Object-Oriented Programming

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

Final Exam

Format

- Code reading questions
    - Ex. Read this class and write the vtable entries
- Short answers
    - Ex. What is the ‘Rule of Three’?
- Multiple Choice
    - Ex. Which of the following code blocks leak memory?

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 & C++

- Compilation
- Linking
- Source Management
- Binary Management

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

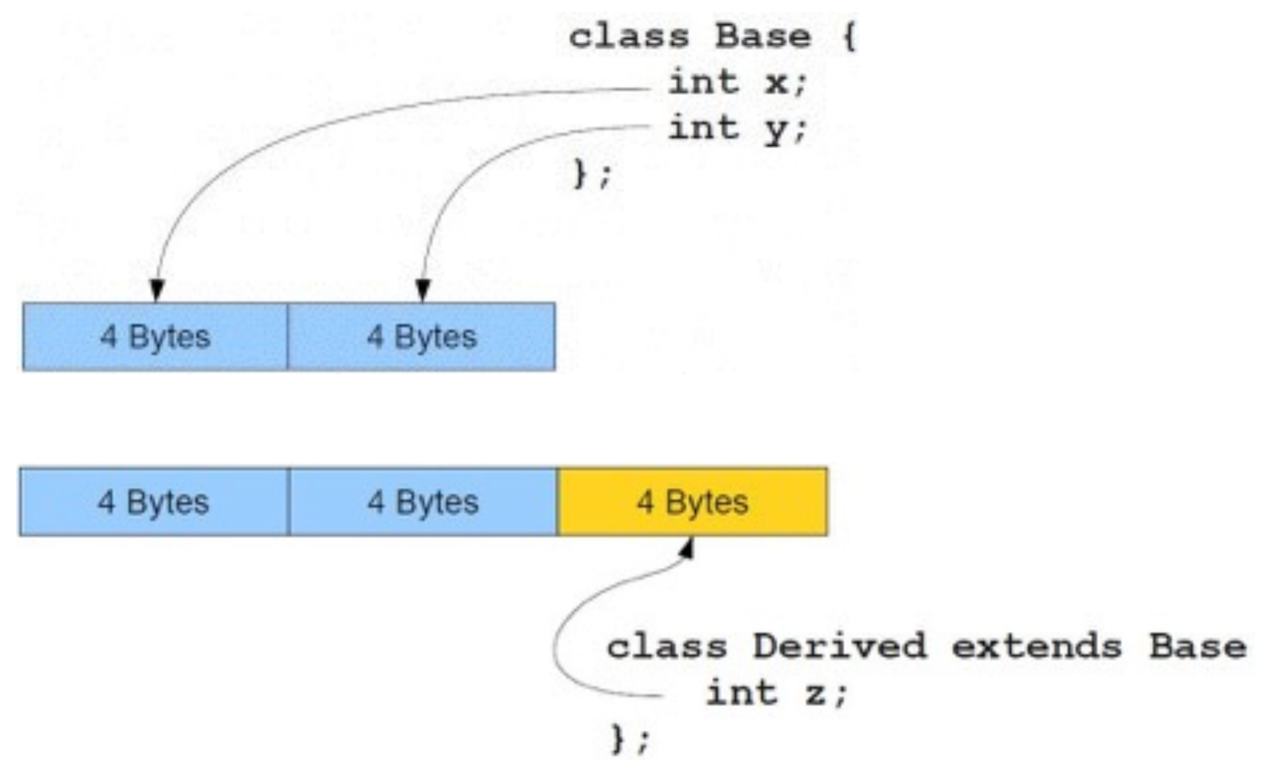
Review: Vtables & Data Layouts

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

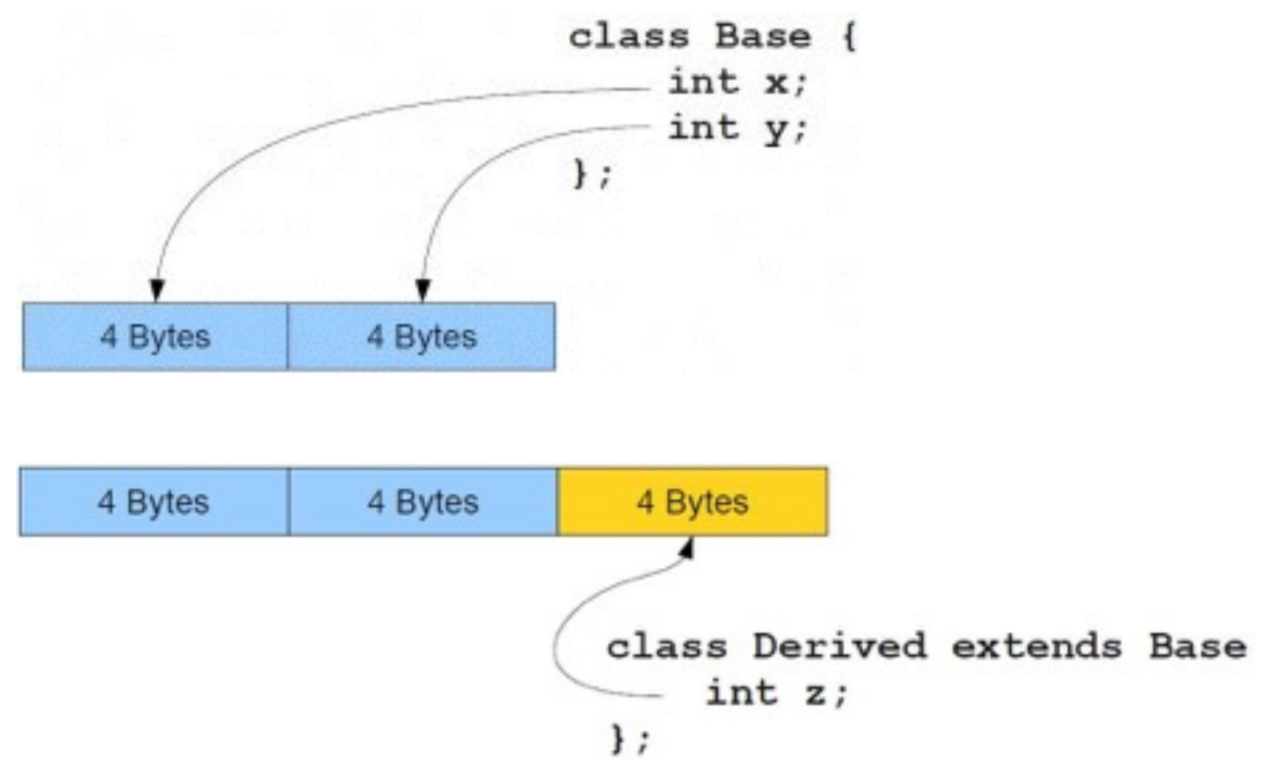
- Data members of a class are laid out contiguously in memory for each instance.
- Can be accessed via an offset.
- Child objects have the same memory layout as parent objects with additional space for subclass members.



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*.

