# Lecture 31: More C++

CS 62
Spring 2013
Kim Bruce & Kevin Coogan

# Assertions

- To write assertion: assert(bool_exp)
  - If fails, throws exception -- you can't control what printed.
  - Must #include <cassert>
- Can turn off if write
  - #define NDEBUG
  - must occur before #include <cassert>

# Running C++

- So far:

> g++ -o name_of_executable fst_source.cpp snd_source.cpp ...

- What is really happening:
  - Runs preprocessor on source:
    g++ -E some_source.cpp
  - Compiles new source:
    g++ -c fst_source.cpp
    g++ -c snd_source.cpp
    ...
  - Generates .o files w/ same name -- machine code
  - Linker links together all files
    g++ first_source.o second_source.o
  - Can specify output via -o option

# C++ Warnings!

- Many errors (e.g. io) revealed only if you ask for them!
- Common errors:
  - bool type: be aware that 0 can be used as false and any non-zero as true:
    - if (x = 0) {...} else {...} // always executes "else" part
  - C++ not check if initialize vbles before use (*& no default*)
  - C++ doesn't check to see if return value from function
  - C++ requires declaration before use:
    - declare prototype: void myFcn(int a);

# Initializers

- Originally wrote constructor for IntCell:

```
IntCell::IntCell(int initValue){
    value = initValue;
}
```

- However, better to use initializer:

```
IntCell::IntCell(int initValue)
                :value(initValue){   }
```

  - General syntax: comma-separated list of instance vbles and values after colon.
  - More efficient since only initialize once rather than run default and then update in constructor.

# More Initialization

- Can use constructors (*sort of*) like Java:
  - Point p(3,5)
    *equivalent to*
  - Point p = Point(3,5)

# Memory & Pointers in C++

- In Java
  - Primitive values live on run-time stack
    - Allocated and deallocated when go in and out of scope
  - Objects allocated in heap
    - Allocated with new, garbage collected when not accessible
    - "Pointer" to object stored on run-time stack (or in heap object)
    - Assignment gives "sharing", point to same spot in heap.
- In C++
  - Explicitly declare pointers
    - Allocated w/new, deallocated with "delete"

# Assignments

- In Java

```
Point p = new Point(1,1); // p is reference to point in heap
                          // the reference is an address or pointer
Point q = p;              // p, q store same location
q.setX(17)                // changes both q and p!
p = new Point(2,7) // p has diff address, q unchanged
```

# In C++

```
int x, y;       // x, y hold ints
x = 10;
int *p, *q;     // p, q pointers to ints
p = new int(3);// p now points to 3
*p = 47;        // p now points to 47
q = p;          // q pts to same int as p
*q = 23;        // both p and q pt to 23
delete p;       // locn pted to by precycled
*q = 17         // error - memory recycled
p = NULL;       // p points to nowhere
p = &x          // p points to address of x
// Dangerous if x on stack!
// Try to avoid this!
cout << *p << endl;  // prints val of x
```

# C++ likes puns

- Declare int * p *means p points to an int*
  - To refer to the int pointed at, write *p
- Arrays similar: int arr[3]; *array w/3 elts*
  - arr[0] is first element of the array