C++ Constructor Insanity

CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:

Andrew Hu
Austin Chan
Brennan Stein
Cheng Ni
Cosmo Wang
Diya Joy
Guramrit Singh
Mengqi Chen
Pat Kosakanchit
Rehaan Bhimani
Renshu Gu
Travis McGaha
Zachary Keyes

Administrivia

- Exercise 10 released today, due Monday
  - Write a substantive class in C++!

- Homework 2 due next Thursday (2/6)
  - File system crawler, indexer, and search engine
  - Note: `libhw1.a` (yours or ours) and the `.h` files from hw1 need to be in right directory (`~yourgit/hw1/`)
  - Note: use Ctrl-D to exit `searchshell`, test on directory of small self-made files

Class Definition (.h file)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(int x, int y);                 // constructor
  int get_x() const { return x_; }     // inline member function
  int get_y() const { return y_; }     // inline member function
  double Distance(const Point& p) const;      // member function
  void SetLocation(int x, int y);             // member function

 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point

#endif  // POINT_H_
```

this const means that this function is not allowed to change the object on which it is called (the implicit "this" pointer)

function definitions

declarations

compiler may chouse to expand inline (like a macro) instead on an actual function call

naming convention for class data members (Google C++ style guide)

Class Member Definitions (.cc file)

Point.cc

BAD STYLE
used here on purpose

```
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int y) {
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}
```

equivalent to y_ = y;

"this" is a (Point * const)

makes "this" a (const Point * const)

```
double Point::Distance(const Point& p) const {
  // We can access p's x_ and y_ variables either through the
  // get_x(), get_y() accessor functions or the x_, y_ private
  // member variables directly, since we're in a member
  // function of the same class.
  double distance = (x_ - p.get_x()) * (x_ - p.get_x());
  distance += (y_ - p.y_) * (y_ - p.y_);
  return sqrt(distance);
}
```

equivalent to p.x_

can't be const because we are mutating "this"

```
void Point::SetLocation(const int x, const int y) {
  x_ = x;
  y_ = y;
}
```

Class Usage (.cc file)

usepoint.cc

```
#include <iostream>
#include <cstdlib>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
  Point p1(1, 2);  // allocate a new Point on the Stack
  Point p2(4, 6);  // allocate a new Point on the Stack

  cout << "p1 is: (" << p1.get_x() << ", ";
  cout << p1.get_y() << ")" << endl;

  cout << "p2 is: (" << p2.get_x() << ", ";
  cout << p2.get_y() << ")" << endl;

  cout << "dist : " << p1.Distance(p2) << endl;
  return EXIT_SUCCESS;
}
```

calls defined constructor

"dot notation" used for member functions

Point* p;

p->get_x(); ⟺ (*p).get_x();

`struct` vs. `class`

STYLE TIP

- 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 ← public data members with names like x, y
  - Use `class` for abstractions with data + functions

↖ private data members with names like x_, y_

Lecture Outline

- **Constructors**
- Copy Constructors
- Assignment
- Destructors

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:

```
Point(const int x, const int y);
```

  created for you zero-argument

  - 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

```
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
```

default behavior → primitives: just allocate space (garbage)
→ objects: default construct

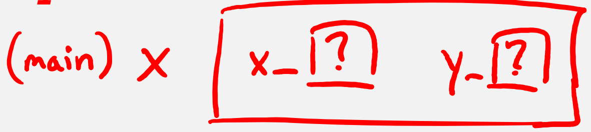
SimplePoint.h

```
#include "SimplePoint.h"

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

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

SimplePoint.cc



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

```
#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
}
```

added, so no synthesized def ctor

Multiple Constructors (overloading)

```
#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
}
```

added, so now there is a def. ctor

int: a | ? | ? | ? |

SimplePoint: a | x_ 0 y_ 0 | x_ 0 y_ 0 | x_ 0 y_ 0 |

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:

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

can be expressions

```
// 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;
}
```

body can be empty { }

member names

Initialization vs. Construction

STYLE TIP

```
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.

②set y_ ①set x_ ③set z_ (garbage)

Next, constructor body is executed.

④ set z_

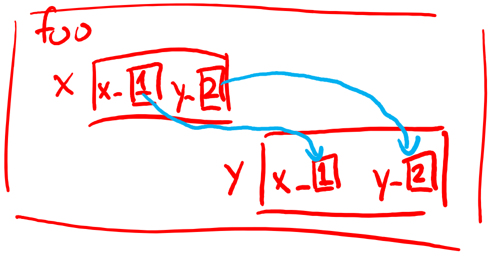
#1 #2 #3

- 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
- Destructors

Copy Constructors



STYLE TIP

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

