Inheritance and Polymorphism
Assignment 11

- **Gradebook.h** *(no need to modify)*
- **Gradebook.cpp** *(no need to modify)*
- **GradebookRunner.cpp** *(no need to modify)*
- **Student.h** *(no need to modify)*
- **Student.cpp** *(this is where you'll do your work)*

You need to implement the Student class. We’ll go into this a little more in a bit.
Gradebook Interface

typedef Student* StudentPtr;
class Gradebook {
public:
    Gradebook(CString aName);
    ~Gradebook();
    CString getName() const;
    void addStudent(const Student& aStudent);
    void addGrade(CString aStudentId, const string& aGrade);
    void passByValue(Student aStudent) const;
private:
    CString mName;
    StudentPtr mStudents;
    int mStudentSize;
    map<CString, string*> mGrades;
    map<CString, int> mNumOfGrades;
};
typedef char* CString;
#include <cstddef> // NULL

class Student {
public:
  Student();
  Student(const CString aName, const CString aId);
  ~Student();
  CString getName() const;
  void setName(const CString aName);
  CString getStudentId() const;
  void setId(const CString aId);
  Student& operator=(const Student aStudent);
private:
  CString mName;
  CString mId;
};
GradebookRunner - main()

cout << "********************Student Declaration*******************\n";
Student student1;
student1.setName("John Doe");
student1.setId("00002222");

cout << "********************Gradebook Declaration*******************\n";
Gradebook csci123GB("CSCI 123 Gradebook");

cout << "********************Add a Student to Gradebook*******************\n";
csci123GB.addStudent(student1);

cout << "********************Add a Grade to Gradebook*******************\n";
csci123GB.addGrade(student1.getStudentId(), "A");

cout << "********************Add a Grade to Gradebook*******************\n";
csci123GB.addGrade(student1.getStudentId(), "B");

cout << "********************Pass a Student by value*******************\n";
csci123GB.passByValue(student1);

cout << "********************End Pass a Student by value*******************\n";
What you need to do

That explains what you need to do. Again you only need to modify the Student.cpp file.

Here's the modifications:

1. add any namespaces and/or includes
2. Implement the default constructor add code so that the student name is set to "No Name" and the student id is set to "No Id" by default
3. Implement the two parameter constructor set the object's name and the id to the parameters passed to the constructor
4. Implement the code for the copy constructor. Set the object's members to the Student passed to the copy constructor
5. Implement the destructor for the Student class. It should remove any allocated memory from the heap.
6. Implement the assignment operator. It should assign the left hand Student to the right hand student. Don't remove my return statement.
7. Implement the setName member function.
8. Implement the setId member function.
Each of the functions that you need to implement has the following comment above the function.

The function also has couts in the function bodies, you want to leave these in the file or you won’t get the proper output.

The assignment should show you how the constructors, destructors, and assignment operators are called.
Gradebook Implementation

Gradebook::Gradebook(const CString aName) : mStudentSize(0) {
    cout << "Gradebook() - Default Constructor\n";
    cout << "\tAllocating memory for a name of length " << strlen(aName) << endl;
    mName = new char[strlen(aName)+1];
    strcpy(mName, aName);
    cout << "\tAllocating memory for 10 students\n";
    mStudents = new Student[10];
}
Gradebook::~Gradebook() {
    cout << "~Gradebook() - Destructor\n";
    cout << "\tDeleting Gradebook Name\n";
    delete [] mName;
    cout << "\tDeleting Gradebook Students\n";
    delete [] mStudents;

    int i = 0;
    for (map<CString, string*>::iterator p = mGrades.begin(); p != mGrades.end(); ++p) {
        cout << "Deleting the collection of grades ";
        cout << "for student " << p->first << endl;
        cout << endl;
        // p-> returns a string pointer
        delete [] p->second; // delete the collection of grades
        // for this student
    }
}
Gradebook Implementation

