Object-Oriented Programming

CSCI-UA 0470-001
Class 27
Instructor: Randy Shepherd

Final Exam

Format

- Here is some code, either..
  - Tell me what it does
  - Find a bug
  - Describe some characteristics of it in some notations
  - What does it print
  - …

Object-Oriented Design & Analysis

- Motivation for OOP
- Principles of OOP
- Phases: Analysis, Design & Implementation
- The Pillars of OOP
- Substitution Principle
- Single Responsibility Principle
- Composition
- Design Patterns
- Implementation & Definition Inheritance

Java

- Packages
- Enums & Type Safety
- Classes & Objects
- Generics
- java.lang.Object Hierarchy
- Arrays
- Interfaces
- Inheritance
- Overriding
- Overloading
- Garbage Collection

C++

- Namespaces
- Preprocessor & Directives
- References & Pointers
- Higher Order Functions
- Function Pointers
- Classes
- Class Inheritance
- Virtual Inheritance
- Multiple Inheritance
- Templates & Specializations
- Value & 'Reference' Semantics
- Operator Overloading
- Destructors
- Copy Constructors
- Rule of Three
- Memory Management

Translation Concepts

- Virtual Method Dispatch
- Vtables & Data Layouts
- Covariance
- Static vs Dynamic Typing
- Abstract Syntax Trees
- Visitor Pattern
- Scope
- Symbol Tables
- Overload Resolution
- Reference Counting
- Smart Pointers

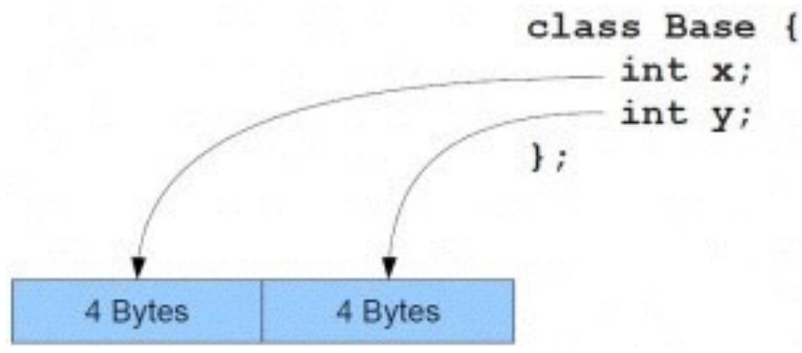
Review: Vtables & Data Layouts

Objects in Memory

- For any given object, how is the data organized in memory?
- How does the runtime find a particular property?
- How do these things work when dealing with inheritance hierarchies?

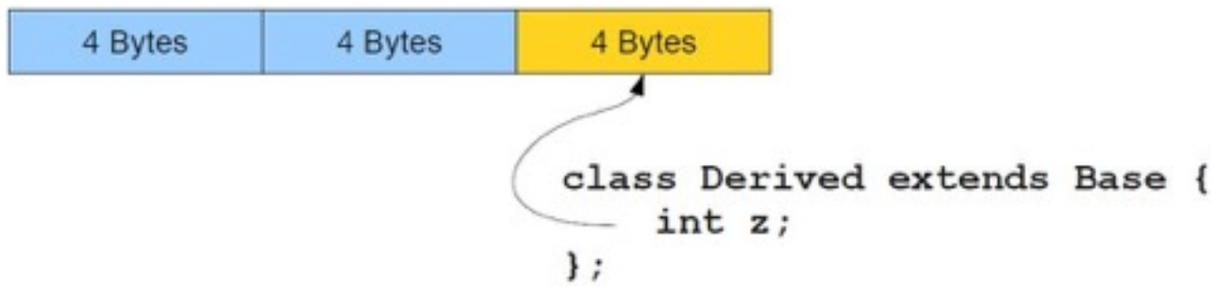
Object Data

- Data members of a class are laid out contiguously in memory for each instance.



- Can be accessed via an offset.

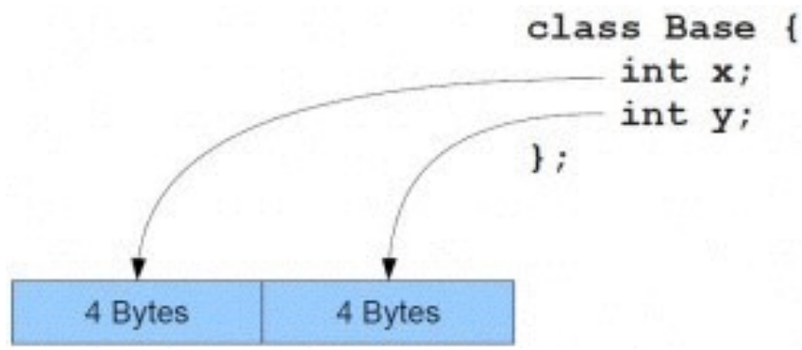
- Child objects have the same memory layout as parent objects with additional space for subclass members.

