

Independence of implementation

Lecture 7

Writing C++ classes during Object-Based Programming

In the last lecture, we learnt what is data abstraction and data encapsulation. In this lecture, we will have a more complete look at how to write classes.

References:

- Textbook Chapter 3, pages 77–89, and Chapter 4 115–142, ignore paragraphs about Eiffel and Smalltalk.
- For invariants and assertions: page 57–60.
- For Copy-On-Write, pages 161–168.

PMOOP(0396A)

Suppose we decide to modify *TIntStack* so that it uses a less heavy data structure, say a plain array:

```
class TIntStack {
public:
    TIntStack();
    void Push(int data);
    void Pop();
    int Top() const;
    bool Empty() const;
private:
    int *_stackData;
    int _capacity, _count; // # of elements allocated and used in _stackData
};
```

How much modifications we need for the **user** code of *TIntStack*?

Answer: **no modification at all**—we know that without looking at it!
The user code has no way to depend on our implementation!

PMOOP(0396A)-7.1

Difficulty of class designers

- The implementer of the class knows the **data representation** of the class that the user either don't know or don't have access to.
- So **it is the responsibility of class implementors** to make sure the class work 100% according to the specification of the interface.
The user can't help at all: all he can see is a simple abstraction.
- Unlike the programmer writing the application program, **class writers typically don't know the order of calls to the class code**.
In contrast, application programmers usually have perfect idea about the exact order in which functions of the programs are called—at least when the input is as expected.
- In other words, **the code must work** no matter how the user call its functions (any ordering, argument, etc)!
- How the class writer can guarantee such **generality**?

PMOOP(0396A)-7.2

Class invariants

- To make this generality possible, **class designers always expect data within an object to be in a certain “shape”** whenever a public member function is called. These assumptions are called “**class invariants**”.
- E.g., in our last example, the designer will **expect the _capacity field of any TIntStack object to be the size of array allocated**.
- What if that **assumption does not hold**? **The class designer won't guarantee anything about it**: any bad thing can happen.
E.g., if *_capacity* is too large, the class code may write into the wrong memory and causes segmentation fault.
- Then **how the class designer can say the class will “work in all situations”**? Because the class designer will **make sure that the assumption will not be violated by member functions**.
- Because the object data can be accessed **only through the member functions**, they can be sure that the assumption is true.

PMOOP(0396A)-7.3

How to make class invariants hold initially?

- When we have a new local **int** variable, its value is undefined.
- Similarly, **all fields of a new object hold undefined values**.
- But we **must not let the user see such an object**: the class invariants do not hold at this time.
E.g., we don't know what *_capacity* is holding, whether *_stackData* is a valid pointer, whether *_count* ≤ *_capacity*, etc.
- So **how this problem is solved**? C++ solves the problem with a special kind of member functions called **constructors**.
- Whenever a new object is created, **a constructor is called**. This is completely automatic: there is no way to create an object without calling a constructor.
- There can be many constructors for the same class, which differ in the **number and types** of arguments, in just the same way as **function overloading**.

PMOOP(0396A)-7.4

Defining constructors

Let's see two constructors that would be useful for our class.

```
TIntStack::TIntStack():           TIntStack::TIntStack(int initCapacity) {
    _stackData(new int [5]),      _capacity = initCapacity;
    _capacity(5), _count(0) {}    _count = 0;
                                   _stackData = new int[_capacity];
                                   }
}
```

- The name of a constructor is fixed: it is **the same as the class name**.
- Constructor has **no return type**—not even **void**.
- There are two ways to initialize a data member: **after the function header**, or use normal assignment in constructor body.
- In fact, we can merge the two constructors into one: just give *initCapacity* a default value of 5. In general this is a good thing to do.
- Uninitialized fields have **unknown initial value**. Except that uninitialized objects are initialized using the default constructor.

PMOOP(0396A)-7.5

Using constructors

When you **create an object**, you can select which constructor to call. E.g., the following statements create variables using the two constructors:

```
// Global or local object
TIntStack intStack1;
// Temporary object
MyStackFunc(TIntStack());
// Dynamic object
TIntStack *intStack2
    = new TIntStack();
// Global or local array
TIntStack intStacks1[10];
// Dynamic array
TIntStack *intStacks2
    = new TIntStacks1[10];
```

Temporary object: objects that are created only for a statement, and will be destroyed immediately afterwards.

