CS 162
Intro to Computer Science II
Lecture 12
Accessors vs. Mutators
“this” keyword, const
Constructors
2/9/24
Odds and Ends

• Assignment 2 due Sunday midnight via TEACH

• Assignment 1 demo due today
Today’s Topics:

• Accessor vs. Mutator functions
• this keyword (pointer)
• Separate class files
Basic Example

• Suppose that we create a Point class
  • It contains an X value and a Y value
  • We can create member functions to move the point, display the value, or perform other manipulations

• Demo...
C++ Access Specifiers

• C++ includes the concept of access specifiers (modifiers)
• For now, we will introduce two:
  • public: these variables and functions are accessible and modifiable to any part of the program
  • private: can only be accessed or modified by code within the same class

• Why would we want to make something private?
Introducing Encapsulation

• Hide the details of your class from others
  • Make your class easier to maintain
  • Helps avoid broken code

• Consider the Point class
  • What if we change int x; → int x_position;
  • That’s a problem for anyone who was using our Point class
Example of Broken code

class Point {
public:
    int x_position;
    int y_position;

    void move_left(int);
};

int main () {
    Point p1, p2;
    p1.x = 8;
    p1.y = 4;
}

The variable names no longer match
How to avoid this problem?

• Make x and y **private**!
  • So they cannot be modified outside of our class

```cpp
class Point {
private:
    int x;
    int y;
public:
    void move_left(int);
};
```

• Demo...

• Wait a second... Once x and y are private, how to we access or modify the state of an object now?
How to Implement Encapsulation?

• Introduce the concept of **accessor** functions
  • Functions that are used to get (or access) values of an object from outside (or inside) the class
  • E.g. implement `get_x()` and `get_y()`
  • Now there’s a layer of abstraction between the implementation (your code) and the interface (how people interact with your code)
  • Details such as internal variable names no longer matter

• **mutator** functions are used to set (or mutate) values of an object from outside (or inside) the class
  • E.g. `set_x()` and `set_y()`
Accessor and Mutator Functions

• Use a consistent naming scheme
  • `get_grade()`, `get_location()`, `get_name()`
  • `set_grade()`, `set_location()`, `set_name()`

• **Accessors** are commonly known as “getters”
• **Mutators** are commonly known as “setters”

• Demo...
Why are accessors and mutators critical?

• In combination with access specifiers, accessors and mutators allow us to control access

• Especially useful when you want to have “read-only” member variables
  • Users can retrieve the variable using a public “getter” function
  • They cannot modify a private value unless you provide a “setter”
How secure are access specifiers?

• This is not meant to prevent people from looking at your source code
• A programmer could still open your .cpp file and look at the names of “private” variables
• The concept of public and private members is enforced by the compiler
• You will receive a compile-time error if you try to access unauthorized variables or functions
Classes vs. Structs

• Structs
  • Convention: No functionality
  • Default public

• Classes
  • Functionality
  • Default private
  • Convention:
    • member variables: private
    • member functions: public
Vocab

• **Struct**: a type definition without any member functions; collection of data items of diverse types

• **Class**: a type definition with both member variables and member functions

• **Object**: instance of the class

• **Member Variable**: variable that belongs to a particular struct/class

• **Member Function**: function that belongs to a particular class

• **Encapsulation**: the details of implementation of a class are hidden from the programmer who uses the class
• Abstraction vs. Encapsulation
  • Abstraction: hide unwanted details while giving out most essential details
    • i.e. 10,000 feet view
  • Encapsulation: hide the code and data into a single unit
  • In short, abstraction hides details at the design level, while encapsulation hides details at the implementation level
• Classes have member variables and functionality
• Contents are private by default
  • Traditionally member variables are private with member functions being public
  • Use accessors and mutators to work with private member variables
    • get_grade(), get_location(), get_name()
    • set_grade(), set_location(), set_name()
• Classes are typically written with their own header (.h) and implementation (.cpp) files
**this** Keyword

• Can be used inside any class functions as a **pointer** to the object with which the function was called
  • **“this” always points to the object being operated on**

• Using **this** can be helpful
  • Make sure we’re referring to the data members of a class, not to other variables that might be in scope.
  • E.g. when a function parameter has the same name as one of its data members
    ```cpp
    void Point::set_x(int x) {
      this->x = x;
    }
    ```

• Demo...
Const

• To prevent changes to an object being passed, put `const` the parameter listing
  • E.g. `bool is_greater (const Point& a, const Point& b);`
• If a function isn’t supposed to change anything, put a `const` at the end
  • e.g. `void print() const;`
  • `void Point::print() const { /* definition */}`
  • Will cause an error if the code in print changes anything
• If using `const` member variable, it has to be initialized in constructor(s) using initialization list
  • E.g. `Point::Point():z(5){} //where z is defined as a const int`
• Demo...
Today’s Topics:

• Constructors
  • Default vs. non-default
Implementing a Class

• Let’s use what we’ve learned so far to create a Course class
  • Create header and implementation files
  • Basic properties include:
    • course name
    • roster
    • current enrollment
    • instructor

• Demo...
Implementing a class

• Now our Course class ...
  • Has a name
  • Contains roster information with student names
  • Tracks number of enrolled students

• New question... how do we initialize the member variables?
  • Use mutators
  • Umm... calling each individual mutator function is cumbersome
  • Fortunately, we have a better way!
Introducing Constructor

• Constructor – a specially defined function
• Automatically called when the object is created
• Sets up (initializes) the object with appropriate values
  • Member variable values
  • Allocating memory for member variables
  • *Opening a file to read from or write to
• If a constructor is not provided by the programmer, one will be automatically generated (implicitly) but will not initialize any values
More details on Constructors

• **Must** have the same name as the class
• Not allowed to return anything
• May have parameters
  • If no parameters provided, referred to as default constructor
  • If parameters are provided, referred to as non-default constructor (a.k.a. parameterized constructor).
• It can be defined in a couple ways:
  • Option 1: Use assignment statements
    ```cpp
    Point::Point (){ 
      this->x = -1;
      this->y = -1;
    }
    ```
  • Option 2: Use initialization list
    ```cpp
    Point::Point() : x(-1), y(-1) {} 
    Point::Point(int a, int b) : x(a), y(b) {} 
    ```
More details on Constructors

• Each class may have **at most one** default constructor, and **any number** of non-default ones

• If you define any non-default constructors for a class, you **must** define a default one

• If constructors are explicitly defined for a class, the compiler will not generate one for you
  • Typical compile time error: a class has non-default constructors, but no default one. Create objects using default constructor → NoNo!!!

• Can’t be called using the dot operator

• Can be called after the object is created

  ```c
  next_point = Point (3,3);
  ```
Demo...