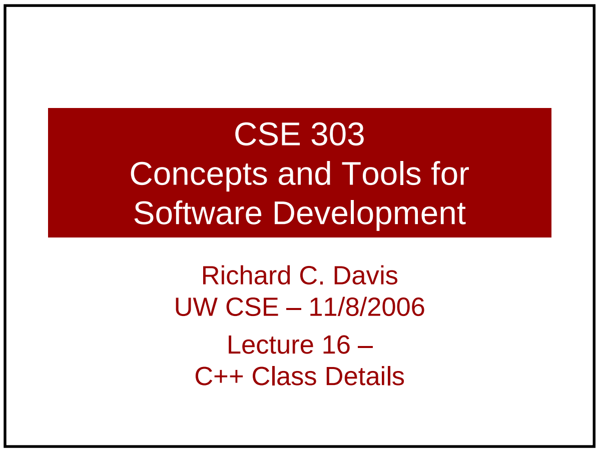
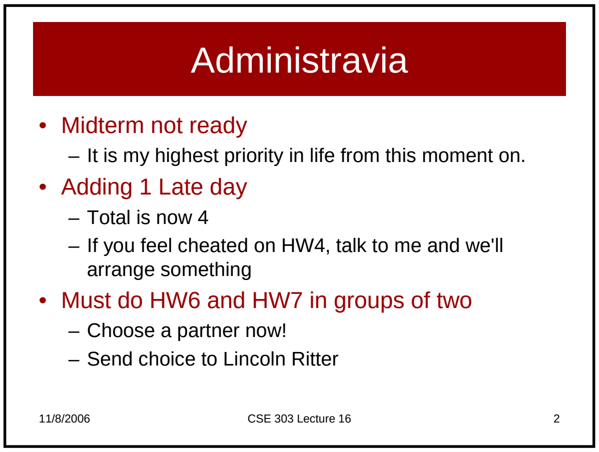


CSE 303 Concepts and Tools for Software Development

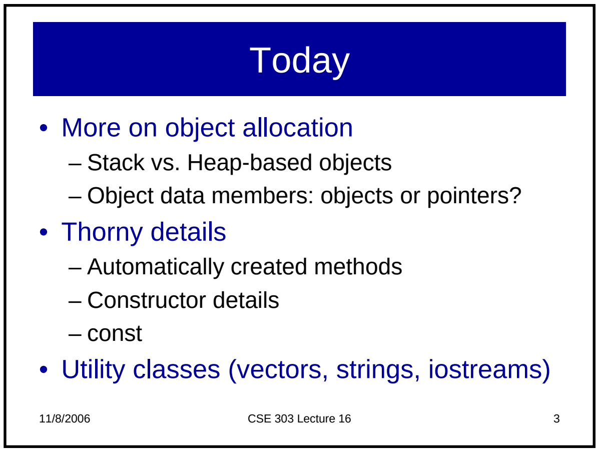
Richard C. Davis
UW CSE – 11/8/2006

Lecture 16 –
C++ Class Details



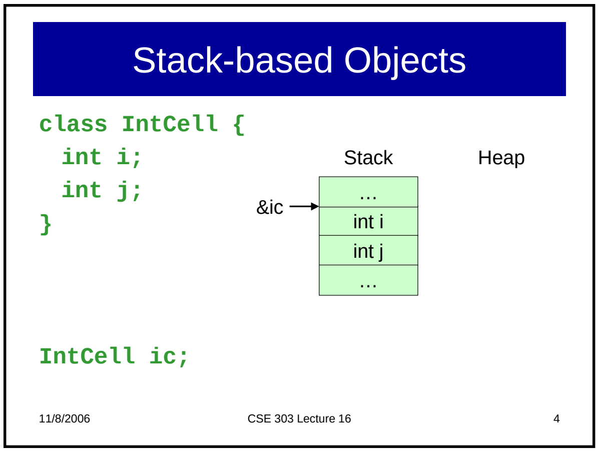
Administravia

- Midterm not ready
  - It is my highest priority in life from this moment on.
- Adding 1 Late day
  - Total is now 4
  - If you feel cheated on HW4, talk to me and we'll arrange something
- Must do HW6 and HW7 in groups of two
  - Choose a partner now!
  - Send choice to Lincoln Ritter



