About how long did Exercise 4 take you?

A. [0, 2) hours
B. [2, 4) hours
C. [4, 6) hours
D. [6, 8) hours
E. 8+ Hours
F. I didn’t submit / I prefer not to say
C++ Constructor Insanity (part 1)
CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:

Ann Baturytski  Humza Lala
Yuquan Deng   Alan Li
Noa Ferman    Leanna Mi Nguyen
James Froelich   Chanh Truong
Hannah Jiang    Jennifer Xu
Yegor Kuznetsov
Relevant Course Information

- Exercise 6 released today, due next Monday (10/23)
  - Write a substantive class in C++ (uses a lot of what we will talk about in lecture today)

- Midterm in next Friday’s class (10/27)
  - See course website for details & sample midterms
  - See Ed post about potential review session

- Homework 2 due on 10/30
  - See Ed post about partner finding & confirmation
Lecture Outline (cont’d from last lecture)

- C++ Classes Intro
struct vs. class

- In C, a struct can only contain data fields
  - No methods and all fields are always accessible

- In C++, struct and class are (nearly) the same!
  - Both can have methods and member visibility (public/private/protected)
  - Minor difference: members are default public in a struct and default private in a class

- Common style convention:
  - Use struct for simple bundles of data
  - Use class for abstractions with data + functions
Memory Diagrams for Objects

- An **object** is an instance of a class that maintains its *state* independent from other objects
  - This state is the collection of its data members
  - Conceptually, an object acts like a collection of data fields (plus class metadata)
    - Layout is *not* specified or guaranteed, unlike structs in C

- **Drawn out as variables within variables:**

  ```
  class Point {
    ...
    private:
      int x_;  // data member
      int y_;  // data member
  };  // class Point
  ```
Lecture Outline

- Constructors
- Copy Constructors
- Assignment (next lecture)
- Destructors (next lecture)
Constructors

- A constructor (`ctor`) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
  - A constructor *must* be invoked when creating a new instance of an object – which one depends on *how* the object is instantiated

- Written with the class name as the method name:
  
  ```cpp
  Point(const int x, const int y);
  ```
  
  - C++ will automatically create a *synthesized default constructor* if you have *no* user-defined constructors
    - Takes no arguments and calls the default ctor on all non-“plain old data” (non-POD) member variables
    - Synthesized default ctor will fail if you have non-initialized const or reference data members
Synthesized Default Constructor Example

```cpp
class SimplePoint {
    public:
        // no constructors declared!
        int get_x() const { return x_; } // inline member function
        int get_y() const { return y_; } // inline member function
        double Distance(const SimplePoint& p) const;
        void SetLocation(int x, int y);
    private:
        int x_; // data member
        int y_; // data member
}; // class SimplePoint
```

```
#include "SimplePoint.h"
...
// definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
    SimplePoint x; // invokes synthesized default constructor
    return EXIT_SUCCESS;
}
```
Synthesized Default Constructor

- If you define any constructors, C++ assumes you have defined all the ones you intend to be available and will not add any others

```cpp
#include "SimplePoint.h"

// defining a constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
    x_ = x;
    y_ = y;
}

void Foo() {
    SimplePoint x; // compiler error: if you define any ctors, C++ will NOT synthesize a default constructor for you.

    SimplePoint y(1, 2); // works: invokes the 2-int-arguments constructor
}
```
Multiple Constructors (overloading)

```cpp
#include "SimplePoint.h"

// default constructor
SimplePoint::SimplePoint() {
    x_ = 0;
    y_ = 0;
}

// constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
    x_ = x;
    y_ = y;
}

void Foo() {
    SimplePoint x;       // invokes the default constructor
    SimplePoint y(1, 2); // invokes the 2-int-arguments ctor
    SimplePoint a[3];    // invokes the default ctor 3 times
}
```
Initialization Lists

- C++ lets you **optionally** declare an **initialization list** as part of a constructor definition
  - Initializes fields according to parameters in the list
  - The following two are (nearly) identical:

```cpp
Point::Point(const int x, const int y) { 
    x_ = x;
    y_ = y;
    std::cout << "Point constructed: (" << x_ << ",";
    std::cout << y_<< ")" << std::endl;
}
```

```cpp
Point::Point(const int x, const int y) : x_(x), y_(y) {
    std::cout << "Point constructed: (" << x_ << " ,";
    std::cout << y_<< ")" << std::endl;
}
```
Initialization vs. Construction

Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering (!)

Data members that don’t appear in the initialization list are default initialized/constructed before body is executed

Initialization preferred to assignment to avoid extra steps

• Real code should never mix the two styles
Lecture Outline

- Constructors
- **Copy Constructors**
- Assignment (next lecture)
- Destructors (next lecture)
Copy Constructors

- C++ has the notion of a **copy constructor** (`cctor`)
  - Used to create a new object as a copy of an existing object

```cpp
Point::Point(const int x, const int y) : x_(x), y_(y) {} // copy constructor

Point::Point(const Point& copyme) {
    x_ = copyme.x_;  
    y_ = copyme.y_;  
}

void Foo() {
    Point x(1, 2);  // invokes the 2-int-arguments **constructor**
    Point y(x);    // invokes the **copy constructor**
    // could also be written as "Point y = x;"
}
```

- Initializer lists can also be used in copy constructors (preferred)
Synthesized Copy Constructor

- If you don’t define your own copy constructor, C++ will synthesize one for you
  - It will do a **shallow** copy of all of the fields *(i.e., member variables)* of your class *(can be problematic with pointers)*
  - Sometimes the right thing; sometimes the wrong thing

```cpp
#include "SimplePoint.h"
...
// definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
    SimplePoint x;
    SimplePoint y(x); // invokes synthesized copy constructor
    ...
    return EXIT_SUCCESS;
}
```
When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:
    
    ```
    Point x;       // default ctor
    Point y(x);    // copy ctor
    Point z = y;   // copy ctor
    ```
  
  - You pass a non-reference object as a value parameter to a function:

    ```
    void Foo(Point x) { ... }
    Point y;         // default ctor
    Foo(y);          // copy ctor
    ```

  - You return a non-reference object value from a function:

    ```
    Point Foo() {
        Point y;     // default ctor
        return y;    // copy ctor
    }
    ```
Compiler Optimization

- The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies.
- Sometimes you might not see a constructor get invoked when you might expect it.

```cpp
Point Foo() {
    Point y;    // default ctor
    return y;   // copy ctor? optimized?
}

int main(int argc, char** argv) {
    Point x(1, 2); // two-ints-argument ctor
    Point y = x;   // copy ctor
    Point z = Foo(); // copy ctor? optimized?
}
```
Extra Exercise #1

- Write a C++ program that:
  - Has a class representing a 3-dimensional point
  - Has the following methods:
    - Return the inner product of two 3D points
    - Return the distance between two 3D points
    - Accessors and mutators for the $x$, $y$, and $z$ coordinates
Extra Exercise #2

- Write a C++ program that:
  - Has a class representing a 3-dimensional box
    - Use your Extra Exercise #1 class to store the coordinates of the vertices that define the box
    - Assume the box has right-angles only and its faces are parallel to the axes, so you only need 2 vertices to define it
  - Has the following methods:
    - Test if one box is inside another box
    - Return the volume of a box
    - Handles <<, =, and a copy constructor
    - Uses const in all the right places