MTH 4300, Lecture 12

Defining a Class: Step 1;
Step 2: Creating Member Functions;
Step 2a: What is -> and Omitting `this->`;
Step 2b: Accessors vs Mutators;
Step 3: Constructors

E Fink

March 12, 2025

1. Defining a Class: Step 1

The minimal class definition looks like:

```
class YourClassName {
public:
   // Declarations of member (attribute) variables
};  // <--- SEMI-COLON!
```

So, for example, if I had a program where keeping track of Dogs was really important, I could declare

```
class Dog {
public:
   int age;
   double weight;
   string owner;
};  // <--- SEMI-COLON!
```

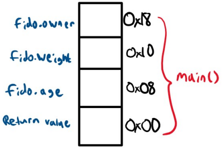
L12x1_dog.cpp

Defining a Class: Step 1

Then, you can declare a `Dog` object just the way you would declare any other variable:

```
Dog fido;
```

The presence of this `Dog` object causes three variables to be created: `fido.age`, `fido.weight`, and `fido.owner`. These variables get placed in the appropriate stack frame just like any other variable.



For now, to give the member variables values, you can assign by, e.g. `fido.age = 5;` Later, we won't do things like this, for two reasons:

- We will soon start restricting direct access to member variables from within `main()`. This is called *information hiding*, and despite how it may sound, it is a helpful thing to do.
- We will also start using *constructors* to initialize our objects.

Defining a Class: Step 1

L12x2_fundy.cpp

Imagine that you are creating an app called Fundy, where users can create their own fundraisers.
Create the definition of a class called `Fundraiser`, whose objects are meant to represent individual fundraisers. `Fundraiser` objects should each have three member variables:

- the `cause` for the fundraiser (i.e., a message describing what the fundraiser is for)
- the `target` value for the fundraiser, in dollars (i.e., the amount that the fundraiser is hoping to raise)
- and the `current` amount raised.

Then, in `main()`, create just one `Fundraiser` object – this fundraiser should be for the survivors of the Krakatoa volcano, and the fundraising target should be $1,000,000 – and assign its members appropriately.

2. Step 2: Member Functions

Member functions represent behaviors done by/with/to objects.

As we've seen, calling a member function looks like

```
object_name.fn_name();
```

Let's break this down. First, the parentheses at the end indicate that we are calling a function. The syntax of the call is different than the functions we've seen in C++, but similar to Python method calls. You should read it as "do `fn_name()` to (or with) `object_name`."

The variable `object_name` is, in a way, a silent argument to the function. Depending on the behavior we are talking about, there may be other arguments necessary; these would go within the parentheses. It is also possible that the function should have a return value – in that case, it'd be likely that you would instead write something like

```
x = object_name.fn_name();
```

With the variable `x` catching the return value.

Step 2: Member Functions

Fine, so how do you create member functions?

To declare a member function: put its prototype (return type, name, *ADDITIONAL* parameters) inside the class definition:

```
class YourClassName {
public:
   // member variables
   ret_type fn_name(param1_type, param2_type);
};  // <--- SEMI-COLON!
```

E.g.

```
class Dog {
public:
   // members omitted
   int age_in_dog_years();
   void bark_num_times(int);
};
```

Step 2: Creating Member Functions

Later, you have to actually define the member function, which is often done outside (but below) the class definition. This is actually similar to writing a normal function, with two major differences:

- Instead of
  `ret_type fn_name(param_type param_name);`
  you write
  `ret_type YourClassName::fn_name(param_type param_name);`
- You don't need to declare or pass the *member variables* of the object which called the member function – just refer to them by `this->member_var_name`. (We will explain later exactly what `this` means, exactly what `->` means, and that neither of these are truly needed for the current purpose.) You may think of `this->` as being akin to `self.` in Python.

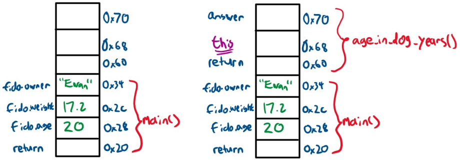
Example:

```
int Dog::age_in_dog_years() {
   int answer = 7 * (this->age); // age is a member
   // Q: WHOSE age?
   // A: this object that just called the function
   return answer;
}
```

Step 2: Creating Member Functions

Methods are just functions; when they are called, a stack frame gets added to the call stack. Of course, all (outside) parameters have spaced reserved in that frame, as do all local variables.

But there is one extra special variable that gets put into the stack frame: `this`, which is a pointer to the object who called it.



E.g. suppose that `fido` is a `Dog` object. Then of course `&fido` would provide `fido`'s address (the address of the first attribute, really). Let's say you add the line

```
cout << "Here is what this contains:  " << this;
```

to the function `age_in_dog_years()`. Then, a call to `fido.age_in_dog_years()` would cause this cout line to display this address.

Step 2: Member Functions

L12x3_methods.cpp

Create methods for the `Fundraiser` class.

- `.donate()`, which receives a `double` named `x` as outside argument, and adds `x` to the `current` (and returns nothing);
- `.met()`, which receives no outside arguments, and returns a `bool`, reflecting if the amount `current` is above the `target` value;
- `.beats()`, which receives another `Fundraiser` named `other` as an outside argument, and returns a `bool` which is `true` if the `Fundraiser` in question has more money `currently` that the `other` one.

(For the last one: if we have a `Fundraiser` object as an OUTSIDE argument, you still need to use the `.` notation that access that object's member variables. Only *the object which calls the function* can have its members referenced with the `this->` notation.)

3. 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.

4. 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.

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.)

L12x4_constructdog.cpp **L12x5_constructmario.cpp**