601.220 Intermediate Programming

C++ default constructors

C++ classes

```cpp
// rectangle_ex1.cpp:
#include <iostream>
#include "rectangle3.h"

using std::cout;
using std::endl;

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

- 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

C++ classes: 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() {
    // behind the scenes, Rectangle's
    // default constructor is called
    Rectangle r;
    ...
}
```

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 built-in types (int, doubles,. . .), instance variables aren't initialized (so they have garbage values)
  - For instance variables of class types, 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

```
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;
};
```

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

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: constructors

Compare these default constructors:

```
// defCtor1.h:
class IntAndString1 {
public:
    IntAndString1() {
        i = 7;
        s = "hello";
    }

    int i;
    std::string s;
};

class IntAndString2 {
public:
    IntAndString2() : i(7), s("hello") { }
    //                ^^^^^^^^^^^^^^^^
    //              "initializer list"

    int i;
    std::string s;
};
```

C++ classes: constructors

```
// defCtor1.cpp:
#include <iostream>
#include "defCtor1.h"

using std::cout;
using std::endl;

int main() {
    IntAndString1 is1;
    IntAndString2 is2;
    cout << "is1.i=" << is1.i << ", is1.s=" << is1.s << endl;
    cout << "is2.i=" << is2.i << ", is2.s=" << is2.s << endl;
    return 0;
}
```

```
$ g++ -c defCtor1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o defCtor1 defCtor1.o
$ ./defCtor1
is1.i=7, is1.s=hello
is2.i=7, is2.s=hello
```

C++ classes: constructors

- 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 nor Python have initializer list syntax

- stackoverflow.com/questions/7154654

C++ classes: 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: constructors

- 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

Zoom poll!

What is the correct output?

```
class Foo {
public:
  Foo() : i(5), s("hi") {
    i = 10; s = "bye";
  }
  int getI() {return i;}
  string getS() {return s;}
private:
  int i; string s;
};
int main() {
  Foo f;
  cout << f.getI() << " " << f.getS() << endl;
  return 0;
}
```

A. 10 bye

B. 5 hi

C. 5 bye

D. 10 hi

E. does not compile/work and/or undefined behavior

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