C++ Constructor Insanity
CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:
Matthew Arnold    Nonthakit Chaiwong    Jacob Cohen
Elizabeth Haker   Henry Hung          Chase Lee
Leo Liao          Tim Mandzyuk        Benjamin Shmidt
Guramrit Singh
Administrative Request

Please fill out a mid-quarter assessment survey
https://docs.google.com/forms/d/e/1FAIpQLSeW05jSv2UvDFeOtTmIM-h5oRTBn8XWYoahWvXsmJhmjveUWA/viewform
Lecture Outline

- Constructors / Destructors: The Issues
  - Constructors
  - Copy Constructors
  - Assignment
  - Embedded and Global Objects
Constructors / Destructors Demonstration

- A **constructor** is **always** run when an object is created
  - Just like in Java
- A **destructor** is **always** run when an object is destroyed
  - Not at all familiar from Java

- Constructors/destructors are the source of a lot of complexity in C++
- We’ll explain why
Example Class

class Example
{
    public:

        Example(std::string &name)
        {
            my_name = name;
            std::cout << "Created " << my_name << "\n";
        }

        Example(Example& other)
        {
            my_name = "copy of " + other.my_name;
            std::cout << "Created " << my_name << "\n";
        }

        ~Example()
        {
            std::cout << "Destroying " << my_name << std::endl;
        }

    private:
        std::string my_name;
};
Example Use of Class

```cpp
Example sub(Example arg)
{
    return arg;
}

int main(int argc, char *argv[])
{
    Example local("local");
    Example clone(local);

    Example *pExample = new Example("newed");

    {
        Example local_scope("local_scope");
    }

    Example arg("arg");
    sub(arg);

    delete pExample;

    Example *pNeverDeleted = new Example("never deleted");
    *pNeverDeleted = local;

    return EXIT_SUCCESS;
}
```
Example Use of Class

Example sub(Example arg)
{
  return arg;
}

int main(int argc, char *argv[])
{
  Example local("local");
  Example clone(local);

  Example *pExample = new Example("newed");

  Example local_scope("local_scope");
}

Example arg("arg");
sub(arg);

delete pExample;

Example *pNeverDeleted = new Example("never deleted");
pNeverDeleted = local;

return EXIT_SUCCESS;