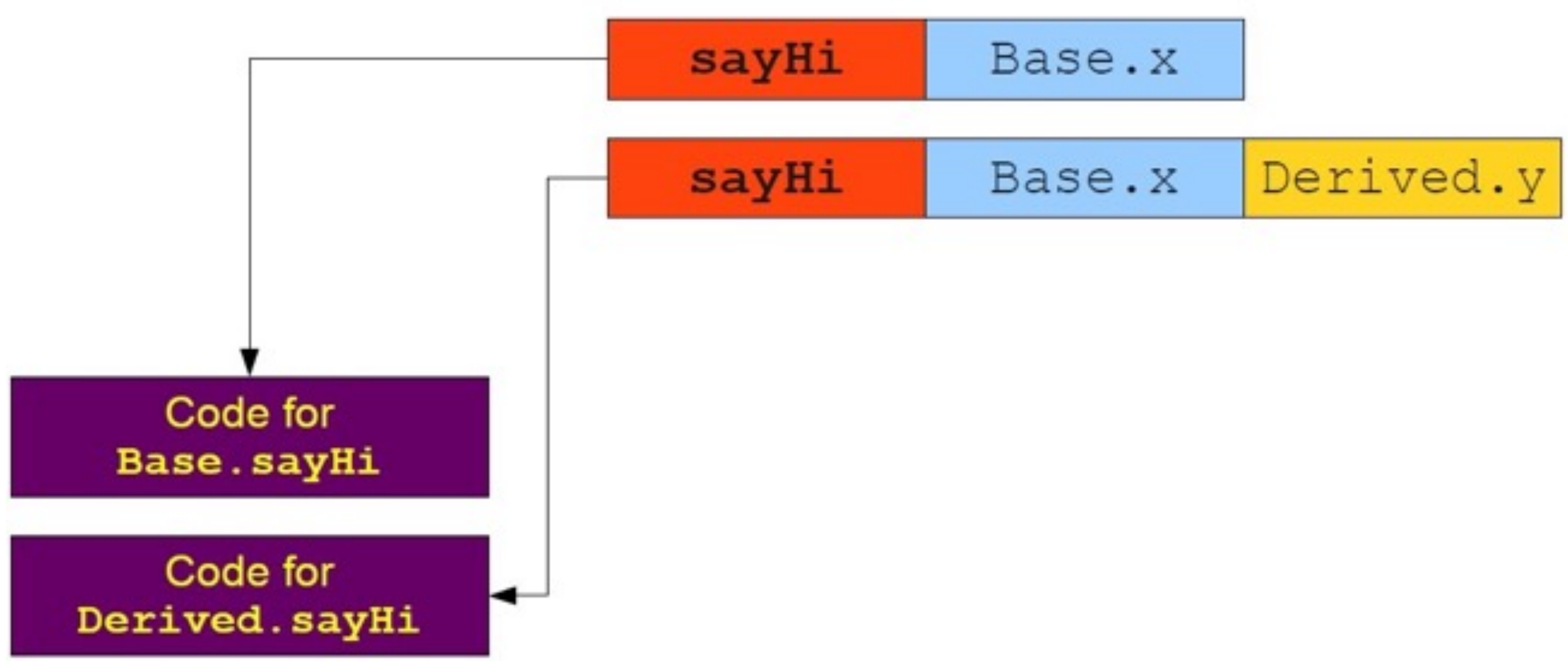


Object Methods

- 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");
    }
}
```



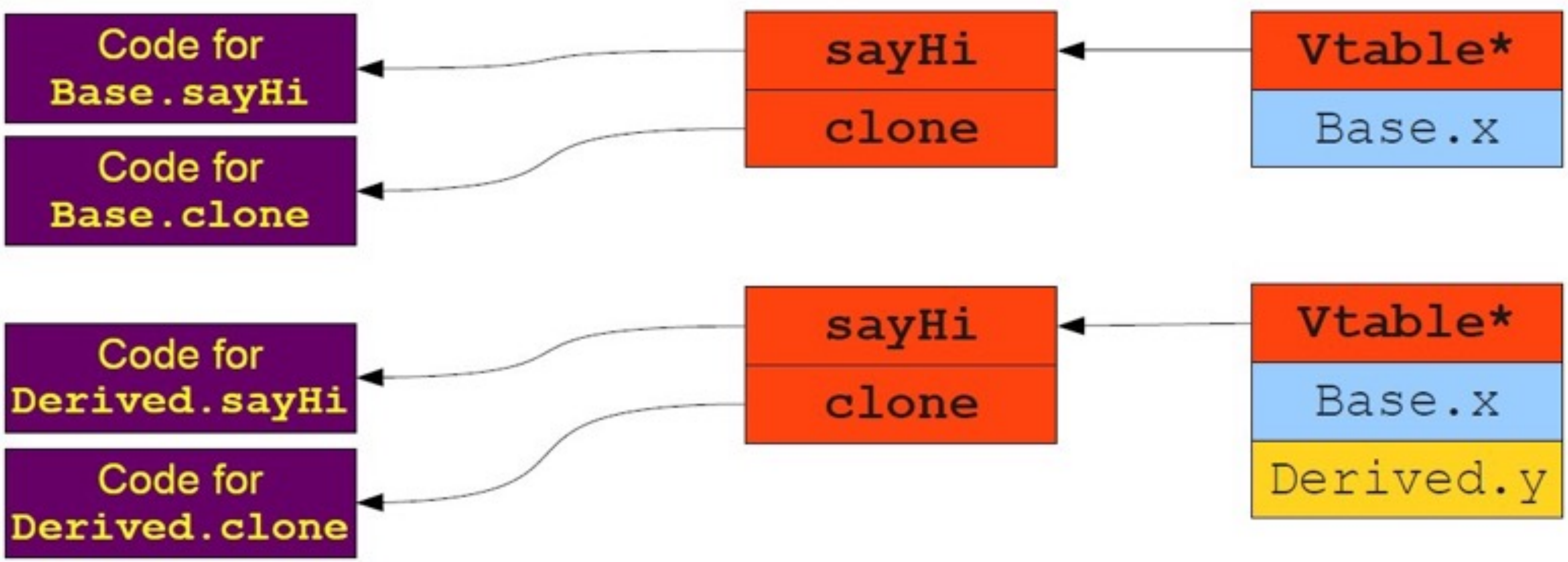
Per-Data Layout Method Pointers

- 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 pointer is to the ‘vtable’, or ‘virtual member table’.
- At instance creation (at runtime) table pointer will be set to point to the right vtable. i.e. the implementation for the *dynamic type*.
- This is what enables *dynamic dispatch*

VTables & Data Layouts

- What are the data layouts of A & B?

```java
class A {
    int a = 0;
    public int method() { return 12345;}
    public String toString() {
        return "A";
    }
}

class B extends A {
    int b = 0;
    public String toString() { return "B"; }
}
```

VTables & Data Layouts

- What are the data layouts of A & B?

- A

  A::a

- B

  A::a
  B::b

```java
class A {
    int a = 0;
    public int method() { return 12345;}
    public String toString() {
        return "A";
    }
}

class B extends A {
    int b = 0;
    public String toString() { return "B"; }
}
```

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)
```

```java
class A {
    int a = 0;
    public int method() { return 12345;}
    public String toString() {
        return "A";
    }
}

class B extends A {
    int b = 0;
    public String toString() { return "B"; }
}
```

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)
```

```
class A {
    int a = 0;
    public int method() { return 12345;}
    public String toString() {
        return "A";
    }
}

class B extends A {
    int b = 0;
    public String toString() { return "B"; }
}
```

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?

