Object Oriented Programming Part II: C++

Topics

1. Recap: Encapsulation
2. Recap: Abstraction
3. Inheritance

   3.1 Virtual methods
4. Polymorphism
5. Abstract classes
   - Pure virtual methods
6. Composition

Recap: Encapsulation

- Encapsulation hides the details of an object's state from unauthorized parties.
- The values of attributes and the existence of certain methods can be hidden from other objects or users.
- Achieved in C++ by maintaining **private** attributes: the attribute is only accessible from within members of the same class. We learned on Tuesday about how the default access specifier for classes is "private" (All attributes and methods will be private unless declared otherwise).

```
#include <iostream>
```

```
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;
    };

    bool operator==(const Student_Encapsulation_Example& s2){
        return (this->id_ == s2.id_);
    }
```

```
}
```

```
Student_Encapsulation_Example s = Student_Encapsulation_Example(25);
```

```
Student_Encapsulation_Example s_2 = Student_Encapsulation_Example(24);
```

```
Student_Encapsulation_Example s_3 = Student_Encapsulation_Example(25);
```

```
// what will happen when I run this code?
s.id_
```

```
if (s == s_3){
    std::cout << "Equal" << std::endl;
} else {
    std::cout << "Not Equal" << std::endl;
}
```

Note in the example above that the comparison operator `==` for s is able to access the 'id' attribute of s3, because both are instances of the StudentEncapsulationExample class.

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){};
```

- `: id_(id)` is the initializer list
- Note that in general, before the statement that forms the function body of the constructor begins executing, initialization of all direct bases, virtual bases, and non-static data members is finished. So the initializer list is where you can specify non-default initialization of these values.

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.

```
class Student_Encapsulation_Example2 {
    // By default, these variables are private
    int id_ = 0;
    double gpa_ = 0.0;
    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_;
    }
}
```

```
Student_Encapsulation_Example2 se = Student_Encapsulation_Example2(25);
```

```
se.get_id()
```

```
Student_Encapsulation_Example2 default_se =  Student_Encapsulation_Example2();
```

```
// What will this return?
default_se.get_id()
```

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

```
class Student_Abstraction_Example1 {
    int id_;
    double gpa_;
    // In our first implementation, courses and grades are stored in two
vectors
    // the student's grade for courses_[i] = course_grades_[i]
    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 / 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_;
    }
}
```

```
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_;
    }

}
```

```
Student_Abstraction_Example1 s1 = Student_Abstraction_Example1(3);
s1.add_course("CME 211", 3.4);
s1.add_course("CME 212", 3.2);
s1.print_course_roster();
```

```
Student_Abstraction_Example2 s2 = Student_Abstraction_Example2(3);
s2.add_course("CME 211", 3.4);
s2.add_course("CME 212", 3.2);
s2.print_course_roster();
```

```
s1.get_gpa()
```

```
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

```
#define STUDENT_HPP

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

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

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

#endif /* STUDENT_HPP */
```

file: Student.cpp

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

#include "Student.hpp"

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

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

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

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

Sample *main.cpp*

```
#include <iostream>
#include <vector>
#include "Student.hpp"

int main(){
    Student s = Student(25);
    s.add_course("CME 211", 3.4);
    s.add_course("CME 212", 3.2);
    s.print_course_roster();
    return 0;
}
```

And we compile with: `g++ -std=c++11 -Wall -Wconversion -Wextra -Wpedantic main.cpp Student.cpp -o main`

What if instead of just changing the implementation details of our Student class, we actually wanted to create different kinds of Students, that had differing functionality but still shared the attributes and methods we had previously defined?

A (silly) 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**.

Inheritance

```
#include <iostream>
```

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

  public:
    std::string student_type_ = "Student";

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

    Student(const Student &s):id_(s.id_), gpa_(s.gpa_), courses_(s.courses_),
course_grades_(s.course_grades_) {};

    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";
    }

}
```

```
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) {
        student_type_ = "SPCD Student";
    };

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

}
```

```
class Local_Student : public Student {
    std::string dorm_;
  public:
    Local_Student(int id, std::string dorm) : Student(id), dorm_(dorm) {
        student_type_ = "Local Student";
    };

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

}
```

```
SCPD_Student remote_student = SCPD_Student(34, "Minneapolis");
Local_Student local = Local_Student(25, "Lyman");
```

```
// Parent methods are inherited by children automatically
local.get_gpa()
```

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

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

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.

