

# CSE 333

## Lecture 11 - constructor insanity

**Hal Perkins**

Department of Computer Science & Engineering  
University of Washington





# Administrivia

HW2 out now, due a week from Tuesday

Midterm the following Friday

- Everything up to C++ basics (e.g., exercises)
- Old exams on the web now; topic list will be updated if needed by early next week

New exercise posted today, due Monday before class



# Today's goals

More details on constructors, destructors, operators

Walk through *complex\_example/*

- pretty hairy and complex
- a lesson on why using a **subset of C++** is often better

*new / delete / delete[ ]*

- *str/* example



# Constructors

*A constructor initializes a newly instantiated object*

- a class can have multiple constructors
  - ▶ they differ in the arguments that they accept
  - ▶ which one is invoked depends on how the object is instantiated

*You can write constructors for your object*

- but if you don't write any, C++ might automatically synthesize a *default constructor* for you
  - ▶ the default constructor is one that takes no arguments and that calls default constructors on all non-POD member variables
  - ▶ C++ does this iff your class has no const or reference data members, and no other user-defined constructors



# Example of synthesis

*see SimplePoint.cc, SimplePoint.h*



# Constructors, continued

You might choose to define multiple constructors:

```
Point::Point() {
    x_ = 0;
    y_ = 0;
}

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

void foo() {
    Point x; // invokes the default (argument-less) constructor
    Point y(1,2); // invokes the two-int-arguments constructor
}
```



# Constructors, continued

You might choose to define only one:

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

void foo() {
    // Compiler error; if you define any constructors, C++ will
    // not automatically synthesize a default constructor for you.
    Point x;

    // Works.
    Point y(1,2); // invokes the two-int-arguments constructor
}
```



# Initialization lists

As shorthand, C++ lets you declare an initialization list as part of your constructor declaration

- initializes fields according to parameters in the list
- the following two are (nearly) equivalent:

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

```
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

When a new object is created using some constructor:

- first, the initialization list is applied to members
  - in the order that those members appear within the class definition, not the order in the initialization list (!)
- next, the constructor is invoked, and any statements within it are executed

Prefer initializations to assignment

- Objects are already initialized by some constructor before they can be assigned - initializer avoids two separate steps



# Initialization vs. construction

```
#ifndef _POINT_H_
#define _POINT_H_

class Point {
public:
    Point(const int x, const int y, const int z)
        : x_(x), y_(y) {
        z_ = z;
    }

private:
    int x_, y_, z_;
}; // class Point

#endif // _POINT_H_
```



# Initialization vs. construction

first,  
initialization  
list is  
applied

```
#ifndef _POINT_H_
#define _POINT_H_

class Point {
public:
    Point(const int x, const int y, const int z)
        : x_(x), y_(y) {
        z_ = z;
    }

private:
    int x_, y_, z_;
}; // class Point

#endif // _POINT_H_
```



# Initialization vs. construction

```
#ifndef _POINT_H_
#define _POINT_H_

class Point {
public:
    Point(const int x, const int y, const int z)
        : x_(x), y_(y) {
        z_ = z;
    }

private:
    int x_, y_, z_;
}; // class Point

#endif // _POINT_H_
```

next,  
constructor  
is executed





# Copy constructors

C++ has the notion of a **copy constructor**

- 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) { }

Point::Point(const Point &copyme) { // copy constructor
    x_ = copyme.x_;
    y_ = copyme.y_;
}

void foo() {
    // invokes the two-int-arguments constructor
    Point x(1,2);

    // invokes the copy constructor to construct y as a copy of x
    Point y(x); // could also write as "Point y = x;"
}
```



# When do copies happen?

The copy constructor is invoked if:

- you pass an object as an parameter to a call-by-value function
- you return an object from a function
- you initialize an object from another object of the same type

```
void foo(Point x) { ... }  
  
Point y; // default cons.  
foo(y); // copy cons.
```

```
Point foo() {  
    Point y; // default cos.  
    return y; // copy cons.  
}
```

```
Point x; // default cons.  
Point y(x); // copy cons.  
Point z = y; // copy cons.
```



# But...the compiler is smart...

It sometimes uses a “return by value optimization” to eliminate unnecessary copies

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

```
Point foo() {  
    Point y; // default constructor.  
    return y; // copy constructor? optimized?  
}  
  
Point x(1,2); // two-ints-argument constructor.  
Point y = x; // copy constructor.  
Point z = foo(); // copy constructor? 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

*see SimplePoint.cc, SimplePoint.h*



# assignment != construction

The “=” operator is the assignment operator

- assigns values to an existing, already constructed object
- you can overload the “=” operator

```
Point w;           // default constructor.  
Point x(1,2);     // two-ints-argument constructor.  
Point y = w;      // copy constructor.  
y = x;           // assignment 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) { // always check against this  
        x_ = rhs.x_  
        y_ = rhs.y_  
    }  
    return *this; // always return *this from =  
}  
  
Point a; // default constructor  
a = b = c; // works because "=" returns *this  
a = (b = c); // equiv to above, as "=" is right-associative  
(a = b) = c; // works because "=" returns a non-const
```