PMOOP(0396A)-7.6

Default constructor

A **default constructor** is one that can be called with no argument.

The following serves as default constructor:

```
TIntStack::TIntStack();
TIntStack::TIntStack(int initSize = 10);
TIntStack::TIntStack(int initSize = 10, bool autoResize = true);
```

Of course, your program can only have one of them.

If your class has no constructor at all, a default constructor is **automatically defined**. The constructor does nothing apart from default initializations. Any explicit constructor will suppress this default constructor.

The **default initialization** of a class is to **call the default constructor**.

You also need a default constructor if you want an array of objects. Without default constructor, you have to create an array of pointers.

It is also needed if you want it to be used with some STL classes.

PMOOP(0396A)-7.7

To be on the safe side: checking invariants in methods

- Once the invariants are set up by the constructor, each member function will strike to **keep the invariants held**.
- It is a **bug** if **public** functions return with **unsatisfied invariants**.
- In an ideal world, the class is implemented correctly, so there is no need to check the invariants (member functions guarantee them).
- In practice, class sometimes has bugs (especially during heavy development), and **we want to know early that invariants are violated**.
- **Only some invariants can be checked**. E.g., it is very hard to check whether `_stackData` is really pointing to an array of `_capacity` items. Because we don't have a way to know the size of an array.
- But checking only invariants that are easy to check (e.g., whether `_count` is at most `_capacity`, or whether `_stackData` is non-NULL) is still a good idea.

PMOOP(0396A)-7.8

Simple invariant checking

One simple way to check invariants: use `assert`. E.g.:

```
inline void TIntStack::CheckInvariants() const {
    assert(_count ≥ 0 && _capacity > 0);
    assert(_stackData ≠ 0);
    assert(_count ≤ _capacity);
}
void TIntStack::Push(int data) { // Buggy
    CheckInvariants();
    _stackData[_count++] = data;
    CheckInvariants();
}
```

When the `_count` becomes too large, one will **immediately know** this: the program will terminate with a message like this:

```
a.out: TIntStack.cc:15: void TIntStack::CheckInvariants() const: Assertion
'_count <= _capacity' failed.
```

PMOOP(0396A)-7.9

More about assert

- The `assert()` function is a handy C function, defined in the `<cassert>` header file. (So be sure to `#include` it!)
- It is good in that it **leaves a stack trace** so that it is possible to run program in the debugger until it fails, and view the stack using "bt".
- It can also be used to **check pre-conditions and post-conditions**.
- After debugging, you might want to **turn off assertion** (to speed up your program). You can do this by `#define NDEBUG` in all source files. Or use "g++ -DNDEBUG" to compile the program, which is more convenient since you don't need touching any source code.
- What's bad about assertion: **no conditional assert**. You can't say "just turn on the assertion for memory checks".
- Also, **no recovery**: the program will always be terminated.. Later we will learn a mechanism called exceptions to deal with these problems.

PMOOP(0396A)-7.10

Destructors: when we dispose the things we allocate?

- There is a **big problem** in our class that prevent it from real use: the **memory allocated for `_stackData` is never freed!**
- We call it **memory leak**: after the object is destroyed (i.e., it is **delete**'ed, or it falls out of scope), there is no way to access the memory.
- So where should we free up the memory? In a **member function**?
Not a good idea: what if the user continue to use the object ?!
- In C++, there is a special member called the **destructor** which is automatically called when an object is destroyed.
Unlike constructor, no overloading of destructor is allowed.
- Like constructors, the destructor has a **fixed name**: `~` followed by the class name, e.g., `~TIntStack`.
- Apart from returning memory, destructors are also used to close files, close windows, etc.

PMOOP(0396A)-7.11

Example destructors

Let's see what we will do in our destructor:

```
TIntStack::~TIntStack() {
    CheckInvariant();
    delete [] _stackData;
}
```

- Note that the destructor **has no return value**, just like the constructors.
- It **must not take any argument**.
- For data member like `_capacity` or `_count` that has nothing to deallocate, we can just ignore them.
The storage of the object itself will be reclaimed after `delete`, only the data that is pointed to by data fields need manual recovery.
- After the destructor, **the class invariants will not hold**.
In particular, `_stackData` is no longer pointing to an allocated array of `_capacity` int elements.

