C++ Constructor Insanity
CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:
Deeksha Vatwani   Hannah Jiang   Jen Xu
Leanna Nguyen   Nam Nguyen   Sayuj Shahi
Tanay Vakharia   Wei Wu   Yiqing Wang
Zohar Le
Lecture Outline

❖ Constructors
❖ Copy Constructors
❖ Assignment
❖ Destructors
❖ An extended example
Constructors

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
  - Which one is invoked depends on how the object is instantiated

- Written with the class name as the method name:

  ```cpp
  Point(const int x, const int y);
  ```
Default Constructor

- The **default constructor** does not take any parameters
  
  ```cpp
  Point();
  ```

- C++ will automatically **synthesize a default constructor** if you have *no* user-defined constructors
  - Calls the default ctors on all non-“plain old data” (non-POD) member variables
  - Will fail if you have non-initialized const or reference data members
Synthesized Default Constructor

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(const int x, const 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 0;
}
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 "Point.h"

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

void foo() {
    Point x; // compiler error: if you define any ctors, C++ will NOT synthesize a default constructor for you.
    Point y(1, 2); // works: invokes the 2-int-arguments constructor
}
```
Multiple Constructors (overloading)

```cpp
#include "Point.h"

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

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

void foo() {
  Point x;       // invokes the default constructor
  Point a[3];    // invokes the default ctor 3 times
                  //   (fails if no default ctor)
  Point y(1, 2); // invokes the 2-int-arguments ctor
}
```
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;
}

// constructor with an initialization list
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

```cpp
class Point3D {
public:
    // constructor with 3 int arguments
    Point3D(const int x, const int y, const int z)
        : y_(y), x_(x) {
        z_ = z;
    }
private:
    int x_, y_, z_;  // data members
};  // class Point3D
```

First, initialization list is applied. Next, constructor body is executed.
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 of default initialization (construction) followed by assignment

- (and no, real code should never mix the two styles this way 😇)
Initialization vs. Construction

- The difference between initialization and assignment start to matter when we have:
  - objects as member variables
  - const member variables
  - reference member variables

```cpp
class Triangle {
public:
  Triangle(const Point& p1, const Point& p2, const Point& p3)
    : p1_(p1.get_x(), p1.get_y()) {  
    // constructor body
  }

private:
  Point p1_, p2_, p3_;  
  const Point kOrigin;
};  // class Triangle
```

Triangle.h

2-parameter constructor called on p1_, but default constructor called on p2_, p3_, and kOrigin – is the default constructor's behavior what we want?
Lecture Outline

❖ Constructors
❖ Copy Constructors
❖ Assignment
❖ Destructors
❖ An extended example
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_;  // copyme.x_
  y_ = copyme.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)
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?
}

Point x(1, 2);  // two-ints-argument ctor
Point y = x;     // copy ctor
Point z = foo(); // copy ctor? optimized?
```
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
  - Sometimes the right thing; sometimes the wrong thing

```cpp
#include "SimplePoint.h"

int main(int argc, char** argv) {
    SimplePoint x;
    SimplePoint y(x); // invokes synthesized copy constructor
    ...
    return 0;
}
```
Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- Destructors
- An extended example
Assignment != Construction

❖ “=” is the assignment operator
  ▪ Assigns values to an existing, already constructed object

```cpp
Point w;            // default ctor
Point x(1, 2);     // two-ints-argument ctor
Point y(x);        // copy ctor
Point z = w;       // copy ctor
y = x;             // assignment operator
```

❖ How can you tell the difference between assignment operator= and a copy constructor that uses =?
  • Answer: are you creating/initializing a new object? If so, it’s a copy constructor; if you are just updating an existing object it’s assignment
How many of each method gets called?

- Default constructor
- Two-parameter constructor
- Copy constructor
- Assignment operator

```cpp
class Triangle {
    public:
    Triangle(const Point& p1, const Point& p2, const Point& p3)
        : p1_(p1.get_x(), p1.get_y()) {
        // constructor body
    }

    private:
    Point p1_, p2_, p3_;  // class Triangle
};
```

Having a constant as a member variable is not a good design – better to have only one copy of a constant!
Overloading the “=” Operator

❖ You can choose to define the “=” operator
   ▪ But there are some rules you should follow:

```cpp
Point& Point::operator=(const Point& rhs) {
  if (this != &rhs) { // (1) always check against this
    x_ = rhs.x_;  
    y_ = rhs.y_;   
  }
  return *this;   // (2) always return *this from op=
}
```

```cpp
Point c;             // default constructor
a = b = c;          // works because = return *this
a = (b = c);        // equiv. to above (= is right-associative)
(a = b) = c;        // "works" because = returns a non-const
```
Synthesized Assignment Operator

- If you don’t define the assignment operator, C++ will synthesize one for you
  - It will do a shallow copy of all of the fields (i.e. member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing

```c++
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
    SimplePoint x;
    SimplePoint y(x);
    y = x; // invokes synthesized assignment operator
    return 0;
}
```
Lecture Outline

❖ Constructors
❖ Copy Constructors
❖ Assignment
❖ Destructors
❖ An extended example
Destructors

- C++ has the notion of a destructor (dtor)
  - Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)
  - Place to put your cleanup code – free any dynamic storage or other resources owned by the object
  - Standard C++ idiom for managing dynamic resources
    - Slogan: “Resource Acquisition Is Initialization” (RAII)

```cpp
Point::~Point() {  // destructor
    // do any cleanup needed when a Point object goes away
    // (nothing to do here since we have no dynamic resources)
}
```
How many times does the destructor get invoked?

- Assume `Point` with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

```cpp
Point PrintRad(Point& pt) {
    Point origin(0, 0);
    double r = origin.Distance(pt);
    double theta = atan2(pt.get_y(), pt.get_x());
    cout << "r = " << r << endl;
    cout << "theta = " << theta << " rad" << endl;
    return pt;
}

int main(int argc, char** argv) {
    Point pt(3, 4);
    PrintRad(pt);
    return 0;
}
```

A. 1  
B. 2  
C. 3  
D. 4  
E. We’re lost...
Lecture Outline

❖ Constructors
❖ Copy Constructors
❖ Assignment
❖ Destructors
❖ An extended example
Complex Example Walkthrough

See:

Complex.h
Complex.cc
testcomplex.cc

❖ (Some details like friend functions and namespaces are explained in more detail next lecture, but ideas should make sense from looking at the code and explanations in C++ Primer.)
Extra Exercise #1

- Modify your Point3D class from Lec 10 Extra #1
  - Disable the copy constructor and assignment operator
  - Attempt to use copy & assignment in code and see what error the compiler generates
  - Write a `CopyFrom()` member function and try using it instead
    - (See details about `CopyFrom()` in next lecture)
Extra Exercise #2

❖ Write a C++ class that:
  ▪ Is given the name of a file as a constructor argument
  ▪ Has a `GetNextWord()` method that returns the next whitespace- or newline-separated word from the file as a copy of a `string` object, or an empty string once you hit EOF
  ▪ Has a destructor that cleans up anything that needs cleaning up