CSE 333
Lecture 11 - constructor insanity

Steve Gribble
Department of Computer Science & Engineering
University of Washington
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[ ]
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 initializes all member variables to 0-equivalents (0, NULL, false, ..)
  - C++ does this iff your class has no const or reference data members
Example of synthesis

see SimplePoint.cc, SimplePoint.h
Constructors, continued

You might choose to define multiple constructors:

```cpp
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:

```cpp
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:

```cpp
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;
}
```
Copy constructors

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

- used to **create a new object** as a copy of an existing object

```cpp
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 a parameter to a call-by-value function

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

- you return an object from a function

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

- you initialize an object from another object of the same type

```cpp
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

```cpp
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

```cpp
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

```cpp
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
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

```c++
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

```cpp
class Point {
    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);
};
```

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

To dynamically allocate an array

- use \texttt{"type \*name = new type[size];"}

To dynamically deallocate an array

- use \texttt{"delete[] name;"}

- it is an error to use \texttt{"delete name;"} on an array
  \begin{itemize}
  \item the compiler probably won’t catch this, though!!
  \item it can’t tell if it was allocated with \texttt{"new type[size];"} or \texttt{"new type;"}
  \end{itemize}

\texttt{see arrays.cc}
## malloc vs. new

<table>
<thead>
<tr>
<th></th>
<th>malloc( )</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>what is it</strong></td>
<td>a function</td>
<td>an operator and keyword</td>
</tr>
<tr>
<td><strong>how often used in C</strong></td>
<td>often</td>
<td>never</td>
</tr>
<tr>
<td><strong>how often used in C++</strong></td>
<td>rarely</td>
<td>often</td>
</tr>
<tr>
<td><strong>allocates memory for</strong></td>
<td>anything</td>
<td>arrays, structs, objects, primitives</td>
</tr>
<tr>
<td><strong>returns</strong></td>
<td>a (void *)&lt;br&gt;<em>needs a cast</em></td>
<td>appropriate pointer type&lt;br&gt;<em>(doesn’t need a cast)</em></td>
</tr>
<tr>
<td><strong>when out of memory</strong></td>
<td>returns NULL</td>
<td>throws an exception</td>
</tr>
<tr>
<td><strong>deallocating</strong></td>
<td>free</td>
<td>delete or delete[]</td>
</tr>
</tbody>
</table>
Overloading the “=” operator

Remember the rules we should follow?
- here’s why; hugely subtle bug

```cpp
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
  a = a;    // crash above; dereference delete’d pointer!!
}
```
Overloading the “=” operator

Remember the rules we should follow?

- here’s why; hugely subtle bug

This is yet another reason for disabling the assignment operator, when possible!!
Exercise 1

Modify your 3D Point class from lec12 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-separate 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 Friday!