# CMPT 115: Principles of Computer Science

## Object Oriented Programming in C++

Department of Computer Science
University of Saskatchewan

March 28 - April 1, 2016

# Laboratory 10 Overview

**Today's Goal**

- To write simple operations for the Point class, and test them.

Exercises (to hand in with Assignment 10)

- `Point.h`, `Point.cc`, `testPoint.cc`

# Part I

# Pre-Lab 10 Reading

# Object Oriented Programming in C++

- Until recently, we've been programming using C++ and its *procedural* features, such as `char`, `int`, and `float`, as well as the creation of new types using `structs`.
- We'll spend some time with the *object-oriented* features of C++ in this lab.
- The `class` construct is what gives C++ its object-oriented features.
- Classes facilitate modularity and information hiding (see lectures).
- A `class` defines the characteristics of objects in terms of attributes (also called "fields") and methods (also called messages, methods, or operations).
- We will see that the class concept helps us build ADTs. The two concepts, classes and ADTs, are useful together, but they are not the same concept!

# Large Programs

In `object-oriented C++`, we divide up large programs into classes. Each class can be broken into two parts:

1. header files (or .h files),
2. code files (or .cc) files.

For very large classes, an implementation can be broken into several smaller .cc files.

# Large Program Organization

## Usually, we put the following in header files:

- Defined constants (`const int` or `#define`).
- Function prototypes with function headers.
- Only those `#include` statements which are necessary for the prototypes.
- `structs` and `typedefs`.

## Usually, we put the following in .cc files:

- Function implementations.
- Any `#include` statements that are used in the functions not already declared in the header file.

# Classes

- Earlier in the course, we put each ADT in its own set of files, such as `List.cc` and `List.h`.
- This is typically what is done with classes as well.
- If we define a `Person` class, we will have files `Person.cc` and `Person.h`.
- We put the class definition of `Person` in the `.h` file, and we put its method implementations in the `.cc` file.

# Attributes and methods

## Class Data Attributes

- The data Attributes can be of any legal data type (`int`, `float`, `string`, etc.), or even any class type, etc.
- Data Attributes can also be references to any other type.

## Class Methods

- These are functions which are considered part of the object and are declared in the class definition.
- Methods are ordinary functions defined with parameters and a return type.
- They can directly access the attributes of their class and manipulate them.

# Class Syntax

- The syntax of a class declaration is:

```
class ClassName {
  // attributes and methods
};
```

- Typically we list the data structures first, then the method prototypes.

# The Player class

```
class Player {
  private:
    int health;    //these are the attributes
    int strength;
    int agility;
  public:
    void move();    //these are the method prototypes
    void attackEnemy();
    void getTreasure();
};
```

- Every `Player` object which gets created has three integers called `health`, `strength` and `agility` for its attributes.
- The methods we can perform on a `Player` object are `move`, `attackEnemy` and `getTreasure`.

# Access Levels

## Public vs Private

**public**: If an attribute or method is public, it can be accessed from anywhere in your code.

**private**: The members are accessible only from within the class. For data members, this means that the data can be accessed or modified only by code in a method for the class. For methods, this means that the function can be called only while in another method of the class. This is the default, if no specifiers are used.

Using public attributes is discouraged. If you want to access a data attribute from outside the class, write a method to return that value.

# Access Example cont'd

- The `public` or `private` specifier affects all attributes and methods until the next occurrence.
- In the example above, the `private` keyword begins a private section encompassing all three attributes.
- The `public` keyword specifies that all three methods should be public.
- So, only an `Player` method can access the object's attributes, but any code in the program is free to call the methods in the `Player` class.

# Creating objects on the stack and on the heap

Given the Player class definition:

```
class Player {
  private:
    int health;    // attributes
    int strength;
    int agility;
  public:
    void move();    // method prototypes
    void attackEnemy();
    void getTreasure();
};
```

If we wanted to use it, we could have the following main function.

```
int main(){
  Player p1;
  Player *p2 = new Player;

  p1.move();
  p1.getTreasure();
  p2->attackEnemy();
  p2->move();
  p1.move();
}
```

When we have a reference to an object, we use the "arrow" notation. When a local variable contains object itself, we use the "dot" notation (this only happens when the object is on the STACK).

# Point Example

- Here is the definition of the class Point.
- We only put the attributes and method *prototypes* here.
- You can find this in a file called `Point.h`.

```
class Point { // sample class
  private:
    float x; // stores the x coordinate
    float y; // stores the y coordinate

  public:
    void setX(float newX);
    void setY(float newY);
    float getX();
    float getY();

};
```

# Class Syntax

- Then we *implement* each method inside a file called `Point.cc`.
- A method declaration has the format:

```
<return type> ClassName::FunctionName(<type> <name> , <type> <name>
, ...);
```

# Point Example

We need the classname before the method name.

```
void Point::setX(float newX){
    x = newX;
}

void Point::setY(float newY){
    y = newY;
}
```

- This indicates that you are implementing the `setX` method which is in the `Point` class.
- The `x` and `y` are the attributes (instance variables) of the object.
- Notice `x` and `y` are neither local variables nor parameters.

