

An example in daily life

Lecture 6

Data abstraction and Encapsulation

Now that we know how to physically break up the source file into many, we are going to write programs that consists of many source files.

But before we really do so, let's first examine a technique called data abstraction, which allows us to easily make up the pieces of code without reading every line of it.

References:

- Textbook chapter 2, pages 27–60, except “What is a Multi-thread safe class” (page 55).

PMOOP(0396A)

PMOOP(0396A)-6.1

An operation

Or is it? Let's examine what happens **when one presses the PLAY button...**

- Check whether the **battery is enough**, and if so, ...
Otherwise it shows “no battery” a while and stop.
- Check whether a Minidisc** can be found in the disc compartment, ...
- If so, **slide open the MD** and **activate the laser beam and the motor** to read the disc header...
- Check the **format of the disc** and see whether it is good,
E.g., the disc might be encoded in a way not supported by the MD walkman...
- and check **whether there is any track to play...**
- If everything goes well, **start decoding the data** and put it into the Digital-to-Analog converter so that sound is generated, ...
- and show message in the **LCD display**.

PMOOP(0396A)-6.2

The shifting focus

- But **who cares about all these details?**
- As far as the user of the walkman is concerned, **all she is interested is that she can hear the content of the MD.**
She probably don't care that a laser beam is operating.
- On the other hand, **a designer of MD players** will treat it rather differently: he will treat the MD not as something that contains music, but as something that will **make the laser sense digital signals.**
- He will also be aware that **there are different disc formats**, some **space is needed to place the circuitry and battery**, the **DAC need to have the same resistance (“impedance”)** to electricity as that of the earphone, ...
- How about **makers of MDs titles?** He will have another set of ideas about the MD walkman. E.g., some **copy control** mechanism is built into the system, how good are the **quality of audio** reproduced, a MD-player designer consider MD as storing a sequence of bytes, ...

PMOOP(0396A)-6.3

Abstraction: highlight the focus

- So why **different people have completely different idea about the same thing?**
- Why don't have a **single idea about the MD-walkman**, and teach everybody about what it is?
- The primary reason is that **by focusing on just a few things** of the system **that really makes a difference for the individual**, we make the system **easier to understand, use and design.**
- An **abstraction** is a **simplifying view** of a system. E.g., the abstraction of MD walkman as **something containing button to control playback of Minidisc**, MD as **something that stores tracks of music**, etc.
- The **abstraction used by different type of users can be very different.** E.g., a MD-walkman repair engineering will think the MD walkman is a piece of electronics reading the bits of MD using laser.

PMOOP(0396A)-6.4

Another example: recursive functions

Let's see how the idea is used in programming...

Suppose we see the following function for **sorting**, how we understand it?

```
void quicksort(int array[], int left, int right) {
    if (right <= left)
        return;
    // the following rearrange the array so that mid is between left and
    // right-1, with everything in array[left] to array[pivot] smaller than
    // everything in array[pivot+1] to array[right]
    int pivot = partition(left, right);
    quicksort(array, left, pivot);
    quicksort(array, pivot+1, right);
}
```

Will we understand it by *dry running* the program, i.e., pretending that we are the computer, **run one statement each time**, stepping into the function calls, remembering all the contexts? **Luckily, no.**

PMOOP(0396A)-6.5

Abstraction in recursive functions

- To analyse quicksort, first concentrate on **very small cases**: what if there is either only 0 or 1 element between *array[left]* and *array[right]*?
- The function works correctly: it **simply returns**.
Of course, an array of 0 or 1 element is always sorted.
- Now we **abstract out the quicksort function** for all smaller cases as something which sorts the elements between *array[left]* and *array[right]*.
- When quicksort is called for a larger array, **the partition function split it into two strictly smaller arrays**, with all smaller elements in the first half and all larger elements in the second.
That "strictly smaller" is important: we only abstract smaller cases.
- Now we invoke quicksort for each side. **We don't need to go into it: we already know what will happen due to abstraction**: both sides will be sorted, giving us a completely sorted array.
- And we are done! No complicated fiddling with multiple context...

PMOOP(0396A)-6.6

Interface and implementation

Most object oriented languages allow **two abstractions** of each "class": the users' view and the implementor's view.

N.B.: (1) We will see what is classes shortly. (2) The OS and the network usually serve to provide more abstractions. (3) We will later see C++ actually provides an intermediate abstraction.

- The users view the class as something that **provide a number of public functions and data items**.
- The implementers view the class as **the code written to support the public functions**, including all the public and private functions and data.
- When we speak of **abstraction**, we normally means the abstraction used by the **user** of the class.
- The public functions and data items are called **the interface** of the class: it **provides users with the methods to access the class**.

PMOOP(0396A)-6.7

Implementers' abstraction

- The abstraction of the user is a **strict subset of that of the implementer**. **The life of the implementer** must be too difficult, right?
- No. The implementer has a **simplified view**: he **doesn't care about the rest of the program**. The user of the class will handle that.
Meaning that users are tough. Simple interfaces are really needed.
- So **the interface allows the user and implementer to focus on his own job**, without worrying about the other party.
- This is possible because
 1. The user only requires **the interface being implemented** when he write his program. He doesn't care about the actual implementation.
 2. The implementer does not care what the user do, **as long as the published interface is used** to access the class.
- Interfaces serve as **contracts between users and implementers**.