```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();
    }
}
```

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?

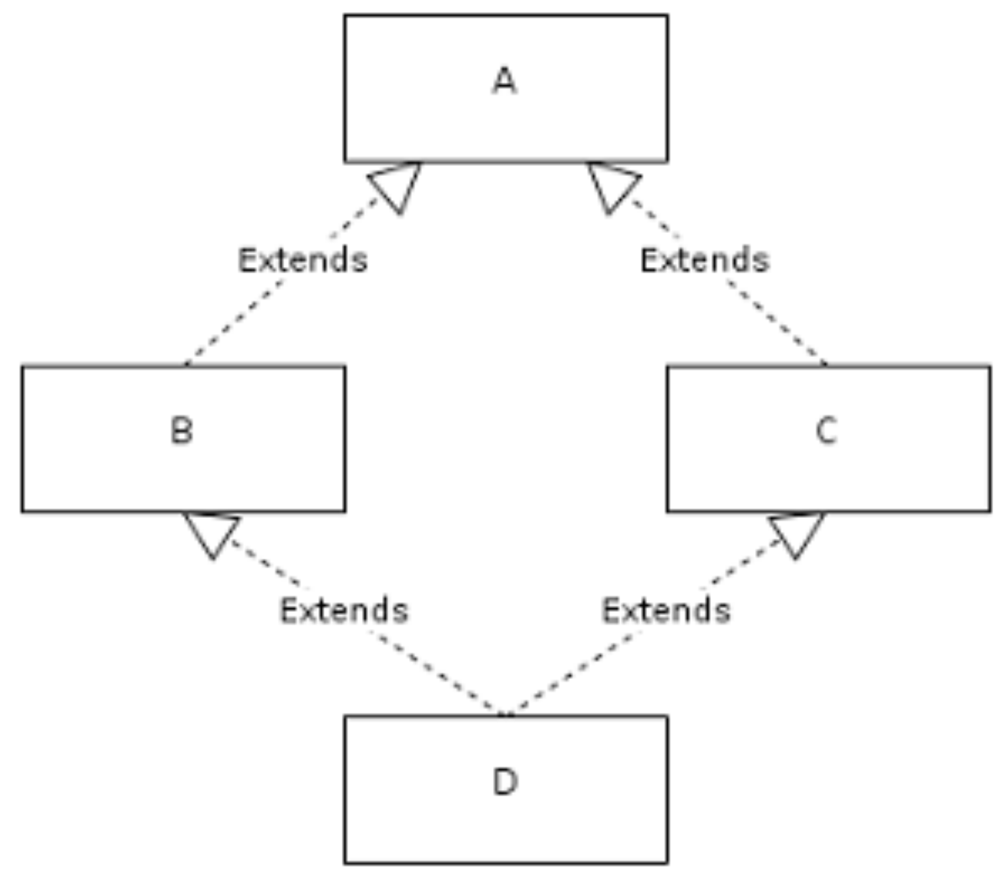
```
m1: B
m2: A
m3: A
```

```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’.

```
class Top {
public:
   int a;
};

class Left : public Top {
public:
   int b;
};

class Right : public Top {
public:
   int c;
};

class Bottom : public Left, public Right {
public:
   int d;
};
```

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	**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

- 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.

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;

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!

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

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

Review: Covariant Arrays

Java Arrays

```java
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What do lines 5, 6 and 7 print?

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What do lines 5, 6 and 7 print?
  - line 5: `[Ljava.lang.String;`
  - line 6: `[I`
  - line 7: `[[Ljava.lang.String;`
- Java initializes each element of an array to some sensible default value.
- What do all those brackets and characters mean though?

Java Arrays

- The JVM uses this shorthand notation to indicate the type of the array.
- Primitives are denoted with a single letter
- [indicates an array
- L is used for a class (terminated by a ;)
- Why no closing ‘]’?

```
[Z = boolean
[B = byte
[S = short
[I = int
[J = long
[F = float
[D = double
[C = char
[L = any non-primitives(Object)
```

Java Arrays

```
String[] ss = new String[5];
String[] tt = new String[5];

System.out.println(ss.equals(tt));
System.out.println(ss.getClass().getName());
System.out.println(new int[7].getClass().getName());
System.out.println(new String[5][5].getClass().getName());
System.out.println(ss.getClass().getSuperclass().getName());
```

- Java arrays exhibit *external containment* for non-primitives.
- For non-primitive types, Java does not provide C-style arrays in which all array *values* are stored in a contiguous block of memory.
- This is also true for *multidimensional* arrays of *primitive* types. Why?
- Remember that asterisk back on the 'Array Review' slide?

Java Arrays

```
1  String[] ss = new String[5];
2  String[] tt = new String[5];
3
4  System.out.println(ss.equals(tt));
5  System.out.println(ss.getClass().getName());
6  System.out.println(new int[7].getClass().getName());
7  System.out.println(new String[5][5].getClass().getName());
8  System.out.println(ss.getClass().getSuperclass().getName());
```

- What does line 8 print?

Java Arrays

