the gamedesigninitiative at cornell university

Lecture 7

C++ Overview

the gamedesigninitiative at cornell university

Lecture 7

C++ Overview

So You Think You Know C++

- Most of you are experienced Java programmers
  - Both in 2110 and several upper-level courses
  - If you saw C++, was likely in a systems course
- Java was based on C++ syntax
  - Marketed as "C++ done right"
  - Similar with some important differences
- **This Lecture**: an overview of the differences
  - If you are a C++ expert, will be review

So You Think You Know C++

- Most of you are experienced Java programmers
  - Both in 2110 and several upper-level courses
  - If you saw C++ [illegible] course
- Java [illegible]
  - Ma[illegible]
  - Simi[illegible] some important differences
- **This Lecture**: an overview of the differences
  - If you are a C++ expert, will be review

All the sample code is online.
Download and **play with it**.

Comparing Hello World

Java

```
/* Comments are single or multiline
 */

// Everything must be in a class
public class HelloWorld {

  // Application needs a main method
  public static void main(String arg[]){

    System.out.println("Hello World");

  }

}
```

C++

```
/*Comments are single or multiline
 */

// Nothing is imported by default
#include <stdio.h>

// Application needs a main FUNCTION
int main(){

   printf("Hello World");
   printf("\n"); // Must add newline

   // Must return something
   return 0;
}
```

Comparing Hello World

Java

```
/* Comments are single or multiline
 */

// Everything must be in a class
public class HelloWorld {

  // Application needs a main method
  public static void main(String arg[]){

    System.out.println("Hello World");

  }

}
```

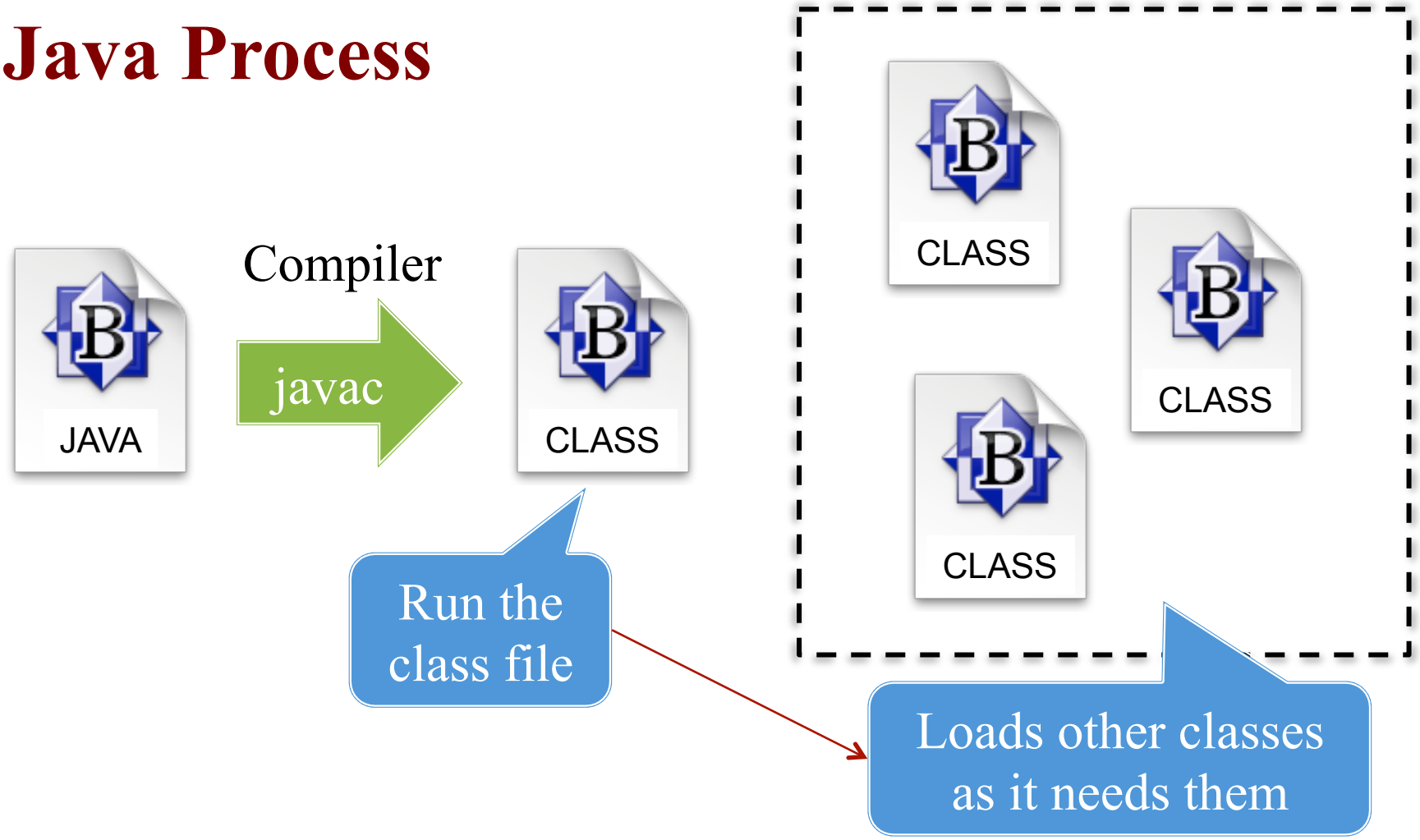
C++

```
/*Comments are single or multiline
 */

// Nothi
#includ

// Appli                    TION
int main(){

  printf("Hello World");
  printf("\n"); // Must add newline

  // Must return something
  return 0;
}
```

