C++ Contructor  INSANITY 

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

pollev.com/cse333

About how long did Exercise 8 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn't submit / I prefer not to say

Administrivia

- Exercise 9 released today, due Monday
  - Write a substantive class in C++ (but no dynamic allocation – yet)
  - First submitted Makefile!
- Homework 2 due next Thursday (10/24)
  - More complex than HW1: file system crawler, indexer, and search engine

Lecture Outline

- **Intro to Classes in C++**
- Constructors
- Copy Constructors
- Assignment

Classes

- Class definition syntax (in a `.h` file):

```
class Name {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
}; // class Name
```

  - Members can be functions (methods) or data (variables)

- Class member function definition syntax (in a `.cc` file):

```
retType Name::MethodName(type1 param1, …, typeN paramN) {
  // body statements
}
```

  - (1) *define* within the class definition or (2) *declare* within the class definition and then *define* elsewhere

Class Organization

- Similar conceptually to C when modularizing with `structs`:
  - Class definition is part of interface and should go in `.h` file
    - Private members still must be included in definition (**!**)
  - Usually put member function *definitions* into companion `.cc` file
    - Common exception: setter and getter methods
  - Both can also include non-member functions that use the class

- Unlike Java, you can name files anything you want
  - Typically `Name.cc` and `Name.h` for `class Name`

Class Definition (.h file)

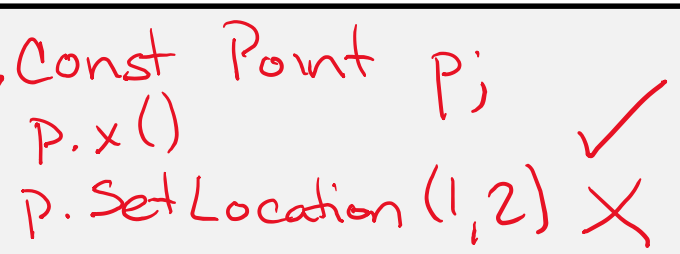
Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point() { }                 // inline constructor defn
  Point(int x, int y);        // constructor
  int x() const { return x_; }  // inline method defn
  int y() const { return y_; }  // inline method defn
  double Distance(const Point &p) const;  // method decl
  void SetLocation(int x, int y);         // method decl

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

#endif  // POINT_H_
```



Class Member Definitions (. cc file)

Point.cc

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

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

double Point::Distance(const Point &p) const {
  // We can access p's x and y values either through (1) the x()
  // and y() accessor functions or (2) the x_ and y_ private
  // member variables directly (since we're in a member
  // function of the same class).
  double distance = (x_ - p.x()) * (x_ - p.x());
  distance += (y_ - p.y_) * (y_ - p.y_);
  return sqrt(distance);
}

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

inconsistent.
DO NOT DO THIS

Class Usage (. cc file)

usepoint.cc

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

using std::cout;
using std::endl;

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

  cout << "p1 is: (" << p1.x() << ", ";
  cout << p1.y() << ")" << endl;

  cout << "p2 is: (" << p2.x() << ", ";
  cout << p2.y() << ")" << endl;

  cout << "dist : " << p1.Distance(p2) << endl;

  p1.SetLocation(2, 1);   // YES: non-const method on
                          // non-const instance
  p2.SetLocation(6, 4);   // NO: non-const method on const instance
  return 0;
}
```

`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

- Intro to Classes in C++
- **Constructors**
- Copy Constructors
- Assignment

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
- The "default constructor" takes no arguments
  - Eg, it's invoked for every element in an array
- Written with the class name as the method name:

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

```
Point();
```

  - 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 Point {
 public:
  // no constructors declared!
  int x() const { return x_; }      // inline member function
  int 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
```

Point.h

```
#include "Point.h"

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

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

Point.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 "Point.h"

// defining a constructor with two arguments
Point::Point(int x, 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)

Point.cc

```
#include "Point.h"

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

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

void foo() {
  Point x;         // invokes the default constructor
  Point a[3];      // invokes the default ctor 3 times
  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:

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

```
// constructor with an initialization list
Point::Point(int x, int y) : x_(x), y_(y) {
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

can put any valid expr inside parens. Eg:
y_(x*x)
y_(1000)
y_(CalculatePi())

Point.cc

Initialization vs. Construction

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(int x, int y, 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.

- Member variables are constructed in the order they are defined in the class, not by the initialization list ordering (**!**)
  - Member construction *always* happens before ctor body is executed
  - Data members that don't appear in the initialization list are *default initialized/constructed*

- Initialization preferred to assignment to avoid extra steps
  - Real code should never mix the two styles

Lecture Outline

- Intro to Classes in C++
- Constructors
- **Copy Constructors**
- Assignment

Copy Constructors

If it came into existance at the same time as the copy, it's the copy constructor

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

```
Point::Point(int x, 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)
  - Sometimes the right thing; sometimes the wrong thing

```
#include "Point.h"

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

int main(int argc, char **argv) {
  Point x;
  Point 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 "return by value optimization" or "move semantics" to eliminate unnecessary copies
  - May not see a copy constructor 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

- Intro to Classes in C++
- Constructors
- Copy Constructors
- **Assignment**

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
```

(because y already existed)

Overloading the "=" Operator

Point x;
Point &y=x
have same address, ie
&x == &y
Can use this fact to check for self assign:
x=x x=y
y=y y=x

- 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, to
    x_ = rhs.x_;       // protect against self-assignment
    y_ = rhs.y_;
  }
  return *this;        // (2) always return *this from op=
}

Point a, b, 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) Sometimes the right thing; sometimes the wrong thing

```
#include "Point.h"

... // no decl/defn for operator=

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

What Gets Called When?

noisycopies.cc

```
#include "Point.h"

Point Helper(const Point &parg) {  // 4) no ctor; pass-by-ref
  Point p4;                        // 5) default ctor
  p4 = parg;                       // 6) assignment operator
  return p4;                       // 7) copy ctor copies p4 into
                                   //    main()'s stack frame
}

int main(int argc, char **argv) {
 Point p1(1, 1);           // 1) 2-arg ctor
 Point p2 = p1;            // 2) copy ctor
 Point p3 = Helper(p1);    // 8) p3 initialized by copy ctor from
                           //    Helper()'s also-copied instance
                           //    (see step 7)

 return 0;
}
```

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