PMOOP(0396A)-7.12

Invocation

Let's make it more concrete about when destructor is invoked...

```
int f(TIntStack s) {
    TIntStack s1;
    TIntStack &s2 = s1;
    s2 = TIntStack(); // destructor called for temporary object
    TIntStack *s3 = new TIntStack;
    TIntStack *s4 = new TIntStack;
    delete s4; // destructor called for the dynamic object pointed to by s4
    TIntStack s5[5];
    return 5;
    // destructor called for the 5 TIntStack of s5
    // destructor called for s1
    // destructor called for s
    // Note: destructor never called for s2 (no need) and *s3 (not automatic)
};
```

PMOOP(0396A)-7.13

Construct by copying

C++ needs to **construct a new object by copying** at many times:

- When passing **arguments of object type**.
Not for reference to object types, which is not copied.
- When returning **values of object type**.
- When creating an object in the form "`TClass c = d;`".
- When throwing an **exception** of an object type.
We will examine exceptions in the third part of the course.
- When constructing a **larger object containing** that type.

All these are done by a **special constructor** called the copy constructor. We are talking about constructors, when a **new object** needs to be created. Distinguish it with assignments, where object creation is not needed (i.e., old object reused).

PMOOP(0396A)-7.14

Example Invocation

```
TIntStack CombineStacks(const TIntStack &s, TIntStack t)
{
    // Argument t is created by copy constructor, but s is not copied
    TIntStack ret = s; // Copy constructor used to make ret
    while (!t.Empty()) {
        ret.Push(t.Top());
        t.Pop();
    }
    return ret; // Done by copy constructor
}

int main()
{
    TIntStack s1, s2;
    s1.Push(1);
    s2.Push(2);
    TIntStack s3 = CombineStacks(s1, s2); // Copy constructor used once more
}
```

PMOOP(0396A)-7.15

Default copy constructor

Question: So will our class compile with the above code?
Problem: we haven't define a copy constructor.

Answer: Yes.
In other words, C++ defines a copy constructor automatically. Technically, **every** class has a copy constructor.

Question: Will our class and program performs satisfactorily?
In other words, is the copy constructor satisfactory?

Answer: No, not quite. The program will probably crash—and even if not, it won't work correctly.

Why?!

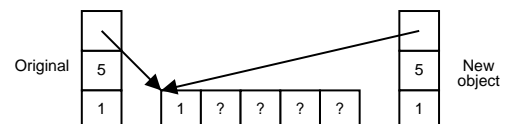
Answer: The automatically defined copy constructor is not smart enough.

PMOOP(0396A)-7.16

Default copy constructor

What is done by the default copy constructor?

- **All the data fields are copied.** In our case, this means `_stackData`, `_capacity` and `_count` are copied.
- **Pointers are copied by copying the pointer values, not their contents.** In our case, the resulting objects will have `_stackData` pointing to the same place.



The problem: our `TIntStack` class **expects each object to have its own array allocated in `_stackData`**.

PMOOP(0396A)-7.17

Copy semantics

We want to specify **what should happen when our objects are copied**: the semantics (meaning) of copying.

Conceptually, it determines **what constitutes as “part of the object”**.

- The default copy semantics is called **shallow copy**: we copy the “first layer” of the object, i.e., not into the objects pointed to by data fields.
- Conceptually, the data pointed to by pointer fields is **not part of the objects**.
- If this is really our intent, **the objects should consider that the pointed-to array as something controlled by someone else**, e.g., user code.
- E.g., the destructor should **never deallocate** the object, there should be a way for the user to give us an object (e.g., through constructor), etc.

But... this is **not** our intent.

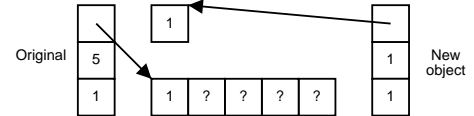
PMOOP(0396A)-7.18

Deep copying

We want `_stackData` to be considered **part of our object**.

More precisely, after copying, pushing into a stack shouldn't affect the other stack.

- One alternative: **deep copy**. We want to copy the content of the data pointed to by the pointer, not the pointer itself.
- Our **custom copy constructor will allocate storage** for `_stackData`.
- It is not strictly necessary that we copy the whole structure in exactly the same way as the original object. As long as **the new object will behave the same way as the old one**, it is okay.
E.g., we don't really need to create an array of `_capacity` elements. We can create one of a different size, e.g., `_count` elements.



