MTH 4300, Lecture 23

CUNY First;
Inheritance;
Protected Access Specifier

E Fink

May 7, 2025

1. CUNY First

Imagine that you are writing a system like CUNY First. As part of the system, all users could be represented as objects. Every user, be they student, faculty or staff, has a first and last name and an EMPLID. However, students and faculty have class schedules; students have the ability to enroll in classes; faculty and staff have pay rates.

L23x1_cuny.cpp

We will start out considering three classes: `Faculty`, `Student`, `Staff`. Our `Faculty` class will be defined by

```
class Faculty {
private:
    string first, last;
    int EMPLID;
    double pay_rate;
    std::vector<string> teaching_schedule;
public:
    Faculty(string f, string l, int emp, double pr);
    double get_pay(double hours) const;
    double add_course(string &course);
};
```

CUNY First

Our `Student` class will have the declaration

```
class Student {
private:
    string first, last;
    int EMPLID;
    std::vector<string> class_schedule;
public:
    Student(string f, string l, int emp);
    double enroll_course(string &course);
};
```

Pretty similar to `Faculty`, except that a `Student` has a `course_schedule` instead of a `teaching_schedule`, and they have an `enroll()` function instead of `add_course()`, and no pay :(

And `Staff` will have the declaration

```
class Staff {
private:
    string first, last;
    int EMPLID;
    double pay_rate;
    string role;
public:
    Staff(string f, string l, int emp, double pr, string
        r);
    double get_pay(double hours) const;
};
```

CUNY First

Oh, whoops. I forgot that everyone needs to have an email address. Ok, I'll go add that attribute – to all three classes.

Hmmm . . . a `display()` method would be nice for each user. Ok, I'll go add that – to all three classes.

And sometimes names need to be updated, so we can add a `set_name()` function – three times.

(And this is on top of the fact that we've already written three very similar classes.)

`Faculty`, `Students` and `Staff` are definitely different types, with different attributes, and different versions of shared methods. But at their core, they are all `Users` of the system – and in fact, all the commonalities in the three classes flow from this fact!

If we can exploit these commonalities, we can make code construction vastly simpler, as well as more flexible.

2. Inheritance

Inheritance refers to the act of extending an existing *base* class (e.g. `User`) to create new *derived* classes (e.g. `Faculty`, `Student`, `Staff`), in such a way that the objects of the derived class will still also be objects of the original base class.

In particular, derived class objects will possess all the attributes of an element of the base class, and they will also have access to all the member functions of the base class, although some of these may be "rewritten" in the derived class. The derived classes may also have additional member variables and functions.

Inheritance is used to express an "is a" relationship between objects. For example, every `Student` IS A `User` of CUNY First. Every operation that is valid for every `User` will certainly be valid for a `Student`.

Inheritance

In general, to create an inherited class, the basic declaration looks like

```
class Derived_Type: public Base_Type {
   // Member variables/functions specific to Derived_Type
   //
   Derived_Type(args): Base_Type(args), other_attr{arg} {}
}
```

Every object of the `Derived_Type` will automatically possess all of the members of the `Base_Type`.

Notice the form of the constructor of the derived type: a call to the `Base_Type` constructor – with parentheses! – ought to be the first item in the initializer list.

L23x2_employee.cpp

Notice that the `Employee` object we've created has all the attributes of a `User` – every public `User` attribute is public for `Employees`, and every private `User` attribute is private for `Employees`.

Also notice that you can call `User` member functions (aside from any constructors, destructors, friends or operator=='s that exist) directly on our `Employee` objects. That's because an `Employee` IS A `User`.

Inheritance

In a derived class, sometimes one provides a different version of a defined in the Base class. For example, when we `display()` an `Employee`, we should be showing slightly different information. If we explicitly include a function named `display()` in the `Employee` class declaration, that is referred to as *hiding* the Base class's function.