C-style console. Similar to CCLog, used by Cocos2d-x

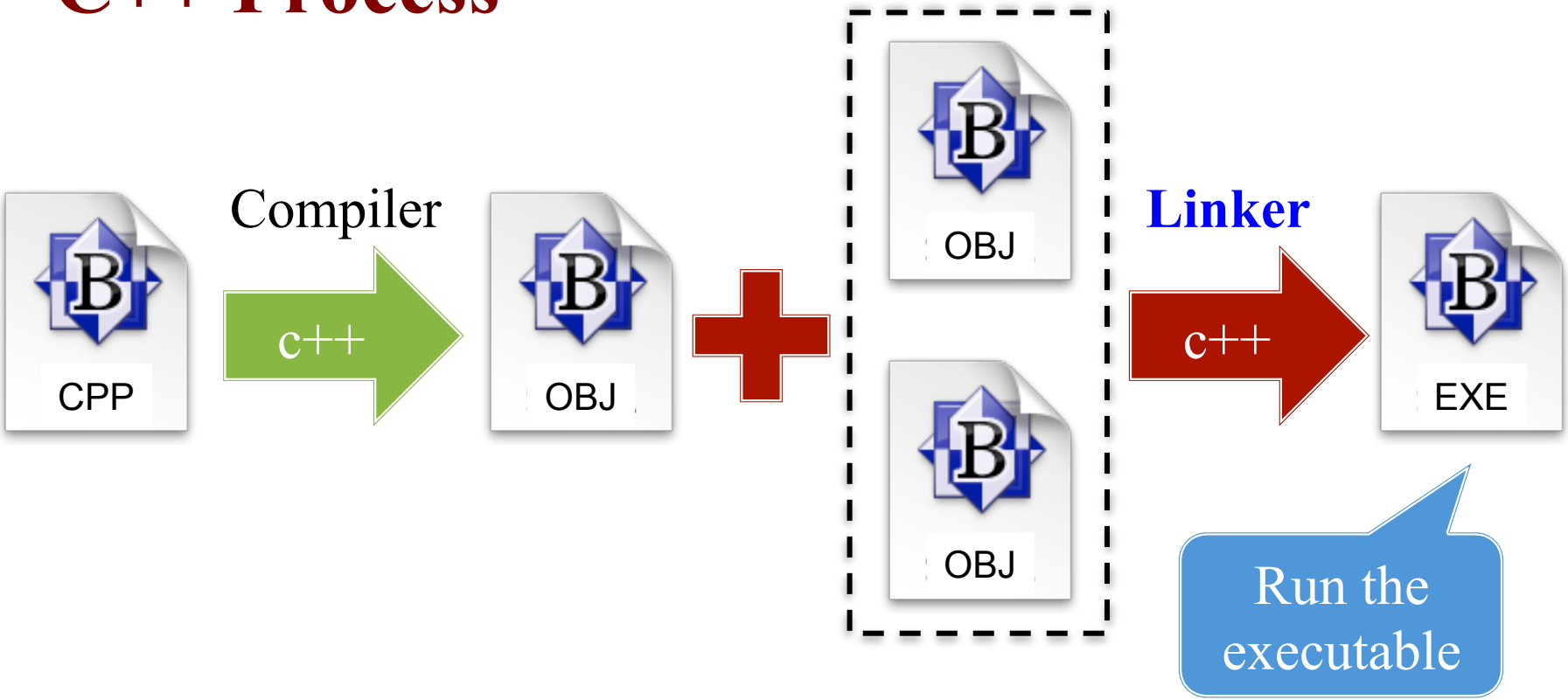
Biggest Difference: Compilation

Java Process

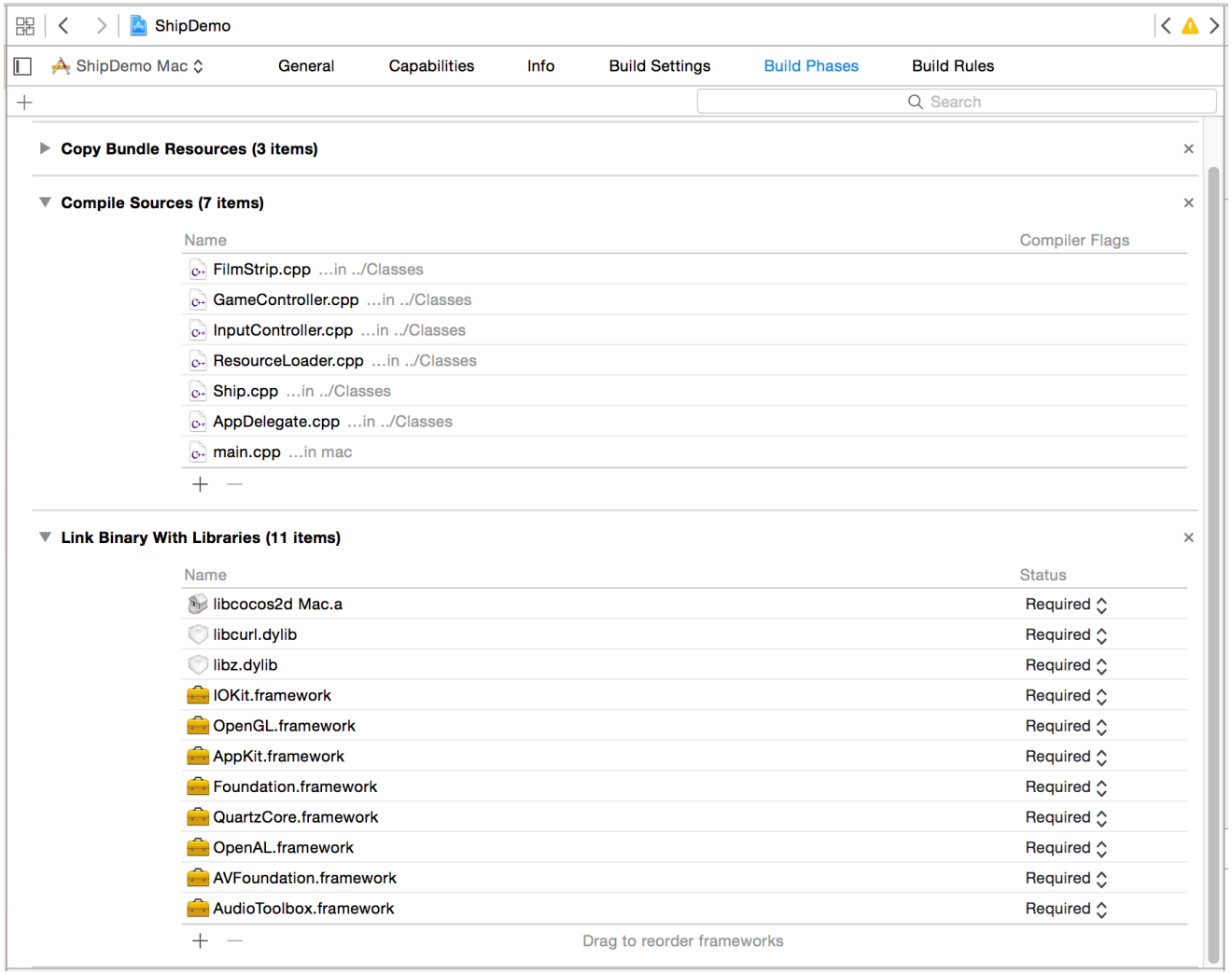


Biggest Difference: Compilation

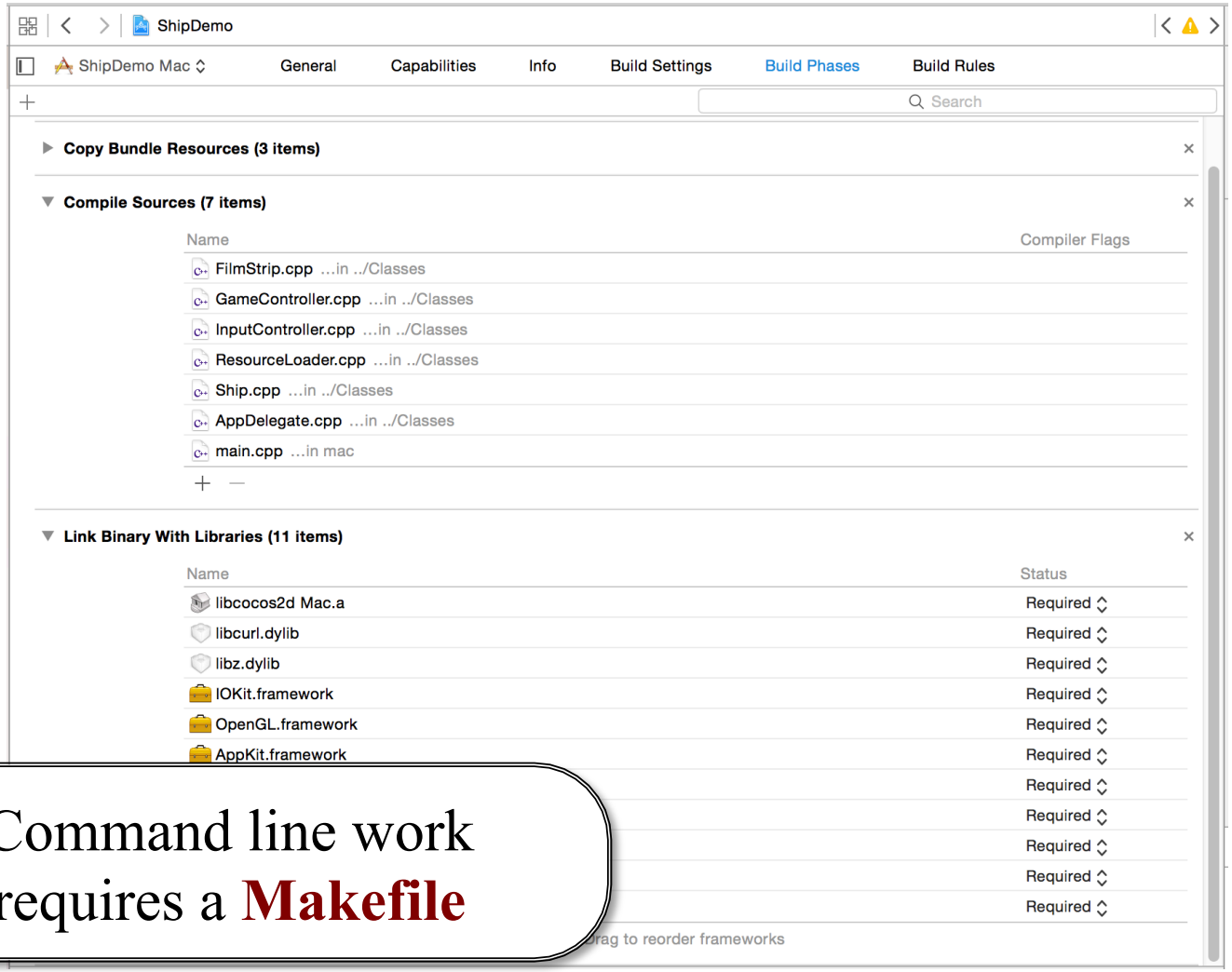
C++ Process



All Handled by the IDE



All Handled by the IDE



Command line work requires a **Makefile**

Separation Requires Header Files

- Need #include for libs
  - But linker adds the libs
  - So what are we including?
- **Function Prototypes**
  - Declaration without body
  - Like an interface in Java
- Prototypes go in .h files
  - Also includes types, classes
  - May have own #includes

```
/* stringfun.h
 * Recursive string funcs in CS 1110