Today

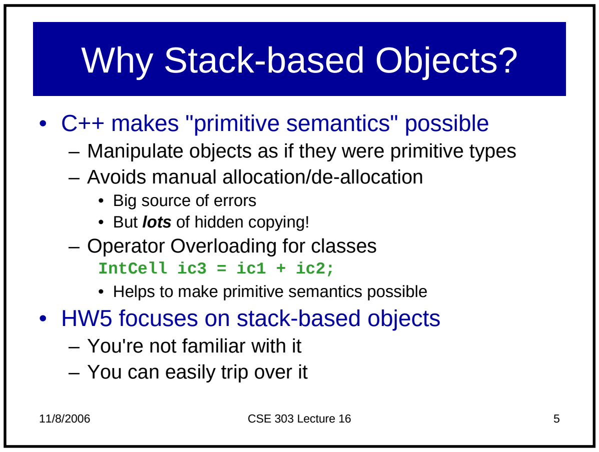
- More on object allocation
  - Stack vs. Heap-based objects
  - Object data members: objects or pointers?
- Thorny details
  - Automatically created methods
  - Constructor details
  - const
- Utility classes (vectors, strings, iostreams)



Stack-based Objects

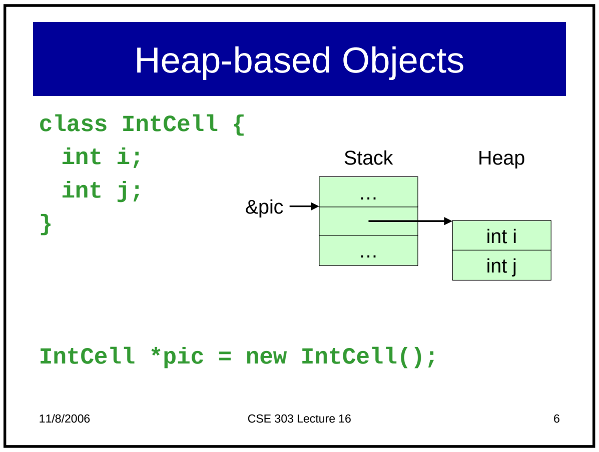
```
class IntCell {
  int i;
  int j;
}
```

```
IntCell ic;
```



Why Stack-based Objects?

- C++ makes "primitive semantics" possible
  - Manipulate objects as if they were primitive types
  - Avoids manual allocation/de-allocation
    - Big source of errors
    - But ***lots*** of hidden copying!
  - Operator Overloading for classes
    `IntCell ic3 = ic1 + ic2;`
    - Helps to make primitive semantics possible
- HW5 focuses on stack-based objects
  - You're not familiar with it
  - You can easily trip over it



Heap-based Objects

```
class IntCell {
  int i;
  int j;
}
```

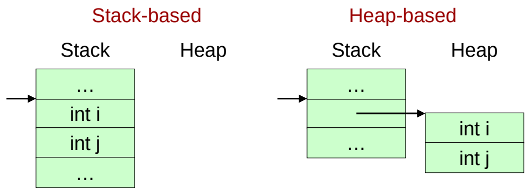
```
IntCell *pic = new IntCell();
```

Why Heap-based Objects?

