CSC 305: Object-Oriented Programming in C++

Academic year 2018-2019

Dr. M. NYAMSI and Mme F. TIAKO

Introduction to OOP

Tell me and I forget.

Show me and I remember.

Let me do and I understand.

Chinese Proverb

Introduction to OOP

- What is OOP: Principles and problem solving approach
- Why OOP: Limitations of imperative and other programming paradigms, advantages of OOP
- Case study:
  - Simple automated teller machine (ATM);
  - requirements specifications.
- Classes, objects, member functions and data members (identification and simple programming).

Introduction to OOP

- Practical: simple application of what we learned
- Assignment (personal work): Application to the case study
  - ATM Classes: identification and implementation
  - ATM Objects: identification and implementation
- Classes: Scope, accessing class members, separating interface from implementation, constructors and destructors
- Assignment: Application to case study

Introduction to OOP: what is OOP?

- Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.
- Object-oriented programming (OOP) is a **programming model** organized around **objects** rather than "actions" and **data** rather than logic.
- The programming challenge was seen as how to write the logic, not how to define the data.
- Object-oriented programming takes the view that **what we really care about are the objects we want to manipulate** rather than the logic required to manipulate them.

Introduction to OOP: what is OOP?

- The first step in OOP is to identify all the objects the programmer wants to manipulate and how they relate to each other, an exercise often known as **data modeling.**
- Once an object has been identified, it is generalized as a class of objects
  - which defines the kind of data it contains and
  - any logic sequences that can manipulate it.
- Each distinct logic sequence is known as a **method.**
- Objects communicate with well-defined interfaces called **messages**.

Introduction to OOP: Principles

- **SOLID:**
  - **Single responsibility:** a **class** should have only a single responsibility
  - **Open-closed:** software entities … should be **open** for extension, but **closed** for modification
  - **Liskov substitution:** objects in a program should be **replaceable** with instances of their **subtypes** without altering the correctness of that program

Introduction to OOP: Principles

- **SOLID:**
  - **Interface segregation:** **many** client-specific interfaces are better than **one** general-purpose interface
  - **Dependency inversion:** one should Depend upon Abstractions. Do not depend upon concretions

Introduction to OOP: Principles

In other words: principles are

- **Encapsulation:**
  - We will learn to **hide unnecessary details** in our classes and provide a clear and simple interface for working with them.
- **Inheritance**
  - We will explain how **class hierarchies** improve code readability and enable the reuse of functionality.

Introduction to OOP: Principles

In other words: principles are

- **Abstraction**
  - We will learn how to **work through abstractions**: to deal with objects considering their important characteristics and ignore all other details.
- **Polymorphism**
  - We will explain how to work in the same manner with different objects, which define a specific implementation of some abstract behavior.

Introduction to OOP: Principles

- **Abstraction:** it means to focus on the essential features of an element (object in OOP), ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used.
- An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

Introduction to OOP: Principles

- **Abstraction: Example**

When a class Student is designed, the attributes

- enrolment_number (MatNumber), name, course and address are included

Characteristics like pulse_rate and size_of_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

Introduction to OOP: Principles

- **Encapsulation:**
  - It is the process of binding both attributes and methods together within a class.
  - Through encapsulation, the internal details of a class can be hidden from outside.
  - The class has methods that provide user interfaces by which the services provided by the class may be used.

Introduction to OOP: Principles

- **Modularity:**
  - It is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem.
  - Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
  - Modularity is intrinsically linked with encapsulation.
  - Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter-module interaction or coupling is low

Introduction to OOP: Principles

- **Hierarchy:**
  - It is the ranking or ordering of abstraction
  - Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached.
  - It uses the principle of “divide and conquer”.
  - Hierarchy allows code reusability

Introduction to OOP: Why OOP

- Keep large software projects manageable by human programmers through:
  - Modularization: Decompose problem into smaller subproblems that can be solved separately
  - Abstraction (understandability): Individual modules are understandable by human readers
  - Encapsulation (information hiding): Hide complexity from the user of a software; protect low-level functionality

Introduction to OOP: Why OOP

- Keep large software projects manageable by human programmers through:
  - Composability (structured design): Interfaces allow to freely combine modules to produce new systems.
  - Hierarchy: Incremental development from small and simple to more complex modules
  - Continuity: Changes and maintenance in only a few modules does not affect the architecture.

