

## Lecture 11

### Object ownership; Run-time Type Identification (RTTI)

In this lecture we will continue to look at various issues when we make an elaborate class scheme like the Mix-in and role-playing class examples.

In particular, we will explore on object ownership and runtime type identification.

#### References:

- Textbook chapter 2: Adopt semantics, pp. 106–111.
- Textbook chapter 6: RTTI, pp. 303–316.

PMOOP(0396A)

PMOOP(0396A)-11.1

#### Concepts of ownership

- **Where the storage comes from?** Two ways:
  1. By **allocation**: when the holder object is created, or when a function that creates a new sub-object is called, the object is created using the **new** operator or calling other memory allocating functions.  
E.g., the memory of our earlier *TIntStack* is acquired by allocation.
  2. By **adoption**: the **user** of the class may **execute a member function**, giving it an object—and the responsibility of its storage.  
E.g., we might write a *TIntStack* constructor which accepts an integer array from the user, and adopt the array from the user. After that the user must not deallocate the memory himself.
- To minimize errors, **consider all objects to have its owner.**
- Note that **only dynamically allocated objects can be given up** to another class: automatically and statically allocated objects can't be freed. Best to consider them owned by the program and the function respectively.

PMOOP(0396A)-11.2

PMOOP(0396A)-11.3

#### Pointers for adoption

How to adopt? E.g., a *TIntStack* constructor that adopts an array:

```
// Adopt array
TIntStack::TIntStack(int array[], int size, int capacity)
: _stackData(array), _size(size), _capacity(capacity)
{ CheckInvariants(); }
```

Such semantics should be clearly documented, like above. This reminds users that **the caller must release its control over the object**. E.g.:

```
int main() {
    int array[10], *dArray;
    TIntStack stack1(array, 0, 10); // Bug!! Can't adopt local var
    dArray = new int[10];
    TIntStack stack2(dArray, 0, 10); // Okay (dynamic var)
    delete[] dArray; // Bug!! Not owning dArray now
}
```

Pointers, not references, should be used for adoption.

PMOOP(0396A)-11.4

PMOOP(0396A)-11.5

#### What about reference-counted objects

- In the last lecture, we have two perhaps confusing relations between classes: **aggregation** and **association**.
- For **aggregation**, the holder class is **responsible for the storage** of the other class.
- E.g., In the *TUniversityMember* example, if a *TPerson* is destroyed, there is no reason to keep the *TUniversityMember*'s around. They should be destroyed.
- For **association**, the two classes are not responsible for the storage of the other class.
- E.g., In the *employer-employee* example, we don't want to destroy a employer object just because an employee left, or vice versa.  
But when an employer is destroyed, we probably needs to find all its employee, and terminate the association, otherwise we have dangling pointers.

- As we have seen, Copy-On-Write can be used to make **multiple objects to share the same sub-object**.
- How we can view the ownership in this case?
- The easiest way to view the picture is that **all the holder objects own the sub-object**, so the sub-object is owned many times.
- There **must** be a **counter** that keeps the number of times the sub-object is owned: a **reference count**.  
It counts the number of objects referring to it, so the name.
- We need special rules for these objects. In particular, **the holder must respect that others may also be owning it**.
- E.g., before destroying the object, it must check the reference count. Depending on its value, it should either destroy it or decrement the count.

#### Differentiation between pointers and references

- We have seen quite a few things that **can be done for both pointers and reference alike**.
- What is the **difference** between a pointer and a reference?
- There is the superficial difference that **pointers need one more level of dereference**. But the difference is only syntactic.  
Adoption uses pointers because of this syntactic difference. We can only give up dynamically allocated variables, so it is more natural to pass pointers.
- The real difference is that **you can have a NULL pointer, but you cannot have a NULL reference**.
- This makes reference passing a **better** choice for **most parameter passing** in function calls: you don't have to check for NULL values.  
Use pointers if you really need a special NULL value, e.g., for allowing some arguments to be omitted.

## Returning pointers and references

How about **return values** that are pointers or references?