- It's similar to Java
- But you have to manually free all memory

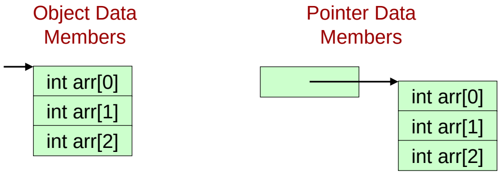
```
IntCell *pic = new IntCell();
delete pic;
```

Stack vs. Heap based Objects



- Which is better? You judge
  - Heap-based: Java-like semantics, manual memory mgmt
  - Stack-based: Primitive semantics, hidden copies

Data members: objects or pointers?



- Which do you choose?
  - Similar to stack-based vs. heap-based object question
  - You must be the judge of this as well
- Compare *ObjData.cpp* and *PtrData.cpp*

Thorny Details

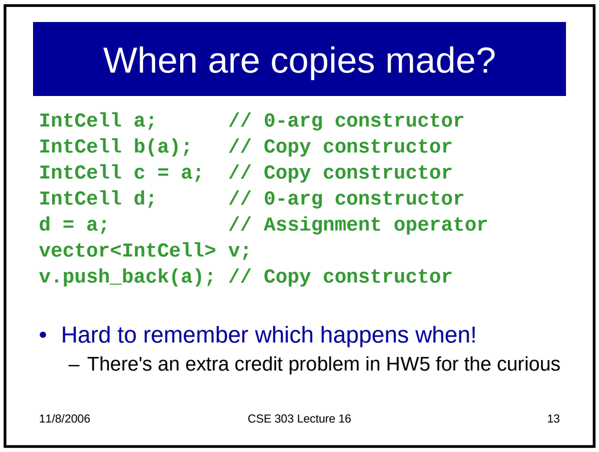
- Complicated C++ object details
  - Can cause unexpected behavior
  - We'll cover the high-level bits
  - HW5 helps you think about these

Thorny Details: Automatically Created Methods

- C++ Defines several hidden methods
  - The "Big Three"
    - Destructor
    - Copy Constructor
    - Assignment Operator
  - Default Constructor
- You can define all of these yourself
  - Doing so overrides default
- Warning: Be aware of these!

The "Big Three"

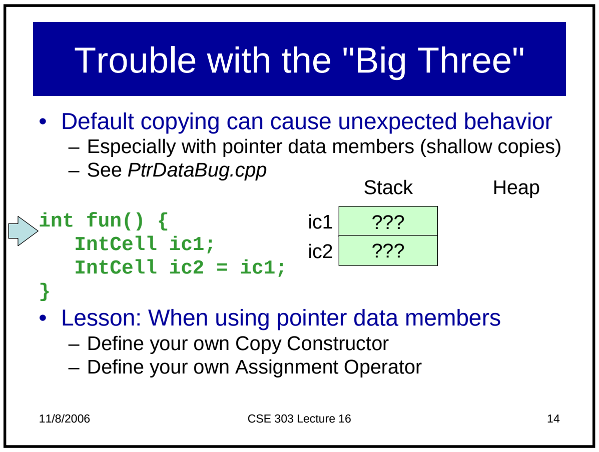
- Copy constructor
  `Class(const Class &rhs);`
  - Default calls copy constructor on all data members
  - copy constructor for primitive types copies bits
- Assignment operator
  `const Class &operator=( const Class &rhs);`
  - Default calls assignment operator on all data members
  - Assignment operator for primitive types copies bits
- Destructor
  `~Class();`
  - Does NOT call delete on any data members!



When are copies made?

```
IntCell a;       // 0-arg constructor
IntCell b(a);    // Copy constructor
IntCell c = a;   // Copy constructor
IntCell d;       // 0-arg constructor
d = a;           // Assignment operator
vector<IntCell> v;
v.push_back(a);  // Copy constructor
```

- Hard to remember which happens when!
  - There's an extra credit problem in HW5 for the curious