C++ Constructor Insanity
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:46:15: error: cannot bind non-const lvalue reference of type 'std::string&' {aka 'std::__cxx11::basic_string<char>&'} to an rvalue of type 'std::string' {aka 'std::__cxx11::basic_string<char>'}
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:43:25: error: cannot bind non-const lvalue reference of type 'std::string&' {aka 'std::__cxx11::basic_string<char>&'} to an rvalue of type 'std::string' {aka 'std::__cxx11::basic_string<char>'}
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:9:24: note: initializing argument 1 of 'Example::Example(std::string&)'
example.cc:37:17: error: cannot bind non-const lvalue reference of type 'std::string&' {aka 'std::__cxx11::basic_string<char>&'} to an rvalue of type 'std::string' {aka 'std::__cxx11::basic_string<char>'}
example.cc: In function 'int main(int, char**)':
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/iostream:39,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/ostream:38,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/ios:42,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/bits/locale_classes.h:40,
  from example.cc:1:
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/iostream:39,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/ostream:38,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/ios:42,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/bits/locale_classes.h:40,
  from example.cc:1:
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/iostream:39,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/ios:42,
  from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/bits/ios_base.h:41,
Example Use of Class

```
Example sub(Example arg)
{
  return arg;
}

int main(int argc, char *argv[])
{
  Example local("local");
  Example clone(local);

  Example *pExample = new Example("newed");

  {
    Example local_scope("local_scope");
  }

  Example arg("arg");
  sub(arg);

  delete pExample;

  Example *pNeverDeleted = new Example("never deleted");
  *pNeverDeleted = local;

  return EXIT_SUCCESS;
}
```

```}
attu6> g++ -std=c++17 -g -Wall example.cc
example.cc: In function ‘int main(int, char**)’:
example.cc:37:17: error: cannot bind non-const lvalue reference of type ‘std::string&’ {aka ‘std::__cxx11::basic_string<char>&’} to an rvalue of type ‘std::string’ {aka ‘std::__cxx11::basic_string<char>’}
   Example local("local");
   ^~~~~~~
In file included from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/string:55,
   from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/bits/locale_classes.h:40,
   from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/bits/ios_base.h:41,
   from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/ios:42,
   from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/ostream:38,
   from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/iostream:39,
   from example.cc:1:
   /opt/rh/gcc-toolset-9/root/usr/include/c++/9/bits/basic_string.h:525:7: note: after user-defined conversion: ‘std::__cxx11::basic_string<_CharT, _Traits, _Alloc>::basic_string(const _CharT*, const _Alloc&) [with <template-parameter-2-1> = std::allocator<char>; _CharT = char; _Traits = std::char_traits<char>; _Alloc = std::allocator<char>]
      basic_string(const _CharT* __s, const _Alloc& __a = _Alloc())
      ^~~~~~~~~~~
example.cc:9:24: note: initializing argument 1 of ‘Example::Example(std::string&)’
9 |   Example(std::string &name)
   | ^~~~~~~~~~~~~~
example.cc:40:35: error: cannot bind non-const lvalue reference of type ‘std::string&’ {aka ‘std::__cxx11::basic_string<char>&’} to an rvalue of type ‘std::string’ {aka ‘std::__cxx11::basic_string<char>’}
   Example *pExample = new Example("newed");
   ^~~~~~~
In file included from /opt/rh/gcc-toolset-9/root/usr/include/c++/9/string:55,
   from /opt/rh/
Example Use of Class

Example sub(Example arg)
{
    return arg;
}

int main(int argc, char *argv[])
{
    Example local("local");
    Example clone(local);

    Example *pExample = new Example("newed");

    {
        Example local_scope("local_scope");
    }

    Example arg("arg");
    sub(arg);

    delete pExample;

    Example *pNeverDeleted = new Example("never deleted");
    *pNeverDeleted = local;

    return EXIT_SUCCESS;
}
Example Use of Class

class Example
{
    public:

