



# CS 11 C++ track: lecture 6

---

- Today:
  - Default function arguments
  - **friend** classes
  - Introduction to exception handling



# Default function arguments (1)

---

```
class Foo {  
private:  
    int x, y;  
public:  
    Foo() { x = y = 0; }  
    Foo(int x, int y) {  
        this->x = x;  
        this->y = y;  
    }  
    // ...  
};
```



# Default function arguments (2)

---

```
class Foo {  
private:  
    int x, y;  
public:  
    Foo(int x=0, int y=0) {  
        this->x = x;  
        this->y = y;  
    }  
    // ...  
};
```



## Default function arguments (3)

---

- Now we can do this:

```
Foo f(); // x = y = 0;  
Foo f(2); // x = 2; y = 0;  
Foo f(2,3); // x = 2; y = 3;
```

- Default args filled in from left to right



## Default function arguments (4)

---

- CANNOT do this:

```
Foo(int x=0; int y) { /* ... */ }
```

- All default args must be at *end* of argument list
- Otherwise ambiguous



# friend classes (1)

---

- Last time saw **friend** functions
- Way to allow special access privileges for a non-member function
- Can declare an entire class as a friend



## friend classes (2)

---

```
// in Vector.hh:  
  
class Matrix; // empty decl  
  
class Vector {  
  
    friend class Matrix;  
  
    // ...  
  
};
```



# friend classes (3)

---

```
// in Matrix.hh:  
  
class Vector; // empty decl  
  
class Matrix {  
  
    friend class Vector;  
  
    // ...  
  
};
```



# friend classes (4)

---

- Now `Vectors` and `Matrixes` can access each others' internal representations
- Useful for efficiently defining e.g. vector/matrix multiplication
- Empty decls avoid `#include` loop
- Use sparingly!



# Exception handling (1)

---

- Exceptions are better way to handle errors
- Error occurs → **throw** an exception
- Can **catch** an exception and deal with it
- Can create own exception objects *e.g.*

```
class ArrayOutOfBounds { /* ... */ };
```



# Exception handling (2)

---

```
// in Matrix.cc  
  
int Matrix::getelem(int row, int col) {  
    if ((row < 0) || (col < 0)) { // etc.  
        throw ArrayOutOfBounds();  
    }  
  
    // get and return the element  
}
```



# Exception handling (3)

---

```
// the try block
try {
    Matrix m(10, 10);
    cout << m.getelem(100, 100) << endl;
} catch (ArrayOutOfBounds) {
    cout << "Oops!" << endl;
}
```



# Exception handling (3)

---

```
// Can identify the exception object:
try {
    Matrix m(10, 10);
    cout << m.getelem(100, 100) << endl;
} catch (ArrayOutOfBounds a) {
    // a is now the exception object
    cout << "Oops!" << endl;
}
```



# Exception handling (4)

---

- Exception classes can save state:

```
class ArrayOutOfBounds {  
    private:  
        int row, col;  
    public:  
        ArrayOutOfBounds(int row, int col) {  
            this->row = row; this->col = col;  
        }  
        void display();  
};
```



# Exception handling (5)

---

```
void ArrayOutOfBounds::display() {  
    cout << "Invalid array access: "  
        << "row = " << row  
        << "col = " << col  
        << endl;  
}
```



# Exception handling (6)

---

```
try {  
    Matrix m(10, 10);  
    cout << m.getelem(100, 100) << endl;  
} catch (ArrayOutOfBounds a) {  
    a.display();  
}
```



# Exception handling (7)

---

- Can catch multiple exceptions in one try block:

```
try {  
    // ...  
} catch (Exception1 e1) {  
    // ...  
} catch (Exception2 e2) {  
    // ...  
} catch (...) {  
    // Catches ANY exception!  
    // Try not to use this.  
}
```



# Exception handling (8)

---

- Exceptions not caught propagate up the stack to caller of function, then its caller, etc. etc.
- all the way up to `main()` (top level)
- If not caught there, program aborts



# Next week

---

- Templates!
- By far the most interesting and tricky part of C++
- Answers the burning question:
  - how do I reuse the same code for **Matrix** of **ints**, **doubles**, etc.?