson* class, and its users, **assume that calling Print() for a TPerson object will achieve the effect** specified by *TPerson*.

PMOOP(0396A)-9.11

PSP with dynamic binding

- We want the **Polymorphic Substitution Principle** to hold—even after the derived class overrides some methods.
- Let's look at PSP again:

If we have a derived type object d, then we have another base type object b such that the behaviour of the same function applied to d and to b is the same—at least conceptually.

- What it means “conceptually” is that it should have the same semantics: should do the things specified by the base class.
- Now it is clear that **there is a contract between the base class and the derived class**: it must supply the right sort of functions.
- If the contract is **violated**, there is **no guarantee that the other functions using the base class will still be correct** when given a pointer to the derived class.

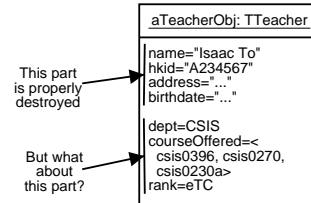
PMOOP(0396A)-9.12

Destroying a derived class through a base class pointer

What will happen if we have the following code?

```
TTeacher *teacher = new TTeacher(...);
TPerson *person = teacher; // Okay: polymorphic
delete person; // What is destroyed?
```

The answer: it won't work!! Polymorphic behaviour comes from virtual functions, while we have nothing virtual related to destruction...



PMOOP(0396A)-9.13

Fixing the problem

If we think that a base class pointer can point to a derived class object (i.e., use polymorphism), this is a big problem.

Later we will see that we usually create an object of derived class, give it to someone that expect a base class pointer, and forget it.

To fix the problem, we define a **virtual destructor** in the base class.

This is needed **even if there is nothing to do in the destructor**: the sole purpose is then to tell that destruction must be dynamically binded. E.g.,

```
class TPerson {
public:
    ...
    virtual ~TPerson() {}
};
```

On the other hand, the derived class **need not provide destructor** if there is nothing to be done.

PMOOP(0396A)-9.14

Implementation and Interface reuse

With polymorphism, inheritance achieves **two purposes**: it allows **implementation reuse**, and it allows **interface reuse**.

- **Implementation reuse**: whenever we derive a new class from the base class, all the implementations are inherited, and need not be written again.
- **Interface reuse**: whenever we have a base class pointer, we can assume all the base class interface is available for use.

In many object-oriented designs, **interface reuse is more important** than implementation reuse.

- We may want to make sure that every object of a base class will have a certain function, but **no default implementation** is provided.
- The sole purpose of such member functions in the base class is to make sure that **we can call them**.

PMOOP(0396A)-9.15

More about interface reuse

- Note that the existence of a common base class is **very important**.
- Suppose we have two classes **TPerson** and **TStudent**, that TStudent contains **all data members and member functions** of TPerson, but TStudent is not declared as a TPerson subclass.
- Then a **TPerson pointer cannot point to an object of type TStudent**—even if it seems make sense to us!
- If you **expand the words** to the terminology which we have used before, it is clear why it can't. It says **TPerson pointer cannot point to the TPerson subpart of a TStudent object**—which is of course true: there is no such subpart.
- So even if somehow we never **make any instance** of TPerson, we still want it: it defines a common point in which all other classes are derived from.

PMOOP(0396A)-9.16

Example: Pure virtual functions

- In a graphics system, we might have a base class **TShape**. We want to **be able to find the area of any TShape** using member function **Area**.
- We know how to compute the area of a TCircle, or a TSquare, or a TRectangle. But how to compute the area of a TShape?
- But of course... we **shouldn't have a TShape at the first place!** Our graphical objects can be a circle, a square, a rectangle, ...; but we don't expect a "shape" object in our system.
- The best way to implement this is to have **no implementation** for **TShape::Area**. We only want its interface.
We want the interface because, given a **TShape * pshape**, we want to be able to use **pshape->Area()** to find its area.
- Such a function is called a **pure virtual function**. It says "=0" after the "prototype", meaning "must be overridden".
It can be implemented, but such implementation can only be accessed directly: it will never be written into the vtable.

