

C++ classes

```
// rectangle.h
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle {
public:
    double area() const {
        return width * height;
    }
private:
    double width, height;
};
#endif // RECTANGLE_H

// main.cpp
#include <iostream>
#include "rectangle.h"
int main() {
    Rectangle r;
    // What are the values of r.width and r.height right now?
    // I haven't set them to anything
    // Do they get set to reasonable defaults?
    return 0;
}
```

C++ classes: constructors

- How do `classes` get initialized?
- Who decides what values the fields should have initially?
- Often, **you** want to decide how fields should be initialized, and you do this by writing a **constructor** member function
- Java and Python also have constructors
- **Default constructor** for a class is a member function that C++ calls when you declare a new variable of that class without any initialization

```
int main() {  
    Rectangle r;  
    // behind the scenes, the default constructor is called  
    ...  
}
```

C++ classes: constructors

- A **constructor** is a member function you can define yourself
- If you define it, it should be **public**
- The function name must match the class name exactly
- Called a **default** constructor if it takes no arguments

```
class Rectangle {  
public:  
    // default constructor for Rectangle  
    Rectangle() { ... }  
    ...  
}
```

C++ classes: constructors

- Either you provide at least one constructor or the compiler generates a default one for you
- For `Rectangle` class we saw last time, the compiler generated one for us
- What does a compiler-generated default constructor do? For each member field,
 - If it is a **built-in type** (`int`, `doubles`,...), it isn't initialized (so it has a **garbage value**)
 - If it is of a **class type**, the **default constructor** for that class type is called

C++ classes: constructors

We've been using default constructors behind the scenes. For example:

```
// invokes string's default constructor  
// initializes word to be empty string  
std::string word;
```

```
// invokes vector's default constructor  
// initializes v to be empty vector  
std::vector<int> v;
```

C++ classes: constructors

A constructor is called implicitly when a new object is declared or explicitly when one is created using `new`.

```
int main() {  
    // calls default constructor for r  
    Rectangle r;  
  
    // calls default constructor for *rp  
    Rectangle *rp = new Rectangle();  
}
```

C++ classes: constructors

If we create our own constructor (default or otherwise), the compiler won't generate any constructor for us.

```
class Rectangle {  
public:  
    // Here we define our own "default constructor,"  
    // to initialize values to zero  
    // (because we don't want garbage)  
    Rectangle() : width(0.0), height(0.0) { }  
    ...  
private:  
    double width, height;  
};
```

C++ classes: constructor initializer list

```
class Rectangle {  
public:  
    // Here we define our own "default constructor,"  
    // to initialize values to zero  
    Rectangle() : width(0.0), height(0.0) { }  
    //                ~~~~~ ~~~~~  
    //                Initializes dimensions by setting  
    //                them equal to specified values.  
    //                If these were objects themselves,  
    //                we could've called THEIR constructors  
    //                e.g. list() where list is a vector<int>  
    ...  
private:  
    double width, height;  
};
```


C++ classes: constructor initializer list

Compare these default constructors:

// constructor1.h

```
1  class IntAndString1 {
2  public:
3      IntAndString1() {
4          i = 7;
5          s = "hello";
6      }
7
8      int i;
9      std::string s;
10 };
11
12 class IntAndString2 {
13 public:
14     IntAndString2() : i(7), s("hello") { }
15     //
16     //          "initializer list"
17
18     int i;
19     std::string s;
20 };
```

C++ classes: constructor initializer list

```
// constructor1.cpp

1  #include <iostream>
2  #include "constructor1.h"
3
4  int main() {
5      IntAndString1 is1;
6      IntAndString2 is2;
7      std::cout << "is1.i=" << is1.i << ", is1.s=" << is1.s << std::endl;
8      std::cout << "is2.i=" << is2.i << ", is2.s=" << is2.s << std::endl;
9      return 0;
10 }
```

\$ g++ -o constructor1 constructor1.cpp -std=c++11 -pedantic -Wall -Wextra

\$./constructor1

is1.i=7, is1.s=hello
is2.i=7, is2.s=hello

C++ classes: constructor initializer list

The results are the same. Which one is better?

- The "initializer list" is usually the better choice:
 - works as expected, even for reference variables
 - can use default and non-default constructors to initialize fields

// this is the "initializer list" style

```
IntAndString() : i(7), s("hello") { }
```

- Neither Java or Python have initializer list syntax!
(<https://stackoverflow.com/questions/7154654>)

C++ classes: constructor initializer list

Why is the "initializer list" usually the better choice?

```
// this is the "initializer list" style  
IntAndString() : i(7), s("hello") { }
```

```
// this is the other option  
IntAndString() {  
    i = 7;  
    s = "hello";  
}
```

It has to do with how 's' is initialized.

C++ classes: constructor initializer list

- With initializer list, `string s` is initialized by calling appropriate non-default constructor
 - We can call whatever non-default constructor we want
- Without initializer list, `'string s'` is first initialized with default constructor, then later set using `'s = "hello"'`, wastefully

Quiz!

What is the correct output?

```
1  class Foo {
2  public:
3      Foo() : i(5), s("hi") {
4          i = 10; s = "bye";
5      }
6      int getI() {return i;}
7      string getS() {return s;}
8  private:
9      int i; string s;
10 };
11
12 int main() {
13     Foo f;
14     cout << f.getI() << " " << f.getS() << endl;
15     return 0;
16 }
```

- A. 10 bye
- B. 5 hi
- C. 5 bye
- D. 10 hi
- E. does not compile/work and/or undefined behavior

Quiz - answers

What is the correct output?

```
1  class Foo {
2  public:
3      Foo() : i(5), s("hi") {
4          i = 10; s = "bye";
5      }
6      int getI() {return i;}
7      string getS() {return s;}
8  private:
9      int i; string s;
10 };
11
12 int main() {
13     Foo f;
14     cout << f.getI() << " " << f.getS() << endl;
15     return 0;
16 }
```

At line 14:

| Symbols (Scope) | Values |
|------------------|--------|
| f(main).i | 10 |
| f(main).s | "bye" |
| (f.getI())(main) | 10 |
| (f.getS())(main) | "bye" |

C++ classes: recap

- `const` protect the object by appending to the end of the method header
- `private:` or `public:` scope of data and function members
- constructors can use `initializer list`
- class definition can/should be `split between .h and .cpp files`