Intro to Classes (in C++)

CSci-3081W: Program Design and Development
Professor Daniel Keefe

First, any questions?

- On the syllabus you read for homework?
- On the course in general?
- On the reading (McConnell Chp 6)?

Intro to Classes (in C++)

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Classes are an example of Abstract Data Types

- McConnell discusses ADTs in Section 6.1, which I asked you to just skim.
- First of all, what is an Abstract Data Type?
- Think of an example, how does programming with ADTs compare to programming without them?

Benefits of Using ADTs (classes)

Let's explain these:

1. You can hide implementation details
2. Changes don't affect the whole program
3. You can make the interface more informative
4. It's easier to improve performance
5. The program is more obviously correct
6. The program becomes more self-documenting
7. You don't have to pass data all over your program
8. You're able to work with real-world entities rather than low-level implementation structures

In C++, Abstract Data Types are Classes

The Basics: How do you create a class in C++?

Class Interfaces are Typically Specified in .h Files

```
class Date {
public:
  // .. public interface here
private:
  // .. private interface here
};
```

It defines the types of all the public and private methods and fields (member functions and data members).

Still in the .h file, let's fill in a bit more detail:

```
class Date {
 public:
  Date(int y, int m, int d);
  virtual ~Date();

  string print();

 private:
  int year, month, day;
};
```

Notice, there's a method called print() here but we haven't written any code yet to define what happens when print is called.

Class Implementation

Specify the interface in the Date.h file:

```
class Date {
public:
  Date(int y, int m, int d);
  virtual ~Date();
  string print();

private:
  int year;
  int month;
  int day;
};
```

Define the details of what happens when each method is called within the Date.cpp file:

```
Date::Date(int y, int m, int d) {
  year = y;
  month = m;
  day = d;
}

Date::~Date() {
}

string Date::print() {
  cout << year << “ “ << month
       << “ “ << day << endl;
}
```

Using Classes

- As programmers, **we should only need to read the .h file to know how to use the class**. (If the programmer has done a good job.)
- Likewise, the compiler will only use the info the .h file in order to compile code that depends upon the Date class.
- Example (main.cpp):

We need to include the Date class’s .h file before we use the Date class.

Every C++ program starts with a special function called main(). Each program has exactly one main() in it. Usually, I put it inside its own file main.cpp.

```
#include “Date.h”

int main(int argc, char* argv[]) {
  Date d1(2010, 12, 27);
  d1.print();

  Date *d2 = new Date(2010, 12, 28);
  d2->print();
  delete d2;
}
```

Constructors

- The special methods that create objects.
- Can take any number of arguments.
  - Standard constructors
  - Default constructor -- takes 0 arguments
  - Copy constructor -- takes a single argument of the same class
- A class can have multiple different constructors:

```
class Date {
public:
  Date();
  Date(int y, int m, int d);
...
```

Destructors

- The destructor is called when the object is deallocated.
- Only one destructor per class, it takes no parameters and is named based on the name of the class with a “~” prepended.
- There’s an important case (we’ll discuss later) where destructors must be marked “virtual” — I just always make them virtual to be safe.
-
  ```
  class Date {
   public:
     Date();
     Date(int y, int m, int d);
     virtual ~Date();
   ...
  ```
- For an object on the stack, called when the block in which it was allocated exits.
- For an object on the heap (allocated with a call to **new**), called when the **delete** operation is used on the pointer to the object.

Done. That was the smallest possible technical intro to the syntax of how to create C++ classes.

Now, I want to talk about good design, which is much more difficult to learn from a textbook or google:

What makes a good class Interface?

Why is this so critical to good software design?