PMOOP(0396A)-7.19

Example copy constructor

```
TIntStack::TIntStack(const TIntStack &other) {
    if (other._count == 0) {
        _count = 0; _capacity = 5;    // Create a brand new one
        _stackData = new int [5];
    } else {
        _count = _capacity = other._count;
        _stackData = new int [_count];
        for (int i = 0; i < _count; ++i)
            _stackData[i] = other._stackData[i];
    }
}
```

- The prototype of a **copy constructor** is always of this form (except that const can be dropped). In particular, one **cannot pass by value**.
It is not possible: we can't use the copy constructor, as we are still defining it.

PMOOP(0396A)-7.20

What private really means

- If you are careful, you should notice that we have **use a private member** of an object other than our own object.
- In particular, we have used `other._stackData` and `other._count`.
- This is **allowed in C++**: within a class (i.e., within the definitions of member functions of a class), we can use the private member of **any object** of the same class.
- We say that in C++, **encapsulation is at the class level**, not at the object level.
- On the other hand, it is impossible to access a member, e.g., `_count`, outside the class.
Of course: this is encapsulation.

PMOOP(0396A)-7.21

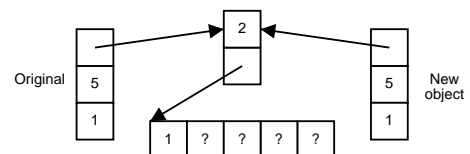
Copy-On-Write

- **For some classes, the copying is usually wasted**: the copied object is discarded quickly.
- E.g., in our case, if most copying is really used just to pop everything out of the array (no push), it would be wasteful to do the copying.
We won't touch the data for the whole lifetime of the object.
- **Is it possible to avoid the copying in this case?**
- In other words, **is it possible for two objects to share** the common data until we actually make the two stacks different?
- The answer is **yes, but we need a more complicated object arrangement** to support it.
- We call such ideas **copy-on-write (COW)**: do the copy only when we need to write over the external data.

PMOOP(0396A)-7.22

Object arrangement

How complicated is it? Let's see:



- There is an **auxiliary struct** for each array we allocate. Its sole purpose is to count the number of objects sharing the array (here, 2).
- When we copy, we do a **shallow copy**, and then **increment the count**.
- When we **destroy**, we **decrement the count**. If the count becomes 0, we deallocate the array and the auxiliary struct.
- Before we push, **we first check whether the data is shared** (i.e., `count > 1`). If so, do a copy.

PMOOP(0396A)-7.23

Example class

How to implement that? We will have to modify the data fields:

```
class TIntStack {
public:
    TIntStack();
    TIntStack(const TIntStack &);
    ... // The other functions
private:
    struct TCowObj { // Type of auxillary object
        int _numShare;
        int *_stackData;
    };
    TCowObj *_cowObj;
    int _capacity, _count;
};
```

Note that the user don't see the complicated change of implementation at all except that copies get faster and some Push get slower—the power of encapsulation.

PMOOP(0396A)-7.24

Example functions to handle COW

How about the other functions? E.g., copy constructor and Push:

```
TIntStack::TIntStack(const TIntStack &other) {
    _cowObj = other._cowObj;
    _capacity = other._capacity;
    _count = other._count;
    ++_cowObj._numShare;
};
TIntStack::Push(int data) {
    if (_cowObj._numShare > 1) { // Copy!
        --_cowObj._numShare;
        int *newObj = new TCowObj;
        newObj._numShare = 1;
        newObj._stackData = new int [++_capacity];
        for (int i = 0; i < _count; ++i)
            newObj._stackData[i] = _cowObj._stackData[i];
        _cowObj = newObj;
        _cowObj._stackData[_count++] = data;
    } else ... // Similar to original version
}
```

PMOOP(0396A)-7.25

Disable copying

- What if my object is **not suitable for copying**?
- E.g., if you have a class encapsulating a file, it is probably not good to copy objects of such class.
- Technically, **all objects have a copy constructor**.
Either default or manually written.
- But you can limit who can **use** the copy constructor.
- In other words, we can have a private copy constructor, so that the **user will be unable to use it**.
The content might just print an error message and exit the program.
- But be sure you know the **consequence**: users will **not be able to pass the object by value** as arguments or return values of functions.

PMOOP(0396A)-7.26