- Most such return values would **transfer the ownership** from the function to the user. In this case reference shouldn't be used.  
E.g., the function helps the user to create an object. Like the functions `Clone` and `MakeChangedExpression` in our assignment.
- Sometimes there is **no transfer of ownership**, and the object returned is owned by the class.
- For most such case, **the returned value should be const** (pointer or reference) to maintain object integrity (i.e., invariants).
- The only reason that this rule should be violated is **when there is no invariants about the values of the owned objects**.  
E.g., for a vector of int's, it is okay for a function to return a writable pointer or reference to its contents because the class does not care what is stored in the object that the pointer points to.

PMOOP(0396A)-11.6

## The difficulty to finding types

Let's turn back to our classes that represent people in the university.

- We have a solution that uses **diamond-shaped inheritance**. But when there are more and more classes, it becomes difficult to manage.  
But it is the only way that a type of person can act in place of two types of person.
- We have one alternative which uses **mix-in classes** and avoid diamond-shape inheritance.
- Our last alternative uses **role-playing classes**. Instead of deriving a student class from the person class, we have a pointer to another object which represent the role as a student.

We have fairly concrete idea about how to do diamond-shaped multiple inheritance. But there are **more trouble** on the other schemes.

PMOOP(0396A)-11.7

## Problem using the Mix-in class

Suppose we use a mix-in class to represent the people. Now we want to have **some data structure to store all teachers** teaching a student.

- What should be the data type of each element of the data structure?
- We can't use `TTeacher`: some "teachers" might not be a `TTeacher`.
- `TPerson` is no good either: we **don't know whether a TPerson can offer courses**. And we don't have the needed operations.  
E.g., what to do when the student quit school? How to find all the relevant `MCanTeach` objects and notify them?
- How about `MCanTeach`? This is much more reasonable. But... we get the reverse problem. **How can we find the personal data** from the `MCanTeach` part of the object?  
E.g., what to do when we want to print out all students of a course?  
We can't do a simple `static_cast`: at compile time, you don't know the relative position of a `MCanTeach` part and a `TPerson` part.

PMOOP(0396A)-11.8

## Problem using the role playing class

But is the role-playing class any better? Let's see...

- Now we have a even more basic problem: Given a `TPerson`, how he can offer a course?  
The `TPerson` doesn't support the interface needed to offer a course! Only `TTeacher` does that, but the `TTeacher` is in another object...
- And... this can fail (because the `TPerson` does not have a `TTeacher` role)! How we know that?
- Essentially, how to **find the correct role** that we want, from the set of all possible roles played by the person?
- To complicate the matters, we can usually accept a role that **is a derived class** of the role we want.
- E.g., if we want a student role, and the `TPerson` object plays a `TGradStudent` role, than it should be allowed. How this can be done?

PMOOP(0396A)-11.9

## Dynamic cast

- What we require is a mechanism that **given an object, we can ask whether it is of a particular type or its derived type**.  
This makes `static_cast` unsuitable for the purpose: it just assumes that the object is of the right type, and offset the pointer. If the pointer is of wrong type, the program crash.
- If so, we must also **get a pointer** to the type we want.
- We have a special casting operator for the purpose: **dynamic\_cast**.  
This is the 3rd C++ casting operator we know. We already know `static_cast` and `reinterpret_cast`, and there is also a `const_cast` to turn const things to non-const (used to override missed const's we mention earlier).
- Sometimes we need to cast from base class to derived class: a "down-cast". E.g., from `TUniversityMember*` to `TStudent*`.
- Sometimes we need to cast between two unrelated classes: a "cross-cast". E.g., from `MCanTeach*` to `TPerson*`.

PMOOP(0396A)-11.10

## Dynamic casting of pointers

Suppose `p` is a pointer to a **polymorphic** type (i.e., with virtual functions). Conceptually, `dynamic_cast<T*>(p)` does the following tasks:

- `p` might be pointing to a **sub-part** of a larger object. `Dynamic_cast` finds the **"real" object `p0` and its type `T0`**.
- Check whether `T` is the **same as `T0`**. If so, return `p0`.
- Check whether it is **possible to cast `p0` to `T` using `static_cast`**. If so, return `static_cast<T*>(p0)`.  
This is non-trivial, because this is done at run-time, not compile time.
- If both the above is not possible, either because `T0` does not contain a public part of type `T`, or because `T0` contains multiple public parts of type `T`, **return 0**.  
`Dynamic cast` can also be done to references, like `dynamic_cast<T&>(refobj)`. But we can't have a NULL reference in case it fails. Instead, it "throws an `bad_cast` exception"—to be examined soon.