Good and Bad Class Interfaces

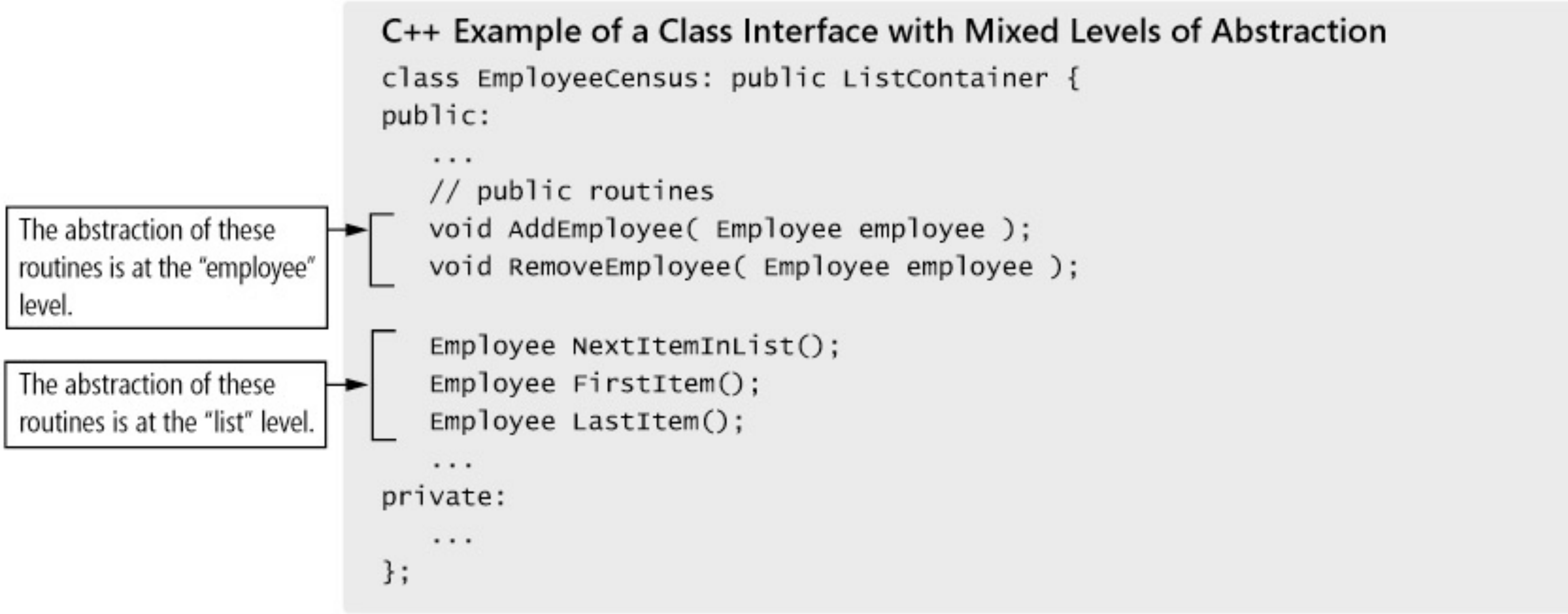
- See McConnell for a list of rules of thumb.
- We’ll look at some specific examples.

Good Abstraction in C++

```cpp
class Employee {
public:
   // public constructors and destructors
   Employee();
   Employee(FullName name,
            String address,
            String workPhone,
            String homePhone,
            TaxId taxIdNumber,
            JobClassification jobClass);
   virtual ~Employee();

   // public routines
   FullName GetName() const;
   String GetAddress() const;
   String GetWorkPhone() const;
   String GetHomePhone() const;
   TaxId GetTaxIdNumber() const;
   JobClassification GetJobClassification() const;
   ...

private:
   ...
};
```