Introduction to OOP: presentation

- ***Class case study:***
  - **Integrated Grade Book**
    - The Grade Book case study uses classes and objects to incrementally build a Grade Book class that represents an instructor's grade book and performs various calculations based on a set of student grades, such as calculating the average grade, finding the maximum and minimum, and printing a bar chart.
- Assignment case study: attached document
  - Simple automated teller machine (ATM);
  - requirements specifications.
  - Identifications of objects and attributes

Introduction to OOP: Classes, Objects …

- Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class.
- Someone has to build a car from its engineering drawings before you can actually drive the car,
  - you **must** create an **object of a class** before you can get a program to perform the tasks the class describes.
- As many cars can be built from the same engineering drawing, many objects can be built from the same class.

Introduction to OOP: Class and instance …

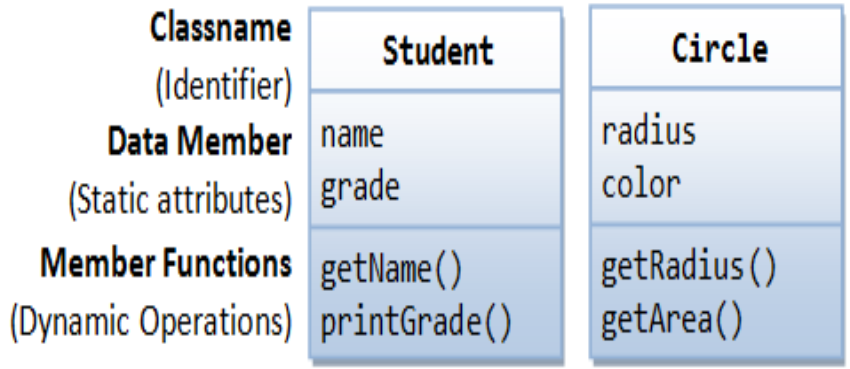
- **Class:**
  - A *class is a definition of objects of the same kind*.
  - In other words, a *class* is a blueprint, template, or prototype that defines and describes the *static attributes* and *dynamic behaviors* common to all objects of the same kind.
- A class can be visualized as a three-compartment box:
  - ***Classname*** (or identifier): identifies the class.
  - ***Data Members*** or ***Variables*** (or *attributes*, *states*, *fields*): contains the *static attributes* of the class.
  - ***Member Functions*** (or *methods*, *behaviors*, *operations*): contains the *dynamic operations* of the class.

Introduction to OOP: Classes, Objects …

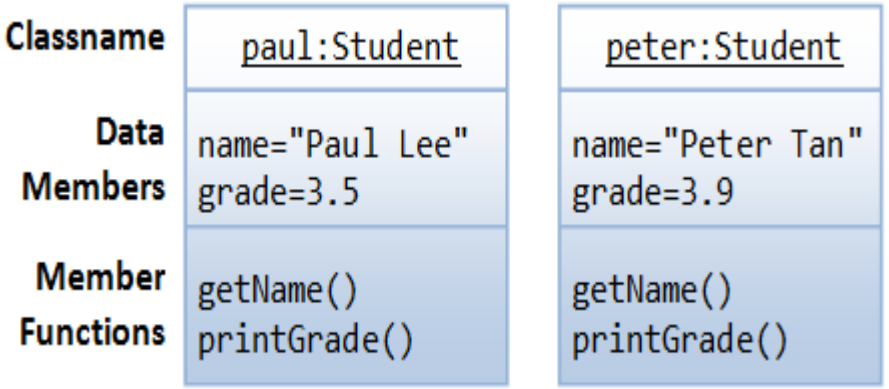
- **Instance:**
  - An *instance* is *a realization of a particular item of a class*.
  - In other words, an instance is an *instantiation* of a class.
  - All the instances of a class have similar properties, as described in the class definition.
- For example
  - You can define a class called "Student"
  - You can create three instances of the class "Student" for "Peter", "Paul" and "Pauline".
- The **data members** and **member functions** are collectively called **class members**.

