

C++ Constructor Insanity

CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:

Danny Allen

Dennis Shao

Eddie Huang

Kevin Bi

Jack Xu

Matthew Neldam

Michael Poulain

Renshu Gu

Robby Marver

Waylon Huang

Wei Lin

Administrivia

- ❖ Exercise 9 released today, due Friday
 - First submitted Makefile!
- ❖ Homework 2 due next Thursday (4/26)
 - 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(const int x, const 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(const int x, const int y); // member function

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

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

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

Compiler may choose to expand inline (like a macro) instead on an actual function call

```
#endif // _POINT_H_
```

Class Member Definitions (.cc file)

Point.cc

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

Point::Point(const int x, const int y) {
    x_ = x;           equivalent to y_ = y;
    this->y_ = y;    // "this->" is optional unless name conflicts
}   "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.                                equivalent to p.x-
    double distance = (x_ - p.get_x()) * (x_ - p.get_x());
    distance += (y_ - p.y_) * (y_ - p.y_);
    return sqrt(distance);
}

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

BAD STYLE
used here on purpose

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

Class Usage (.cc file)

usepoint.cc

```
#include <iostream>
#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 } calls defined
                    constructor

    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 0;
}
```

"dot notation" used for member functions

Point* p;

p->get_x(); \Leftrightarrow (*p).get_x();

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

Lecture Outline

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

Constructors

- ❖ A **constructor (ctor)** initializes a newly-instantiated object
 - A class can have multiple ctors 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::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

```
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
```

} default ctor just allocates space (garbage!)

SimplePoint.h

```
#include "SimplePoint.h"  
... // definitions for Distance() and SetLocation()  
  
int main(int argc, char** argv) {  
    SimplePoint x; // invokes synthesized default constructor  
    SimplePoint y(x);  
    y = x;  
    return 0;  
}
```

SimplePoint.cc

Synthesized Default Constructor

```
#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

```
#include "SimplePoint.h"

// default constructor
SimplePoint::SimplePoint() {
    x_ = 0;
    y_ = 0; } we now initialize to zero
}

// 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 a[3]; // invokes the default ctor 3 times
    SimplePoint y(1, 2); // invokes the 2-int-arguments ctor
}
```

must construct each element of the array



Initialization Lists

- ❖ C++ lets you *optionally* declare an **initialization list** as part of your 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;  
}
```

```
// 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;  
}
```

can be expressions

Initialization vs. Construction

```
class 3DPoint {  
public:  
    // constructor with 3 int arguments  
    3DPoint(const int x, const int y, const int z) : y_(y), x_(x)  
    {  
        z_ = z; // ①  
    }  
  
private:  
    int x_, y_, z_; // data members  
}; // class 3DPoint
```

First, initialization list is applied.

Next, constructor body is executed.

- Data members are initialized in the order they are defined, 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

Lecture Outline

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

Copy Constructors

- ❖ 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;"

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

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

reference to object of same class

alias to binds to object

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 parameter to a function:

```
void foo(Point x) { ... }
Point y;
foo(y);
```

pass-by-value of an object

- You return a non-reference object 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

```
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?
```

↑ can read up on your own
if interested

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

```
#include "SimplePoint.h"

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

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

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
                  (method operator=( ))
```