CString Gradebook::getName() const {
    return mName;
}

Gradebook Implementation

```cpp
void Gradebook::addStudent(const Student& aStudent) {
    cout << "Adding a student\n";
    if(!atoi(aStudent.getStudentId())) {
        cout << "Invalid student id!\n";
        cout << "You can only add a student with a valid student id\n\n";
    } else if(mStudentSize >= 0 && mStudentSize <= 9) {
        mStudents[mStudentSize++] = aStudent;
        cout << "Allocating memory for ten grades for this student\n";
        mGrades[aStudent.getStudentId()] = new string[10];
        // set the number of grades for this student to
        // zero
        mNumOfGrades[aStudent.getStudentId()] = 0;
    } else {
        // only allow 10 grades for each student
        cout << "Gradebook is full of students\n";
    }
}
```
Gradebook Implementation

```cpp
void Gradebook::addGrade(CString aStudentId, const string& aGrade) {
    if(!atoi(aStudentId)) {
        cout << "Invalid student id!\n";
        cout << "You can only add a grade with a valid student id\n\n";
    } else if(mNumOfGrades[aStudentId] >= 0 &&
               mNumOfGrades[aStudentId] <= 9) {
        cout << "Adding a grade for student with id " << aStudentId <<
             endl;
        mGrades[aStudentId][mNumOfGrades[aStudentId]] = aGrade;
        mNumOfGrades[aStudentId] = mNumOfGrades[aStudentId] + 1;
    } else {
        cout << "Gradebook is full of grades\n";
    }
}
```
void Gradebook::passByValue(Student aStudent) const {
    cout << "PassByValue aStudent\n";
    // what gets called here?
}

Gradebook Implementation
Overview

15.1 Inheritance Basics
15.2 Inheritance Details
15.3 Polymorphism
Inheritance
Inheritance Basics

- Inheritance is the process by which a new class, called a derived class, is created from another class, called the base class
  - A derived class automatically has all the member variables and functions of the base class
  - A derived class can have additional member variables and/or member functions
  - The derived class is a child of the base or parent class
Shape Classes

**Shape**

- color: string
- filled: bool

+Shape()
+Shape(color: string, filled: bool)
+getColor(): string
+setColor(color: string): void
+isFilled(): bool
+setFilled(filled: bool): void
+toString(): string

The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
Creates a Shape.
Creates a Shape with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns a string representation of this object.

**Circle**

- radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: bool)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+toString(): string

**Rectangle**

- width: double
- height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: bool)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

A Base Class

- We will define a class called Shape for all Shapes (Rectangles and Circles)
- The Shape class will be used to define classes for Rectangles and Circles

- A definition of the Shape class is found in the following files:
  Shape.h and Shape.cpp
  Circle.h and Circle.cpp
  Rectangle.h and Rectangle.cpp
  ShapeRunner.cpp
Class Rectangle & Circle

- Rectangle and Circle are derived from the Shape class
  - Rectangle and Circle inherit all member functions and member variables of Shape
  - The class definition begins

```cpp
class Rectangle : public Shape
```

- :public Shape shows that Rectangle is derived from Shape
- Rectangle declares additional member variables width and height
- Circle declares additional member variable radius
Inherited Members

- A derived class inherits all the members of the parent class
  - The derived class does not re-declare or re-define members inherited from the parent, except...
  - The derived class re-declares and re-defines member functions of the parent class that will have a different definition in the derived class
  - The derived class can add member variables and functions
Implementing a Derived Class

- Any member functions added in the derived class are defined in the implementation file for the derived class
  - Definitions are not given for inherited functions that are not to be changed

- The Rectangle and Circle implementation can be found in the Rectangle.cpp and Circle.cpp files
Function toString()

- Function toString() will have different definitions in the Shape and Circle class
  - toString prints the color and whether the shape is filled
  - toString is redefined in the Circle class
  - the toString() function in the Circle class hides the function in the base class, Shape
Parent and Child Classes

- Recall that a child class automatically has **all** the members of the parent class.
- The parent class is an ancestor of the child class.
- The child class is a descendent of the parent class.
- The parent class (Shape) contains all the code common to the child classes.
  - You do not have to re-write the code for each child.
Derived Class Types

- An Circle is a Shape
  - In C++, an object of type Circle can be used where an object of type Shape can be used
  - An object of a class type can be used wherever any of its ancestors can be used
  - An ancestor cannot be used wherever one of its descendents can be used
A base class constructor is not inherited in a derived class
- The base class constructor can be invoked by the constructor of the derived class
- The constructor of a derived class begins by invoking the constructor of the base class in the initialization section:

```cpp
Circle::Circle : Shape()
  radius(1)
{ //no code needed }
```