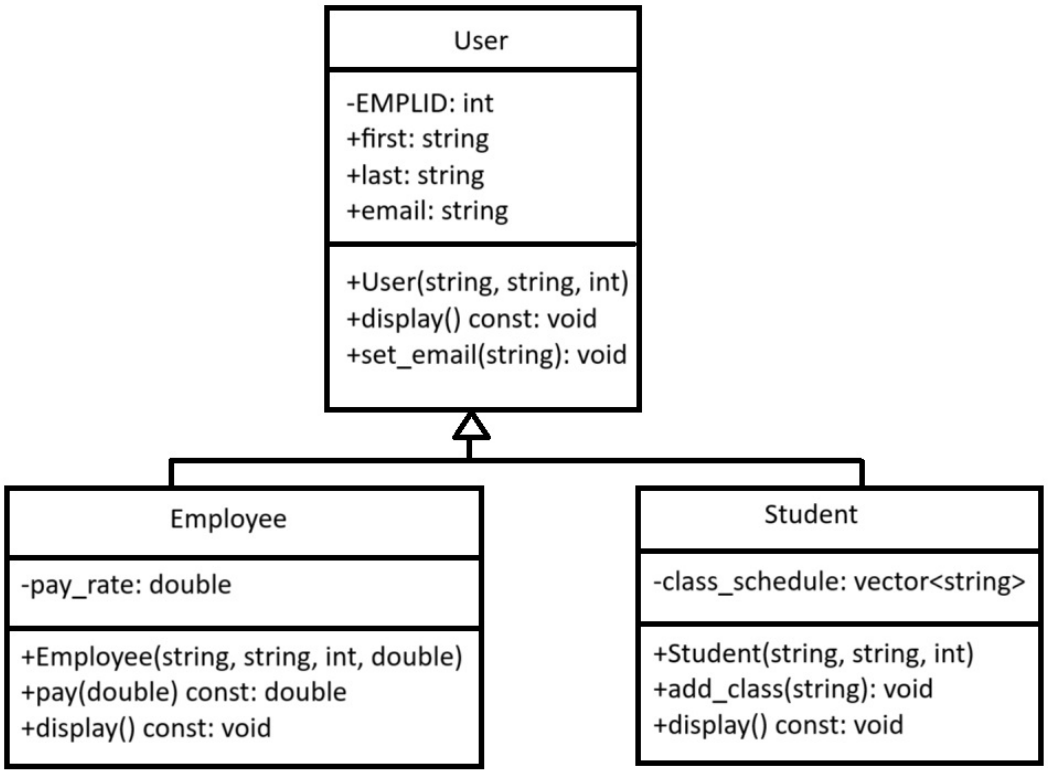
Note that you can still make reference to the hidden member function, by calling `Base::member_fn()`. The `::` operator is known as the *scope resolution operator*.

L23x3_hiding.cpp

Inheritance

The following is a UML (Unified Modeling Language) diagram showing our class hierarchy, with a student class we didn't include in our code. The items above the midline are member variables; the ones below are member functions. Each + denotes a public member, and each – denotes a private member. The triangle-head arrow coming out of the bottom of the `User` class denotes that the two lower classes are derived from `User`; in the derived classes, it is implied that there is access to members from the base class (hidden ones using scope resolution).

The point of this is to give a snapshot of the data types, the contained data and operations, and the relationships between the types.



Inheritance

L23x4_animal.cpp

I am making a game with all sorts of `Animals` who run around. Each `Animal` has `name` and `species` attributes, and supports `eat()` and `display()` methods.

Create a `Dog` derived subclass. This class should be special in the following ways:

- There should be a `breed` attribute (public).
- In the constructor, the `breed` attribute should be set, while the `species` attribute should be automatically set to `Dog`.
- The `display()` from `Animal` should be hidden by a modified version, which displays the *breed* rather than the species (along with the name).
- There should be a new `bark()` method that simply prints `BARK` .

Try to draw a UML diagram!

Inheritance

Notice: when we take the `Dog` named `d`, and perform the construction

```
Animal doggy = d;
```

it works perfectly well – because a `Dog` IS A `Animal`, and because a copy constructor for `Animals` is automatically defined if you don't define one yourself.

However, when this takes place, the constructor for `Animal` only reserves space for the attributes declared in `Animal`, `name` and `species`. The `breed` attribute gets dropped. Likewise, if you call a member function upon `doggy`, then that has to be a member of the `Animal` class – for example,

```
doggy.bark();
```

won't work!

3. Protected Access Specifier

L23x5_access.cpp

In this example, observe the following:

- In the `User` subclass, I have made all the attribute variables **private**. This is generally good practice, but ...
- ...it complicates the new version of `set_email()` that I am writing in the `Employee` derived class.

In general, inherited classes cannot directly access the private members of their base classes. However, there is an intermediate level of access between public and private: *protected*. Protected members can be accessed within the class they are defined in *and* within classes that directly inherit from.

Making members protected is a compromise, for internal details of a class that subclasses would nonetheless need access to.