The Conceptual Side of Classes

- Earlier we defined classes as a user defined data type
  - It could have member functions
  - It could have member variables
  - This is a technical, concrete definition
- The conceptual definition of classes is just that--it's a concept
  - Usually a noun
    - trees, birds, people, a person, dog, food, hot dogs, computers, etc.
  - Verbs don't usually make good classes
    - thinking, running, listening, laughing, crying
- When we define a class, we're providing a description for a class of "things".
- A variable or instance of a class is thought of as one "member" of the class.
- For example:

```cpp
class Student
{
    string name;
    string address;
    string localPhone;
    int studentID;
};
```

The Conceptual Side of Classes (cont)

- Sometimes multiple classes have similarities:
  - class Student
    { string name; string address; string localPhone; int studentID; }
  - class Instructor
    { string name; string address; string phone; string employeeID; }

The Conceptual Side of Classes (cont)

- Sometimes the similarities are common to a broader class than the class being defined
  - In the case of Student and Instructor, consider the common fields:
    - name
    - address
    - phone
  - Suppose we create a class called "Person", as follows:

```cpp
class Person
{
    string name;
    string address;
    string phone;
};
```
The Conceptual Side of Classes (cont)

• Now maybe you’d think that we could do this:

```cpp
class Student
{
    Person imAPerson;
    int studentID;
};
class Instructor
{
    Person imAPerson;
    int employeeID;
};
```

• We can, in fact do this. 
  - But then any instance would have to access fields in Person through the `imAPerson` member variable.

Inheritance

• A better way to do this is with Inheritance
  - In C++, when one class inherits another 
    - all public (and protected) member variables in the “base class” are accessible from the “derived class” as if they were declared right in the derived class.
  - In our example:
    - Person is the base class
    - Student is the derived class
  - To declare Student as being a derivation of Person, do this:
```cpp
class Student : public Person
{ 
    int studentID;
};
class Instructor : public Person
{ 
    int employeeID;
};
```

Inheritance (cont)

• Now, given the following declarations:
```cpp
class Person
{ 
    public:
        string name;
        string address;
        string localPhone;
    
};
class Student : public Person
{ 
    public:
        int studentID;
    
};
```

• We can write the following code:
```cpp
int main()
{
    Student aStudent;
    aStudent.name = "Jon Doe";    // Defined in Person
    aStudent.address = "12 Park Place"; // Defined in Person
    aStudent.phone = "555-1212";    // Defined in Person
    aStudent.studentID = 442221;    // Defined in Student
    
    return 0;
}
```

Let’s see this in action:

Protected Members

• A derived class may access any of the public members of the base class, and so can anyone else using the base class directly.
• A derived class may NOT access any of the private members of the base class, nor may anyone else using the base class directly.
• A derived class may access any of the protected members of the base class, but no one using the base class directly may access them.
• To mark a member variable or function as protected, do the following:
```cpp
class Person
{ 
    protected:
        string name;
        string address;
        string phone;
    
};
```
• To clarify, when a member function or variable follows a protected keyword:
  – Only member functions defined in a derived class may access the protected member functions/variables in the base class
  – All other classes (not derived from the base class) may not access the protected member functions/variables

• Let’s look at some code:

```cpp
class Person
{
public:
    void setInfo(string Name, string Addr, string Phone);
protected:
    string name;
    string address;
    string phone;
};
```

• Now Consider a Derived Class:

```cpp
class Student: public Person
{
public:
    void printInfo();
    int getId() { return studentID; } // getID() is.
private:
    int studentID;
};
```

```cpp
void Student::printInfo()
{
    cout << "Name: " << name << endl; // name, address and
    cout << "Addr: " << address << endl; // phone are
    cout << "Phone: " << phone << endl; // in the base class
}
```

• Finally, let’s use it...

```cpp
int main()
{
    Student aStudent;
    aStudent.name = "Joe Student"; // ??
    aStudent.address = "166 Phelps Lane"; // ??
    aStudent.phone = "555-1212"; // ??
    aStudent.printInfo();
}
```

• Since name, address and phone are declared as protected members of the Person class...
  – They cannot be accessed “outside” of the class

```cpp
void Person::setInfo(string Name, string Addr, string Phone)
{
    name = Name;
    addr = Addr;
    phone = Phone;
}
```

• So the right way to do it (in this particular case) is:

```cpp
int main()
{
    Student aStudent;
    // Now set the information. Remember, setInfo() is
    // defined in the "Person" class
    aStudent.setInfo("Joe Student", "166 Phelps Lane", "555-1212");
    aStudent.printInfo();
}
```

• Let’s see this in action...

```cpp
Demonstration #2
```

```
Protected Members (cont)

• Since name, address and phone are declared as protected members of the Person class...
  – They cannot be accessed “outside” of the class

• So the right way to do it (in this particular case) is:

```cpp
int main()
{
    Student aStudent;
    // Now set the information. Remember, setInfo() is
    // defined in the "Person" class
    aStudent.setInfo("Joe Student", "166 Phelps Lane", "555-1212");
    aStudent.printInfo();
}
```

• Let’s see this in action...