PMOOP(0396A)-6.8

A sufficient interface

- Since interface is used to **support the users' abstraction**, it is clear that **simple is good**.
Recall that abstraction is simplifying view.
- But an abstraction **must support all the operations needed by the users**. Otherwise the user will bypass the interface.
- E.g., for the MD-walkman, is it possible for the interface not to include a stop button? Clearly not: we want to stop it at some time.
- Less obviously, can we just drop out the LCD display?
- Not quite: if the user needs to find a specific position within a song she want to fast-forward to, she needs some indication that he has reached where she want.
- The important message: **Interface design requires good knowledge about what the user want to do with the system**.

PMOOP(0396A)-6.9

Implementation protection

- When you press the PLAY button on the MD-walkman, it will activate the motor and laser **only if the door is closed**.
- If the user can directly control the walkman to activate the motor and laser, the resulting action can **damage the walkman and its user**.
- The designers assume that **the user uses the published interface** to access the walkman, which allows activation of motor and laser only when door is closed.
- If the **assumption is violated**, e.g., the user trigger the door close detector in another way, **there is no guarantee of correct operation**.
- **For an interface to work, the user must access the system using the interface**. If it is bypassed, there is no guarantee that mishaps won't happen.

PMOOP(0396A)-6.10

Encapsulation: no access to implementation!

An interface is said to achieve **encapsulation** if there is **no way to access the implementation except through the interface**.

Encapsulation is also known as **information hiding**.

Why encapsulation?

- To **guarantee the correct operation** of the system.
- To achieve **implementation independence**:

If you change the implementation, the user does not need to update his own view of the system—as long as the interface is not changed.

E.g., if you replace your MD-walkman with an upgraded version which supports a new method of noise control, you can still use the walkman as before.

PMOOP(0396A)-6.11

Summary

- (Users') Abstraction: clients' simplified view of a system.
- Implementation: everything that makes up the system.
- Interface: the methods provided for users to access the system. In other words, the methods to support the users' abstraction.
- Sufficient interface: An interface that allows the user to do everything she want to do with the system.
- Encapsulation: To enforce that interface be used to access a system.
- Implementation independence: the virtue that the way the users access a system is dependent only on the interface, not on its implementation.

PMOOP(0396A)-6.12

Data abstractions and Data encapsulation

- In **procedural programming** (i.e., when you separate programs into functions), the primary abstraction is **functions**, which abstracts out the code that implements the functions.
- In many applications, **functions are used to manipulate data**. E.g., a pointer to a **struct** can be given to the function for manipulation.
- Unluckily, such uses **cannot support encapsulation**: the user **can do anything he want to do** by inspecting and modifying the struct directly.
- Note our implicit desire to **treat the struct** as our primary abstraction rather than the **function**: **Data abstraction**.
Why we want to abstract data rather than code? In fact, we want both. The procedural approach supports code abstraction. It is not that "we only need data abstraction"—despite of what Java programmers want to convince you.
- With data abstraction, we use **Data encapsulation**. We want the user to use **only some public operations** (interface) to access the struct.

PMOOP(0396A)-6.13

Example: stack

Suppose we want to implement an integer stack.

- The **first** and **most important** thing to do: analyse how the user would like to use the stack.
So that we can write a sufficient but yet simple interface.
- Suppose the user wants to **push** an element into the stack, **pop** out an element, **examining the top** of the stack, and **checking** whether the stack is empty.
- The interface then contains these four functions. Everything else has to be in the implementation.
- The implementation may use a linked list of integers, a vector of integers, a simple array, etc to store the data. **The user don't care**.
- **Auxillary functions** might be required to implement the functions required. Just like the implementation data, this shouldn't be exposed.

PMOOP(0396A)-6.14

Without class

Without class, you'd write something like this:

```
struct TStack {  
    vector<int> _stack_data;  
};  
void push(TStack &s, int data) { s.push_back(data); }  
void pop(TStack &s) { ... }  
int top(TStack &s) { ... }  
bool empty(TStack &s) { ... }
```

Is it good? Reasonable, but it **doesn't support encapsulation**: anybody can go into the `_stack_data` and put **invalid data** there.

And if you really do so, your program will be **dependent on that the `_stack_data` being a vector**. If later you change it to a linked list, your program will break.

PMOOP(0396A)-6.15

Data abstraction with C++

- C++ supports data abstractions and encapsulation using **class**.
- **class** is very similar to **struct**: it is used to define new **data types**, which **instances** (i.e., variables of that type) can be used to hold **data members**.
- But there are two differences: you can **define operations** on classes, and selectively **hide data members and operations** from the users.
Both are really not their differences: both can be used for **struct** as well. But normally we don't do this. We want the keyword to tell what the data type is for: **class** for data types with data encapsulation, **struct** for data types which expose all data members.
- The operations of a class is called a **member function**.
- A member (data or function) is said to be **public** if the users can use it, and is said to be **private** if they can't.
There is also something called **protected** which we will explore later.

