

601.220 Intermediate Programming

Alternate constructors, default arguments, and `this`

C++ classes: non-default constructors

Constructors can also take arguments, allowing caller to "customize" the object

```
// string has a non-default constructor taking a string  
// argument; initializes s1 to a copy of the argument  
string s1("hello");
```

```
// this looks similar, but it actually calls the *default*  
// constructor first, *then* does the assignment afterward  
string s2 = "world";
```

C++ classes: non-default constructors

```
// constructors1.cpp
1  #include <iostream>
2
3  class DefaultSeven {
4  public:
5      // default constructor commented out
6      // DefaultSeven() : i(7) { }
7
8      // non-default constructor
9      DefaultSeven(int initial) : i(initial) { }
10     // can still use initializer list ^^
11
12     int get_i() { return i; }
13 private:
14     int i;
15 };
16
17 int main() {
18     DefaultSeven s(10);
19     std::cout << "s.get_i() = " << s.get_i() << std::endl;
20     return 0;
21 }
```

C++ classes: non-default constructors

```
$ g++ -o constructors1 constructors1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ ./constructors1  
s.get_i() = 10
```

NOTE: Because we supplied an alternate (that is, non-default) constructor, there is no implicitly-created default constructor

C++ default arguments

- In C++ we can specify default values for function arguments in the definition
- We can then omit parameters when calling the function, but only sequentially from right to left (**can't skip middle params**)
- Default argument values create several functions in one
- This applies to functions in classes, as well as any other function
- Can be really useful for creating multiple constructors
 - If include default values for all arguments, this results in usage as a default (parameter-less) constructor

C++ default arguments

```
// constructors2.cpp

1  #include <iostream>
2
3  class DefaultSeven {
4  public:
5      // default value gives us 3 ways to call
6      DefaultSeven(int initial = 7, double val = .5) : i(initial), v(val) { }
7      int get_i() { return i; }
8      double get_v() { return v; }
9  private:
10     int i;
11     double v;
12 };
13
14 int main() {
15     DefaultSeven one(10, 20), two(2), tre;
16     std::cout << one.get_i() << " " << one.get_v() << std::endl;
17     std::cout << two.get_i() << " " << two.get_v() << std::endl;
18     std::cout << tre.get_i() << " " << tre.get_v() << std::endl;
19     return 0;
20 }
```

C++ default arguments

```
$ g++ -o constructors2 constructors2.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./constructors2
```

```
10 20
```

```
2 0.5
```

```
7 0.5
```

C++ classes: variable name conflicts

What happens if a constructor parameter has the **same name** as the instance variable it is supposed to initialize?

```
class MyThing {  
public:  
    MyThing(int init) : init(init) { }  
    // initializer list is ok ^^^^  
  
    int get_i() { return init; }  
private:  
    int init;  
};
```

Initializer list is a good choice - context makes it ok.

C++ classes: variable name conflicts

```
// constructors3.cpp
1  #include <iostream>
2
3  class MyThing {
4  public:
5      MyThing(int init) : init(init) { }
6      // initializer list is ok ^^^^
7
8      int get_i() { return init; }
9  private:
10     int init;
11 };
12
13 int main() {
14     MyThing s(10);
15     std::cout << "s.get_i() = " << s.get_i() << std::endl;
16     return 0;
17 }
$ g++ -o constructors3 constructors3.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./constructors3
s.get_i() = 10
```

C++ classes: `this` pointer

- What happens if another member function has a parameter with the same name as the instance variable it is supposed to initialize?
- Local variable (parameter) **hides** the instance variable shadowing. We could change the parameter name, but...
- `this` is **a pointer to the instance variable** and can be used to clarify
 - `this->init` always refers to the instance variable in our example
- We don't use `this` unless necessary in C++, unlike Java where it is a good style to always qualify instance members.

C++ classes: `this` pointer usage

```
// constructors4.cpp

1  #include <iostream>
2
3  class MyThing {
4  public:
5      MyThing(int init) : init(init) { }
6      // initializer list is ok ^^^^
7
8      int get_i() { return init; }
9
10     void set_i(int init) { this->init = init; }
11     // using this pointer ^^^^^ to clarify
12 private:
13     int init;
14 };
15
16 int main() {
17     MyThing s(10);
18     s.set_i(20);
19     std::cout << "s.get_i() = " << s.get_i() << std::endl;
20     return 0;
21 }

$ g++ -o constructors4 constructors4.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./constructors4

s.get_i() = 20
```

C++ classes: arrays of objects

What happens if we declare an array with our own class type?

Declaring an array of a class type makes **all** the objects, **calling a default constructor** to create each one. Thus, **requires** the class to have **a default constructor**!

// constructors5.cpp

```
1  #include <iostream>
2
3  class MyThing {
4  public:
5      // no default constructor
6      MyThing(int init) : init(init) { }
7      int get_i() { return init; }
8
9  private:
10     int init;
11 };
12
13 int main() {
14     MyThing s[10]; // tries to call default constructor
15     std::cout << "s[0].get_i() = " << s[0].get_i() << std::endl;
16     return 0;
17 }
```

C++ classes: arrays of objects

```
$ g++ -o constructors5 constructors5.cpp -std=c++11 -pedantic -Wall -Wextra

constructors5.cpp: In function int main():
constructors5.cpp:14:17: error: no matching function for call to MyThing::MyThing()
    MyThing s[10]; // tries to call default constructor
                  ^
constructors5.cpp:6:5: note: candidate: MyThing::MyThing(int)
    MyThing(int init) : init(init) { }
    ~~~~~
constructors5.cpp:6:5: note:   candidate expects 1 argument, 0 provided
constructors5.cpp:3:7: note: candidate: constexpr MyThing::MyThing(const MyThing&)
    class MyThing {
    ~~~~~
constructors5.cpp:3:7: note:   candidate expects 1 argument, 0 provided
constructors5.cpp:3:7: note: candidate: constexpr MyThing::MyThing(MyThing&&)
constructors5.cpp:3:7: note:   candidate expects 1 argument, 0 provided
```

Well... then what's the alternative if I don't really want to have a default constructor?

C++ classes: arrays of objects

Alternative 1: list-initialization

// constructors6.cpp

```
1  #include <iostream>
2
3  class MyThing {
4  public:
5      // no default constructor
6      MyThing(int init) : init(init) { }
7      int get_i() { return init; }
8
9  private:
10     int init;
11 };
12
13 int main() {
14     // use list-initialization to initialize the array
15     MyThing s[10] = {{0},{1},{2},{3},{4},{5},{6},{7},{8},{9}};
16     std::cout << "s[0].get_i() = " << s[0].get_i() << std::endl;
17     return 0;
18 }
```

```
$ g++ -o constructors6 constructors6.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./constructors6
```

```
s[0].get_i() = 0
```

C++ classes: arrays of objects

Alternative 2: use STL. e.g. `std::vector`

// constructors7.cpp

```
1  #include <iostream>
2  #include <vector>
3
4  class MyThing {
5  public:
6      // no default constructor
7      MyThing(int init) : init(init) { }
8      int get_i() { return init; }
9
10 private:
11     int init;
12 };
13
14 int main() {
15     // use empty vector and reserve 10 element size
16     std::vector<MyThing> s;
17     s.reserve(10);
18     // initialization using emplace_back
19     for (int i = 0; i < 10; ++i) s.emplace_back(i);
20     std::cout << "s[0].get_i() = " << s[0].get_i() << std::endl;
21     return 0;
22 }

```

\$ g++ -o constructors7 constructors7.cpp -std=c++11 -pedantic -Wall -Wextra

\$./constructors7

s[0].get_i() = 0