```
1  String[] ss = new String[5];
2  String[] tt = new String[5];
3
4  System.out.println(ss.equals(tt));
5  System.out.println(ss.getClass().getName());
6  System.out.println(new int[7].getClass().getName());
7  System.out.println(new String[5][5].getClass().getName());
8  System.out.println(ss.getClass().getSuperclass().getName());
```

- What does line 8 print?
    - `java.lang.Object`
- Again, in Java arrays are objects and extend Object.
- Again, arrays do not override any methods of class Object.

Java Arrays

```
1  String[] sa = {"I", "am", "an", "array"};
2  Object o = sa;
3  Object[] oa = sa;
4  for(Object object : oa)
5    System.out.print(object);
```

- Is line 2 legal?

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object o = sa;
3 Object[] oa = sa;
4 for(Object object : oa)
5   System.out.print(object);
```

- Is line 2 legal?
    - Yes of course. Arrays extend Object.

Java Arrays

```java
1 String[] sa = {"I", "am", "an", "array"};
2 Object o = sa;
3 Object[] oa = sa;
4 for(Object object : oa)
5   System.out.print(object);
```

- Are lines 3 and 4 legal?

Java Arrays

```
1  String[] sa = {"I", "am", "an", "array"};
2  Object o = sa;
3  Object[] oa = sa;
4  for(Object object : oa)
5    System.out.print(object);
```

- Are lines 3 and 4 legal?
  - Yes!
  - Java arrays are typed *covariantly.* *(What a great final exam question subject.)*
  - Moreover.. if S extends T, then S[] extends T[]

Java Arrays

```java
String[] sa = {"I", "am", "an", "array"};
Object[] oa = sa;
oa[3] = new Object();
sa[3].charAt(3);
```

- Ok, if thats the case then, is this legal Java?

Java Arrays

```
1  String[] sa = {"I", "am", "an", "array"};
2  Object[] oa = sa;
3  oa[3] = new Object();
4  sa[3].charAt(3);
```

- Ok, if thats the case then, is this legal Java?
  - line 2 is ok, because of covariant array sub-typing
  - line 3 is ok because Object is a subtype of Object
  - line 4 is ok because sa is an array of Strings
- That doesn’t seem right… does it?

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object[] oa = sa;
3 oa[3] = new Object();
4 sa[3].charAt(3);
```

- Lets look at that again..
  - After executing line 2, the arrays sa and oa reference the same array object in memory.
  - Hence, line 3 would store an *Object* into the 3rd position of the ss array.
  - The third line would then attempt to call the *charAt* method on an *Object*.
  - *charAt* can only be called safely on objects of class *String* and its subclasses.

Java Arrays

```java
1 String[] sa = {"I", "am", "an", "array"};
2 Object[] oa = sa;
3 oa[3] = new Object();
4 sa[3].charAt(3);
```

- To prevent such unsafe behavior, the JVM will throw a runtime ArrayStoreException before executing line 3.
- Moreover, because of covariant subtyping of arrays, all store operations on arrays incur an additional *dynamic type* check.
- As seen with the design decisions of vtables, Dynamic type checks are undesirable due to the performance cost.

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

```java
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 );
```

```cpp
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.

```
Ptr& operator=(const Ptr& right) {
  TRACE(addr);
  if (addr != right.addr) {
    if (0 == --(*counter)) {
      TRACE("cleanup");
      delete_policy::destroy(addr);
      delete counter;
    }
    addr = right.addr;
    counter = right.counter;
    ++(*counter);
  }
  return *this;
}
```

Destructor

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

```
~Ptr() {
  TRACE(addr);
  if (0 == --(*counter)) {
    delete_policy::destroy(addr);;
    delete counter;
  }
}
```

Study Tip

- Very likely that homeworks and in-class exercises will inform the test.