PMOOP(0396A)-11.11

### Working with Mix-ins

Now our first problem becomes trivial to solve. E.g., a function that takes a list of *MCanTeach* objects can be printed like this:

```
void PrintTeachers(vector<MCanTeach> teachers) {
    for (int i = 0; i < sizeof(teachers); ++i) {
        TPerson* p = dynamic_cast<TPerson>(teachers[i]);
        assert(p); // all MCanTeach should be TPerson!
        p->Print();
    }
}
```

We use **assert** here because we really **don't expect that our system contains an MCanTeach that is mixed with no or multiple TPerson**.

This is why this arrangement is better than passing a vector of *TPerson* and *dynamic\_cast* it to *MCanTeach*: it catches errors easier.

How about role-playing classes? It is more complicated to achieve the effect we want. (The book actually didn't tell it.)

PMOOP(0396A)-11.12

### The role-playing classes

Suppose we have role classes like this:

```
class TPerson; // Forward declaration so that we can write TPerson*

class TUniversityMember {
public:
    TUniversityMember(TUniversityMember&) {}
    TPerson* GetRolePlayer() { return _player; }
private:
    TUniversityMember(const TUniversityMember&) {} // No copying
    TPerson* _player;
};

class TStudent: public TUniversityMember {
public:
    bool EnrollCourse(TCourse* course);
    ...
private:
    vector<TCourse> _courses;
};
```

PMOOP(0396A)-11.13

### The person holding many role-playing objects

How to find a needed role? If we always need *TStudent*:

```
class TPerson {
public:
    TPerson(string name, string hkid, ...) ...
    void AddRole(TUniversityMember* role); // adopt role
    void DelRole(TUniversityMember* role);
    TStudent* FindStudentRole() {
        for (unsigned int i = 0; i < _roles.size(); ++i) {
            TStudent* ret = dynamic_cast<TStudent>(list[i]);
            if (ret)
                return ret;
        }
    }
private:
    ...
    vector<TUniversityMember> _roles;
};
```

PMOOP(0396A)-11.14

### The user code

The user code then looks like this:

```
TPerson* CreateGradTeachingStudent() {
    TPerson* pPerson = new TPerson(...); // Create the person and
    pPerson->AddRole(new TGradStudent); // add the needed roles
    pPerson->AddRole(new TTeacher);
    return pPerson;
}

bool EnrollCourse(TPerson* pp, TCourse* course) {
    TStudent* pStudent = pp->FindStudentRole(); // Find the right role
    pStudent->EnrollCourse(course); // and call enroll there
    ...
}
```

But then there is a maintainance problem: the *TPerson* class constantly needs to be modified to accept find new roles: *TGradStudent*, *TTeacher*, *TResearcher*, ...

PMOOP(0396A)-11.15

### Fixing the problem

There is a better method, using "template member functions". We will examine templates in a few weeks.

```
template<class T> // Allow the function to find any T, not just TStudent
T* TPerson::FindRole() {
    for (unsigned int i = 0; i < _roles.size(); ++i)
        if (T* ret = dynamic_cast<T>(list[i])) // Find a T, not a TStudent
            return ret;
    return 0;
}
```

Now, for the user code to find a *TStudent* role, he does the following:

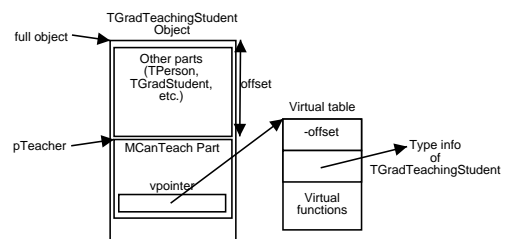
```
bool EnrollCourse(TPerson* pp, TCourse* course) {
    TStudent* pStudent = pp->FindRole<Student>();
    pStudent->EnrollCourse(course);
    ...
}
```

PMOOP(0396A)-11.16

### How it is implemented: finding the real object

To allow finding the "real" object: the vtbl of each sub-part contains an offset from the real object.