C++ Example of a Class Interface with Mixed Levels of Abstraction

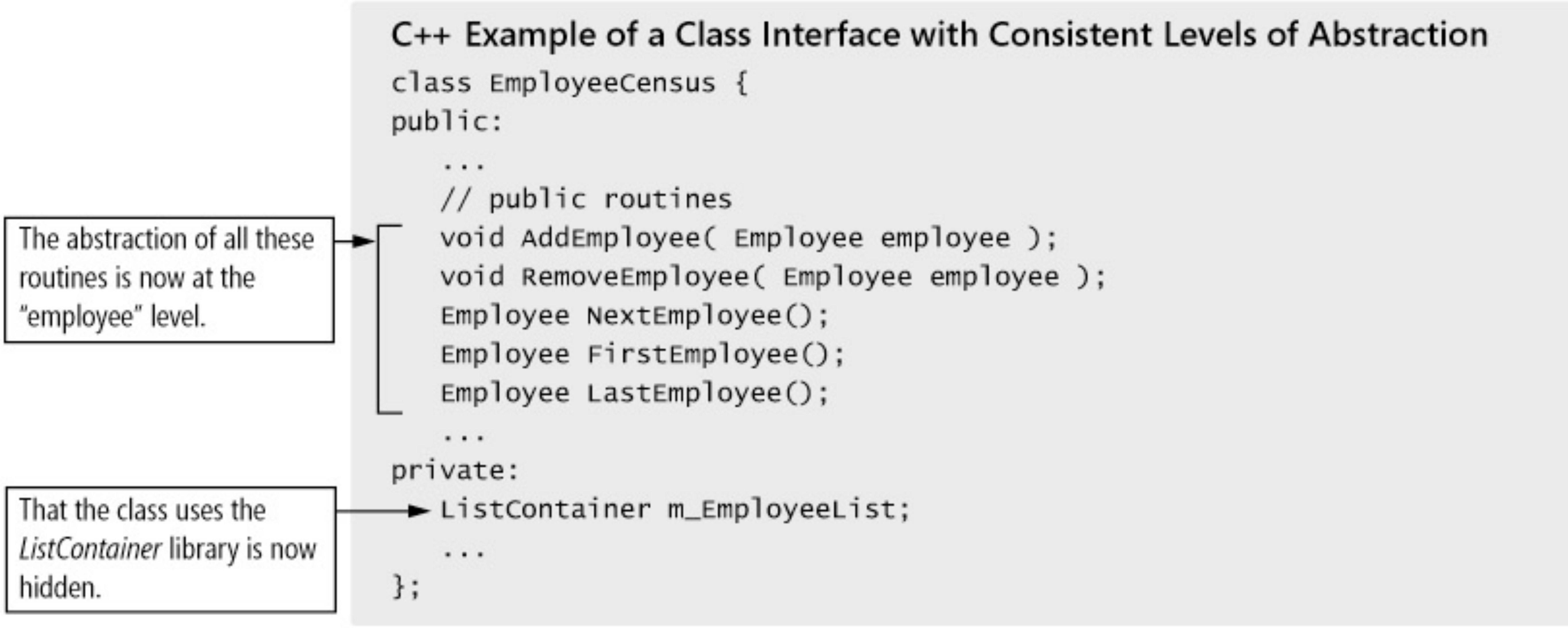
```
class EmployeeCensus: public ListContainer {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );

   Employee NextItemInList();
   Employee FirstItem();
   Employee LastItem();
   ...
private:
   ...
};
```

The abstraction of these routines is at the "employee" level.

The abstraction of these routines is at the "list" level.

- Poor abstraction, mixed levels of abstraction.
- Each class should implement one and only one abstract data type.



C++ Example of a Class Interface with Consistent Levels of Abstraction

```
class EmployeeCensus {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );
   Employee NextEmployee();
   Employee FirstEmployee();
   Employee LastEmployee();
   ...
private:
   ListContainer m_EmployeeList;
   ...
};
```

The abstraction of all these routines is now at the "employee" level.

That the class uses the *ListContainer* library is now hidden.

- Better. The level of abstraction is consistent. Everything is at the Employee level.

```
class Program {
public:
   ...
   // public routines
   void InitializeCommandStack();
   void PushCommand( Command command );
   Command PopCommand();
   void ShutdownCommandStack();
   void InitializeReportFormatting();
   void FormatReport( Report report );
   void PrintReport( Report report );
   void InitializeGlobalData();
   void ShutdownGlobalData();
   ...
private:
   ...
};
```

- Poor Abstraction, collection of miscellaneous functions:

```
class Program {
public:
   ...
   // public routines
   void InitializeUserInterface();
   void ShutDownUserInterface();
   void InitializeReports();
   void ShutDownReports();
   ...
private:
   ...
};
```

- Better Abstraction, consistent.

Class Interfaces Can Erode Over Time

- After a number of new features are added, our beautiful Employee class now looks like this:

```
class Employee {
public:
   ...
   // public routines
   FullName GetName() const;
   Address GetAddress() const;
   PhoneNumber GetWorkPhone() const;
   ...
   bool IsJobClassificationValid( JobClassification jobClass );
   bool IsZipCodeValid( Address address );
   bool IsPhoneNumberValid( PhoneNumber phoneNumber );

   SqlQuery GetQueryToCreateNewEmployee() const;
   SqlQuery GetQueryToModifyEmployee() const;
   SqlQuery GetQueryToRetrieveEmployee() const;
   ...
private:
   ...
};
```

No logical connection between Employee and zip code checks, etc.
SQL Details are at a different level of abstraction than Employee

Don’t Expose Member Data in Public

- A Point class needs x,y data, which is better?

```
class Point {

public:
   float x;
   float y;
};
```

```
class Point {

public:
   float GetX();
   float GetY();

   void SetX( float x );
   void SetY( float y );

private:
   float x;
   float y;
};
```

Can you design a good C++ class interface now?

- We just looked at a 2D point.
- Now, how about an ADT for a Circle? You design it.