```
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;"
}
```

reference to object of same class

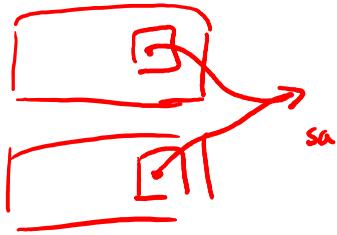
alias to binds object

a ctor must be called because the object didn't exist previously.

constructing from existing object, so we use the copy ctor.

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

Synthesized Copy Constructor



now point to the same thing!

- 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

```
#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
```

pass-by-value of an object

  - 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

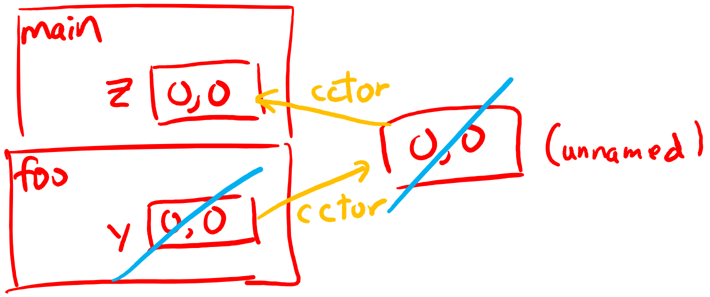
  can read up on your own if interested

  (unnamed temporary object)

  - Sometimes you might not see a constructor get invoked when you might expect it

```
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?
```



Lecture Outline

- Constructors
- Copy Constructors
- **Assignment**
- Destructors

Assignment != Construction

- "=" is the assignment operator
  - Assigns values to an *existing, already constructed* object

```
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
```

z did not exist →

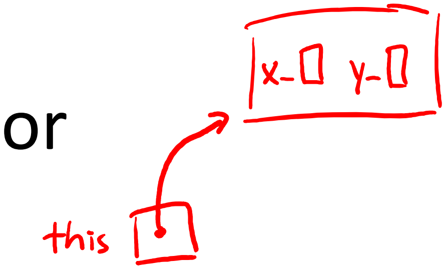
y exists →

method operator=( )

STYLE TIP

Overloading the "=" Operator

- You can choose to define the "=" operator
  - But there are some rules you should follow:



```
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=
}

Point a;         // 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
```

more important when dealing with dynamically allocated memory

returns reference to class object (allows for chaining)

a.operator=(b.operator=(c))

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

```
#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 EXIT_SUCCESS;
}
```

Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- **Destructors**

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)

tilde

no parameters

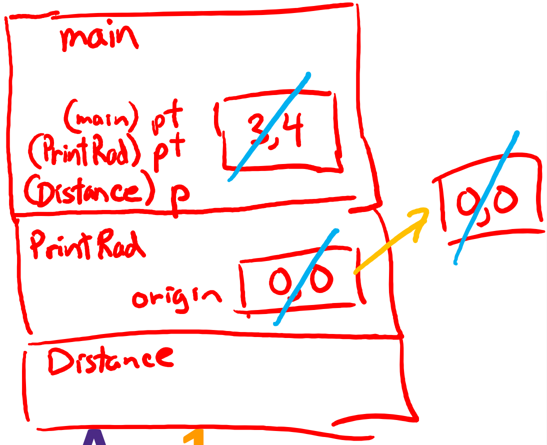
```
Point::~Point() {    // destructor
  // do any cleanup needed when a Point object goes away
  // (nothing to do here since we have no dynamic resources)
}
```

executed in reverse order as ctor: ① body of dtor
② destruct members in reverse order of declaration

Polling Question

ctor	cctor	op=	dtor
2	1	0	3

- How many times does the ***destructor*** get invoked?
  - Assume `Point` with everything defined (ctor, cctor, =, dtor)
  - Assume no compiler optimizations



test.cc

```
Point PrintRad(Point& pt) {
  Point origin(0, 0);  // ② ctor called
  double r = origin.Distance(pt); // Distance takes ref, so object NOT copied
  double theta = atan2(pt.get_y(), pt.get_x());
  cout << "r = " << r << endl;
  cout << "theta = " << theta << " rad" << endl;
  return pt; // ③ PrintRad returns an object, so cctor is called to create a temp
}            // ④ while cleaning up, origin is destructed

int main(int argc, char** argv) {
  Point pt(3, 4);  // ① ctor called
  PrintRad(pt);    // PrintRad takes ref, so pt is NOT copied
  return 0;        // ⑤ return value of PrintRad ignored; temp is destructed
}                  // ⑥ while cleaning up, pt is destructed
```

A. 1

B. 2

[✓ C. 3]

D. 4

E. We're lost…

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