# Point Example

..continued.

```
float Point::getX(){
    return x;
}

float Point::getY(){
    return y;
}
```

This is given to you in `Point.cc` on Moodle.

# Constructors

- A constructor is a method that gets called immediately when an object is allocated (on the stack or the heap).
- It is the constructor's job to initialize the object's attributes to sensible initial values.
- A constructor may have parameters that can inform the initial values.
- A constructor has the same name as the class.
- You can have more than one constructor.
- A constructor has no return type and no return statement.
- C++ gives every class a default (implicit) constructor that takes no arguments, and does nothing.

# Destructors

- A destructor is a method that gets called immediately when an object is de-allocated.
- It is the destructor's job tidy up. It may need to deallocate memory on the heap, or close a file.
- A destructor may not have parameters.
- A destructor has the same name as the class, preceded with a "$\sim$".
- You can have only one constructor.
- A constructor has no return type and no return statement.
- C++ gives every class a default (implicit) destructor that does nothing.

# Point Example

- Let's add in a constructor and destructor to our Point class

```
class Point { // sample class
  private:
    float x; // stores the x coordinate
    float y; // stores the y coordinate

  public:
    Point(); //the constructor
    void setX(float newX);
    void setY(float newY);
    float getX();
    float getY();
    ~Point(); //the destructor
};
```

# Simple implementations of constructors and destructors

- Now let's add the implementation.

```
Point::Point(){
    x = 0;
    y = 0;
}
```

- Whenever a Point object is created, this constructor is called. Thus, its `x` and `y` coordinate are initialized to be 0.

```
Point::~Point(){
    //do nothing
}
```

- The destructor is called whenever a Point object is deallocated. This one still does nothing, because the attributes are simple variables.

# Declaring and allocating objects

- To create a new object of type Point on the system stack:

```
Point p1;
```

- To create a new object of type Point on the heap:

```
Point *p2 = new Point;
```

- In both cases, the constructor is called.

# De-allocating objects

- An object allocated on the stack is deallocated automatically, like other variables on the stack.
- The destructor method is called implicitly.

```
void example() {
  Point p1;  // automatic allocation
}            // automatic deallocation when function returns
```

- To deallocate an object on the heap:

```
Point *p2 = new Point;
delete p2;
```

- delete automatically calls the destructor.

# Part II

## Lab activities and exercises

# Exercises

- The files `Point.h`, `Point.cc`,`testPoint.cc` can be found on Moodle.
- ACTIVITY: Download them and compile them using

```
g++ -Wall -pedantic Point.cc testPoint.cc -o testPointApp
```

Test it out!

# Exercises: allocating and deallocating

ACTIVITY: add code in `testPoint.cc` as follows

1. Declare and allocate a new point on the heap.
2. Ask the user for two floating point numbers from the console
3. Sets the x and y coordinate of the point to the input,
4. Print them out to the screen.
5. Deallocates the object on the heap.

# Exercises: access

ACTIVITY: In `testPoint.cc`

1. Try to access a Point object's attributes directly, e.g.,

```
p2->x = 55.5;
cout << "The x value is: " << p2->x << endl;
```

2. What does the compiler tell you?
3. Temporarily move the attribute `x` below the `public:` specifier in Point.h. Try compiling again.
4. Now move the attribute `x` back to where it was. It's a good place for attributes.

# Exercises: adding a new method

ACTIVITY: add a new method called displacement()

1. It will take no parameters
2. It will calculate and return the distance from the Point's $(x, y)$ to the origin (0,0). Hint: It doesn't matter how you calculate distance here, but you can use the so-called Euclidean distance in two dimensions.
3. Add the method header to `Point.h`.
4. Add the method implementation to `Point.cc`.
5. Test the new method in `testPoint.cc`.

# Constructors with parameters

- Because a constructor is just a method, it can have parameters as well!
- ACTIVITY: Let's change the constructor for the Point class so that it has two parameters, `xCoord` and `yCoord`, and it set the x and y the attributes of these values.
- Then you could make a new Point object as follows:

```
Point p(10,15);
Point *p2 = new Point(5,7);
```

- On the heap, it looks like a call to the constructor.
- On the stack, it looks like `p` is a function, but the arguments are sent to the constructor anyway.

# Constructors with parameters

ACTIVITY:

- Change the constructor in the example above so that it takes in as parameter two integers, `xCoord` and `yCoord` and sets the x and y attributes respectively.
- Then, modify the main function that you created above so that it:
  1. asks the user for two integers,
  2. creates a new Point using the new constructor that initializes the x and y coordinate of the point,
  3. then prints them out to the screen.
  4. Test the `displacement` method on your point, as in the previous exercise.