This is why we insist that the from-type of *dynamic\_cast* is polymorphic: only polymorphic class have the needed virtual table.

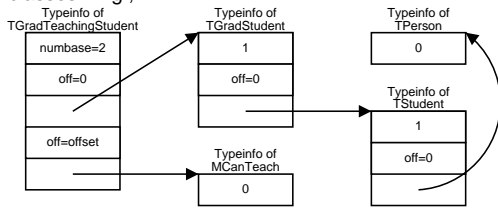


Given an *MCanTeach* part, the program can use the virtual pointer to find the offset, and add the offset to the address to find the full *TGradTeachingStudent* object. The type information of *TGradTeachingStudent* can be found with the Type info pointer.

PMOOP(0396A)-11.17

### How is it implemented: making static cast

The **type info object** is critical in making **static cast** during runtime. It contains a list of pointers to type info objects, and offsets to the object of the base classes. E.g.,



E.g., to cast the *TGradTeachingStudent* back to a *TPerson*, each base class is queried to find a pointer being the same as Typeinfo of *TPerson*. This is done recursively, and the total offset is used to find the final address.

It is correct to think that *dynamic\_cast* is not very fast as it needs to traverse the whole class tree. Some optimization can be done by the compiler, though.

PMOOP(0396A)-11.18

### Type identifiers

- If there is a type identifier for each type, is it possible to retrieve it and use the information in our program?
- You can **find the type identifier** of an object, e.g., `typeid(pStudent)` (need `#include <typeinfo>`). It returns an object of type `type_info`.
- This works even if the object is a **primitive type**. E.g., `typeid(42) ...`
- However, you can do very little with these `type_info` object. You can **compare** for equality using the `==` operator (e.g., `typeid(p) == typeid(p2)`), and you can **get the name** (e.g., `typeid(42).name()` returns `"i"` in g++).
- There is a fixed ordering defined, which can be checked using `typeid(p).before(typeid(q))`.
- There is **no support for class hierarchy navigation**, yet. In g++ 3.0, there is an ABI that you can use the find type identifiers of base classes.

PMOOP(0396A)-11.19

### How type identifiers can be used

- Type identifier identifies a single class. E.g., if you want the `FindRole` function to prefer exact match of roles:

```
template<class T> T* TPerson::FindRole() {
    for (unsigned int i = 0; i < _roles.size(); ++i)
        if (typeid(*list[i]) == typeid(T))
            return dynamic_cast<T*>(list[i]);
    for (unsigned int i = 0; i < _roles.size(); ++i)
        if (T* ret = dynamic_cast<T*>(list[i]))
            return ret;
    return 0;
}
```

- Sometimes we want to make an arbitrary total ordering of all the types (e.g., for making a map). This might makes the `before` function of a `type_info` object useful.

PMOOP(0396A)-11.20

### Important note: Big misuse of polymorphism

Many early OO programmers write something like this:

```
// Misuse of typeid and dynamic cast---don't follow!!
double FindArea(TShape* s) {
    if (typeid(*s) == typeid(TRectangle)) {
        TRectangle* rect = dynamic_cast<TRectangle*>(s);
        return s.GetWidth()*s.GetHeight();
    } else if (typeid(*s) == typeid(TCircle)) {
        TCircle* circle = dynamic_cast<TCircle*>(s);
        return s.GetRadius()*s.GetRadius()*M_PI;
    } else if ...
}
```

The problem of such code is that once you add a new type, your **old** code need to be updated, and the update is **scattered** around the program.

As a rule, if you find code which **explicitly tells what to do for each type**, with **no sensible default that works for all future types**, there must be **something wrong**.

PMOOP(0396A)-11.21

### How that can be avoided

Of course, this can easily be avoid using dynamic binding:

```
class TShape {
public:
    double Area() = 0;
    ...
};

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

Note that code are grouped **in the ways extensions are made**, not in the ways functionalities are implemented.

This also allows extensions to be easily thrown away.

PMOOP(0396A)-11.22