Overloading the “=” Operator

- ❖ You can choose to **overload** 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 =
}
```

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)

Synthesized Assignment Operator

- ❖ If you don't overload 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( )

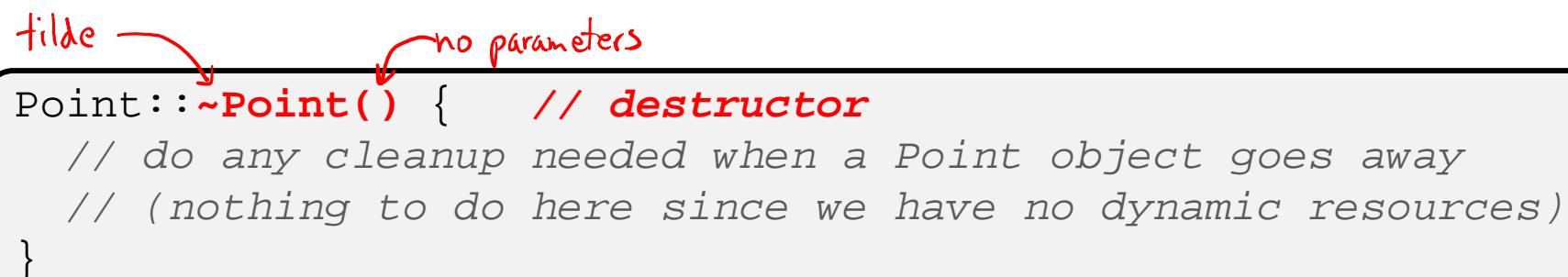
int main(int argc, char** argv) {
    SimplePoint x;
    SimplePoint y(x);
    y = x;           // invokes synthesized assignment operator
    return 0;
}
```

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)



The diagram shows a code snippet for a destructor in the Point class. Red annotations with arrows point to specific parts of the code:

- A red arrow points to the tilde character (~) in the method name, with the word "tilde" written above it.
- A red arrow points to the parameter list in the method signature, with the words "no parameters" written above it.

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

Peer Instruction Question

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

A. 1

B. 2

C. 3

D. 4

E. We're lost...

```
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
```

Rule of Three

- ❖ If you define any of:
 - 1) Destructor
 - 2) Copy Constructor
 - 3) Assignment (`operator=`)
- ❖ Then you should normally define all three
 - Can explicitly ask for default synthesized versions (C++11):

```
class Point {  
public:  
    Point() = default;                                // the default ctor  
    ~Point() = default;                                // the default dtor  
    Point(const Point& copyme) = default;                // the default cctor  
    Point& operator=(const Point& rhs) = default; // the default "="  
    ...
```

Dealing with the Insanity

- ❖ C++ style guide tip:
 - If possible, **disable** the copy constructor and assignment operator by declaring as private and not defining them

Point.h

```
class Point {  
public:  
    Point(const int x, const int y) : x_(x), y_(y) {} // ctor  
    ...  
private:  
    Point(const Point& copyme); // disable cctor (no def.)  
    Point& operator=(const Point& rhs); // disable "=" (no def.)  
    ...  
}; // class Point  
  
Point w; // compiler error (no default constructor)  
Point x(1, 2); // OK!  
Point y = w; // compiler error (no copy constructor)  
y = x; // compiler error (no assignment operator)
```

Disabling in C++11

- ❖ C++11 add new syntax to do this directly
 - This is the better choice in C++11 code

Point_2011.h

```
class Point {  
public:  
    Point(const int x, const int y) : x_(x), y_(y) {} // ctor  
    ...  
    Point(const Point& copyme) = delete; // declare cctor and "=" as  
    Point& operator=(const Point& rhs) = delete; // as deleted (C++11)  
private:  
    ...  
}; // class Point  
  
Point w; // compiler error (no default constructor)  
Point x(1, 2); // OK!  
Point y = w; // compiler error (no copy constructor)  
y = x; // compiler error (no assignment operator)
```

CopyFrom

- ❖ C++11 style guide tip
 - If you disable them, then you should instead probably have an explicit “CopyFrom” function

Point.h

```
class Point {  
public:  
    Point(const int x, const int y) : x_(x), y_(y) {} // ctor  
    void CopyFrom(const Point& copy_from_me);  
    ...  
    Point(Point& copyme); // disable cctor (no def.)  
    Point& operator=(Point& rhs); // disable "=" (no def.)  
private:  
    ...  
}; // class Point
```

sanepoint.cc

```
Point x(1, 2); // OK  
Point y(3, 4); // OK  
x.CopyFrom(y); // OK
```

Extra Exercise #1

- ❖ Modify your 3DPoint 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

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