```
local.get_dorm()
```

```
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 and protected in the Local_Student class, which is why we were able to call `get_gpa()`.

private inheritance is also an option, though less common (`public` and `protected` members of the Base class become `private` members of the Derived class).

Protected access specifier

We have introduced a new access specifier: `protected`. `Protected` means that the attribute or method is only accessible from within members of the same class *or from within members of derived classes*.

Let's rewrite our Student classes above to illustrate the use of the `protected` keyword:

```
class Student_protected {
    int id_;

  protected:
    std::string dorm_ = "No dorm assigned";

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

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

```
class Local_Student_protected : public Student_protected {
  public:
    Local_Student_protected(int id, std::string dorm) : Student_protected(id){
        dorm_ = dorm;
    };

}
```

```
Local_Student_protected loc = Local_Student_protected(23, "Munger");
```

```
loc.get_dorm()
```

```
// What happens if we try to access `loc.dorm_` from outside of the class?
loc.dorm_
```

```
// What happens if we go back and remove the "protected" keyword?
```

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

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.

```
#include <iostream>
```

```
std::vector<Student*> students_;
```

```
students_.push_back(&local);
```

```
students_.push_back(&remote_student);
```

```
for (int i = 0; i < students_.size(); i++){
    std::cout << students_[i]->student_type_ << " " << 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.

Note: We used a vector of pointers to Students in our example above:

`std::vector<Student*> students_;`

Would we still have been able to take advantage of Polymorphism with a vector of Student objects?

```
std::vector<Student> students_2;
```

```
students_2.push_back(local);
```

```
students_2.push_back(remote_student);
```

```
for (int i = 0; i < students_2.size(); i++){
    std::cout << students_2[i].student_type_ << " " << students_2[i].get_id()
<< ": "
        << students_2[i].get_dorm() << std::endl;
}
```

```
students_2[1].get_location()
```

What happened when instead of creating a vector of pointers to Student objects, we created a vector of Student objects?

- We were still able to add the "local" and "remote_student" objects to the vector, but the copy constructor of the "Student" class was called, creating new Student objects and implicitly casting the derived objects to the base class
- That's why the type changed to "Student", and get_dorm() follows the Student class behavior

TL;DR: In order to make use of polymorphism, use pointers to objects of the Base class type

Abstract Classes

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

```
Student base = Student(22);
```

```
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 Object Oriented Programming lecture.

Abstract Class 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++:

```
#include <iostream>
```

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

```
#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_;
    };

}
```

```
class Rectangle: public Shape {
```

```
  protected:
    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_);
    }

}
```

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

```
Rectangle rect = Rectangle(0,0,2,5);
```

```
rect.GetArea()
```

```
Circle circ = Circle(0,0,6);
```

```
circ.GetArea()
```

```
// You can also inherit from an inherited class!
class Square: public Rectangle {
  public:
    Square(double x0,
           double y0,
           double length):Rectangle(x0, y0, x0 + length, y0 + length){};

    double GetArea(){
        // Notice: We are able to access the "protected" member variables of
the Rectangle class
        double side_length = abs(x0_ - x1_);
        return side_length*side_length;
    }
}
```

```
Square sq = Square(0,0,5);
```

```
sq.GetArea()
```

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 any derived class must override that function.
- 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.

```
std::vector<Shape*> my_vector;

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

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

```
total_area
```

Composition

Composition is another type of relationship between objects. Composition occurs 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.

Note: We have already made use of the concept of composition in the `Student` examples, where our `Student_Course` member objects only existed in the context of a particular student, and the `Student_Course` objects didn't "know" about the `Student`.

```
#include <iostream>
```

```
class Point2D
{
private:
    double x_ = 0.0;
    double y_ = 0.0;

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;
    }

};
```

```
Point2D p = Point2D(4,5)
```

```
std::cout << p << std::endl;
```

```
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_;
    };

    Point2D GetLocation(){
        return center_;
    }

}
```

```
Circle2 circle = Circle2(4,3,2);
```

```
circle.GetArea()
```

```
std::cout << circle.GetLocation() << std::endl;
```

```
Circle2 circle2 = Circle2();
```

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