    Example(std::string &name)
    {
        my_name = name;
        std::cout << "Created " << my_name << "\n";
    }

    Example(Example& other)
    {
        my_name = "copy of " + other.my_name;
        std::cout << "Created " << my_name << "\n";
    }

    ~Example()
    {
        std::cout << "Destroying " << my_name << std::endl;
    }

    private:
    std::string my_name;
};
Example Use of Class

class Example
{
    public:
        Example(const std::string &name)
        {
            my_name = name;
            std::cout << "Created " << my_name << "\n";
        }

        Example(Example& other)
        {
            my_name = "copy of " + other.my_name;
            std::cout << "Created " << my_name << "\n";
        }

        ~Example()
        {
            std::cout << "Destroying " << my_name << std::endl;
        }

    private:
        std::string my_name;
};

Example local("local");
Example *pExample = new Example("newed");

attu6> g++ -std=c++17 -g -Wall example.cc
 attu6>
Let’s Run It – Part 1

Example sub(Example arg)
{
    return arg;
}

int main(int argc, char *argv[])
{
    Example local("local");
    Example clone(local);

    Example *pExample = new Example("newed");

    Example local_scope("local_scope");

    Example arg("arg");
    sub(arg);

    delete pExample;

    Example *pNeverDeleted = new Example("never deleted");
    *pNeverDeleted = local;

    return EXIT_SUCCESS;
}
Let’s Run It – Part 2

Example sub(Example arg)
{
    return arg;
}

int main(int argc, char *argv[])
{
    Example local("local");
    Example clone(local);
    Example *pExample = new Example("newed");
    {
        Example local_scope("local_scope");
    }
    Example arg("arg");
    sub(arg);
    delete pExample;
    Example *pNeverDeleted = new Example("never deleted");
    *pNeverDeleted = local;
    return EXIT_SUCCESS;
}
Let’s Run It – Part 3

Example sub(Example arg)
{
    return arg;
}

int main(int argc, char *argv[])
{
    Example local("local");
    Example clone(local);

    Example *pExample = new Example("newed");

    {Example local_scope("local_scope");
    }

    Example arg("arg");
    sub(arg);
    delete pExample;

    Example *pNeverDeleted = new Example("never deleted");
    *pNeverDeleted = local;

    return EXIT_SUCCESS;
}
Let’s Run It – Part 4

Example sub(Example arg)
{
    return arg;
}

int main(int argc, char *argv[])
{
    Example local("local");
    Example clone(local);
    Example *pExample = new Example("newed");
    {
        Example local_scope("local_scope");
    }
    Example arg("arg");
    sub(arg);
    delete pExample;
    Example *pNeverDeleted = new Example("never deleted");
    *pNeverDeleted = local;
    return EXIT_SUCCESS;
}

attu6> ./a.out
Created 'local'
Created 'copy-of-local'
Created 'newed'
Created 'local_scope'
Destroying local_scope
Created 'arg'
Created 'copy-of-arg'
Created 'copy-of-copy-of-arg'
Destroying copy-of-copy-of-arg
Destroying copy-of-arg
Destroying newed
Created 'never-deleted'
Destroying arg
Destroying copy-of-local
Destroying local

attu6>

This is assignment. No construction. No destruction.
Construction / Destruction / Assignment

- **Construction**
  - Initial Values

- **Destruction**
  - Some Values

- **Assignment**
  - Memory Copy (sort of)
  - Object

*Nothing special about this.*
Destruction – Something Special

- **Construction**

- **Destruction**

* Might have to deallocate on destruction → destructor method: `~classname()`
Assignment – Something Special

- Assignment

Might have to deallocate on assignment → overloaded assignment operator method: `operator=()`
Copy Construction – Something Special

- Copy Construction

What happens when either object is destroyed?
Destructor frees allocated space, and remaining object has a dangling pointer

Need a Copy Constructor: `myclass(myclass& other)`
Lecture Outline

- Constructors / Destructors: The Issues
- Constructors
  - Copy Constructors
  - Assignment
- Embedded and Global Objects
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
- Declared with the class name as the method name and no return type:
  ```
  void Point(const int x, const int y);
  ```

- A constructor that “requires no arguments” is called the default constructor
  ```
  void Point();
  void Point(const int x=0, const int y=0);
  ```

Default arguments
Aside: Ambiguities and Surprises