 */

#ifndef_STRINGFUN_H_
#define_STRINGFUN_H_

#include <string>

/* True if word a palindrome */
bool isPalindrome(string word);

/* True if palindrome ignore case */
bool isLoosePalindrome(string word);

#endif
```

Separation Requires Header Files

- Need #include for libs
  - But linker adds the libs
  - So what are we including?
- **Function Prototypes**
  - Declaration without body
  - Like an interface in Java
- Prototypes go in .h files
  - Also includes types, classes
  - May have own #includes

```
/* stringfun.h
 * Recursive string funcs in CS 1110
 */

#ifndef _STRINGFUN_H_
#define _STRINGFUN_H_

#include <...>

/* Tr
bool i

/* Tr
bool isLoosePalindrome(string word);

#endif
```

Prevents inclusion more than once (which is an error)

Separation Requires Header Files

- Need #include for libs
  - But linker adds the libs
  - So what are we including?
- **Function Prototypes**
  - Declaration without body
  - Like an interface in Java
- Prototypes go in .h files
  - Also includes types, classes
  - May have own #includes

```
/* stringfun.h
 * Recursive string funcs in CS 1110
 */

#ifndef_STRINGFUN
#define_STRINGFUN_

#include <string>

/* True if word a palindrome */
bool isPalindrome(string word);

/* True if palindrome ignore case */
bool isLoosePalindrome(string word);

#endif
```

Type not built-in

Pointers vs References

Pointer

- Variable with a * modifier
- Stores a memory location
- Can modify as a parameter
- Must dereference to use
- Can allocate in heap

Reference

- Variable with a & modifier
- Refers to another variable
- Can modify as a parameter
- No need to dereference
- Cannot allocate in heap

Java's reference variables are a combination of the two

Pointers vs References

Pointer

- Variable with a * modifier
- Stores a
- Can mo
- Must de
- Can allocate in heap

Reference

- Variable with a & modifier
- Refers to another variable
- Can modify as a parameter
- No need to dereference
- Cannot allocate in heap

Safer!
Preferred if do not need heap

Java's reference variables are a combination of the two

When Do We Need the Heap?