Any Shape constructor can be invoked
Default Initialization

- If a derived class constructor does not invoke a base class constructor explicitly, the base class default constructor will be used.
- If class B is derived from class A and class C is derived from class B:
  - When a object of class C is created:
    - The base class A's constructor is the first invoked.
    - Class B's constructor is invoked next.
    - C's constructor completes execution.
Calling Base Class Constructors

A constructor is used to construct an instance of a class. Unlike data fields and functions, the constructors of a base class are not inherited in the derived class. They can only be invoked from the constructors of the derived classes to initialize the data fields in the base class. The syntax to invoke it is as follows:

```cpp
DerivedClass(parameterList): BaseClass() {
    // Perform initialization
}

Or

DerivedClass(parameterList): BaseClass(argumentList) {
    // Perform initialization
}
```
Circle constructor:

```cpp
Circle::Circle(double radius) : Shape("black", false) {
    this->radius = radius;
}
```
Private is Private

- A member variable (or function) that is private in the parent class is not accessible to the child class
  - The parent class member functions (accessors) must be used to access the private members of the parent

This code would be illegal:

```cpp
string Circle::toString() {
    return "Circle color " + this->color +
    " filled " + ((this->filled) ? "true" : "false");
}
```

- color and filled is a private member of Shape!
The protected Qualifier

- protected members of a class appear to be private outside the class, but are accessible by derived classes.

If member variables name, color, and filled are listed as protected (not private) in the Shape class, this code, illegal on the previous slide, becomes legal:

```cpp
string Circle::toString() {
    return "Circle color " + this->color + " filled " + ((this->filled) ? "true" : "false");
}
```
Programming Style

- Using **protected** members of a class is a convenience to facilitate writing the code of derived classes.
- Protected members are not necessary
  - Derived classes can use the public functions of their ancestor classes to access private members
- Many programming authorities consider it **bad** style to use protected member variables
Redefinition of Member Functions

- When defining a derived class, only list the inherited functions that you wish to change for the derived class
  - The function is declared in the class definition
  - Circle has its own definition of toString()

- How does C++ know to call the derived class definition of toString()?
Redefining or Overloading

- A function redefined in a derived class has the same number and type of parameters
  - The derived class has only one function with the same name as the base class

- An overloaded function has a different number and/or type of parameters than the base class
  - The derived class has a function with the same name as the base class
    - One (function – toString() ) is defined in the base class (Shape), one in the derived class (Circle)
Function Signatures

- A function signature is the function's name with the sequence of types in the parameter list, not including any const or '&'
  - An overloaded function has multiple signatures
- Some compilers allow overloading based on including const (Visual C++) or not including const
Access to a Redefined Base Function

- When a base class function is redefined in a derived class, the base class function can still be used.
- To specify that you want to use the base class version of the redefined function:

```cpp
cout << "Base Class toString()\n";
cout << circle.Shape::toString() << endl;
```
Section 15.1 Conclusion

- Can you
  - Explain why the declaration for getColor is not part of the definition of Circle?
  - Give a definition for a class Triangle derived from class Shape with one additional string called title? Add two member functions getTitle and setTitle.
Inheritance Details

15.2
Inheritance Details

- Some special functions are, for all practical purposes, not inherited by a derived class
  - Some of the special functions that are not effectively inherited by a derived class include
    - Destructors
    - Copy constructors
    - The assignment operator
Copy Constructors and Derived Classes

- If a copy constructor is not defined in a derived class, C++ will generate a default copy constructor
  - This copy constructor copies only the contents of member variables and will not work with pointers and dynamic variables
  - The base class copy constructor will not be used

- Why won’t the generated copy constructor work for pointers?
Operator = and Derived Classes

- If a base class has a defined assignment operator = and the derived class does not:
  - C++ will use a default operator that will have nothing to do with the base class assignment operator
Destructors and Derived Classes

- A destructor is not inherited by a derived class
- The derived class should define its own destructor
The Assignment Operator