  - I'd like to be able to adjust its radius as my program executes.
  - I'd also like to be able to calculate the area of the circle.
  - And, I might want to move the center point (x,y) of the circle.

The next slides contain the most important concept for designing good object oriented programs!

Good Class Design: Two Types of Relationships between Classes: “has a” and “is a”

(note the triple underline,
the professor must think this is important)

C++ Example

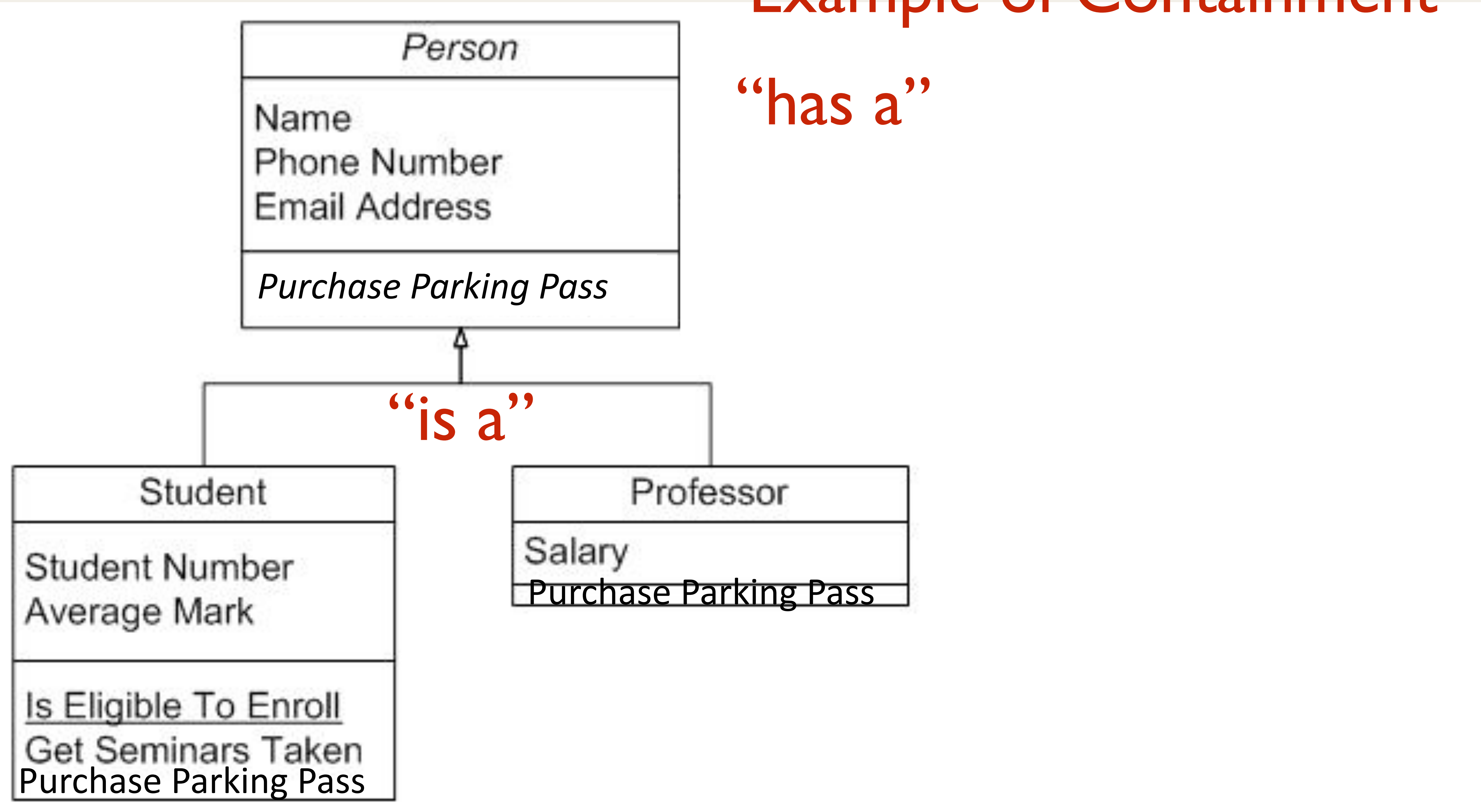
```
class Shape {
public:
  // virtual means subclasses can override this function
  virtual float area();
}

class Square : public Shape {
public:
  float area();
}
```

Another Example (by the way this is a UML diagram)

Example of Containment

“has a”



Example of Inheritance

Before next time…