Introduction to OOP: Classes, Objects …

- Example of classes



- Example of instance
  - Object



Introduction to OOP: Classes, Objects …

- In C++, the definition is:

```
class Circle {          // classname
private:
   double radius;
   string color;
public:
   double getRadius();
   double getArea();
}
```

This class has :

2 data members (variables)

2 member functions (methods)

Variables are private and Methods are public

Introduction to OOP: Classes, Objects …

- In C++, the definition is:
  - The example above is a class with data and function members
  - Member functions are without parameters
  - If you define class data member without specifying Private, data members are automatically private.
  - private members of a class are accessible only from within other members of the same class (or from their *"friends“ to see later*). Declaring data members with access specifier private is known as **data hiding**
  - protected members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes (to see later).
  - Finally, public members are accessible from anywhere where the object is visible.

Introduction to OOP: Classes, Objects …

- In C++, the definition is:
  - // Instantiation:
    - Circle C1(0.5,Blue);
    - Circle C(1.2);
    - Circle C2;

are 3 instances of the class Circle: They are objects of class Circle

- C1 has radius of 0.5 and color is blue (called **Constructor**)
- C has radius 1.2 and the default color
- C2 has the default radius and the default color

Introduction to OOP: Classes, Objects …

- In C++, the definition is:
  - Constructors:
    - Each class you declare can provide a constructor that can be used to initialize an object of the class when the object is created.
    - A constructor is a special member function that **must be** defined with the same name as the class, so that the compiler can distinguish it from the class's other member functions.
    - An important difference between constructors and other functions is that constructors cannot return values, so they cannot specify a return type (not even void). Normally, constructors are declared public.
    - If a class does not explicitly include a constructor, the compiler provides a default constructor (a constructor with no parameters).

Introduction to OOP: Classes, Objects …

- In C++, the definition is:
  - Constructors: Example

```
Circle(double r = 1.0, string c = "red")
 {
    radius = r;
    color = c;
 }
```

Introduction to OOP: Classes, Objects …

- Data members, Set and Get functions:
  - A class normally consists of one or more member functions that manipulate the attributes that belong to a particular object of the class.
  - Attributes are represented as variables in a class definition.
  - Such variables are called data members and are declared inside a class definition but outside the bodies of the class's member-function definitions.
  - Each object of a class maintains its own copy of its attributes in memory.
  - Example: Circle class that contains a radius data member to represent a particular Class object's circle. Same thing with color data member

Introduction to OOP: Classes, Objects …

- Data members, Set and Get functions:
  - Example: Circle class that contains a radius data member to represent a particular Class object's circle. Same thing with color data member.

```
class Circle {          // classname
   double radius;       // Data members (variables)
   string color;
    ……..
}; // end of class definition
```

  - You can use set and get functions to set and have the values of these data members

Introduction to OOP: Classes, Objects …

- Set and Get functions:
  - A class's private data members can be manipulated only by member functions of that class (and by "friends" of the class).
  - So a client of an object (any class or function that calls the object's member functions from outside the object) calls the class's public member functions to request the class's services for particular objects of the class.
  - Classes often provide public member functions to allow clients of the class to
    - set (assign values to) or
    - get (obtain the values of)