- In implementing an overloaded assignment operator in a derived class:
  - It is normal to use the assignment operator from the base class in the definition of the derived class's assignment operator
  - Recall that the assignment operator is written as a member function of a class
The Operator = Implementation

- This code segment shows how to begin the implementation of the = operator for a derived class:

```cpp
Derived& Derived::operator=(const Derived& rhs) {
    Base::operator=(rhs)    // call the base class operator
}
```

- This line handles the assignment of the inherited member variables by calling the base class assignment operator
- The remaining code would assign the member variables introduced in the derived class
The Copy Constructor

- Implementation of the derived class copy constructor is much like that of the assignment operator:
  
  ```
  Derived::Derived(const Derived& object)
  : Base(object), <other initializing>
  {...}
  ```

- Invoking the base class copy constructor sets up the inherited member variables
  - Since object is of type Derived it is also of type Base
Destructors in Derived Classes

- If the base class has a working destructor, defining the destructor for the derived class is relatively easy
  - When the destructor for a derived class is called, the destructor for the base class is automatically called
  - The derived class destructor need only use delete on dynamic variables added in the derived class, and data they may point to
Destruction Sequence

- If class B is derived from class A and class C is derived from class B...
  - When the destructor of an object of class C goes out of scope
    - The destructor of class C is called
    - Then the destructor of class B
    - Then the destructor of class A
  - Notice that destructors are called in the reverse order of constructor calls
Section 15.2 Conclusion

- Can you
  - List some special functions that are not inherited by a derived class?
  - Write code to invoke the base class copy constructor in defining the derived class's copy constructor?
Polymorphism

15.3
Polymorphism

- Polymorphism refers to the ability to associate multiple meanings with one function name using a mechanism called late binding.
- Polymorphism is a key component of the philosophy of object oriented programming.
A Late Binding Example

- Imagine a graphics program with several types of figures
  - Each figure may be an object of a different class, such as a Circle, Oval, Rectangle, etc.
  - Each is a descendant of a class Shape
  - Each has a function `draw( )` implemented with code specific to each shape
  - Class Shape has functions common to all shapes
A Problem

- Suppose that class Shape has a function center
  - Function center moves a shape to the center of the screen by erasing the shape and redrawing it in the center of the screen
  - Function center is inherited by each of the derived classes
    - Function center uses each derived object's draw function to draw the shape
    - The Shape class does not know about its derived classes, so it cannot know how to draw each figure
Virtual Functions

- Because the Shape class includes a method to draw shapes, but the Shape class cannot know how to draw the different shapes, virtual functions are used.

- Making a function virtual tells the compiler that you don't know how the function is implemented and to wait until the function is used in a program, then get the implementation from the object.
  - This is called late binding.
void displayObject(BaseClass someClass) {
    cout << someClass.toString().data() << endl;
}

// each class has its own toString() function
int main() {
    displayObject(BaseClass());
    displayObject(DerivedClass1());
    displayObject(DerivedClass2());
    return 0;
}
No Polymorphic Behavior

- Application Output:

Base Class
Base Class
Base Class
Polymorphic Behavior

class BaseClass {
    public:
        virtual string toString() {
            return "Base Class";
        }
    }
};
class DerivedClass1: public BaseClass {
    string toString() {
        return "DerivedClass 1";
    }
};
Polymorphic Behavior

```cpp
void displayObject(BaseClass *someClass) {
    cout << someClass->toString().data() << endl;
}

int main() {
    BaseClass *bc = new BaseClass();
    DerivedClass1 *dc1 = new DerivedClass1();
    DerivedClass2 *dc2 = new DerivedClass2();
    displayObject(bc);
    displayObject(dc1);
    displayObject(dc2);
    return 0;
}
```
Polymorphic Behavior

- BaseClass toString is declared as virtual, so that polymorphism is turned on.
- Meaning the compiler will wait till runtime to determine which function is called.
- The parameter to displayObject is changed to a pointer.
Define Virtual Functions

To enable dynamic binding for a function, you need to do two things:

- The function must be declared `virtual` in the base class.
- The variable that references the object for the function must contain the address (be a pointer) of the object.