# Synthesized assignment oper.

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

*see SimplePoint.cc, SimplePoint.h*



# Destructors

C++ has the notion of a **destructor**

- invoked automatically when a class instance is deleted / goes out of scope, etc., even via exceptions or other causes
- place to put 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)

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



*see complex\_example/\**



# Dealing with the insanity

## C++ style guide tip

- if possible, disable the copy const. and assignment operator
  - *not possible if you want to store objects of your class in an STL container, unfortunately*

```
class Point {
public:
    Point(int x, int y) : x_(x), y_(y) { }

private:
    // disable copy cons. and "=" by declaring but not defining
    Point(Point &copyme);
    Point &operator=(Point &rhs);
};

Point w;           // compiler error
Point x(1,2);     // OK
Point y = x;      // compiler error
x = w;            // compiler error
```



# Dealing with the insanity

## C++ style guide tip

- if you disable them, then you should instead have an explicit “CopyFrom” function

```
class Point { Point.cc, h
public:
    Point::Point(int x, int y) : x_(x), y_(y) { }
    void CopyFrom(const Point &copy_from_me);

private:
    // disable copy cons. and "=" by declaring but not defining
    Point(const Point &copyme);
    Point &operator=(const Point &rhs);
};
```

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

sanepoint.cc



# new

To allocate on the heap using C++, you use the “new” keyword instead of the “malloc( )” `stdlib.h` function

- you can use `new` to allocate an object
- you can use `new` to allocate a primitive type

To deallocate a heap-allocated object or primitive, use the “delete” keyword instead of the “free( )” `stdlib.h` function

- if you’re using a legacy C code library or module in C++
  - ▶ if C code returns you a `malloc( )`’d pointer, use `free( )` to deallocate it
  - ▶ **never** `free( )` something allocated with `new`
  - ▶ **never** delete something allocated with `malloc( )`



new / delete

*see [heappoint.cc](http://heappoint.cc)*



# C++11 nullptr

C and C++ have long used `NULL` as a pointer value that references nothing

C++11 introduces a new literal for this: `nullptr`

- New reserved word
- Interchangeable with `NULL` for all practical purposes
  - ▶ But it has type  $T^*$  for any/every  $T$ , and is not an integer value
    - Avoids funny edge cases (see C++ references for details)
    - Still can convert to/from integer 0 for tests, assignment, etc.
- Advice: prefer `nullptr` in C++11 code, but `NULL` will also be around for a long, long time.



# Dynamically allocated arrays

To dynamically allocate an array

- use `“type *name = new type[size];”`

To dynamically deallocate an array

- use `“delete[] name;”`
- it is an error to use `“delete name;”` on an array
  - ▶ the compiler probably won't catch this, though!!!
  - ▶ it can't tell if it was allocated with `“new type[size];”` or `“new type;”`

*see arrays.cc*



# malloc vs. new

	<b>malloc( )</b>	<b>new</b>
<b>what is it</b>	a function	an operator and keyword
<b>how often used in C</b>	often	never
<b>how often used in C++</b>	rarely	often
<b>allocates memory for</b>	anything	arrays, structs, objects, primitives
<b>returns</b>	a (void *) <i>(needs a cast)</i>	appropriate pointer type <i>(doesn't need a cast)</i>
<b>when out of memory</b>	returns NULL	throws an exception
<b>deallocating</b>	free	delete or delete[ ]



*see str/\**



# Overloading the “=” operator

Remember the rules we should follow?

- here's why; hugely subtle bug

```
Foo::Foo(int val) { Init(val); }
Foo::~~Foo() { delete my_ptr_; }

void Foo::Init(int val) { my_ptr_ = new int; *my_ptr_ = val; }

Foo &Foo::operator=(const Foo& rhs) {
    // bug...we forgot our "if (self == &rhs) { ... }" guard
    delete my_ptr_;
    Init(*(rhs.my_ptr_)); // might crash here (see below)
    return *this; // always return *this from =
}

void bar() {
    Foo a(10); // default constructor
    Foo b(20); // default constructor
    a = a; // crash above; dereference delete'd pointer!!
}
```



# Overloading the “=” operator

Remember the rules we should follow?

*This is yet another reason for disabling the assignment operator, when possible*



# Exercise 1

Modify your 3D Point class from lec10 exercise 1

- disable the copy constructor and assignment operator
- attempt to use copy & assign in code, and see what error the compiler generates
- write a CopyFrom( ) member function, and try using it instead



# 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



# Exercise 3

Write a C++ function that:

- uses `new` to dynamically allocate an array of strings
  - ▶ and uses `delete[]` to free it
- uses `new` to dynamically allocate an array of pointers to strings
  - ▶ and then iterates through the array to use `new` to allocate a string for each array entry and to assign to each array element a pointer to the associated allocated string
  - ▶ and then uses `delete` to delete each allocated string
  - ▶ and then uses `delete[]` to delete the string pointer array
  - ▶ (whew!)



See you on Monday!