PMOOP(0396A)-9.17

Code: Pure Virtual Function

```
class TShape {
public:
    // Pure virtual function
    virtual double Area() = 0;
};

class TCircle: public TShape {
public:
    ...
    virtual double Area() {
        return _r * _r * M_PI;
    }
private:
    double _r;
};

class TSquare: public TShape {
public:
    ...
    virtual double Area() {
        return _s * _s;
    }
private:
    double _s;
};

class TRectangle: public TShape {
public:
    ...
    virtual double Area() {
        return _w * _h;
    }
private:
    double _w, _h;
};
```

PMOOP(0396A)-9.18

Consequence of pure functions

- A class with at least one pure function is called an **abstract class**.
E.g., TShape is an abstract class, TCircle is not.
- An **abstract class cannot be instantiated**, although you can have a pointer to an abstract class. E.g.:

```
TShape shape; // Invalid
TShape *pshape; // Okay
pshape = new TShape; // Invalid
pshape = new TCircle; // Okay, and is the primary use of TShape
TCircle circle;
TShape &rshape = circle; // Okay
```
- This makes it a great way to make sure that **all derived classes will override the function**: if a derived class doesn't override the function, it can't be used to create objects.

PMOOP(0396A)-9.19

3 types of classes

So far, we have seen 3 different types of classes. Each has advantages and disadvantages, and should be **chosen by individual basis**.

- Classes with **no virtual functions**. Most of these classes don't expect users to use it as a base class. You can still do so, to add some capabilities. But they **must not be used in a polymorphic way**. They are **very efficient**: no extra storage or dereference, inline is possible, etc.
- Classes which has **some virtual functions**. These virtual functions should have the **associated semantics**, and derived classes can be written **conforming** to these semantics. **Virtual destructor** should be provided. The base class itself is still usable. **Less efficient**.
- Classes which has **some pure virtual function**. Like the above, but the methods **must be overridden** before it can be used, thus making sure that the derived classes provide their implementation. The base class itself is cannot be instantiated.

PMOOP(0396A)-9.20

When not to use inheritance

- When **interface reuse is not needed**, i.e., when our new class doesn't have the same interfaces as the base class.
E.g., we probably don't want to inherit a car object from an engine object: we don't need the interface of engine, like "ignite", etc. Use containment instead.
- When **base class has already committed into something that our new class does not want**.
E.g., even if we are having an immutable *TRectangle* object, we probably don't want to inherit from it to make *TSquare* because the base class has committed into two pieces of memory to hold the width and height of the rectangle, while we need only one.
In many such situations, we will instead create a third class to act as the **common base class** of the two classes.

PMOOP(0396A)-9.21

When not to use inheritance (cont'd)

- When we want to **add the same capabilities to** not just one class TB1, but **all derived classes of a class TB** (where TB is derived from). If we subclass from TB1, we will end up having to write a lot of classes.
E.g., suppose we want to add a variant of TCircle which is hollow, a TRectangle which is hollow, etc., then we probably shouldn't derive THollowCircle from TCircle, THollowRectangle from TRectangle, etc. Doing so will cause **code duplications** because code in THollowCircle and THollowRectangle would be the same.
In these scenarios, a better design is needed, which will be taught in the 0402 course.

PMOOP(0396A)-9.22

When not to use dynamic binding

- When **cost of dynamic binding is too large**.
Example: when you create a pointer of 4 bytes which you want to create a million instances, a 4 byte vpointer is probably big.
- When it **doesn't make sense for a method to be overridden**.
Example: if you have a function to return an identifier that is stored in the constructor of the base class, it probably shouldn't be allowed for overriding.
- When the different behaviours of the method **can be easily captured by a simple number**. It is more economical to store it within the object.
Example: if you have a library system and have many kinds of materials, e.g., books, periodical, ... each with a different maximum borrowing period, it is probably better to record the period in the state rather than in a virtual function.

PMOOP(0396A)-9.23

Static data members

- We have seen that the implementation of **dynamic binding** requires some data to be **shared among all objects of the same class**.
In particular, there is one vtable per class.
- How it is **implemented**? Actually, the C++ compiler will **generate one vtable** in each object file that creates a constructor for the object. These vtables are **merged** during static/dynamic linking.
- Is it possible to define **data member** to be shared among the objects? Yes: we can use static data member. E.g.,

```
class TPerson {  
private:  
    static int _numStudents;  
};  
// Somewhere in implementation  
int TPerson::_numStudents;
```

PMOOP(0396A)-9.24

Static members functions

- A static data member is very similar to a global variable (using static allocation), except that it is defined in the class.

To access it, you can use `TPerson::_numStudents`.

Or, within a `TPerson` method, simply `_numStudents`.

- Similarly, you can define a **static member function** by adding a **static** keyword before the declaration. It is very similar to a global function (i.e., needs no object to call).

Such functions have no **this** pointer, so we cannot directly name a data member within such functions.

To call these functions, use `TPerson::MyStaticFunction()`.

- We can call static functions **via an object**, but it is **better not to do so**.
No dynamic binding is done for static functions—even if you call it through an object. It is impossible to define a static virtual member function.

PMOOP(0396A)-9.25