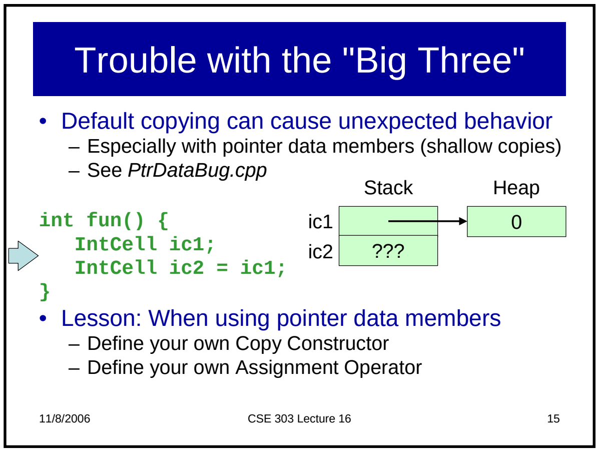


Trouble with the "Big Three"

- Default copying can cause unexpected behavior
  - Especially with pointer data members (shallow copies)
  - See *PtrDataBug.cpp*

```
int fun() {
   IntCell ic1;
   IntCell ic2 = ic1;
}
```

- Lesson: When using pointer data members
  - Define your own Copy Constructor
  - Define your own Assignment Operator

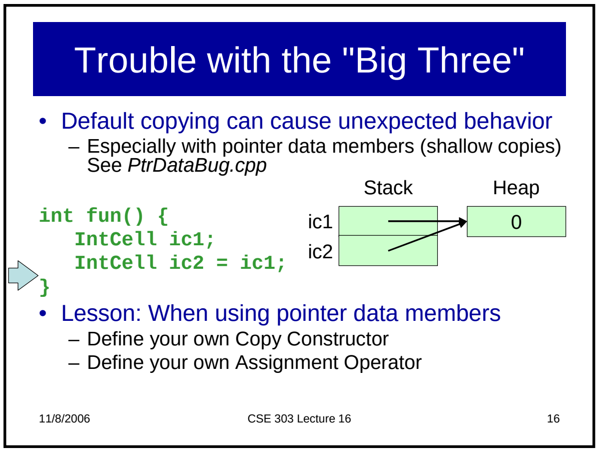


Trouble with the "Big Three"

- Default copying can cause unexpected behavior
  - Especially with pointer data members (shallow copies)
  - See *PtrDataBug.cpp*

```
int fun() {
   IntCell ic1;
   IntCell ic2 = ic1;
}
```

- Lesson: When using pointer data members
  - Define your own Copy Constructor
  - Define your own Assignment Operator

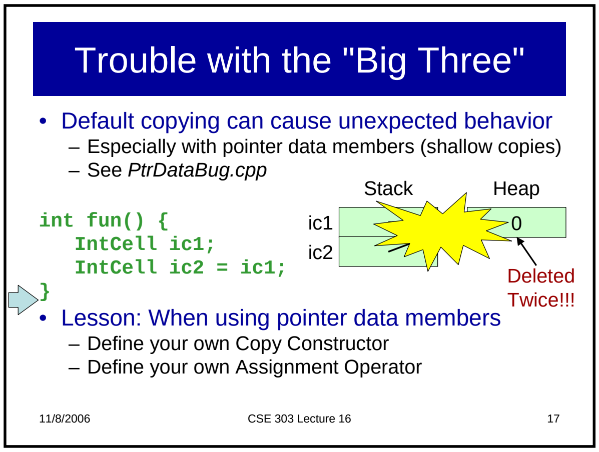


Trouble with the "Big Three"

- Default copying can cause unexpected behavior
  - Especially with pointer data members (shallow copies) See *PtrDataBug.cpp*

```
int fun() {
   IntCell ic1;
   IntCell ic2 = ic1;
}
```

- Lesson: When using pointer data members
  - Define your own Copy Constructor
  - Define your own Assignment Operator

