string const & findSmallest(vector<string> const &v) {
    string smallest = v[0];
    for (unsigned int i = 0; i < v.size(); i++)
        if (v[i] < smallest)
            smallest = v[i];
    return smallest;
}

Hint: It will crash badly!

(Next slide please...)
Is this better?

```cpp
string const & findSmallest(vector<string> const &v)
{
    string const & smallest = v[0];
    for (unsigned int i = 0; i < v.size(); i++)
        if (v[i] < smallest)
            smallest = v[i];
    return smallest;
}
```

Hint: Now it won’t compile!

(Next slide please...)
Without const?

string const & findSmallest(vector<string> const &v) {  
    string const & smallest = v[0];  
    for (unsigned int i = 0; i < v.size(); i++)  
        if (v[i] < smallest)  
            smallest = v[i];  
    return smallest;  
}

Hint: Try running this!

(Next slide please...)
How about this?

```cpp
string const & findSmallest(vector<string> const &v) {
    string const * smallest = &v[0];
    for (unsigned int i = 0; i < v.size(); i++)
        if (v[i] < *smallest)
            smallest = &v[i];
    return *smallest;
}
```

(Next slide please...)

© 2011 Fawzi Emad, Computer Science Department, UMCP
Another approach. How should we call it?

```cpp
string const & findSmallest(vector<string> const &v) {
    int smallestIndex = 0;
    for (unsigned int i = 0; i < v.size(); i++)
        if (v[i] < v[smallestIndex])
            smallestIndex = i;
    return v[smallestIndex];
}
```

How should we call it?
```cpp
string s = findSmallest(...);
string &s = findSmallest(...);
string const &s = findSmallest(...);
```
Inheritance: Java vs. C++

Many aspects in C++ similar to Java, but many differences also.

Let’s start with the Lingo:

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>super class</td>
<td>base class</td>
</tr>
<tr>
<td>subclass</td>
<td>derived class</td>
</tr>
</tbody>
</table>
# Base Class Syntax: Java vs. C++

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Animal {</td>
<td>class Animal</td>
</tr>
<tr>
<td>private int size, age;</td>
<td>{</td>
</tr>
<tr>
<td>public Animal(int s, int a) {</td>
<td>private:</td>
</tr>
<tr>
<td>size = s;</td>
<td>int size, age;</td>
</tr>
<tr>
<td>age = a;</td>
<td>public:</td>
</tr>
<tr>
<td>}</td>
<td>Animal(int s, int a) : size(s), age(a)</td>
</tr>
<tr>
<td>}</td>
<td>{</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public void print() {</td>
<td>void print()</td>
</tr>
<tr>
<td>System.out.print(&quot;S: &quot; + size + &quot;, A: &quot; + age);</td>
<td>{</td>
</tr>
<tr>
<td>}</td>
<td>cout &lt;&lt; &quot;S: &quot; &lt;&lt; size &lt;&lt; &quot;A: &quot; &lt;&lt; age;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
# Derived Class Syntax: Java vs. C++

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Cat extends Animal {</td>
<td>class Cat : public Animal</td>
</tr>
<tr>
<td>private int whiskers;</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>private:</td>
</tr>
<tr>
<td></td>
<td>int whiskers;</td>
</tr>
<tr>
<td></td>
<td>public:</td>
</tr>
<tr>
<td>public Cat(int s, int a, int w){</td>
<td>Cat(int s, int a, int w) : Animal(s, a), whiskers(w)</td>
</tr>
<tr>
<td>super(s, a);</td>
<td>{</td>
</tr>
<tr>
<td>whiskers = w;</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>void print()</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td>public void print() {</td>
<td>Animal::print();</td>
</tr>
<tr>
<td>super.print();</td>
<td>cout &lt;&lt; &quot;,W: &quot; &lt;&lt; whiskers;</td>
</tr>
<tr>
<td>System.out.print(&quot;&quot;, W: &quot; + whiskers);</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>};</td>
</tr>
</tbody>
</table>
### Slicing: Java vs. C++

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat c = new Cat(8, 4, 200);</td>
<td>Cat c(8, 4, 200);</td>
</tr>
<tr>
<td>Animal a = c;</td>
<td>Animal a = c;</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>Animal a;</td>
</tr>
<tr>
<td></td>
<td>a = c;</td>
</tr>
</tbody>
</table>

- a and c are references to the same object, which is class Cat.
- c views the object as type Cat; a views it as type Animal.

