cxx-oop-partII

November 11, 2018

1 OOP Part II: C++

1.1 Topics

1. Recap: Encapsulation
2. Recap: Abstraction
3. Inheritance
    - virtual methods
4. Polymorphism
5. Abstract classes
    - pure virtual methods
6. Composition

1.2 1. Recap: Encapsulation

- Encapsulation hides the details of an object's state from unauthorized parties.
- Only the object's instance and its methods have access to these values.
- Achieved in C++ by maintaining private member variables

```
In [ ]: #include <iostream>
```

```
In [ ]: class Student_Encapsulation_Example {
            // By default, these variables are private
            int id_;
            double gpa_;
            std::vector<std::string> courses_;

          public:
            Student_Encapsulation_Example(int id){
                this->id_ = id;
            };
        }
```

```
In [ ]: Student_Encapsulation_Example s = Student_Encapsulation_Example(25);
```

```
In [ ]: // what will happen when I run this code?
        s.id_
```

Aside: Initializer-list construction In the above example, we used the syntax

```
Student_Encapsulation_Example(int id){
    this->id_ = id;
};
```

for our constructor. In C++11, the initializer list syntax was introduced for constructors, so we could have written:

```
Student_Encapsulation_Example(int id): id_(id){};
```

There are various benefits to using this syntax which we will see later in this lecture. We will also use it for the rest of the examples.

```
In [ ]: class Student_Encapsulation_Example2 {
            // By default, these variables are private
            int id_;
            double gpa_;
            std::vector<std::string> courses_;

          public:
            Student_Encapsulation_Example2(int id): id_(id){};

            //This is the default constructor, which takes no arguments
            //Members are instantiated to their default values
            Student_Encapsulation_Example2(){};

            int get_id(){
                return id_;
            }
        }
```

```
In [ ]: Student_Encapsulation_Example2 se = Student_Encapsulation_Example2(25);
```

```
In [ ]: se.get_id()
```

```
In [ ]: Student_Encapsulation_Example2 default_se =  Student_Encapsulation_Example2();
```

```
In [ ]: // What will this return?
        default_se.get_id()
```

1.2.1 2. Recap: Abstraction

Abstraction is the principle of hiding unecessary details from other objects (Closely related to encapsulation). Other objects don't need to know the details about the inside of another object's class and *how* its methods work. All that is required is knowledge of what the methods do, and how to interact with them.

To illustrate this, we will create two new examples of a Student class, each with the following methods:

- A constructor that takes an integer id as input
- An `add_course()` method that takes a string and gradepoint as input, and returns nothing
- A `print_course_roster()` method that prints out the current roster for the student, and returns nothing
- A `get_gpa()` method that returns the student's current gpa

```
In [ ]: class Student_Abstraction_Example1 {
            int id_;
            double gpa_;
            std::vector<std::string> courses_;
            std::vector<double> course_grades_;

          public:
            Student_Abstraction_Example1(int id):id_(id){};

            void add_course(std::string name, double gradepoint){
                courses_.push_back(name);
                course_grades_.push_back(gradepoint);
                double sumgradepoints = std::accumulate(
                    course_grades_.begin(),
                    course_grades_.end(),
                    0.0);
                gpa_ = sumgradepoints / (double)this->course_grades_.size();
            };

            void print_course_roster(){
                for (int i = 0; i < courses_.size(); i++){
                    std::cout << courses_[i] << std::endl;
                }
            }

            double get_gpa(){
                return gpa_;
            }
        }
```

```
In [ ]: struct Student_Course{
            std::string course_;
            double course_grade_;
            Student_Course(std::string course,
                           double grade):
                course_(course), course_grade_(grade){};
        };

        class Student_Abstraction_Example2 {
            int id_;
            double gpa_;
            std::vector<Student_Course> courses_;
```

```
    public:
      Student_Abstraction_Example2(int id):id_(id){};

      void add_course(std::string name, double gradepoint){
          courses_.emplace_back(name,gradepoint);
          double sumgradepoints =
              std::accumulate(courses_.begin(),
                              courses_.end(),
                              0.0,
                              [](double curr_sum, Student_Course course){
                                  return curr_sum + course.course_grade_;}
                             );
          gpa_ = sumgradepoints / (double)courses_.size();
      }

      void print_course_roster(){
          for (int i = 0; i < courses_.size(); i++){
              std::cout << courses_[i].course_ << std::endl;
          }
      }

      double get_gpa(){
          return gpa_;
      }

  }
```

```
In [ ]: Student_Abstraction_Example1 s1 = Student_Abstraction_Example1(3);
        s1.add_course("CME 211", 3.4);
        s1.print_course_roster();

        Student_Abstraction_Example2 s2 = Student_Abstraction_Example2(3);
        s2.add_course("CME 211", 3.4);
        s2.print_course_roster();
```

```
In [ ]: s1.get_gpa()
```

```
In [ ]: s2.get_gpa()
```

To a user of this code, the two Student classes function identically, though their internal implementations of the course roster are different.