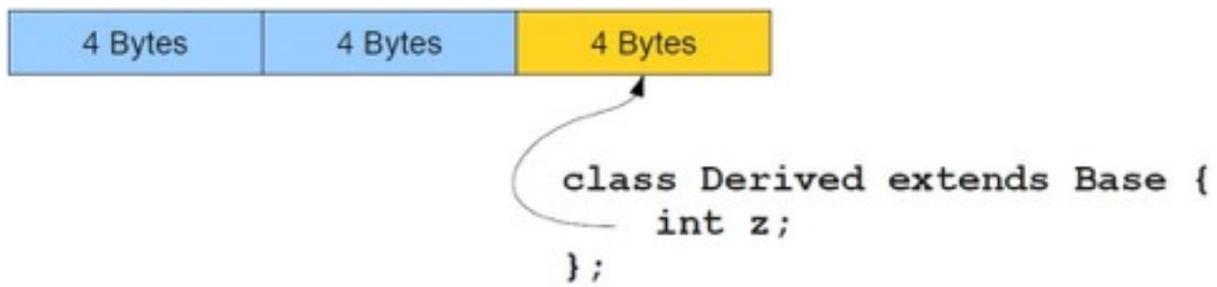


Object Data

- Objects of type *Derived* can be polymorphically operated on as type *Base*, since the offsets are the same.

- No need for the runtime to check the dynamic type of instances of *Base*.