- The keyword `virtual` tells C++ to wait until `displayObject` is used in a program to get the implementation of `displayObject` from the calling object.
Virtual Details

- To define a function differently in a derived class and to make it virtual
  - Add keyword virtual to the function declaration in the base class
  - virtual is not needed for the function declaration in the derived class, but is often included
  - virtual is not added to the function definition
  - Virtual functions require considerable overhead so excessive use reduces program efficiency
Overriding

- Virtual functions whose definitions are changed in a derived class are said to be overridden.

- Non-virtual functions whose definitions are changed in a derived class are redefined.
Type Checking

- C++ carefully checks for type mismatches in the use of values and variables
- This is referred to as strong type checking
  - Generally the type of a value assigned to a variable must match the type of the variable
    - Recall that some automatic type casting occurs
- Strong type checking interferes with the concepts of inheritance
Type Checking and Inheritance

- Consider

  ```
  class Pet {
    public:
      virtual void print();
      string name;
  }
  ```

  and

  ```
  class Dog : public Pet {
    public:
      virtual void print();
      string breed;
  }
  ```
A Sliced Dog is a Pet

- C++ allows the following assignments:
  vdog.name = "Tiny";
  vdog.breed = "Great Dane";
  vpet = vdog;

- However, vpet will lose the breed member of vdog since an object of class Pet has no breed member
  - This code would be illegal: cout << vpet.breed;

- This is the slicing problem
The Slicing Problem

- It is legal to assign a derived class object into a base class variable
  - This slices off data in the derived class that is not also part of the base class
  - Member functions and member variables are lost
Extended Type Compatibility

- It is possible in C++ to avoid the slicing problem
  - Using pointers to dynamic variables we can assign objects of a derived class to variables of a base class without losing members of the derived class object
Dynamic Variables and Derived Classes

- Example:

```cpp
Pet *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;

void Dog::print() {
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}
```

- `ppet->print();` is legal and produces: `name: Tiny` `breed: Great Dane`
Use Virtual Functions

- The previous example:
  `ppet->print();`
  worked because print was declared as a virtual function
- This code would still produce an error:

  ```
  cout << "name: " << ppet->name
       << "breed: " << ppet->breed;
  ```
Why?

- `ppet->breed` is still illegal because `ppet` is a pointer to a `Pet` object that has no `breed` member.
- Function `print()` was declared virtual by class `Pet`:
  - When the computer sees `ppet->print()`, it checks the virtual table for classes `Pet` and `Dog` and finds that `ppet` points to an object of type `Dog`:
    - Because `ppet` points to a `Dog` object, code for `Dog::print()` is used.
Remember Two Rules

- To help make sense of object oriented programming with dynamic variables, remember these rules
  - If the domain type of the pointer pParent is a base class for the for the domain type of pointer pChild, the following assignment of pointers is allowed
    
    \[
    \text{pParent} = \text{pChild};
    \]
    
    and no data members will be lost
  
  - Although all the fields of the pChild are there, virtual functions are required to access them
  
  - Think about the object that was created. What was the constructor that was called? What is at the address?
Virtual Compilation

- When using virtual functions, you will have to define each virtual function before compiling
  - Declaration is no longer sufficient
  - Even if you do not call the virtual function you may see error message:
    "undefined reference to ClassName virtual table"
Virtual Destructors

- Destructors should be made virtual?
  - Consider `Base *pBase = new Derived;
    ...
    delete pBase;
  - If the destructor in Base is virtual, the destructor for Derived is invoked as `pBase` points to a Derived object, returning Derived members to the freestore
    - The Derived destructor in turn calls the Base destructor
Non-Virtual Destructors

- If the Base destructor is not virtual, only the Base destructor is invoked

- This leaves Derived members, not part of Base, in memory
static matching vs. dynamic binding

Matching a function signature and binding a function implementation are two separate issues.

(Static Matching) The declared type of the variable decides which function to match at compile time. The compiler finds a matching function according to parameter type, number of parameters, and order of the parameters at compile time.

(Dynamic Binding) A virtual function may be implemented in several derived classes. C++ dynamically binds the implementation of the function at runtime, decided by the actual class of the object referenced by the variable.
Can you explain why you cannot assign a base class object to a derived class object? Describe the problem with assigning a derived class object to a base class object?