How would a C++ developer communicate to users the functionality of the Student class? One way is through the header file for the student class, which would contain:

file: Student.hpp

```
class Student {
    Student(int id);
    void add_course(std::string name, double gradepoint);
    void print_course_roster();
    double get_gpa();
}
```

file: Student.cpp

```
#include <vector>
#include <iostream>

#include "Student.hpp"

struct Student_Course{
    std::string course_;
    double course_grade_;
    Student_Course(std::string course, double grade):course_(course), course_grade_(grade){};
};

class Student{
    int id_;
    double gpa_;
    std::vector<Student_Course> courses_;

  public:
    Student(int id):id_(id){};

    void add_course(std::string name, double gradepoint){
        courses_.emplace_back(name,gradepoint);
        double sumgradepoints = std::accumulate(courses_.begin(),
                                               courses_.end(),
                                               0.0,
                                               [](double curr_sum, Student_Course course){
                                                   return curr_sum + course.course_grade_;}
                                              );
        gpa_ = sumgradepoints / (double)courses_.size();
    }

    void print_course_roster(){
        for (int i = 0; i < courses_.size(); i++){
            std::cout << courses_[i].course_ << std::endl;
        }
    }

    double get_gpa(){
        return gpa_;
    }

}
```

What if instead of just changing the implementation details of our Student class, we actually wanted to create different kinds of Students?

For example: Suppose that Students could either be SPCD or live on campus, and we wanted the Student object to have different functionality based on this distinction.

This leads us to the principle of **Inheritance**.

1.2.2 3. Inheritance

In [ ]:
```
#include <iostream>
```

In [ ]:
```
class Student {
    int id_;
    double gpa_;
    std::vector<std::string> courses_;
    std::vector<double> course_grades_;

  public:
    Student(int id):id_(id){};

    void add_course(std::string name, double gradepoint){
        courses_.push_back(name);
        course_grades_.push_back(gradepoint);
        double sumgradepoints = std::accumulate(
            course_grades_.begin(),
            course_grades_.end(),
            0.0);
        gpa_ = sumgradepoints / (double)course_grades_.size();
    };

    const void print_course_roster(){
        for (int i = 0; i < courses_.size(); i++){
            std::cout << courses_[i] << std::endl;
        }
    }

    const double get_gpa(){
        return gpa_;
    }

    const double get_id(){
        return id_;
    }

    virtual std::string get_dorm(){
        return "No dorm assigned";
    }

}
```

```
In [ ]: class SCPD_Student : public Student {
            std::string location_;
          public:
            // Note that the constructor for Student
            // is explicitly called with the parameter ("id")
            // To pass a parameter to the parent,
            // we must use the initializer list construction
            // If we don't explicitly call the parent constructor,
            // then the default parent constructor is called
            SCPD_Student(int id, std::string location) : Student(id), location_(location) {};

            const std::string get_location(){
                return location_;
            }

        }
```

```
In [ ]: class Local_Student : public Student {
            std::string dorm_;
          public:
            Local_Student(int id, std::string dorm) : Student(id), dorm_(dorm) {};

            std::string get_dorm(){
                return dorm_;
            }

        }
```

```
In [ ]: SCPD_Student remote_student = SCPD_Student(34, "Minneapolis");
        Local_Student local = Local_Student(25, "Lyman");
```

```
In [ ]: // Parent methods are inherited by children automatically
        local.get_gpa()
```

```
In [ ]: // Methods defined in the child class are also accessible
        remote_student.get_location()
```

```
In [ ]: // What about methods defined in a sibling class?
        local.get_location()
```

3.1 Virtual methods Notice that we used the **virtual** keyword in the Student class when defining our method `get_dorm()`

```
virtual std::string get_dorm(){
    return "No dorm assigned";
}
```

This tells the compiler that the function can be overridden in a derived class, though it doesn't have to be.

```
In [ ]: local.get_dorm()
```

```
In [ ]: remote_student.get_dorm()
```

Let's take a closer look at the syntax that we used to establish the inheritance relationship: `class Local_Student : public Student`

Note that we used the **public** keyword. This meant that all of the `public` and `protected` members and methods of the Student class were also public in the Local_Student class, which is why we were able to call `get_gpa()`.

private inheritance is also an option, though less common.

What is so useful about establishing inheritance relationships this way? **Polymorphism**

1.2.3 4. Polymorphism

The concept that a different version of a method can be called based on the inheritance structure of the classes. This allows us to interact with "Student" objects whose underlying functionality is dictated by their actual type.

```
In [ ]: std::vector<Student*> students_;
```

```
In [ ]: students_.push_back(&local);
```

```
In [ ]: students_.push_back(&remote_student);
```

```
In [ ]: for (int i = 0; i < students_.size(); i++){
            std::cout << "Student " << students_[i]->get_id() << ": "
                << students_[i]->get_dorm() << std::endl;
        }
```

Note that if we hadn't included the `virtual` keyword, then the base class's version of `get_dorm()` would have been called, even for the local student.

The `virtual` keyword signals to the compiler that we don't want **static linkage** for this function (function call determined before the program is executed).