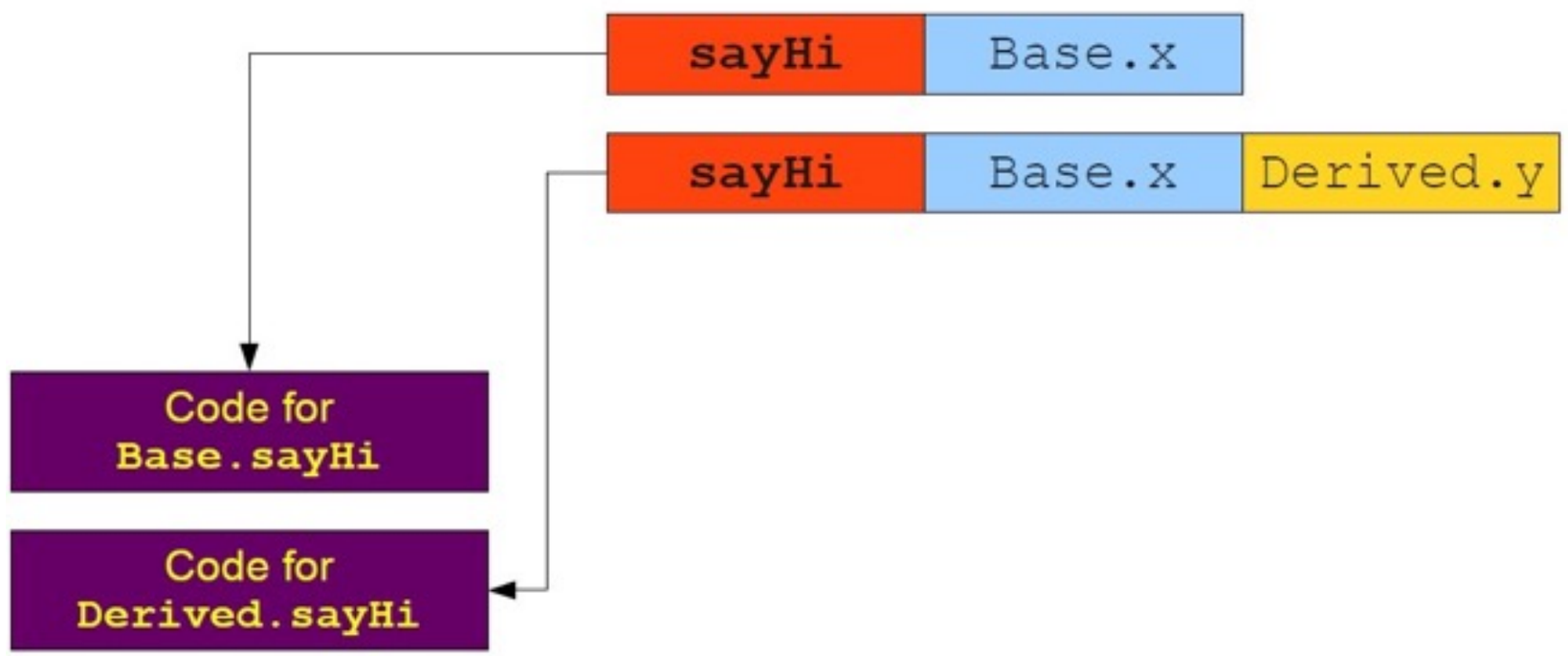


Object Methods

- So could we take this approach with methods?

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
}
```

```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived");
    }
}
```



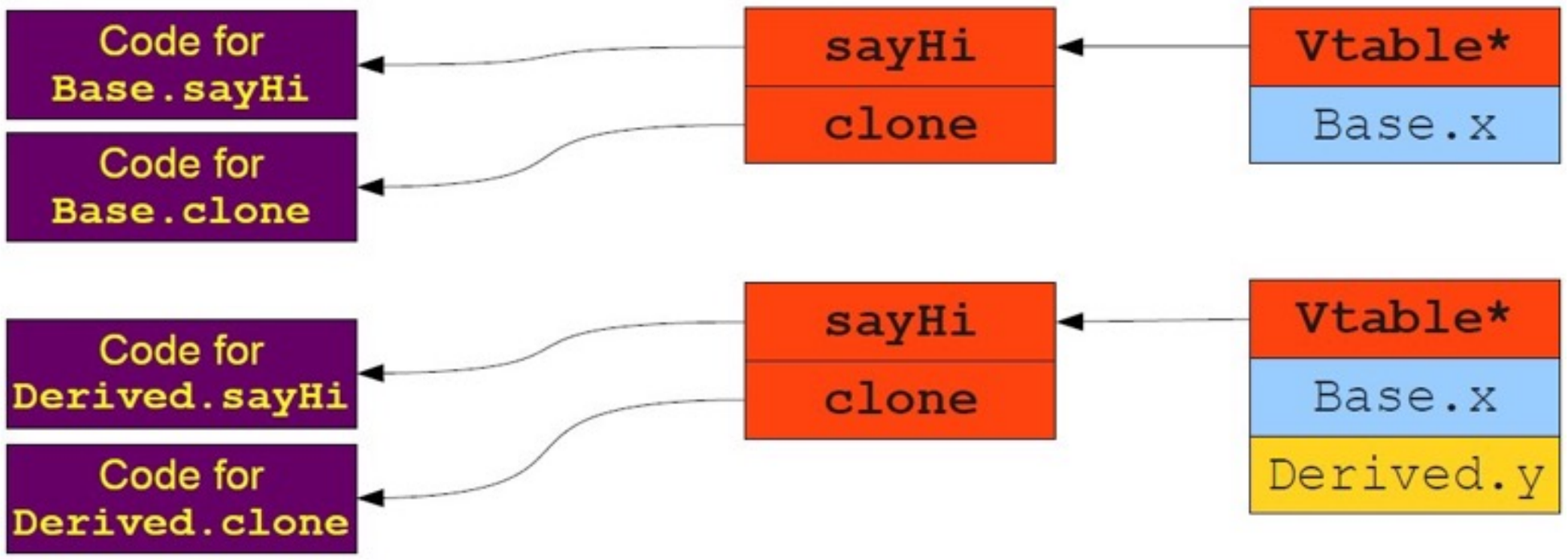
Per-Data Layout Method Pointers

- For every method added in a subclass the size of each instance grows the size of one pointer
- Object creation slower
- Memory consumption higher

A better approach…

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived");
    }
    Derived clone() {
        return new Derived;
    }
}
```



Virtual Table

- When a class defines an *instance* method, the compiler adds a hidden member variable

- That variable is a pointer is to the ‘vtable’, or ‘virtual member table’.

- At instance creation (at runtime) a pointer will be set to point to the right vtable. i.e. the vtable for the *dynamic type*.