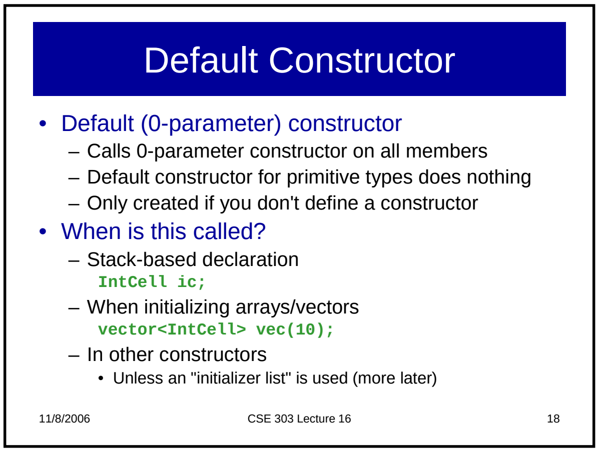


Trouble with the "Big Three"

- Default copying can cause unexpected behavior
  - Especially with pointer data members (shallow copies)
  - See *PtrDataBug.cpp*

```
int fun() {
   IntCell ic1;
   IntCell ic2 = ic1;
}
```

- Lesson: When using pointer data members
  - Define your own Copy Constructor
  - Define your own Assignment Operator



Default Constructor

- Default (0-parameter) constructor
  - Calls 0-parameter constructor on all members
  - Default constructor for primitive types does nothing
  - Only created if you don't define a constructor
- When is this called?
  - Stack-based declaration

    `IntCell ic;`
  - When initializing arrays/vectors

    `vector<IntCell> vec(10);`
  - In other constructors
    - Unless an "initializer list" is used (more later)

Thorny Details: Constructors

- Initializer Lists
  - `IntCell(int x) : i(x) {}`
  - Necessary if data member has no default constr.
- Implicit Type conversions

```
IntCell(int x);
IntCell ic = 5; // Implicit conversion!
```

  - prevented by using explicit

```
explicit IntCell(int i);
```

11/8/2006 CSE 303 Lecture 16 19

Thorny Details: const

- const objects can't be modified

```
const IntCell ic;
ic.setValue(5);  // Compile Error!
```

- How do we identify methods that don't modify?

```
void setValue(int i) const;
```

- const references

```
const Class &operator=(const Class &rhs);
```

  - Reference parameter that works on expressions/casts!
  - Returning references
    - Avoid this in general! It's easy to return dangling references!
    - It works in operators, because they return `*this`

11/8/2006 CSE 303 Lecture 16 20

Note on strings, vectors, and I/O

- All are very complex template classes
  - Nasty compile error messages!
- Why are we learning to use them?
  - They appear in examples and book (and HW5)
  - They make life easier once you know how to use them
- Reference Materials
  - Following pages give overview of everything you need
  - Links to (complex) docs on course web page
    - "Computing Resources" page (toward bottom)

11/8/2006 CSE 303 Lecture 16 21

Class details: vectors

- Initializing & Copying (it's stack-based!)

```
vector<t> vec1;         : Creates Empty Vector
vector<t> vec2(5);      : Calls 0-arg constr. 5 times
vector<t> vec3 = vec1;  : Calls Copy Constructor
```

- Accessing Elements & size

```
int i = vec2[0];
unsigned int j = vec2.size();
```

- Comparing Vectors

```
(vec1 == vec2)  : Compares size & all elements
                : using object's operator== (if defined)
```

11/8/2006 CSE 303 Lecture 16 22

Class details: vectors (cont'd)

- Manipulating Elements

```
vec1[0] = 5;          : Calls operator=
vec1.push_back(5);    : Resizes automatically
vec1.resize(10);      : Manual resize

int *p = &vec1[0];
*(p + 1) = 5;         : Pointer arithmetic works!!
```

- Compare C and C++ in *vectors.cpp*

11/8/2006 CSE 303 Lecture 16 23

Class details: strings

- Use strings like vectors, plus the following
- Initialize from any (char *)

```
string str1 = "Hello";
```