- To **return** a non-primitive
  - Return value is on the stack
  - Copied to stack of caller
  - Cannot copy if size variable
- Important for arrays, objects
  - But objects can cheat

```
int* makearray(int size) {
    // Array on the stack
    int result[size];

    // Initialize contents
    for(int ii = 0; ii < size; ii++) {
        result[ii] = ii;
    }

    return result; // BAD!
}
```

0x7ed508	???
0x7ed528	4
0x7ed548	0
0x7ed568	1
0x7ed588	2
0x7ed5a8	3

Allocation and Deallocation

Not An Array

- Basic format:

```
type* var = new type(params);
...
delete var;
```

- Example:
  - `int* x = new int(4);`
  - `Point* p = new Point(1,2,3);`
- One you use the most

Arrays

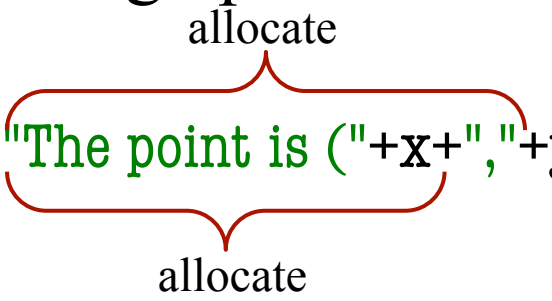
- Basic format:

```
type* var = new type[size];
...
delete[] var; // Different
```

- Example:
  - `int* array = new int[5];`
  - `Point* p = new Point[7];`
- Forget [] == memory leak

Strings are a Big Problem

- Java string operations allocate to the heap
  - `s = "The point is ("+x+","+y+")"`



- How do we manage these in C++?
  - For char*, we don't. Operation + is illegal.
  - For string, it is complicated. Later in lecture
- **Idea**: Functions to remove string memory worries
  - Formatters like printf/CCLog for direct output
  - Stream buffers to cut down on extra allocations

Managing Strings in C++

C-Style Formatters

- `printf(format,arg1,arg2,...)`
  - Substitute into % slots
  - Value after % indicates type
- Examples:
  - `printf("x = %d",3)`
  - `printf("String is %s","abc")`
- Primarily used for output
  - Logging/debug (CCLog)
  - Very efficient for output

C++ Stream Buffers

- `strm << value << value << ...`
  - Easy to chain arguments
  - But exact formatting tricky
- Example:
  - `cout << "x = " << 3 << endl`
  - `stringstream s << "x = " << 3`
- Great if you need to **return**
  - More efficient than + op
  - Can concatenate non-strings

Classes in C++

Declaration

- Like a Java interface
  - Fields, method prototypes
  - Put in the header file

```
class AClass {
private:  // All privates in group
   int field;
   void helper();

public:  // All publics in group
    AClass(int field);  // constructor
    ~AClass();          // destructor

}; // SEMICOLON!
```

Implementation

- Body of all of the methods
  - Preface method w/ class
  - Put in the cpp file

```
void AClass::helper() {
   field = field+1;
}
AClass::AClass(int field) {
   this->field = field;
}
AClass::~AClass() {
   // Topic of later lecture
}
```

Stack-Based vs. Heap Based

Stack-Based

- Object assigned to local var
  - Variable is NOT a pointer
  - Deleted when variable deleted
  - Methods/fields with period (.)
- Example:

```
void foo() {
   Point p(1,2,3); // constructor
   ...
   // Deleted automatically
}
```

Heap-Based

- Object assigned to pointer
  - Object variable is a pointer
  - Must be manually deleted
  - Methods/fields with arrow (->)
- Example:

```
void foo() {
   Point* p = new Point(1,2,3);
   ...
   delete p;
}
```

Stack-Based vs. Heap Based

Stack-Based

- Object assigned to local var
  - Variable is NOT a pointer
  - Deleted when variable deleted
  - Methods/fields wi[illegible]
- Example:

```
void foo() {
   Point p(1,2,3); // constructor
   ...
   // Deleted automatically
}
```