```cpp
class Point {
public:
    Point() { x=-1; y=-1;}
    Point(const int x=0, const int y=0) {}

    int x;
    int y;
};

int main(int argc, char *argv[])
{
    Point p;
    Point q();
    Point r(1);

    std::cout << "p: " << p.x << " " << p.y << std::endl;
    std::cout << "q: " << q.x << " " << q.y << std::endl;
    std::cout << "r: " << r.x << " " << r.y << std::endl;
}
```

attu> g++ -std=c++17 -g -Wall defcon.cc

*What happens?*
Aside:AmbiguitiesandSurprises

classPoint{
  public:
    Point() {x=-1; y=-1;}
    Point(constintx=0, constinty=0) {}

    intx;
    inty;
};

intmain(intargc, char*argv[])
{
    Pointp;
    Pointq();
    Pointr(1);

    std::cout<<"p: "<<p.x<<" "<<p.y<<std::endl;
    std::cout<<"q: "<<q.x<<" "<<q.y<<std::endl;
    std::cout<<"r: "<<r.x<<" "<<r.y<<std::endl;
}

attu>g++-std=c++17-g-Walldefcon.cc

[attu2]~/cse333-21wi/lectures/09-constructor-code>lg
g++-std=c++17-g-WalldefaultCons.cc
defaultCons.cc:Infunction‘intmain(int,char**)’:
defaultCons.cc:15:9: error:call of overloaded ‘Point()’ is ambiguous
15 | Pointp;
  | ^
defaultCons.cc:6:3: note: candidate: ‘Point::Point(int, int)’
  6 | Point(constintx=0, constinty=0);
  | ^~~~~
defaultCons.cc:5:3: note: candidate: ‘Point::Point()’
  5 | Point() {x=-1; y=-1;}
  | ^~~~~
defaultCons.cc:20:27: error:request for member ‘x’ in ‘q’, which is
  of non-class type ‘Point()’
20 | std::cout<<"q: "<<q.x<<" "<<q.y<<std::endl;
   | ^
defaultCons.cc:20:41: error:request for member ‘y’ in ‘q’, which is
  of non-class type ‘Point()’
20 | std::cout<<"q: "<<q.x<<" "<<q.y<<std::endl;
   | ^
Aside: Ambiguities and No Surprise

class Point {
public:
    Point() { x=-1; y=-1;}
    Point(const int x=0, const int y=0) {} 
    int x;
    int y;
};

int main(int argc, char *argv[]) {
    Point p;
    Point q{};
    Point r(1);
    std::cout << "p: " << p.x << " " << p.y << std::endl;
    std::cout << "q: " << q.x << " " << q.y << std::endl;
    std::cout << "r: " << r.x << " " << r.y << std::endl;
}

attu> g++ -std=c++17 –g –Wall defcon.cc

defCon.cc: In function ‘int main(int, char**)’: 
defCon.cc:15:9: error: call of overloaded ‘Point()’ is ambiguous
     15 |   Point p;
         |         ^
   defCon.cc:6:3: note: candidate: ‘Point::Point(int, int)’
     6 |   Point(const int x=0, const int y=0);
        | ^^^^^
   defCon.cc:5:3: note: candidate: ‘Point::Point()’
     5 |   Point() { x=-1; y=-1;}
        | ^^^^^
   defCon.cc:16:11: error: call of overloaded ‘Point(<brace-enclosed initializer list>)’ is ambiguous
     16 |   Point q{};
        | ^
   defCon.cc:6:3: note: candidate: ‘Point::Point(int, int)’
     6 |   Point(const int x=0, const int y=0);
        | ^^^^^
   defCon.cc:5:3: note: candidate: ‘Point::Point()’
     5 |   Point() { x=-1; y=-1;}
        | ^^^^^
**Synthesized Default Constructor**

- If you don’t define any constructors, C++ will create one for you: the *synthesized default*
  - Takes no arguments
  - Calls the *default constructor on all object member variables*
  - Leaves primitive type (e.g., int) members *uninitialized*

- Synthesized default constructor *will fail to compile* if initialization of any member fails to compile
  - There are non-initialized const or reference data members
  - There is an embedded object for which no default constructor exists
    - *(And it hasn’t been explicitly initialized)*
### Synthesized Default Constructor / POD

```cpp
class SimplePoint {
public:
    // no constructors declared!
    int get_x() const { return x_; }       // inline member function
    int get_y() const { return y_; }       // inline member function

private:
    int x_;   // data member - no default initialization
    int y_;   // data member - no default initialization
};  // class SimplePoint

#include "SimplePoint.h"

int main(int argc, char** argv) {
    SimplePoint p;  // invokes synthesized default constructor
    std::cout << "(" << p.get_x() << ", " << p.get_y() << ")\n";
    return 0;
}
```

Output: (4198528, 0)
Synthesized Default Constructor / object

class SimplePoint {
  public:
    // no constructors declared!
    int get_x() const { return pt_.first; }
    int get_y() const { return pt_.second; }
  private:
    std::pair<int, int> pt_;  
};  // class SimplePoint

#include "SimplePoint.h"

int main(int argc, char** argv) {
  SimplePoint p;
  std::cout << "(" << p.get_x() << ", " << p.get_y() << ")\n";
  return 0;
}  

Output:  (0, 0)
Synthesized Default Constructor

- If there are any explicitly defined constructors, C++ will not synthesize additional ones

```cpp
#include "SimplePoint.h"

// defining a constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
    x_ = x;
    y_ = y;
}

void func() {
    SimplePoint x;    // error: no constructor matches
    SimplePoint y(1, 2); // ok: invokes the 2-int-arguments
                           // constructor
}
```
Multiple Constructors (overloading)

// (explicit) 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 sub() {
    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
}
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 are (nearly) identical:

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