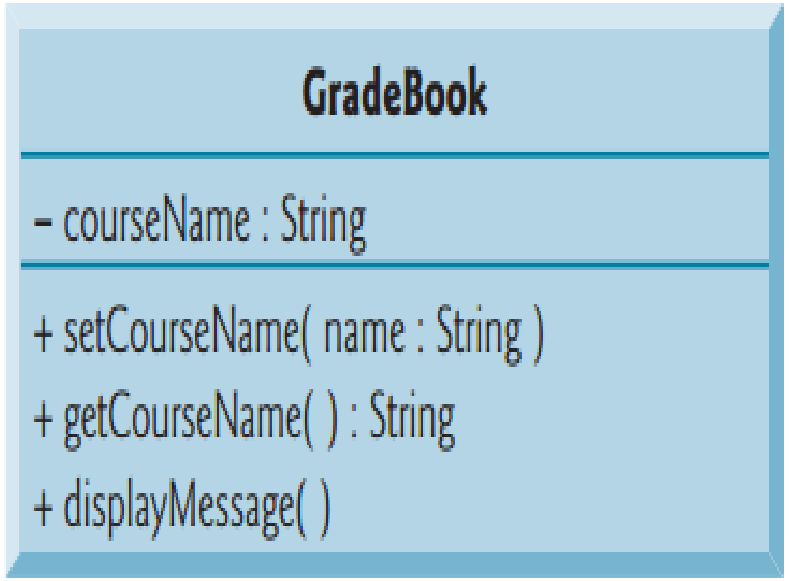
  private data members
  - These member function names need not begin with set or get, but this **naming convention** is common.

Introduction to OOP: Classes, Objects …

- Set and Get functions:
  - Example: getColor, getRadius, getDiameter are member function's to obtain the color, the radius and the diameter of the circle.
  - Exercise: write a setParameter to allow end-user to enter values for data parameters radius and color.
  - Set functions are also sometimes called **mutators** (because they mutate, or change, values)
  - Get functions are also sometimes called **accessors** (because they access values).
  - Providing **public** set and get functions allows clients of a class to access the **hidden data (private)**, but only indirectly.

Introduction to OOP: Classes, Objects …

- Set and Get functions:
  - Consider the class GradeBook given below
  - Define and create corresponding data and function members. Test it with the message ”CSC 305: Object oriented Programming”.



Introduction to OOP: Classes, Objects …

- Accessing or calling a member function in the main
  - Use the dot (.) operator

  Syntax: NameOfObject.nameOfMemeberFunction(Arguments if necessary)

  - Example: c1.getRadius()
- We can also define a class and class function members are defined outside the class definition.
- We then use the (::) colon like for global definition of variables.

Introduction to OOP: Classes, Objects …

- Example:

```
// Class Rectangle
#include <iostream>
using namespace std;
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};
void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}
```

Introduction to OOP: Classes, Objects …

- Header files and main files
  - Rather than to create and use a class in the same file, it is recommended to use two different files.
  - For the re-use of classes (object oriented programming)
  - User defined classes are included in the interface and test files with ”ClassName” not <ClassName> like standard librairy’s.
    - Example: #include "Circle.h"

Introduction to OOP: Classes, Objects …

- Class definitions, when packaged properly, can be reused by programmers worldwide.
- It's customary to define a class in a header file that has a .h filename extension.
- If the class's implementation changes, the class's clients should not be required to change.
- Interfaces define and standardize the ways in which things such as people and systems interact.
- A class's public interface describes the public member functions that are made available to the class's clients.
- The interface describes what services clients can use and how to request those services, but does not specify how the class carries out the services.

