Strathmore University

ICS2101: Object Oriented Programming II[1]

Week 1: Inheritance, Polymorphism & Related concepts

[1]All code for this course will be found in this my git repo: `https://github.com/anyamu/ics2101`

The fundamental data types and their variations are essential, but rather primitive. C and C++ provide tools that allow you to compose more sophisticated data types from the fundamental data types. As you'll see, the most important of these is *struct*, which is the foundation for class in C++.

```
#include <iostream>
#include <string>

struct student {
  std::string name;
  int id;
  double midt, fin, assig;
};

int main() {
  struct student stud;
  // you have to press enter for every value
  std::cout << "What is the student's name & ID?: ";
  std::cin >> stud.name >> stud.id;
  std::cout << "Grades: assignments midterm fin: ";
  std::cin >> stud.assig >> stud.midt >> stud.fin;
  // Print out the results
  std::cout <<"Name: " << stud.name << std::endl
            <<"Student ID: " << stud.id << " Grade: "
            << (stud.midt + stud.fin + stud.assig)
            << std::endl;
}
```

An **object** is a self-contained component that contains properties and methods needed to make a certain type of data useful.
A **class** is a blueprint or template to build a specific type of object and that every object is built from a class.

0.1 Composition, Aggregation & Encapsulation

Composition expresses relationships between objects: e.g. A car *has a* steering wheel. The phrase ''*has a*" implies a relationship where the car

owns the steering wheel or at a minimum uses another object - this phrase is the basis for composition.

```
class SteeringWheel{
};
class Car{
  // composition below
  SteeringWheel s_wheel;
};
```

Aggregation is more subtle but very much like composition: Say for example, someone drives the Car - we can say that the Car ''*has a*" Driver. Certainly, the Driver exists independently of the Car and so if we dispense with the Car we would not dispense with the Driver as well. When designing your classes, always make this distinction between composition & aggregation in that with aggregation, there is that property of ''independence" of the composed object.

```
#include <iostream>
#include <string>
class SteeringWheel{
};
class Driver{
  std::string jina;
public:
  Driver(std::string name = "Dereva")
    : jina(name){}
  std::string getName(){ return jina; }
};
class Car{
  Driver *dere;
  SteeringWheel s_wheel;
  std::string car_type;
public:
  Car(Driver *driver = NULL,
      std::string ct) :
    dere(driver), car_type(ct){}
```

```
20   Car(Driver *driver = NULL) : Car(driver, "Mercedes") {
21   }
22   std::string whosDriving(){
23     return dere->getName()
24       .append(" is driving a ")
25       .append( car_type );
26   }
27 };
28 using namespace std;
29 int main(){
30   Driver *d = new Driver("Ms. Pop Diva");
31   cout << "Our driver is, " << d->getName()
32        << "!" << endl;
33   {
34     // Inside this block, object 'car' exists but is destroyed
35     // once you go out of this block
36     Car car(d);
37     cout << car.whosDriving() << "!!" << endl;
38   }
39   // Has our driver survived the crash!!?
40   cout << "Lone survivor: " << d->getName() << endl;
41   delete d;
42 }
```

Encapsulation / (Information hiding) is about hiding complexity. You don't need to know how an object works in order to use it. For example, you don't need to know how a vehicle engine works in order for you to use the vehicle, and you should be able to use any vehicle whether the engine burns petrol or firewood! There are two main reasons to hide complexity:

- To provide a simplified and understandable way to use the object without the need to understand the complexity inside. The class designer provides an interface: which is a way to interact with the object without needing to know what is happening within the object.
- To manage change. As vehicles have advanced in technology, we now have hybrid engines that also use battery power, but anyone can still drive the same vehicle because the interface has not changed (e.g. there

is still the steering wheel, gas pedal, brakes e.t.c which work similarly as vehicles before.)

At www.learncpp.com, read "Access functions & encapsulation" and create your own example of encapsulation using C++ and/or the UML.

0.2 Inheritance

(Required Reading & Note making)

Bruce Eckel, Thinking in C++, 2nd ed, Volume 1 - Chapter 1 section on inheritance. Please make your own notes too!
Chapter 11 (Inheritance): http://www.learncpp.com/

Inheritance is a way to express a relationship between blueprints (classes). It's a way of saying: I want to build a new object that is similar to one that already exists, and instead of creating the new class from scratch, I want to reference the existing class and simply indicate what's different.A class that is used as the basis for inheritance is called the *superclass* or *base class*, and that inheriting from it is the *subclass* or *derived class*.
Using two concepts of inheritance, *subclassing* (making a new class based on a previous one) and *overriding* (changing how a previous class works), you can organize your objects into a hierarchy. Using inheritance to make this hierarchy often creates easier to understand code, but most importantly it allows you to reuse and organize code more effectively.
The relationship between a subclass and a superclass can be understood in terms of the **is a** relationship. A subclass **is a** more specific instance of a superclass. e.g. An orange **is a** citrus fruit; A cake **is a** kind of pastry. It is logical to see that you can have multiple subclasses of a superclass.

0.2.1 Order of construction for inheritance chains

In chapter 11 of www.learncpp.com , try out the code in that section and upload it to your github.com account. Take note (i.e. make sure you understand) of what's happening in the inheritance hierarchy.