- This is what enables *dynamic dispatch*

VTables & Data Layouts

- What are the data layouts of A & B?

```
1  class A {
2      int a = 0;
3      public int method() { return 12345;}
4      public String toString() {
5          return "A";
6      }
7  }
8
9  class B extends A {
10     int b = 0;
11     public String toString() { return "B"; }
12 }
```

VTables & Data Layouts

- What are the data layouts of A & B?

- A

  A.a

- B

  A.a
  B.b

```
1  class A {
2      int a = 0;
3      public int method() { return 12345;}
4      public String toString() {
5          return "A";
6      }
7  }
8
9  class B extends A {
10     int b = 0;
11     public String toString() { return "B"; }
12 }
```

VTables & Data Layouts

- What is the vtable layout of B?
- You may ignore the __vptr and __isa entries that we used in the translator.
- Recall that the vtable of Object contains the following entries:

```
int Object.hashCode(Object)
boolean Object.equals(Object, Object)
Class Object.getClass(Object)
String Object.toString(Object)
```

```
1  class A {
2      int a = 0;
3      public int method() { return 12345;}
4      public String toString() {
5          return "A";
6      }
7  }
8
9  class B extends A {
10     int b = 0;
11     public String toString() { return "B"; }
12 }
```

VTables & Data Layouts

- What is the vtable layout of B?

```
int Object.hashCode(B)
boolean Object.equals(B, Object)
Class Object.getClass(B)
String B.toString(B)
int32_t A.method(B)
```

- Note the
  typename classname.fieldname
  notation for data layout entries
  and the
  typename classname.methodname(typenames)
  notation for vtable entries.

```
1  class A {
2      int a = 0;
3      public int method() { return 12345;}
4      public String toString() {
5          return "A";
6      }
7  }
8
9  class B extends A {
10     int b = 0;
11     public String toString() { return "B"; }
12 }
```

Private & Static Methods

- So now that we understand the motivation and implementation of dynamic dispatch, we can easily understand what this code prints.
- What does this print?

```
1  class A {
2      public A() {
3          m1();
4          m2();
5          m3();
6      }
7      public void m1() { System.out.println("m1: A"); }
8
9      private void m2() { System.out.println("m2: A"); }
10
11     public static void m3() { System.out.println("m3: A"); }
12 }
13
14 class B extends A {
15     public B() { /* note that B() will implicitly call A() */}
16
17     public void m1() { System.out.println("m1: B"); }
18
19     private void m2() { System.out.println("m2: B"); }
20
21     public static void m3() { System.out.println("m3: B"); }
22 }
23
24 public class WhatIsPrinted {
25     public static void main(String[] args) {
26         A b = new B();
27     }
28 }
```

Private & Static Methods

- So now that we understand the motivation and implementation of dynamic dispatch, we can easily understand what this code prints.

- What does this print?

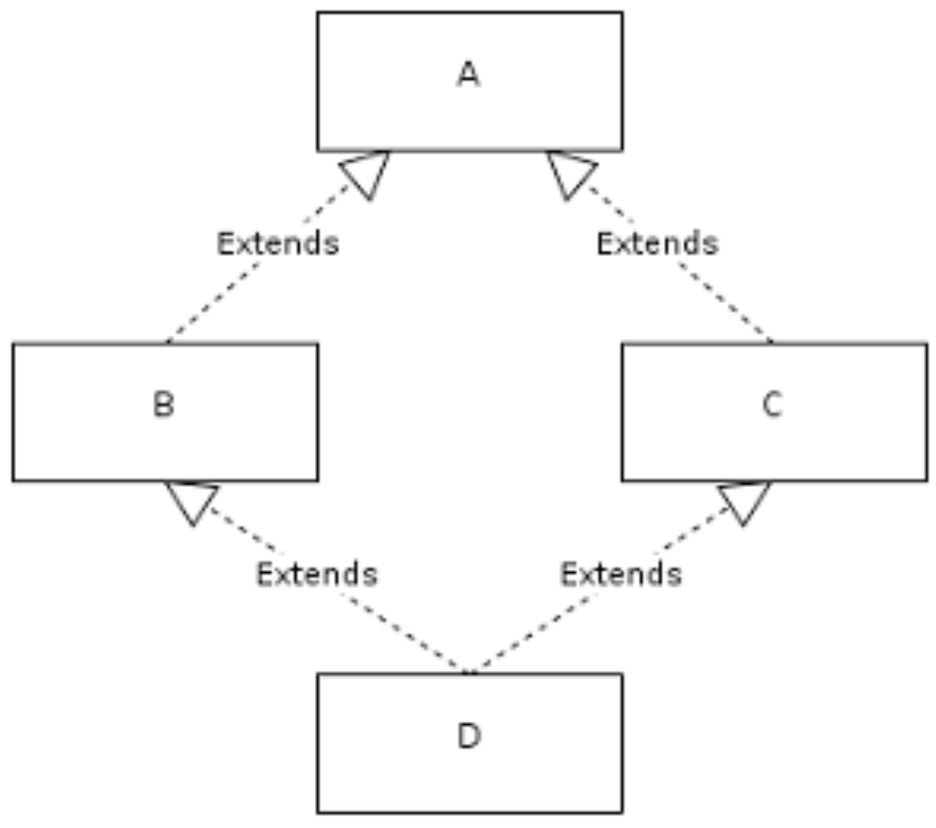
```
m1: B
m2: A
m3: A
```

```java
class A {
    public A() {
        m1();
        m2();
        m3();
    }
    public void m1() { System.out.println("m1: A"); }

    private void m2() { System.out.println("m2: A"); }

    public static void m3() { System.out.println("m3: A"); }
}

class B extends A {
    public B() { /* note that B() will implicitly call A() */}

    public void m1() { System.out.println("m1: B"); }

    private void m2() { System.out.println("m2: B"); }

    public static void m3() { System.out.println("m3: B"); }
}

public class WhatIsPrinted {
    public static void main(String[] args) {
        A b = new B();
    }
}
```