Also if pointer to stack-based

Heap-Based

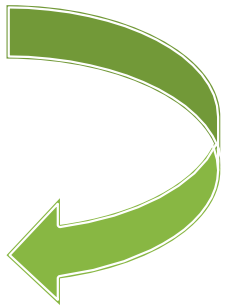
- Object assigned to pointer
  - Object variable is a pointer
  - Must be manually deleted
  - Methods/fields with arrow (->)
- Example:

```
void foo() {
   Point* p = new Point(1,2,3);
   ...
   delete p;
}
```

Returning a Stack-Based Object

- Do not need heap to return
  - Can move to calling stack
  - But this must *copy* object
- Need a special constructor
  - Called **copy constructor**
  - Takes *reference* to object
  - C++ calls automatically
- Is this a good thing?
  - Performance cost to copy
  - Cheaper than heap if small

```
Point foo_point(float x) {
    Point p(x, x);
    return p; // Not an error
}
```



Calls

```
Point::Point(const Point& p) {
    x = p.x;
    y = p.y;
    z = p.z;
}
```

Returning a Stack-Based Object

- Do not need heap to return
  - Can move to calling stack
  - But this must *copy* object
- Need a special constructor
  - Called
  - Take
  - C++
- Is this a good thing?
  - Performance cost to copy
  - Cheaper than heap if small

```
Point foo_point(float x) {
    Point p(x, x);
    return p; // Not an error
}
```

ls

```
Point::Point(const Point& p) {
    x = p.x;
    y = p.y;
    z = p.z;
}
```

What happens when you return a string

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

Caller cannot modify the object returned

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

Caller cannot modify the object returned

Method cannot modify the object passed

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

Caller cannot modify the object returned

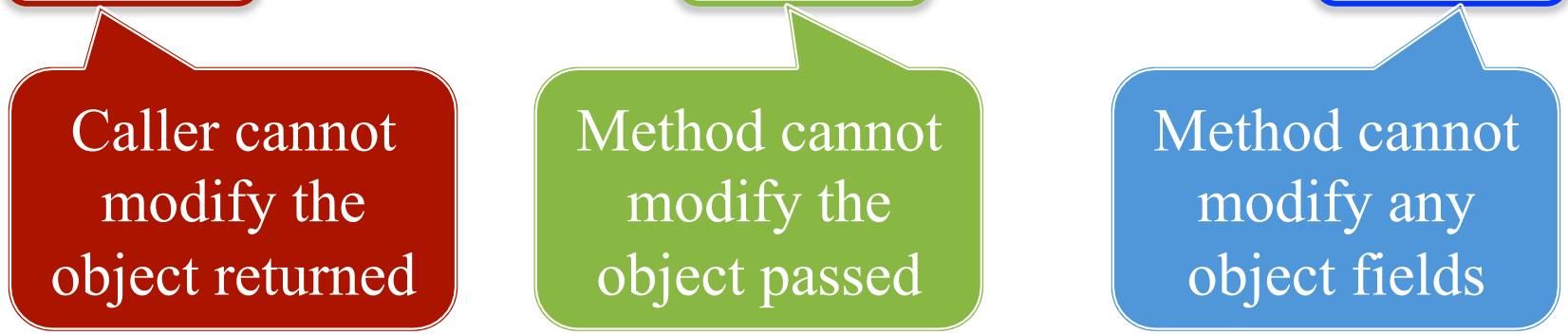
Method cannot modify the object passed

Method cannot modify any object fields

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```



- Believe it or not, these are not the only consts!
  - But these are generally the only ones to use
  - See online tutorials for more

Inlining Method Definitions

- Can implement in .h file
  - Define methods Java-style
  - Will **inline** the methods
- Less important these days
  - Good compilers inline
  - Function overhead is low
- Only two good applications
  - Getters and setters
  - Overloaded operators
  - Use this sparingly

```
class Point {
private:
    float x;
    float y;

public:

    Point(float x, float y, float z);

    float getX() const { return x; }

    void setX(float x) {
        this->x = x;
    }

    ...
};
```