```cpp
Demonstration #2
```

```
Protected Members (cont)

• But they can be accessed inside of the derived class

```cpp
void Student::printInfo()
{
    cout << "Name: " << name << endl; // name, address and
    cout << "Addr: " << address << endl; // phone are
    cout << "Phone: " << phone << endl; // in the base class
}
```

• The Person class had its own public method for setting info:

```cpp
void Person::setInfo(string Name, string Addr, string Phone)
{
    name = Name;
    addr = Addr;
    phone = Phone;
}
```

• Now Consider a Derived Class:

```cpp
class Student: public Person
{
public:
    void printInfo();
    int getId() { return studentID; } // getID() is.
private:
    int studentID;
};
```

```cpp
void Student::printInfo()
{
    cout << "Name: " << name << endl; // name, address and
    cout << "Addr: " << address << endl; // phone are
    cout << "Phone: " << phone << endl; // in the base class
}
```

```
Protected Members (cont)

• Finally, let’s use it...

```cpp
int main()
{
    Student aStudent;
    aStudent.name = "Joe Student"; // ??
    aStudent.address = "166 Phelps Lane"; // ??
    aStudent.phone = "555-1212";            // ??
    aStudent.printInfo();
}
```

• Since name, address and phone are declared as protected members of the Person class...
  – They cannot be accessed “outside” of the class

```cpp
void Person::setInfo(string Name, string Addr, string Phone)
{
    name = Name;
    addr = Addr;
    phone = Phone;
}
```

• So the right way to do it (in this particular case) is:

```cpp
int main()
{
    Student aStudent;
    // Now set the information. Remember, setInfo() is
    // defined in the "Person" class
    aStudent.setInfo("Joe Student", "166 Phelps Lane", "555-1212");
    aStudent.printInfo();
}
```

• Let’s see this in action...

```cpp
Demonstration #2
```

```
Protected Members (cont)

• Finally, let’s use it...

```cpp
int main()
{
    Student aStudent;
    aStudent.name = "Joe Student"; // ??
    aStudent.address = "166 Phelps Lane"; // ??
    aStudent.phone = "555-1212";            // ??
    aStudent.printInfo();
}
```
Cleaning Up our Implementation

* You might think that the Person class should print its own data:

```cpp
class Person {
public:
    void setInfo(string Name, string Addr, string Phone);
    void printInfo();
private:
    string name;
    string address;
    string phone;
};

void Person::printInfo()
{
    cout << "Name: " << name << endl;
    cout << "Addr: " << address << endl;
    cout << "Phone: " << phone << endl;
}
```

Cleaning Up our Implementation

* That makes a certain amount of sense...

```cpp
class Instructor : public Person {
private:
    int employeeID;
};

int main()
{
    Instructor anInstructor;
    anInstructor.setInfo("Ron DiNapoli", "120 Maple Ave", "555-1313");
    anInstructor.printInfo();
}
```

Cleaning Up our Implementation

* But what about things we might want to print out in a derived class that aren’t present in the base class?
  - studentID field in the Student class.
  - employeeID field in the Employee class.
* Is there any way to include them in the Person::printInfo() member function?
* Not really, but we can do the next best thing.
  * We could have a special definition of printInfo which is used when we’re dealing with a Student class instance

```cpp
void Student::printInfo()
{
    cout << "Student ID: " << studentID << endl;
    // Hmmmm, how can I call the printInfo() from Person?
}
```

Cleaning Up our Implementation

* Let’s find out...

Cleaning Up our Implementation

* Wait a minute. If we already have printInfo defined in Person, can we define it Student as well?

```cpp
void Student::printInfo()
{
    cout << "Student ID: " << studentID << endl;
    cout << "Name: " << name << endl;
    cout << "Addr: " << address << endl;
    cout << "Phone: " << phone << endl;

    // Hmmmm, how can I call the printInfo() from Person?
}
```

Overriding

* Yes, it does work.
* Whenever a derived class defines a member function that is also defined in the base class it is said that the definition in the derived class overrides the definition in the base class.
* In our previous example, Student::printInfo() overrides Person::printInfo()
* However, consider the case where we’d like to write a function that can take a Person as an argument and will cause that person’s printInfo method to be invoked:

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

• Let’s consider the following code:

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

Demonstration #4

Redefining Base Class Member Functions II

Overriding (cont)

• So, wait a minute. Did the compiler forget that we overrode `Person::printInfo()` in the derived class `Student`?
  • No, it’s only doing what it was told to do!
• Recall that we didn’t get any complaints from the compiler when we passed `anInstructor` and `aStudent` into 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 #5

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 #6

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()
{
    cout << “This person’s classification is…” << ????
};
```

Pure Virtual Functions (cont)

• The problem here is that the Person class has no idea what “classification” a person is… That is handled in the derived class.
• So it would be very easy to implement printClassification as a member function of each derived class...

```cpp
void Student::printClassification()
{
    cout << “Classification: STUDENT” << endl;
}
void Instructor::printClassification()
{
    cout << “Classification: INSTRUCTOR” << endl;
}
```

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

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)

• 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:
    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)

• 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;
};
```
Pure Virtual Functions (cont)

- 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...

Demonstration #7

Pure Virtual Functions

Pure Virtual Functions & 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 #8

Pure Virtual Functions II

Final Thoughts

- Assignment #3 due Thursday
- This stuff is beginning to get tougher, please visit office hours as needed...
- According to LNG, we have not even started learning C++ yet… :-)

Assignment #3 due Thursday
- This stuff is beginning to get tougher, please visit office hours as needed...
- According to LNG, we have not even started learning C++ yet… :-)