0.2.2 Access Control

public members can be accessed by anybody. **private** members can only be accessed by member functions of the same class. Note that this means derived classes cannot access private members! **protected** access specifier restricts access to member functions of the same class, or those of derived classes.
When a derived class inherits from a base class, the access specifiers may change depending on the method of inheritance. There are three different ways for classes to inherit from other classes: public, private, and protected. There are three ways that members can be accessed:

- A class can always access its own members regardless of access specifier.
- The public accesses the members of a class based on the access specifiers of that class.
- A derived class accesses inherited members based on the access specifiers of its immediate parent. A derived class can always access its own members regardless of access specifier.

We therefore have 9 combinations: 3 member access specifiers (public, private, and protected), and 3 inheritance types (public, private, and protected).

Member access	Class inheritance type		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Table 1: Access specifiers

Quick Quiz & Assignment

See the code below and follow the instructions therein.

```
1 class Base{
2   int priv;
```

```
protected:
  int prot;
public:
  int publ;
};
// What can derived1 access from Base?
// What are the access specifiers of members inherited from Base?
class derived1 : Base {
  int priv1;
protected:
  int prot1;
public:
  int publ1;
};
// What can derived2 access from derived1?
// What are the access specifiers of members inherited  from derived1?
// What about the inherited members of Base in derived1?
class derived2 : public derived1 {
  int priv2;
protected:
  int prot2;
public:
  int publ2;
};
// What can derived3 access from derived2?
// What are the access specifiers of members inherited from derived2?
// What about the inherited members of Base & derived1 in derived2?
class derived3 : protected derived2 {
  int priv3;
protected:
  int prot3;
public:
  int publ3;
};
// Assignment: write out all the remaining inheritance
//             schemes that are possible together with all possible
//             member access specifiers. Upload your code to your
//             github account.
```

0.2.3 Adding, changing and hiding members in a derived class

It's possible to inherit the base class functionality and then add new functionality, modify existing functionality, or hide functionality you dont want (in C++ you cannot remove functionality, but you can hide it).

Adding new functionality

Sometimes you cannot add functionality in some pre-existing code: *why is this sometimes the case?* The solution is to derive your own class and add the functionality you want to the derived class.

```
#include <iostream>
#include <string>

class Shape{
protected:
  std::string name;
public:
  Shape(std::string name = "Amorphous Base Shape"): name(name){}
  std::string getName(){ return name; }
};

class Triangle : public Shape{
public:
  Triangle(std::string name = "Nice Triangle!") : Shape(name){}
};

using namespace std;
int main(){
  Triangle tria;
  cout << tria.getName() << endl;
  return 0;
}
```

```
#include <iostream>
#include <string>
```

```
3  using namespace std;
4
5  class Rectangle{
6    std::string shape;
7    int height, width;
8  public:
9    Rectangle(std::string shape="Rectangle"): shape(shape),
10                                              height(0), width(0){}
11   Rectangle(int h, int w): height(h), width(w){}
12   Rectangle(std::string shape, int h, int w): height(h), width(w){}
13
14   std::string getName(){ return shape; }
15   int area(){return height * width; };
16 };
17
18 class Triangle : public Rectangle{
19 public:
20   Triangle(int h, int b) : Rectangle("Triangle", h, b){}
21   Triangle() : Rectangle("Triangle"){}
22   int area();
23 };
24 int Triangle::area(){
25   // return (int)(0.5 * height * width);
26   return (int)(0.5 * Rectangle::area());
27 }
28 class Square : public Rectangle{
29 public:
30   Square(int side);
31 };
32 Square::Square(int side): Rectangle("Square", side, side){}
33
34 int main(int argc, char **argv){
35   Square sq(10);
36   cout << "Square area: " << sq.area() << endl;
37   Triangle t1(3, 6), t2;
38   cout << "Triangle 1 area: " << t1.area()
39     //!FIXME: t1.getName()
40        << " Name: " << t1.getName() << endl;
```

```
41   cout << "Triangle 2 area: " << t2.area()
42        << " Name: " << t2.getName() << endl;
43   return 0;
44 }
```

0.3 Polymorphism & Virtual Functions

(Coming Soon!!)

Bruce Eckel, Thinking in C++, 2nd ed, Volume 1 - Chapter 1 section on "Interchangeable objects with polymorphism". Please make your own notes too! See: `http://www.learncpp.com/cpp-tutorial/122-virtual-functions/`

0.4 Exercises

Exercise 1. (Modularise! Modularise! Modularise!) (20 points.) For the code in this document, separate the interface & implementation by putting the class definitions and member functions in separate header files. Upload your code to your github.com git repository. (make sure it compiles and that it works ok).

Exercise 2. (UML) (20 points.) Draw up the relevant UML diagrams for the OOP concepts and then do the same for the code in this document. Insert the diagrams in the assignment Google doc that has been shared with you. (please label your work in a neat manner that shows which UML diagram belongs to which OOP concept and/or piece of code).

Exercise 3. (Ms. Saucy Pop Diva) (20 points.) Fix the bug in the 'car steering wheel with driver' example and upload to your github.com git account. Your output should resemble

```
Our driver is, Ms. Pop Diva!
Ms. Pop Diva is driving a Mercedes!!
Lone survivor: Ms. Pop Diva
```

Exercise 4. (Finito!) (40 points.) Look for a '!FIXME' somewhere in the code in this document and actually fix it! Upload all code into your github.com account.