Advanced Inheritance and Virtual Methods

Employee.h

```
class Employee {
   public:
      Employee(const string& name, double attitude, double wage);
      virtual ~Employee();

      const string& getName() const;
      const double getWage() const;

      virtual double getSalary() const;
      virtual double getProductivity() const;
      virtual void print() const;

   private:
      string name;
      double attitude, wage;
      int hours;
      static const int kFullTime = 40;
};
```

Boss.h

```
class Boss: public Employee {
   public:
      Boss(const string& name, const string& title, double attitude, double wage);
      virtual ~Boss();

      void setUnderling(Employee *u);
      Employee *getUnderling() const;

      virtual double getSalary() const;
      virtual double getProductivity() const;
      virtual void print() const;

   private:
      Employee *underling;
      string title;
};
```

Compile-time type versus run-time type. Consider the parameter to this function:

```
void DynamicPrint(Employee *e)
{
   e->print();
}
```

The compile-time type of the parameter is exactly **`Employee *`**. But when a variable is declared as a **`Employee *`** (or **`Employee&`**), that doesn't guarantee that at run-time it points to an **`Employee`**—only that it will be something that is *at least* an **`Employee`**. At

run-time, the pointer could point to an **Employee** or a **Boss** or any **Employee** subclass, and the run-time type can be different for different invocations of the function.

Due to compile-time type checking, within the **DynamicPrint** function, you can send this variable only those messages understood by **Employees** (and not those specific to **Boss**). These messages are safe for all **Employee** subclasses since they inherit all the behavior of their superclass.

If instead the parameter was declared as an object, not a pointer or reference:

```
void StaticPrint(Employee e)
{
   e.print();
}
```

The compile-time type and the run-time type are both exactly **Employee**. For example, since **Employee**s and **Boss**es are not necessarily the same size, it is impossible for a **Boss** object to be wedged into an **Employee**-sized space.

Compile-time binding versus run-time binding.

Go back to considering the **DynamicPrint** function from above. What happens when we try to send the **print** method to the parameter? If it is really pointing to an **Employee**, we expect that it will invoke **Employee's print** method, and that's fine. But what if the parameter is really pointing to a **Boss** which has its own overridden version of the **print** method?

C++ defaults to *compile-time binding*, which indicates the compiler makes the decision at compile time about which version of an overridden method to invoke. Since it is committing at compile time, it has to go with the compile-time type, so in this case it will choose to always use **Employee's print** method, ignoring the possibility that subclasses might provide a replacement.

This most likely is not what you intended. If **Boss** overrides print, you expect that any time you send a **Boss** a **print** message (no matter what the CT type you are working with is), it should invoke the **Boss's** version of the method. Given that OO programming was supposed to be all about making objects responsible for their own behavior, it's unimpressive that it can be so easily convinced to invoke the wrong version!

What makes more sense is to use *run-time binding* where the decision about which version of the method to invoke is delayed until run-time. At the point when the **DynamicPrint** function is called, it can determine what the actual RT type is and dispatch to the correct print function for that type. This means if you call `Apple` passing an **Employee** object, it uses **Employee's print** and if you later call **DynamicPrint** passing a **Boss** object, it uses **Boss's** version of **print**.

Declaring methods `virtual`.

In C++, run-time binding is enabled on a per-method basis (although some compilers have an option to make all methods **`virtual`** even though it's not the standard). In order to make a particular method RT bound, you declare it virtual. Mark it **`virtual`** in the parent class, which makes it **`virtual`** for all subclasses, whether or not they repeat the **`virtual`** keyword on their overridden definitions. In general, you tend to want to make almost all methods **`virtual`** (there are a few exceptions discussed below) so that you guarantee that the right method is sent to the object without fail.

Note that RT binding only applies to those objects accessed through pointers or references. If you are working with an actual object, its CT and RT types are one and the same (for example, consider the **`StaticPrint`** function above) and thus there is never any difference between the CT and RT types, so it might as well bind at CT.

Constructors aren't inherited and can't be virtual.

Constructors are very tightly bound up with a class and each class has its own unique set of constructors. If **`Employee`** defines a 3-arg constructor, **`Boss`** does not inherit that constructor. If it wants to provide such a constructor, it must declare it again and just pass the arguments along to the base class constructor in the constructor initialization list. It is nonsensical to declare a constructor **`virtual`** since a constructor is always called by name (**`Employee("Sally"...)`** or **`Boss("Jane"...)`**) so there is no choice about which version to invoke.

Destructors aren't inherited, but should be `virtual`.

Like constructors, destructors are tightly bound with a class and each class has exactly one destructor of its own. If you don't provide a destructor, the compiler will synthesize one for you that will call the destructors of your member objects and base class and do nothing with your other data members. Note that whether you define your own or let the compiler do it for you, the destruction process will always take care of calling the superclass destructor when finished with the subclass destructor—you never explicitly invoke your parent destructor.