Except Multiple Inheritance

- The described approach to data layout works just fine unless your language allows multiple implementation inheritance.

- Remember the ‘Diamond Problem’?



C++ & Multiple Inheritance

- Consider the relatively simple case of multiple inheritance.
- Note that Top is inherited from *twice* from Bottom via Left and Right.
- This means that an object of type Bottom will have two attributes called ‘a’.

```
1  class Top {
2  public:
3     int a;
4  };
5
6  class Left : public Top {
7  public:
8     int b;
9  };
10
11 class Right : public Top {
12 public:
13    int c;
14 };
15
16 class Bottom : public Left, public Right {
17 public:
18    int d;
19 };
```

C++ & Multiple Inheritance

- How are Left, Right and Botton laid out in memory?

- If we take the approach we are familiar, we just ‘stack’ subclass data below superclass data.

Left
Top::a
Left::b

Right
Top::a
Right::c

Bottom
Left::Top::a
Left::b
Right::Top::a
Right::c
Bottom::d

C++ & Multiple Inheritance

- Now what happens when we upcast a Bottom pointer?
  - Ex. Left* left = bottom;
- No problem.
- We can treat an object of type Bottom as if it were an object of type Left, because the memory layout of both classes agree.

C++ & Multiple Inheritance

- However, what happens when we upcast to Right?
  - Ex. Right* right = bottom;

Left	Right
Top::a	Top::a
Left::b	Right::c

Bottom
Left::Top::a
Left::b
Right::Top::a
Right::c
Bottom::d

C++ & Multiple Inheritance

- However, what happens when we upcast to Right?
  - Ex. Right* right = bottom;
- Uh-oh. The memory layouts are different!
- We cannot use an instance of a Bottom as an instance of a Right, polymorphism is broken!

Review: Method Overload Resolution

Method Overloading

- Java supports overloading methods and can distinguish between methods with he same name but different *signatures*.

- This means that methods within a class can have the same name if they have different parameter lists.

Type & Arity

- Overloaded methods are disambiguated by parameter list *type* and *arity*
  - *type* is a classification identifying one of various types of data, such as real, integer or boolean.
  - *arity* is the number of arguments or operands a function or operation takes.

Overloading Vs Overriding

- When overloading a method, you are really just making a number of different methods that happen to have the same name. It is *resolved statically at compile time* which of these methods are used.

- This should not be confused with overriding where the correct method is chosen at *runtime*, e.g. through virtual functions.

Overloading Caveats

- You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

- The compiler does *not* consider return type when differentiating methods, so you cannot declare two methods with the same signature but with different return types.

- Note though that *overridden* methods have covariant return types.

Implementation Details

- Overloaded methods have separate slots in the vtable because the compiler treats them as totally different, unrelated methods.

- The sameness of their names is a convenience for us as programmers, and does not change the execution model of our language.

Rules for Overload Resolution

- Step 1: Find all methods declarations that share the name with the method invocation call site.

  - Determine the class that is being called.

  - Interrogate that class for methods of that name (That may include methods in the super class!)

