MTH 4300, Lecture 25

Indirect Inheritance; Polymorphism, Static Binding, Static/Dynamic Type; Virtual Methods and Dynamic Binding; Override

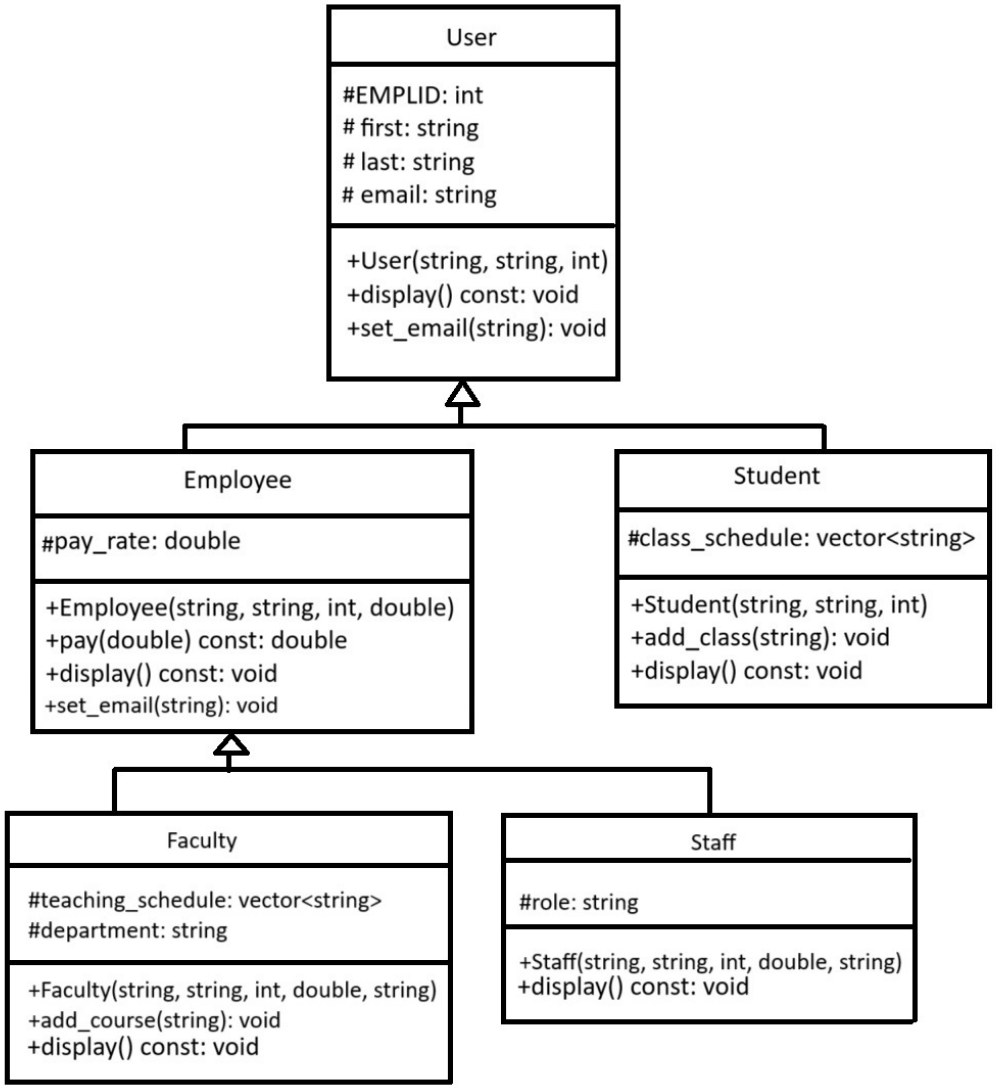
E Fink

May 12, 2025

1. Indirect Inheritance

It is possible to design a class $C1$ which inherits from a class $C2$ which itself inherits from a class $C3$. This is known as *indirect inheritance*.

L25x1_indirect.cpp



Indirect Inheritance

In the UML diagram on the last slide, note that protected members are prefixed by # rather than – or +.

At this juncture, it's probably worth paying attention to the order in which construction takes place. The rule is simply that the constructor for the base class are called first; then the derived-class-specific members are initialized. If we like, we can throw in print statements to verify this.

Indirect Inheritance

Let's address the word `public` that precedes the name of the base class in every inherited class declaration we've seen.

```
class Der:  public Base // PUBLIC inheritance
```

means that every `public` member of `Base` will be a `public` member of `Der` and every `protected` member of `Base` will be a `protected` member of `Der`.
(As we've seen, `private` members of `Base` are simply inaccessible, even from within `Der`.)

You can also have

```
class Der:  protected Base // PROTECTED inheritance
```

which means that means that every `public` or `protected` member of `Base` will be a `protected` member of `Der`; and

```
class Der:  private Base // PRIVATE inheritance
```

means that means that every `public` or `protected` member of `Base` will be a `private` member of `Der`.