Operator Overloading

- Change operator meaning
  - Great for math objects: +, *
  - But can do any symbol: ->
- Method w/ "operator" prefix
  - Object is always on the left
  - Other primitive or const &
- Right op w/ **friend** function
  - Function, not a method
  - Object explicit 2nd argument
  - Has full access to privates

```
Point& operator*=(float rhs) {
   x *= rhs; y *= rhs; z *= rhs;
   return *this;
}

Point operator*(const float &rhs) const {
   return (Point(*this)*=rhs);
}

friend Point operator* (float lhs,
                        const Point& p) {
   return p*lhs;
}
```

Subclasses

- Subclassing similar to Java
  - Inherits methods, fields
  - Protected limits to subclass
- Minor important issues
  - Header must import subclass
  - super() syntax very different
  - See tutorials for more details
- Weird C++ things to avoid
  - No **multiple inheritance**!
  - No **private subclasses**

```
class A {
public:
    float x;

    A(float x) { this->x = x; }
    ...
};

class B : public A {
public:
    float y;

    B(float x, float y) : A(x) {
        this->y = y;
    }
    ...
};
```

Subclasses

- Subclassing similar to Java
  - Inherits methods, fields
  - Protected limits to subclass
- Minor important issues
  - Header must import subclass
  - super() syntax very different
  - See tutorials for more details
- Weird C++ things to avoid
  - No **multiple inheritance**!
  - No **private subclasses**

```
class A {
public:
    float x;

    A(float
    ...
};

class B : public A {
public:
    float y;

    B(float x, float y) : A(x) {
        this->y = y;
    }
    ...
};
```

Weird things if you make it private

Like Java call to super

C++ and Polymorphism

- Polymorphism was a major topic in CS 2110
  - Variable is reference to interface or base class
  - Object itself is instance of a specific subclass
  - Calls to methods are those implementated in subclass
- **Example**:
  - `List<int> list = new LinkedList<int>();`
  - `list.add(10); // Uses LinkedList implementation`
- This is a major reason for using Java in CS 2110
  - C++ does not *quite* work this way

C++ and Polymorphism

- Cannot change stack object
  - Variable assignment copies
  - Will lose all info in subclass
- Only relevant for pointers
  - C++ uses static pointer type
  - Goes to method for type
- **What the hell?**
  - No methods in object data
  - Reduces memory lookup
  - But was it worth it?

```
class A {
public:
   int foo() {return 42;}
};

class B : public A {
public:
   int foo() {return 9000; }
};

B* bee = new B();

x = b->foo();    // x is 9000

A* aay = (A*)bee;

y = a->foo();    // y is 42!!!
```

Fixing C++ Polymorphism

- Purpose of virtual keyword
  - Add to method in base class
  - Says "will be overridden"
- Use optional in subclass
  - Needed if have subsubclass
  - Or if not further overridden
- Hard core C++ users hate
  - Causes a performance hit
  - Both look-up and storage
  - But not a big deal for you

```
class A {
public:
   virtual int foo() {return 42;}
};

class B : public A {
public:
   int foo() {return 9000; }
};

B* bee = new B();

x = b->foo();    // x is 9000

A* aay = (A*)bee;

y = a->foo();    // y is 9000
```

Is There Anything Else?

- C++ has a lot of features not covered lecture
  - **Templates** are the biggest topic skipped
  - **Preprocessor directives** and macros (like #ifndef)
  - **Namespaces** (e.g. packages)
- But you can survive this class without them
  - Need to use templates, but not write them
  - Using templates is close to a Java generic
- Or just look at some tutorials online

Summary

- C++ has a lot of similarities to Java
  - Java borrowed much of its syntax, but "cleaned it up"
- Memory in C++ is a lot trickier
  - Anything allocated with `new` must be deleted
  - C++ provides many alternatives to avoid use of `new`
- Classes in C++ have some important differences
  - Can be copied between stacks if written correctly
  - C++ supports operator overloading for math types
  - C++ needs special keywords to support polymorphism