Intead, we want the selection of which version of `get_dorm()` to call to be dictated by the kind of object for which it is called - this is called **dynamic linkage** or late binding.

1.2.4 5. Abstract Classes

Based on the way we defined our Student class so far, we can still instantiate it (Create objects of type "Student")

```
In [ ]: Student base = Student(22);
```

```
In [ ]: base.get_dorm()
```

What if we wanted to prevent people from creating a Student object on its own, and force all students to belong to one of the child classes (either Local_Students or SCPD_Students)? Then, we would want to create an **abstract class** - a class that specifies some of the functionality of its children, but cannot be instantiated.

We will illustrate this by moving on to another example from the Python OOP lecture.

Example

```
import math

class Shape:
    def GetArea(self):
        raise RuntimeError("Not implemented yet")

class Circle(Shape):
    def __init__ (self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def GetArea(self):
        area = math.pi * math.pow(self.radius, 2)
        return area

class Rectangle(Shape):
    def __init__ (self, x0, y0, x1, y1):
        self.x0 = x0
        self.y0 = y0
        self.x1 = x1
        self.y1 = y1

    def GetArea(self):
        xDistance = self.x1 - self.x0
        yDistance = self.y1 - self.y0
        return abs(xDistance * yDistance)
```

Recall - in this example, Shape is an abstract class which cannot be instantiated. Here is the same code, but implemented in C++:

```
In [ ]: class Shape {
          public:
            //Notice, the virtual keyword and "= 0"
            virtual double GetArea() = 0;
        }
```

```
In [ ]: #include <math.h>

        class Circle: public Shape {
            double x_;
            double y_;
            double radius_;

          public:
            Circle(double x,
                   double y,
                   double radius):
                x_(x), y_(y), radius_(radius){};

            double GetArea(){
                return M_PI * radius_ * radius_;
            };

        }
```

```
In [ ]: class Rectangle: public Shape {
            double x0_;
            double y0_;
            double x1_;
            double y1_;

          public:
            Rectangle(double x0,
                      double y0,
                      double x1,
                      double y1):
            x0_(x0), y0_(y0), x1_(x1), y1_(y1){};

            double GetArea(){
                return abs(x0_ - x1_) * abs(y0_ - y1_);
            }

        }
```

```
In [ ]: // First, what happens if we try to instantiate Shape?
        Shape shape = Shape();
```

```
In [ ]: Rectangle rect = Rectangle(0,0,2,5);
```

```
In [ ]: rect.GetArea()
```

```
In [ ]: Circle circ = Circle(0,0,6);
```

```
In [ ]: circ.GetArea()
```

5.1 Pure virtual methods Recap: What did we just observe?

- We declared a function `virtual double GetArea() = 0;` in the Shape class. The =0 syntax told the compiler that this was a **pure virtual** function, meaning that it cannot be executed in the base class.
- Any class with >= 1 pure virtual function is understood to be an **abstract class** in C++, meaning that it cannot be instantiated.

Q: Does the concept of polymorphism still apply even with an abstract class, such as "Shape"? A: Yes. It is still valid to have a pointer of type Shape.

```
In [ ]: std::vector<Shape*> my_vector;

        my_vector.emplace_back(&circ);
        my_vector.emplace_back(&rect);
```

```
In [ ]: double total_area = 0.0;
        for (int i = 0; i < my_vector.size(); i++){
            total_area += my_vector[i]->GetArea();
        }
```

```
In [ ]: total_area
```

1.2.5 Composition

Composition is another type of relationship between objects. Composition is when objects relate in a "has a" relationship.

Here is an example where we create a `Point2D` class to define point-specific methods, and then re-implement our Circle class to **have** a `Point2D` to represent its center.

```
In [ ]: #include <iostream>
```

```
In [ ]: class Point2D
        {
        private:
            double x_;
            double y_;

        public:
            // A default constructor
            Point2D(){};

            Point2D(double x, double y): x_(x), y_(y){};

            // An overloaded output operator
            friend std::ostream& operator<<(std::ostream& out, const Point2D &point)
            {
                out << "(" << point.x_ << ", " << point.y_ << ")";
                return out;
            }

        };
```

```
In [ ]: Point2D p = Point2D(4,5)
```

```
In [ ]: std::cout << p << std::endl;
```

```
In [ ]: class Circle2: public Shape {
            Point2D center_;
            double radius_;
```

```
          public:
            Circle2(double x, double y, double radius):center_(x, y), radius_(radius){};

            // A default constructor
            Circle2(){};

            double GetArea(){
                return M_PI * radius_ * radius_;
            };

            const Point2D* GetLocation(){
                return &center_;
            }

        }
```

```
In [ ]: Circle2 circle = Circle2(4,3,2);
```

```
In [ ]: circle.GetArea()
```

```
In [ ]: std::cout << *(circle.GetLocation()) << std::endl;
```

```
In [ ]: Circle2 circle2 = Circle2();
```

```
In [ ]: // What value will this return?
        std::cout << *(circle2.GetLocation()) << std::endl;
```