PMOOP(0396A)-6.16

Code

A class is just the same as a struct you have learnt before, except...

```
class TIntStack {  
public:  
    void Push(int data);  
    void Pop();  
    int Top() const;  
    bool Empty() const;  
private:  
    std::vector<int>  
        _stackData;  
};
```

Note that something like *function prototypes* is included. They are the **operations** of the class. In fact, you can even write out the **definitions** of the functions in the class.

C++ allows programmers to destinate part of the class as **public** and part as **private** for encapsulation. These keywords are called **access specifiers**.

Before the first access specifier, the members are private. This default behaviour is the real difference with struct, which defaults to public.

- The member functions take no argument of `TIntStack` type. This might seems strange, until later when you know that **"this"** is automatic.
N.B.: We use the same source code convention as the textbook. See page 42.

PMOOP(0396A)-6.17

Classes vs. objects

- Just like **struct** defines a type (not a variable), **class** defines a type.
- Therefore, there is **no storage allocated for a class**.
Not until we talk about static data member later.
- Storage is allocated when you create a variable of the class-type. E.g.,

```
TIntStack intStack;
```

This creates a variable `intStack` of type `TIntStack`. Depending on whether it is defined globally or within functions, it **may be allocated statically or automatically**. Or you can do **dynamic** allocation by "**new** `TIntStack`".

- A variable of a class-type is called an **object**.
- **Object-Based Programming**: to write large programs by simplifying it using **classes** for **Data Abstractions** and **encapsulation**.

PMOOP(0396A)-6.18

Using the stack

Using the stack is easy. E.g., let's simply test our implementation:
Whenever C++ chooses to make use of implementation easy, it makes use easy.

```
int main() {
    TIntStack intStack;           // Create stack
    intStack.Push(5);             // Push something into it
    intStack.Push(10);
    cout << intStack.Top() << endl; // Get the top element
    intStack.Pop();               // Pop it out
    cout << intStack.Top() << endl;
    cout << intStack.Empty() << endl; // Is it ready?
    intStack.Pop();               // Do it again
    cout << intStack.Empty() << endl;
}
```

Note that **operations are not used in isolation**. It is applied on **some objects**, using the familiar syntax `object.operation(params)`.

PMOOP(0396A)-6.19

Code for stack: the this pointer

Eventually we will have to implement the operations. It is done like this:

```
void TIntStack::Push(int data) {    int TIntStack::Top() const {
    _stackData.push_back(data);    return _stackData.back();
}

void TIntStack::Pop() {            bool TIntStack::Empty() const {
    _stackData.pop_back();         return _stackData.empty();
}
```

But wait... what is meant by `_stackData`?

Of course, it is the private data member of `TIntStack`. But **which `TIntStack` are we talking about?**

Recall that we can talk about data member only if we are given a variable of a struct or class type. We can't talk about data member in isolation.

PMOOP(0396A)-6.20

The this pointer

Let's pick the use and implementation `Pop` and put it side-by-side.

```
// Use                                // Implementation
intStack.Pop();                       void TIntStack::Pop() {
                                        _stackData.pop_back();
}
```

- The intention is quite clear now: we want the `_stackData` of `Pop` to mean `intStack._stackData`. How this is done?
- Whenever a member function is called, it is **called for an object**.
Here, `intStack`. There is no way to call a member function without an object.
- The function call has a hidden argument called **this**. It is **a pointer to the object that the member function is called for**.
- If you use a member within definition of a class function without giving it an object, **this** is used automatically.
So `_stackData` means **this**→`_stackData`.

PMOOP(0396A)-6.21

Const'ness of methods

- Just like function arguments, sometimes we want to make sure "**this**" **cannot be modified by the member function**: it should be "**const**".
- But we have **nowhere to specify the const'ness**: **this** is passed implicitly, and we never write it out!
- So there is a special syntax to do this in C++: to declare it **after the function prototype** like this:

```
int TIntStack::Top() const { // const: no write to _stackData!
    return _stackData.back();
}
```

- If your class is **to be reused**, it is **very important** that you get the **const** right.

Otherwise the user of your class who has a const pointer to your class will not be able to call your supposedly const function.
E.g., we can call `back()` above only because it is a const function.

PMOOP(0396A)-6.22

Mapping to file splitting

Let's go back a bit and see how all classes appear in **header files**.

- If you have a class (type) that you expect others to see, **you will define it in the header file**.
- Just like normal functions, **you will not write it in the header file** unless you want to inline it.
- Writing the function **definition within the class imply inlining**.
- If you have a class that you don't want other to see, **you shouldn't put it into the header**.
But still, if you have a public class containing another class, that contained class must also be in header: it can't compile otherwise.
- The rules for **objects** is just **the same as other variables**: you will define most global private objects as **static** or nameless namespace. Public objects will be declared in headers using **extern** and in one of the source files without **static**.

PMOOP(0396A)-6.23