Critique this Code

```
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?

```
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?

```
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…)

Another approach. How should we call it?

```
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?

```
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:

Java	C++
super class	base class
subclass	derived class

Base Class Syntax: Java vs. C++

Java	C++

public class Animal {
 	
  private int size, age;
 	
  public Animal(int s, int a) {
  size = s;
  age = a;
  }
 	
  public void print() {
  System.out.print("S: " + size + ", A: " + age);
  }
}
	class Animal
	{
	private:
	  int size, age;
	
	public:
	  Animal(int s, int a) : size(s), age(a)
	  {
	  }
	
	  void print()
	  {
	  cout << "S: " << size << "A: " << age;
	  }
	};

Derived Class Syntax: Java vs. C++

Java	C++

public class Cat extends Animal {
 	
  private int whiskers;
 	
  public Cat(int s, int a, int w){
  super(s, a);
  whiskers = w;
  }
 	
  public void print() {
  super.print();
  System.out.print(“, W: “ + whiskers);
  }
}
	class Cat : public Animal
	{
	private:
	  int whiskers;
	
	public:
	  Cat(int s, int a, int w) : Animal(s, a), whiskers(w)
	  {
	  }
	
	  void print()
	  {
	  Animal::print();
	  cout << “,W: “ << whiskers;
	  }
	};

Slicing: Java vs. C++

Java	C++

`Cat c = new Cat(8, 4, 200);`
 	
`Animal a = c;`
 	
•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
	`Cat c(8, 4, 200);`
	
	`Animal a = c;  // Copy constructor`
	
	OR
	
	`Animal a;`
	`a = c;  // Assignment`
	
	• 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++

Java	C++
Cat c = new Cat(8, 4, 100); Animal a = c; a.print(); Output is: S: 8, A: 4, W: 100 • "Late Binding" or "Dynamic Dispatch" • Advantage: Usually what we want	Cat c(8, 4, 100); Animal &a = c; a.print(); Output: S: 8, A: 4 • "Early Binding" or "Static Dispatch" • Advantage: Slightly more efficient

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…

```
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!

```
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.)

```
abstract void foo();
```

- A class with an abstract method must be an abstract class.

```
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…???

```
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
```

(Throws exceptions for illegal casts.)