Lecture 11

More Inheritance, More Constructors

“Absolute C++”
Chapter 15

Overriding
- Let’s recall the code we were working on from last lecture.
- We had a base class named Person and two derived classes named Student and Instructor.
- We defined a method named printInfo() in the base class that prints out generic information, and then override that method in the Student class (but did not define it in the Instructor class).
- We then implemented the following global function:

```cpp
void printPersonInfo(Person &aPerson)
{
    aPerson.printInfo();
}
```

Overriding (cont)
- And tried to call it with the following code:

```cpp
void printPersonInfo(Person &aPerson)
{
    aPerson.printInfo();
}
```

```cpp
int main()
{
    Student aStudent;
    Instructor anInstructor;
    aStudent.setInfo("Joe Student","1 E Main St","555-1212");
    aStudent.studentID = 33445;
    anInstructor.setInfo("Ron D","120 Maple Ave","555-1313");
    anInstructor.employeeID = 12345;
    printPersonInfo(aStudent);
    printPersonInfo(anInstructor);
}
```

Overriding (cont)
- But we didn’t see the printInfo() method defined in Student.
- Did the compiler forget that we override Person::printInfo() in the derived class Student?
- No. Recall that we didn’t get any complaints from the compiler when we passed anInstructor and aStudent in to the function printPersonInfo(Person &).
- It’s legal to do that; since Instructor and Student are derived from Person, the compiler thinks we want to treat whatever argument is passed in as a Person.
- And, since inside the scope of printPersonInfo the argument passed is an instance of a Person, Person::printInfo() is used when we call aPerson.printInfo().
- Well, doesn’t that make overriding somewhat useless?

Virtual Functions
- No, we can achieve the desired behavior by making one minor adjustment to the Person class:

```cpp
class Person
{
    public:
    void setInfo(string Name, string Addr, string Phone);
    virtual void printInfo();

    private:
    string name;
    string address;
    string phone;
};
```

- Does this really make a difference?

Demonstration #1

Virtual Functions
Virtual Functions
- WOW! What just happened?
- By defining `Person::printInfo()` as a virtual function, we told the compiler to keep track of any instances of derived classes which may override this function and make sure the overridden version gets called no matter what type that instance may be cast to.
- This is usually the desired behavior
- When a member function is declared as a virtual function, derived classes have the option of overriding it.
  - If they do not, the member function in the base class is always called
- There’s one catch, though…
  - In order to get this behavior, we needed to declare the argument to `printPersonInfo` as a “Person &” (or Person *). Had we just used `Person`, a copy of the argument passed would have been used and would have retained no knowledge about actually being a derived class...

Demonstration #2
Virtual Functions (Pass by Value)

Pure Virtual Functions
- Suppose the case arose where we wanted to force a derived class to override a specific member function in the base class.
- Why would we want to do that?
- Suppose there were a common function implemented across all the derived classes which didn’t necessarily make sense to include in the base class.
- Consider a simple member function which would print out a Person’s “classification” (student, faculty, staff, etc.).
- Maybe it would look like this

```cpp
void Person::printClassification()
{
    // This is where the classification is
}
```

- Now, this will seemingly fit the bill, but there’s one problem...

Pure Virtual Functions (cont)
- How can we call it from our `printPersonInfo()` function?
- We could add a new member variable to keep track of type...

```cpp
class Person
{
public:
    // setInfo
    void setInfo(string Name, string Addr, string Phone);
    // virtual void printInfo();
    // private:
    string name;
    string address;
    string phone;
    int PersonType;
};
```
- Then make sure we populate this field in the derived class...

Pure Virtual Functions (cont)
- Then do something like the following...

```cpp
void printPersonInfo(Person *aPerson) // have to pass pointer
{
    aPerson->printInfo();
    // Now print classification
    switch( aPerson->personType )
    {
        case kStudentType: // Assume "type" constants exist
            Student *aStudent = (Student *) aPerson;
            aStudent->printClassification();
            break;
        case kInstructorType:
            Instructor *anInstructor = (Instructor *) aPerson;
            anInstructor->printClassification();
            break;
    }
}
```
Pure Virtual Functions (cont)
- I don’t think so!
- C++ gives us a way to declare a member function in the base class and specify that every derived class must implement it (because there is no “default” implementation in the base class)
- This is called a Pure Virtual Function

```cpp
class Person 
{ 
public: 
    void setInfo(string Name, string Addr, string Phone); 
    virtual void printInfo(); 
    virtual void printClassification() = 0; // Pure Virtual 
private: 
    string name; 
    string address; 
    string phone; 
};
```

You declare a member function to be pure virtual by adding a “= 0” initializer right after the declaration.
After doing this, our printPersonInfo() function becomes simple again...

```cpp
void printPersonInfo(Person &aPerson) 
{ 
aPerson.printInfo(); 
aPerson.printClassification(); // Call pure virtual function 
}
```

Let’s see this work...