Indirect Inheritance

A table:

	Public inheritance	Protected inheritance	Private inheritance
Public member of base	Public member of derived	Protected member of derived	Private member of derived
Protected member of base	Protected member of derived	Protected member of derived	Private member of derived
Private member of base	Inaccessible in derived	Inaccessible in derived	Inaccessible in derived

Non-public inheritance is relatively uncommon.

L25x2_access.cpp

We've previously discussed the concept of *polymorphism*: using the same function name to mean different functions, depending on the types of objects it is called upon.

In the next example, being able to use the same function name on different types isn't just a mnemonic device – the entire structure of the program would have to be redesigned if we were unable to achieve the desired polymorphism here.

L25x3_display.cpp

If you have a class, it's not unusual to have many objects of that class type. And if you have many class objects, it's not unusual to have operations that act on many different objects. In this example, we have several different `Users`, of different types, and we wish to `display()` them all – each using their native `display()` method.

Storing all these `Users` in an array doesn't work: even though we call constructors for a `Faculty`, a `Student`, and a `Staff`, when we put them all in a `User` array, they are all stored as `Users`, so `User::display()` is called for all of them.

Polymorphism, Static Binding, Static/Dynamic Type

In the last example, the array of `Users` didn't even reserve space for member variables like `pay_rate`, which not all `Users` have. That approach was inadequate.

Much better (but not without issues): use an array of POINTERS, to heap-allocated objects. Once we're done with that, we'll have a collection of `Users` of various species, and all of them will have all of their pertinent information stored. Let's make the necessary changes now.

Now, try to run the code. The `display()` function is STILL using the generic, `User::` version of `display()`. What went wrong?????

Polymorphism, Static Binding, Static/Dynamic Type

What went wrong?????

Every function call in a program is matched to a particular function with the given name. If there are several functions with a given name, the correct one is chosen based on the types of arguments it is called with. This choice is called a *binding* of a particular call to a particular function.

By default, the binding is performed by the **compiler, at compile-time**. This is called *static binding* (or *early binding*).

Now, let's think like a compiler. Look at **L25x4_display2** . When the compiler sees code like

```
my_users[2]->display();
```

what does it know about `my_users[2]`? It certainly doesn't know the type of the object that `my_users[2]` points to: nobody knows, until the program is running.

The only thing it knows is that `my_users[2]` itself is a `User*`, based on the declaration type of `my_users`. Hence, it chooses to bind the call to `User::display()`.

Polymorphism, Static Binding, Static/Dynamic Type

For expressions involving reference variables or pointer variables, we can distinguish between their *static type* and their *dynamic type*. The static type of a reference/pointer is just the type implied by declarations, whereas the dynamic type is the type of the value the expression has. For example, consider the following bit of code:

```
Student x("Johnny", "Student", 12345678);
User *y = &x;
User &z = x;
```

Here, the static type of `*y` is `User`, and the static type of `z` is `User&`. But the dynamic type of `*y` is `Student`, and the dynamic type of `z` is `Student&`.

Having said that, the moral of what we've said so far is: if `ptr` is a pointer, and `method()` is a member function, then

```
ptr->method()
```

will be bound, by default, to the version of `method()` for the STATIC type of `*ptr`.

3. Virtual Methods and Dynamic Binding

C++ does support *dynamic binding* (or *late binding*): choosing the called function based on the DYNAMIC type of the caller. To achieve this, we use the keyword `virtual`.

Specifically:

- for each method call, the compiler determines the static type of the caller;
- the compiler seeks the method in the class definition for the static type;
- but if that function is a **virtual** function – that is, if its return type in the declaration is preceded by the word `virtual` – then the method will be bound dynamically instead. (When a method is virtual, the derived class's version is said to *override* the base class's method.)

Now, go back to the previous example, and make the `display()` function virtual – it resolves the problem!

Note that if a method is `virtual`, then so are all the functions that hide it in derived classes. Combining this point with what is listed on the previous slide should allow us to predict the behavior of **L25x5_puzzle.cpp** .

4. Override

A warning: if a function in a base class is declared as `virtual`, then the declaration of the corresponding function appearing in the derived class should (usually) have EXACTLY the same number, type, and order of parameters, as well as the same return type and `const` qualifier. If there is a difference, then there could be a compilation error, or worse yet, the dynamic binding could simply fail to work. See **L25x6_exact.cpp**

Placing the keyword `override` after any overriding virtual method is a good practice. This word should be placed at the end of the method's signature – i.e., after the end of the argument list, or for `const` methods, after the word `const`. If there are problems in how you have written your overriding function, the compiler will inform you.

Let's go back to the example, and put `override` as appropriate.