- a and c are separate objects. A is an Animal; c is a Cat.
- a retains a “slice” of c.

Are we doomed? Not completely:

Animal &a = c; // More like Java

Unfortunately, slicing does ruin some things as we’ll see later.
## Binding: Java vs. C++

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat c = new Cat(8, 4, 100); Animal a = c; a.print();</td>
<td>Cat c(8, 4, 100); Animal &amp;a = c; a.print();</td>
</tr>
<tr>
<td>Output is: S: 8, A: 4, W: 100</td>
<td>Output: S: 8, A: 4</td>
</tr>
<tr>
<td>• “Late Binding” or “Dynamic Dispatch”</td>
<td>• “Early Binding” or “Static Dispatch”</td>
</tr>
<tr>
<td>• Advantage: Usually what we want</td>
<td>• Advantage: Slightly more efficient</td>
</tr>
</tbody>
</table>

Can we force these languages to do the opposite if we want to?

(Next slide please...)
Forcing Early/Late Binding: Java vs. C++

Can we force these languages to do the opposite if we want to?

Yes, but in both cases this requires a modification to the base/super class!

- **To force early binding in Java:**
  In the superclass declare the method as “final”. (But then it cannot be over-ridden!)

- **To force late binding in C++:**
  In the base class declare the function as “virtual”.
  It will now be considered “virtual” in all descendents, not just the immediately derived class.

**Rule of thumb for C++:**
If a function *might* be over-ridden one day, declare it as virtual.
Bad News

C++ *can* do dynamic binding, so why is Java favored so heavily for inheritance?

Java:
ArrayList<Animal> list = new ArrayList<Animal>();
list.add(new Cat());
list.add(new Dog());
list.add(new Ostrich());
for (Animal a : list)
  a.print();          // dynamic dispatch

C++:
Try same thing with a vector<Animal>.
Assume print is declared virtual.
Will we reap the benefits of dynamic dispatch?

(Hint: C++ collections store copies.)
Another Example, but not as Bad...

```java
void f(Animal x)
{
    x.print();
}

f(Cat());
f(Dog());
f(Ostrich());
```

This can be corrected easily. How?
Polymorphism: The Bottom Line

To use polymorphism effectively in C++:

• You need to ensure the base class is anticipating it (by use of virtual).

• You may need to use pointers for collections.
Subtle Problem!

Cat *p = new Cat(8, 4, 200);
Animal *q = p;
delete q;

What will happen?

(Next slide please...)
Subtle Problem!

Cat *p = new Cat(8, 4, 200);
Animal *q = p;
delete q;

• Static dispatch! It runs the Animal destructor.
• Only part of the object is destroyed.
• Memory leak!

Rule of thumb for destructors:
If there is any chance that your class may be extended one day, declare the destructor as virtual!
Inheritance: Copy Constructor and Assignment

Default copy constructor:
• Calls copy constructor for base class
• Calls copy constructor for all members

Default operator=:
• Calls operator= for base class
• Calls operator= for all members

Normally this is fine, but what if you need to write your own?

(Next slide please...)
Inheritance: Copy Constructor and Assignment

When implementing these in a derived class, don’t forget to explicitly invoke the base versions!

```cpp
Cat(Cat const &other) : Animal(other), whiskers(other.whiskers)
{
}

Cat & operator=(Cat const &rhs)
{
  if (this != &rhs)
  {
    Animal::operator=(rhs);
    whiskers = rhs.whiskers;
  }
  return *this;
}
```
Abstract: Java vs. C++

Java allows abstract methods and classes:

- An abstract method does not have an implementation (subclasses will implement it.)
  ```java
  abstract void foo();
  ```
- A class with an abstract method must be an abstract class.
  ```java
  public abstract class bar {...
  ```
- An abstract class cannot be instantiated.
- Abstract methods/classes are rarely used in Java; interfaces are favored.

C++:

- Abstract function must be virtual and end with “ = 0”. What the...???
  ```cpp
  virtual void foo() = 0; // completely bizarre
  ```
- A class with an abstract function is automatically abstract.
- An abstract class cannot be instantiated.
- Used frequently in places where Java would use an interface.
Inheritance Casting

Animal *p = new Cat(8, 4, 100);

p->whiskers // doesn't compile

Java-style casting is allowed:
((Cat *) p)->whiskers // works, but...

Will not throw exceptions for illegal casts!

Favor this:
(dynamic_cast<Cat *>(p))->whiskers

(Thows exceptions for illegal casts.)