Pure Virtual Functions and Abstract Classes
- As we just saw, declaring printClassification() as pure virtual caused compiler errors when we tried to work with derived classes which did not define the pure virtual member function.
- The error messages we received made reference to “abstract class”
- An abstract class is simply a base class which contains one or more pure virtual member functions.
- As such, an instance of an abstract class can never be allocated.
- You must always declare or allocate an instance of one of its derived classes.
- This means our printPersonInfo() function must either be passed a reference or pointer to Person.
- Let’s define printClassification() in our derived classes and try again...

Demonstration #3

Demonstration #4

Constructors—Initialization Shorthand
- Sometimes it is tedious to write out all of the initializations like we do below:

```cpp
Course::Course(string theCourseName, string theInstructor, int classSize) 
{ 
    courseName = theCourseName; 
    instructor = theInstructor; 
    size = classSize; 
}
```

There is a “shorthand” we can use to simplify this:
Initialization Shorthand (cont)
- Initialization shorthand:

```cpp
Course::Course(string theCourseName, string theInstructor, int classSize) {
    courseName = theCourseName;
    instructor = theInstructor;
    size = classSize;
}
```

- Any member variable may be initialized in any constructor for the same class in this manner.
- The format is to append the following expression after the parameter list:
  ```cpp
  member-name(expression) {}, member-name(expression)
  ```

Constructors--Quick Summary
- A default constructor is a constructor which takes no arguments
  - If you declare additional constructors you may need to provide a default constructor which does nothing (if you haven’t defined one already)
  - Otherwise you may get “Can’t construct class” errors when trying to create an instance of the class without passing arguments.
- Other constructors may be added which take arguments
  - This is called constructor overloading.
    - A specific form of function overloading, which we’ll discuss a little later
  - The linker will make sure the right one is called, depending on the arguments passed (or lack thereof)
- A shorthand way to initialize member variables in a Constructor’s definition is to follow the parameter list with a colon followed by a comma separated list of member variable names and their initial values in parenthesis.

Constructors and Resource Allocation
- Another common use of constructors is to allocate system resources
  - Memory, GUI objects (Windows, Menus, etc.)
  - Other dynamic structures/classes
  - Consider a modification to the Course class from last lecture which allows us to store a dynamic array of Students as a member variable.

```cpp
class Course {
    public:
        Course();
        Course(string theCourse, string theInstructor, int classSize);
    private:
        string courseName;
        string instructor;
        int size;
        Student *studentList;
        int nextStudent;
    }
```

- It’s OK to move the initializations back into the body of the constructor if you’re starting to make a mess!

Constructors and Inheritance
- Remember our Person class from earlier?

```cpp
class Person {
    public:
        void setInfo(string Name, string Addr, string Phone);
        virtual void printInfo();
        virtual void printClassification() = 0; // Pure Virtual
    private:
        string name;
        string address;
        string phone;
    }
```

- We could apply what we’ve learned about constructors to do the following:

```cpp
class Person {
    public:
        Person(string Name, string Addr, string Phone): name(Name), address(Addr), phone(Phone) {};
        void setInfo(string Name, string Addr, string Phone);
        virtual void printInfo();
        virtual void printClassification() = 0; // Pure Virtual
    private:
        string name;
        string address;
        string phone;
    }
```

- Oh yeah, constructors can be defined in the class definition too!
Constructors and Inheritance (cont)
- Oh, wait. Person is an abstract class (has one or more pure virtual functions).
- That means that we can never create a "standalone" instance of Person.
- Hmmm, can we do something with the constructors of the derived classes?
- Let's look at our Student class again and add a constructor their too...

```cpp
// Demonstrating the Student class

class Student : public Person
{
public:
    Student(string Name, string Addr, string Phone, int id);
    void printInfo();
    int getId() { return studentID; }
    void printClassification();
private:
    int studentID;
};

Student::Student(string Name, string Addr, string Phone, int id):
    Person(Name, Addr, Phone), studentID(id)
{ }

// Let's verify that this does indeed work...
```

Constructors and Inheritance (cont)
- Nope, it won't work.
- You can't access the private members of Person, even from the derived class!
- However, you can call Person's constructor!

```cpp
Student::Student (string Name, string Addr, string Phone, int id):
    Person(Name, Addr, Phone), studentID(id)
{ }
```

Demonstration #5

- Will it work?