- Concatenating

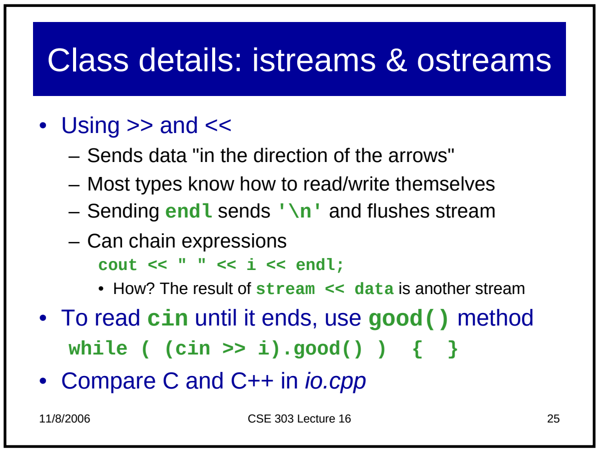
```
str3 = str1 + str2;
```

- Getting C-strings

```
char *cstr = str1.c_str();
```

  - Don't modify the data or free this pointer!
- Compare C and C++ in *strings.cpp*

11/8/2006 CSE 303 Lecture 16 24



Class details: istreams & ostreams

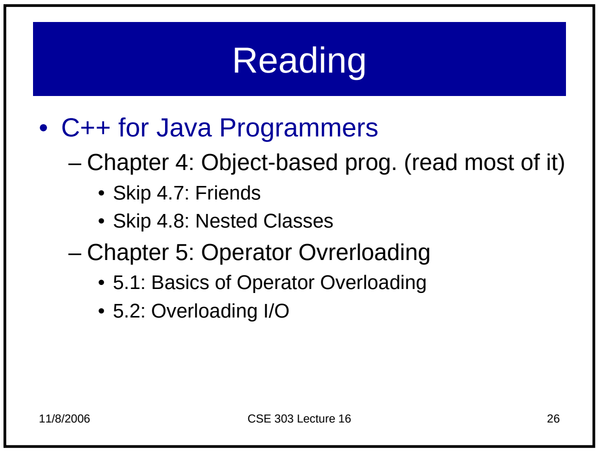
- Using >> and <<
  - Sends data "in the direction of the arrows"
  - Most types know how to read/write themselves
  - Sending `endl` sends `'\n'` and flushes stream
  - Can chain expressions

    ```
    cout << " " << i << endl;
    ```

    - How? The result of `stream << data` is another stream
- To read `cin` until it ends, use `good()` method

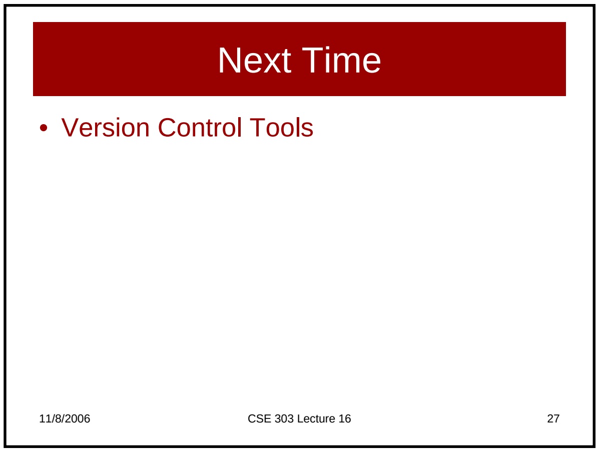
  ```
  while ( (cin >> i).good() )  {  }
  ```

- Compare C and C++ in *io.cpp*



Reading

- C++ for Java Programmers
  - Chapter 4: Object-based prog. (read most of it)
    - Skip 4.7: Friends
    - Skip 4.8: Nested Classes
  - Chapter 5: Operator Ovrerloading
    - 5.1: Basics of Operator Overloading
    - 5.2: Overloading I/O



Next Time

- Version Control Tools