```cpp
// x_ and y_ are instance variables of Points
Point::Point(const int x, const int y) : x_(x), y_(y)
{
}
```
Initialization vs. Construction

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 of default initialization (construction) followed by assignment”
Sometimes Initialization is Required

class Artificial
{
private:
    int  my_a;
    int& my_b;
    int const my_c;
public:
    Artificial(int a, int b, in c) : my_b(b), my_c(c)
    {
        my_a = a;
    }
};
Sometimes Initialization is Required

```cpp
class Artificial
{
private:
    int    my_a;
    int&   my_b;
    int const my_c;
public:
    Artificial(int a, int b, int c)
    {
        my_a = a;
        my_b = b;
        my_c = c;
    }
};
```

```bash
[attu6] > g++ -std=c++17 Artificial.cc
Artificial.cc: In constructor ‘Artificial::Artificial(int, int, int)’:
Artificial.cc:11:3: error: expected identifier before ‘{’ token
  11 |   {
  ^
Artificial.cc:10:3: error: uninitialized reference member in ‘int&’ [-fpermissive]
  10 |   Artificial(int a, int b, int c) :
  ^~~~~~~~~~
Artificial.cc:7:13: note: ‘int& Artificial::my_b’ should be initialized
    7 | int&      my_b;
    |     ^~~~
Artificial.cc:10:3: error: uninitialized const member in ‘const int’ [-fpermissive]
  10 |   Artificial(int a, int b, int c) :
  ^~~~~~~~~~
Artificial.cc:8:13: note: ‘const int Artificial::my_c’ should be initialized
    8 | int const my_c;
    |     ^~~~
Artificial.cc:14:10: error: assignment of read-only member ‘Artificial::my_c’
  14 |     my_c = c;
  ^~~~~^~~
```
Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- Destructors
- Embedded and Global Objects
Copy Constructors

- C++ has the notion of a **copy constructor** (cctor)
  - 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) { }
// copy constructor
Point::Point(const Point& other) {
    x_ = other.x_;  
    y_ = other.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)
Aside: Object Initialization

- What’s the difference between
  
  ```
  Point y(x);  // direct initialization
  
  and
  
  Point y = x;  // copy initialization
  ```
  
  Neither is assignment!
  Both are construction.

- Rules changed with c++17...

- The difference has to do with something we haven’t seen yet (but will)
  - Direct initialization is willing to use all type conversions available to coerce x into a type for which there is a Point constructor defined
  - Copy initialization will not use constructors or conversion operators that have been declared `explicit`
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
    z = y;        // assignment
    ```
  - You call a method that has a non-reference object parameter:
    ```
    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
    }
    ```
Synthesized Copy Constructor

- If you don’t explicitly provide a copy constructor and your code contains something that requires one, then
  - C++ will make one for you

- The synthesized copy constructor will copy values for instance variables of primitive type
  - E.g., int’s, pointers, ...

- For embedded objects, the synthesized copy constructor will use the copy constructor for embedded object’s type
  - And, yes, it will synthesize one if it needs to
Synthesized Copy Constructor Example

- MyClass instance_a(instance_b);

```
int x;
char * p_string;
Bar b;
```

```
x: 6
p_string: 0x4CE...7E0
Bar b = {...}
```

Equivalent to Bar b(instance_b.b);
Synthesized Copy Constructor

- The synthesized copy constructor does a *shallow copy*
  - Sometimes the right thing; sometimes the wrong thing

- If the objects contain pointers to dynamically allocated memory, the shallow copy results in two objects both pointing to the same memory
  - Makes it tricky to figure out which should free/delete the memory...
    - So tricky you probably get it wrong!

- Solution is usually to write the copy constructor yourself and do a deep copy
A Detail: Return Value Optimization

- The compiler sometimes uses a “return value optimization” to eliminate unnecessary copies
  - Sometimes a constructor isn’t invoked when you might expect it to be

```cpp
Point sub() {
    Point y; // default ctor
    return y; // copy ctor? optimized?
}
```

