MTH 4300, Lecture 13

Step 2a: What is `->` and Omitting `this->`;
Step 2b: Accessors vs Mutators;
Step 3: Constructors;
Step 4: `private` and Information Hiding;
Why is Information Hiding Valuable?

E Fink

March 16, 2025

1. Step 2a: What is `->`? (and Omitting `this->`)

As we said, in methods, `this` is a pointer to the whole object which called the method.

If you want to reference the specific attributes of the calling object, it would make sense to write something like `(*this).age`. After all,

- `this` is the address of the calling object;
- then `*this` is the calling object itself;
- and we want to refer to the age attribute of this object.

The combination of `*` and `.` has a special notation: the arrow operator we've been using! In other words,

`x->y` is a shorthand for `(*x).y`.

(So, `->` is "star-then-dot." I find it helpful to repeat that phrase.)

Step 2a: What is ->? (and Omitting `this->`)

The arrow operator can be very useful. However, the one way we've used it so far happens to be unnecessary: if you are referring to the calling object's attributes, **you can actually drop `this->`, and instead just refer to the attribute names!** These attributes will be *understood* to belong to the calling object.

Let's go back to our Dog example and remove the unnecessary `this->`'s. **L13x1_dog.cpp**

2. Step 2b: Accessors vs Mutators

Some member functions *mutate* member variables (modify object state) – e.g. `GameCharacter`'s `.die()` method which lowers `lives` by 1.

Other member functions merely *access* those member variables, and do not change their values – for example, any sort of print function.

For the latter type, we usually include the reserved word `const` after the parentheses of the parameter list, both in the declaration and the definition. E.g.

```
class Dog {
public:
   // ...
   void bark_num_times(int) const; // <-- CONST!
};

void Dog::bark_num_times(int t) const { // <-- CONST!
   for(int i = 1; i <= t; ++i) {cout << "Bark! "; }
}
```

Step 2b: Accessors vs Mutators

The use of the word `const` has at least three benefits:

- `const` allows class *authors* to communicate to class *users* important information about the method (that it's an accessor!).
- `const` allows the compiler to check for bugs (by authors) – if you try to modify a `this->` variable in a `const` function, you probably made a mistake in your function code.
- Later, when we talk about `const` objects, we will see that ONLY `const` methods can be called on `const` objects.

3. Step 3: Constructors

Initializing each member variable individually is tedious. It would be great if there were a function that would allow us to initialize all the attribute variables at once. This functionality is provided by the *constructor*.

Recall from last time, with the `GameCharacter` class, there were two ways we could declare `GameCharacter` objects:

`GameCharacter x;` (gave `x` a generic “Computer Player” name)

or

`GameCharacter y("Mario");`

It turns out that both these declarations were calls to two very special functions: constructors. A constructor is a special type of member function. It has the SAME NAME as the class that it is part of, and NO return type (not even `void`). It generally DOES have arguments – these will be the initializing data.

Step 3: Constructors

Syntax to declare constructors within your class definition:

```
class YourClassName {
public:
   YourClassName(); // Default constructor
   YourClassName(param_type, param_type); // A
       constructor with parameters
   ....
};
```

E.g.

```
class Dog {
public:
   Dog();  // Default constructor
   Dog(int, double, string);
   ....
};
```

If you made a declaration like `Dog d;` then the default constructor would be called to initialize `d`'s attributes. Note: no parentheses after `d`!

If you made a declaration like `Dog x(5, 2.3, "Evan");` then the other constructor would be called to initialize `x`'s attributes.

Step 3: Constructors

To implement a constructor outside of your class definition:

```
YourClassName::YourClassName(parameter list) {
   // Code which initializes all the member variables
}
```

E.g.

```
Dog::Dog() {
   age = -1;
   weight = -1;
   owner = "Unknown";
}
Dog::Dog(int x, double y, string z) {
   age = x;
   weight = y;
   owner = z;
}
```

(Sometimes constructors do more interesting initializations.)

L13x2_constructdog.cpp **L13x3_constructmario.cpp**

4. Step 4: `private` and Information Hiding

A car has a huge number of interacting parts. The **manufacturers** of a car need to understand all those parts, and how they interact with each other.

But the **drivers** of a car need only understand the interface: the steering wheel, the brakes, the radio, etc. A typical driver should NOT be tinkering directly with the engine – this would be at best a distraction, and at worst actively harmful to the correct functioning of the car. The separation of the interior of the car from what is under the hood helps discourage this tinkering.

Similarly, C++ allows class authors to enforce the separation between implementation (the gory details) and interface (the parts that class users should utilize), by specifying some member functions and variables to be `private`.

Step 4: private and Information Hiding

What does this look like? Simple:

```
class GameCharacter {
private:
   int lives;
   int coins;
public:
   string name;
   void collect_coins(int);
   ....
};
```

What effect does this have?

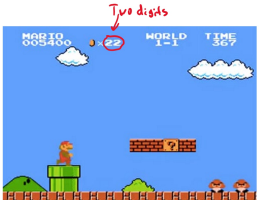
Answer: `.lives` and `.coins` may now only be accessed within `GameCharacter::` functions. Class USERS, on the other hand, who are manipulating `GameCharacters` within `main()`, may NOT directly refer to `x.lives` or `y.coins` – indeed, this will lead to compilation errors.

L13x4_private.cpp

5. Why is Information Hiding Valuable?

Why did I choose `.lives` and `.coins` to make private? Recall that in Mario, the rule is: once a character gets 100 coins, the lives go up by 1, and the coins go back to 0.

A confused USER of the class could be forgetful or unaware of this rule of gameplay. They might change `.coins` independently of `.lives`, in a way that breaks the assumptions about the relationship between these two variables – which OTHER methods might depend on.



(Example: if `.coins` were public, a careless user could accidentally set `m.coins` to have three digits. Will the graphics functionality be prepared for this???)

But this is close-to-impossible if `.lives` and `.coins` are private: the class-user would have to bore into the class implementation to accomplish this.

Why is Information Hiding Valuable?

This illustrates one benefit of *information hiding*: **making sure that class users cannot accidentally corrupt objects**, by putting them in a state that other functions cannot properly handle.

There is a second benefit: **easing the relationship between class AUTHORS and class USERS** ("manufacturers" and "drivers").

Class USERS want a working class, that won't be broken when the class gets updated. Class AUTHORS want the freedom to update their classes, to fix bugs, add features, and improve performance.

These wants are at odds: if a class AUTHOR changes something, client code written by class USERS would ordinarily have to change in response.

Information hiding is a compromise. Class AUTHORS usually won't change what the public members are or what they do, but can change private members to achieve their ends. Meanwhile, class USERS won't have to worry, because they don't access private members.

(Analogy: when manufacturers improve the engine, drivers don't have to relearn how the drive – the interface remains the same!)