# 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

```cpp
// defaultSeven.cpp:
#include <iostream>

class DefaultSeven {
public:
    // default constructor commented out
    // DefaultSeven() : i(7) { }

    // non-default constructor
    DefaultSeven(int initial) : i(initial) { }
    // can still use initializer list ^^^

    int get_i() { return i; }
private:
    int i;
};

int main() {
    DefaultSeven s(10);
    std::cout << "s.get_i() = " << s.get_i() << std::endl;
    return 0;
}
```

# C++ classes: non-default constructors

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

```cpp
// defaultArgs.cpp:
#include <iostream>

class DefaultSeven {
public:
    // default value gives us 3 ways to call
    DefaultSeven(int initial = 7, double val = .5) : i(initial), v(val) { }
    int get_i() { return i; }
    double get_v() { return v; }
private:
    int i;
    double v;
};

int main() {
    DefaultSeven one(10, 20), two(2), tre;
    std::cout << one.get_i() << " " << one.get_v() << std::endl;
    std::cout << two.get_i() << " " << two.get_v() << std::endl;
    std::cout << tre.get_i() << " " << tre.get_v() << std::endl;
    return 0;
}
```

# C++ default arguments

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c defaultArgs.cpp
$ g++ -o defaultArgs defaultArgs.o
$ ./defaultArgs
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 good choice - context makes it ok.

# C++ classes: variable name conflicts

```cpp
// myThing.cpp:
#include <iostream>

class MyThing {
public:
    MyThing(int init) : init(init) { }
    // initializer list is ok ^^^^

    int get_i() { return init; }
private:
    int init;
};

int main() {
    MyThing s(10);
    std::cout << "s.get_i() = " << s.get_i() << std::endl;
    return 0;
}
```

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

# C++ classes: `this` pointer

What happens if another function has a parameter with the same name as the instance variable it is supposed to initialize?

Local variable (parameter) hides the instance variable. 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 good style to always qualify instance members.

# C++ classes: `this` pointer usage

```cpp
// myThing2.cpp:
#include <iostream>

class MyThing {
public:
    MyThing(int init) : init(init) { }
    // initializer list is ok ^^^^

    int get_i() { return init; }

    void set_i(int init) { this->init = init; }
    // using this pointer  ^^^^^^ to clarify
private:
    int init;
};

int main() {
    MyThing s(10);
    s.set_i(20);
    std::cout << "s.get_i() = " << s.get_i() << std::endl;
    return 0;
}
```

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c myThing2.cpp
$ g++ -o myThing2 myThing2.o
$ ./myThing2
s.get_i() = 20
```

# C++ arrays of objects

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

This requires the class to have a default constructor!

```
// myThing3.cpp:
#include <iostream>

class MyThing {
public:
    // no default constructor
    MyThing(int init) : init(init) { }
    int get_i() { return init; }

private:
    int init;
};

int main() {
    MyThing s[10];  // tries to call default constructor
    std::cout << "s[0].get_i() = " << s[0].get_i() << std::endl;
    return 0;
}
```

# C++ default constructor required

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c myThing3.cpp
myThing3.cpp: In function ‘int main()’:
myThing3.cpp:14:17: error: no matching function for call to ‘MyThing::M
   14 |     MyThing s[10];  // tries to call default constructor
      |                 ^
myThing3.cpp:6:5: note: candidate: ‘MyThing::MyThing(int)’
    6 |     MyThing(int init) : init(init) { }
      |     ^~~~~~~
myThing3.cpp:6:5: note:   candidate expects 1 argument, 0 provided
myThing3.cpp:3:7: note: candidate: ‘constexpr MyThing::MyThing(const My
    3 | class MyThing {
      |       ^~~~~~~
myThing3.cpp:3:7: note:   candidate expects 1 argument, 0 provided
myThing3.cpp:3:7: note: candidate: ‘constexpr MyThing::MyThing(MyThing&
myThing3.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

```
// myThing4.cpp:
#include <iostream>

class MyThing {
public:
    // no default constructor
    MyThing(int init) : init(init) { }
    int get_i() { return init; }

private:
    int init;
};

int main() {
    // use list-initialization to initialize the array
    MyThing s[10] = {{0},{1},{2},{3},{4},{5},{6},{7},{8},{9}};
    std::cout << "s[0].get_i() = " << s[0].get_i() << std::endl;
    return 0;
}
```

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c myThing4.cpp
```

# C++ classes: arrays of objects

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

```
// myThing4.cpp:
#include <iostream>
#include <vector>

class MyThing {
public:
    // no default constructor
    MyThing(int init) : init(init) { }
    int get_i() { return init; }

private:
    int init;
};

int main() {
    // use empty vector and reserve 10 elements
    std::vector<MyThing> s
    s.reserve(10);
    // initialization using emplace_back
    for (int i = 0; i < 10; ++i) s.emplace_back(i);
    std::cout << "s[0].get_i() = " << s[0].get_i() << std::endl;
    return 0;
}
```

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c myThing4.cpp
myThing4.cpp: In function ‘int main()’:
```