```cpp
Point y; // default ctor
y = sub(); // return value optimized?
```
Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- Destructors
- Embedded Global Objects
Two Things to Keep In Mind

- "=" is not always assignment
  - It is an assignment operator when it occurs inside an expression
    - x = y + 6;

- "=" indicates initialization when it occurs in a declaration

```cpp
Point w; // default ctor
Point x(w); // copy ctor
Point y = w; // initialization (copy ctor)
y = x; // assignment operator
```
Overloading the “=” Operator

- You can define a procedure (code) to be run when the assignment operator needs to be evaluated in an expression

  - But there are some rules you should follow:

```cpp
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 op=
}
```

```cpp
Point a;          // default constructor
a = b = c;       // works because = return *this
a = (b = c);     // equiv. to above (= is right-associative)
(a = b) = c;     // "works" but different
```
When To Overload Assignment

- When default assignment isn’t correct for your app
  - Often because the object contains pointers to dynamically allocated memory and so you need to do a deep copy on assignment

- “The Rule of Three”
  If you define any of the copy constructor, the assignment operator, or the destructor you very likely need to define all of them
  - Whatever problem made you define one is a problem for the other two as well
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) of your class
    - Does object assignment for embedded objects
- Sometimes the right thing; sometimes the wrong thing
Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- **Destructors**
- Embedded Global Objects
Destructors

- C++ has the notion of a destructor
  - Invoked automatically when a new’ed class instance is deleted or a stack allocated local goes out of scope (even due to exceptions or other causes!)
    - Note: Constructors are run for globals before any of your code runs. Destructors are run for globals after your code is done running.
  
- Place to put your cleanup code – free any dynamic storage or other resources owned by the object

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

- If you wrap a resource in an object declared as a local variable and you define a destructor for that object’s class that release the resource then you are guaranteed that your code will release the resource, even when errors are encountered and/or exceptions thrown.

- Example: `OpenFile in_file("fileToOpen.txt");` Constructor does an `fopen()` and saves `FILE*` value returned. `in_file` provides `FILE*` value whenever needed. Destructor does an `fclose()`.

- You can use this pattern in your code.
- The STL uses it in some important ways (notably for dynamic memory allocation).
RAII

- Common C++ idiom for managing dynamic resources

```cpp
class OpenFile
{
    public:
        OpenFile(const std::string fname) { pFile_ = fopen(...); }
    ...
    FILE* file() { return pFile_; }
    int close() { return fclose(pFile_); }
    ~OpenFile() { close(); }
    private:
        FILE* pFile_;
};

void sub()
{
    OpenFile in_file(“example.txt”);
    // no chance the file won’t be closed when we leave sub
    ...
}
```
Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- Destructors
- Embedded and Global Objects
## Embedded Objects

```cpp
class Embedee
{
public:
  Embedee() { std::cout << "Embedee constructor running\n"; }
  ~Embedee() { std::cout << "Embedee destructor running\n"; }
};

class Embeder
{
public:
  Embeder() { std::cout << "Embeder constructor running\n"; }
  ~Embeder() { std::cout << "Embeder destructor running\n"; }
private:
  Embedee myEe;
};

int main(int argc, char *argv[])
{
  Embeder e;
  return 0;
}
```

`attu> ./a.out`

*Embedee constructor running*

*Embeder constructor running*

*Embeder destructor running*

*Embedee destructor running*

Embedded objects are **constructed** before object constructor is run.

Embedded objects are **destroyed** after object destructor is run.
Global Objects

/* Changed Embeder and Embedee classes to accept a const char* name argument to the constructors and to label output with it */

Embeder global("global");

int main(int argc, char *argv[]) {
    std::cout << "In main" << std::endl;
    Embeder e("local");
    std::cout << "Leaving main" << std::endl;
    return 0;
}

attu> ./a.out
  global embedee constructor running
  global embeder constructor running
  In main
  local embedee constructor running
  local embeder constructor running
  Leaving main
  local embeder destructor running
  local embedee destructor running
  global embeder destructor running
  global embedee destructor running

Global objects are **constructed** before main() is entered

Global objects are **destroyed** after main() is exited