  - Filter by the arity of the argument list vs the arity of the parameter lists.

Rules for Overload Resolution

- Step 2: When selecting candidate methods make sure to use only accessible methods.

- Do not attempt to…
  - access private methods from a subclass
  - access a protected method from outside the hierarchy or a package
  - access an instance method in a static context

Rules for Overload Resolution

- Step 3: Choose the most specific method.
  - One method declaration is more specific than another if the *types* are more specific.
  - So to return to our example from Overloading.java*..
    - m(int) is specific if the call site passes an int
    - There is no m(Exception), the most specific match is m(Object)
    - neither m(A, B) or m(B, A) is more specific than the other, so this is a compiler error.

* https://github.com/nyu-oop/method-overloading-java

Review: Reference Counting & Smart Pointers

Reference Counting

- The problem:
  - we have several references to some data on the heap
  - we want to release memory when there are no more references to it
  - we do not have automatic garbage collection and we do not want to manage memory ‘by-hand’

Reference Counting

- The solution: keep track of how many references point to the object and free it when there are no more.
  - Set reference count to 1 for newly created objects.
  - Increment reference count whenever we copy the pointer to the object.
  - Decrement count when a point to the object goes out of scope or stops pointing to the object.
  - When the count gets to 0, we can free the memory.

Reference Counting

- Advantages:
  - Memory can be reclaimed as soon as no longer needed.
  - Simple, can be done by the programmer for languages not supporting GC.
- Disadvantages:
  - Additional space needed for the reference count.
  - Will not reclaim circular references.

Smart Pointers

- A smart pointer is a class that mimics a regular pointer in syntax and semantics, but it does more.

- Smart Pointers offer *ownership* management in addition to pointer-like behavior.

- Our smart pointer will do *reference counting* to know when it is time to do deallocate the memory on the heap.

Value Semantics

- Smart pointers have value semantics
- Having value semantics means for an object that only its value(s) counts, not its identity.
- Any assignment to an object with value semantics has its values *replaced.*
- Any passing of an object with value semantics to a function creates a *copy.*
- This is different than reference semantics in that an assignment or function invocation would *copy a pointer*.

Value Vs Reference

- What gets printed from line 5 of the Java code?
  - 15,5
  - This is reference semantics

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

- What gets printed from line 5 of the C++ code?
  - 7,5
  - This is value semantics

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Smart Pointers & Value Semantics

- Value semantics are important for the smart pointer because we want to control assignment and copy operations such that we do reference counting.
- To enforce value semantics we must follow the Rule of Three, moreover we must implement…
  - Copy constructor
  - Assignment operator
  - Destructor

Copy Constructor

- Note the copy constructor on line 10

- Consider this code..

```
Ptr<int> original(new int(4));
Ptr<int> copy(p);
```

- If we had reference semantics, the reference count would not increase, even though we created a new reference to the underlying pointer.

```
1  /* size_t is a type used for storing unsigned integer
2     independentof architectures.  */
3  Ptr(T* addr = 0) : addr(addr), counter(new size_t(1)) {
4    TRACE(addr);
5  }
6
7  /* copy constructor for Ptr<U> */
8  template<typename U>
9  Ptr(const Ptr<U>& other)
10   : addr((T*)other.addr), counter(other.counter) {
11   TRACE(addr);
12   ++(*counter);
13 }
```

Assignment Operator

- Consider this code…

```
Ptr<int> a(new int(4));
Ptr<int> b(new int(4));
a = b;
```

- If we had reference semantics we would leak memory.

- While overloading the assignment operator with value semantics we manage memory.

```
1  Ptr& operator=(const Ptr& right) {
2    TRACE(addr);
3    if (addr != right.addr) {
4      if (0 == --(*counter)) {
5        TRACE("cleanup");
6        delete_policy::destroy(addr);
7        delete counter;
8      }
9      addr = right.addr;
10     counter = right.counter;
11     ++(*counter);
12   }
13   return *this;
14 }
```

Destructor

- Now in our destructor we can safely free memory when the reference count decrements to zero.

```
1  ~Ptr() {
2    TRACE(addr);
3    if (0 == --(*counter)) {
4      delete_policy::destroy(addr);;
5      delete counter;
6    }
7  }
```

- Note that this is only possible if the smart pointer has value semantics.

- Remember the delete policy is in place because different allocations require different deallocation commands (ex. arrays vs primitives)

Study Tip

- Very likely that homeworks, lecture demo code and in-class exercises will inform the test.