Introduction to OOP: Classes, Objects …

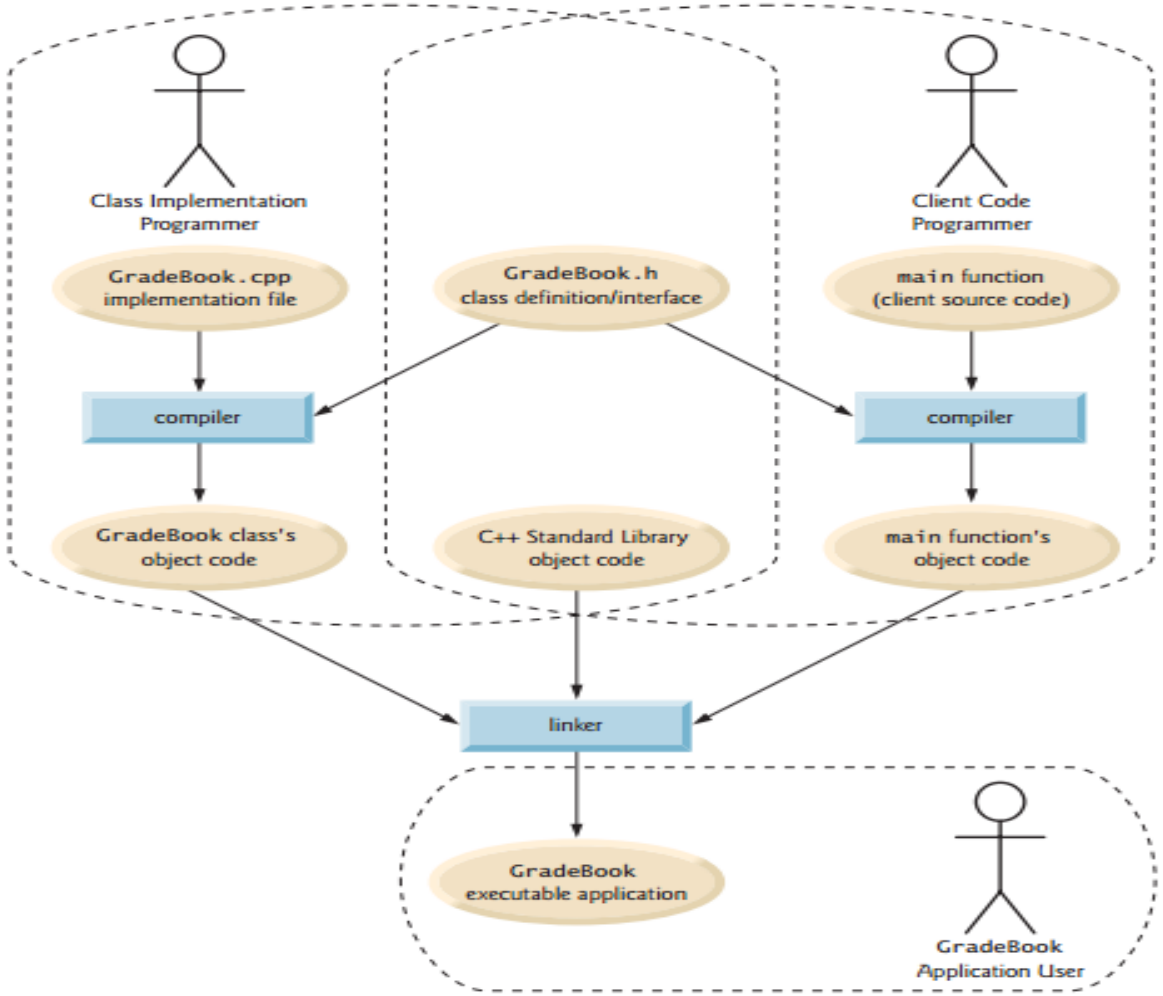
- Consider the Circle class below

Circle
-radius:double = 1.0 -color:string = "red"
+Circle(radius:double,color:string) +getRadius():double +setRadius(radius:double):void +getColor():string +setColor(color:string):void +getArea():double

- We will write
  - Circle.h: defines the public interface of the Circle class.
  - Circle.cpp: provides the implementation of the Circle class.
  - TestCircle.cpp: A test driver program for the Circle class.
- See in the practical sheet

Introduction to OOP: Classes, Objects …

- Compilation and linking process that produces an executable application



Introduction to OOP: Classes, Objects …

- Destructor:
  - Destructors fulfill the opposite functionality of *constructors*
  - They are responsible for the necessary cleanup needed by a class when its lifetime ends.
  - A destructor, similar to constructor, is a special function that has the same name as the classname, with a prefix "**~**" (Tilda).
  - A destructor is a member function very similar to a *default constructor*: it takes no arguments and returns nothing.
  - **Example**: ~Circle().
  - The classes we have defined in previous sections did not allocate any resource and thus did not really require any clean up. Destructor is used with dynamic memory allocation that we will cover later.

Introduction to OOP: Classes, Objects …

- Place to tutorial and practical
  - Do it in class
  - Questions
  - Assignments