C++ Constructor Insanity (part 1)

CSE 333 Spring 2023

Instructor: Chris Thachuk

Teaching Assistants:

Byron Jin
Deeksha Vatwani
Humza Lala
Noa Ferman
Seulchan (Paul) Han
Tim Mandzyuk
CJ Reith
Edward Zhang
Lahari Nidadavolu
Saket Gollapudi
Timmy Yang
Wui Wu

Relevant Course Information

- Exercise 6 released today, due next Monday (4/24)
  - Write a substantive class in C++ (uses a lot of what we will talk about in lecture today)
  - Testable material on midterm, but will count as a "bonus" w.r.t. exercise grading

- Homework 2 due next Thursday (4/27)
  - 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`
  - Tip: test on directory of small self-made files
  - Partner confirmation by *4/20 @ 11:59 PST;* No exceptions!

Lecture Outline (cont'd from last lecture)

- C++ Classes Intro

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

- It's a little more complex than in C when modularizing with `struct` definition:
  - 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 with implementation details
    - Common exception: setter and getter methods
  - These files 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`

Const & Classes

- Like other data types, **objects** can be declared as `const`:
  - Once a `const` object has been constructed, its member variables can't be changed
  - Can only invoke member functions that are labeled `const`

- You can declare a member **function** of a class as `const`
  - This means that it cannot modify the object it was called on
    - The compiler will treat member variables as `const` inside the function at compile time
  - If a member function doesn't modify the object, mark it `const`!

Class Definition (.h file)

STYLE TIP

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

#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;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

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);
}

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;
}
```

Reading Assignment

- ***Read*** the sections in *C++ Primer* covering class constructors, copy constructors, assignment (`operator=`), and destructors
  - Ignore "move semantics" for now
  - The table of contents and index are your friends…

`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:

```
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

```
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
```

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
}
```

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
}
```

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;
}
```

```
// 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


```
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.

- 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

```
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
  - 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
```

  - 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

```
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