The destructor needs to be **`virtual`** for the same reason that normal methods are **`virtual`**: You want to be sure the correct destructor is called, using the RT type of the object, not the CT type.

Consider the **Terminate** function:

```
void Terminate(Employee *e)
{
   delete e;
}
```

If the **Employee** destructor is not virtual, the use of **delete** here will be CT-bound and commit to invoking the **Employee** class destructor. However, if at RT the parameter was really pointing to a **Boss**, we really need to use the **Boss**'s destructor to clean up the any dynamically allocated parts of a **Boss** object. If you declare the base class destructor as **virtual**, it defers the decision about which destructor to invoke until RT and then makes the correct choice. Note that declaring the destructor **virtual** makes the destructor of all subclasses virtual even though the names do not quite match **(~Employee -> ~Boss).**

Some compilers (**g++**, for example) will warn if you define a class with **virtual** methods but a non-**virtual** destructor.

Assigning/copying a derived to a base "slices" the object.

If I have an object of a derived class and try to assign/copy from an object of the base class, what happens? First consider the definition of the assignment operator/copy constructor (whether explicitly defined or synthesized by the compiler). In the **Employee** class, **operator=** it will take a reference to an **Employee** object. It's completely fine to pass it a reference to a **Boss** object, since a **Boss** can always safely stand if for an **Employee**. In the copy/assign operator, it will copy the **Employee** part of the **Boss** object and ignore the extra **Boss** fields, in a sense, "slicing" out the employee fields and throwing the rest away. The result is an **Employee** object that has the same **Employee** data as the **Boss** object did, but none of the **Boss** fields or behavior is kept.

```
void Raspberry()
{
   Employee bob("Bob", 0.8, 10.0);
   Boss sally("Sally", 0.55, 25.0, "Lead Architect");

   bob = sally; // "slices" off extra fields
   bob.print(); // Bob is an *Employee* so uses Employee version
}
```

The same thing is true for the copy constructor. For example, the copy constructor is invoked when passing and returning objects by value. If I were to pass **sally** to the **StaticPrint** function from above, the parameter would be a copy of just the **Employee** fields from **sally**. Slicing is yet another reason to avoid passing object parameters by value.

Be sure that you understand how slicing is different than the "upcast" operation where we assign a **Boss *** to an **Employee ***. In that case, we have not thrown away any

information and if we're correctly using **virtual** methods, we won't lose any behavior either. Declaring a parameter as a reference/pointer to the base is simply generalizing the allowable type so that all derived types can be easily used.

Assigning/copying a base to a derived is not allowed.

In general, copying/assignment in the other direction is not allowed. The rationale goes something like this: If I were to try to assign a **Boss** from an **Employee** object, I could copy all the **Employee** fields, but the extra fields of a **Boss** would left uninitialized. This unsafe operation conflicts with C++'s strong commitment to ensuring all data is initialized before being used.

To be more mechanical about it, consider the compiler-synthesized definitions of the assignment/copy constructor for the **Boss** class. It takes a **const Boss&** as its parameter. Can I pass an **Employee&** to a function that needs a **Boss&**? Nope, an **Employee** does not necessarily have all the data and methods that a **Boss** does. To make assignment work in the other direction, the **Boss** class could define a version of **operator=** that took an **Employee**, copied the **Employee** fields and do something reasonable with the remaining fields. In truth, this is not the common a need (to assign objects of different classes back and forth), but it can be done if necessary.

Calling virtual methods inside other methods.

The binding of methods called from within other methods is basically just like other bindings. For example, assume the body of **Employee::print** makes a call to **getSalary()**. If **getSalary** is not declared **virtual**, the compiler binds the call at CT and within the **print** method of **Employee** "**this**" is of type **Employee *,** so it commits to using **Employee's** version. If **getSalary** is declared **virtual**, it waits until the **print** method is called at RT, at which point the true identity of the object is used to decide which version of **getSalary** is appropriate. It makes no difference whether the **print** method itself is declared **virtual** in deciding how to bind calls to other methods made within the `print` method.

Calling virtual functions in constructors/destructors.

The one place where **virtual** dispatch doesn't enter into the game is within constructors and destructors. If you make a call to a **virtual** function, such as **print**, within the **Employee** constructor or destructor, it will always invoke the **Employee** version of **print**. Even if we are constructing this **Employee** object as part of the constructor of an eventual **Boss** object, at the time of the call to the **Employee** constructor, the object is actually just an **Employee** and thus responds like an **Employee**. Similarly on destruction, if a **Boss** object is being destructed, it first calls its own destructor, "stops being a **Boss**" and then goes on to its parent destructor. At the time of the call to **Employee** destructor, the object is no longer a **Boss**, it's just an **Employee**, and is unwinding back to its beginnings. So in a constructor/destructor the object is always of the stated CT type without exceptions. The rationale for this is that the

virtual function may rely on part of the extra state of a **Boss** (such as the **title** field) and it will not be safe to call it before the **Boss** construction process has occurred, or after the **Boss** destruction has already happened.

Overriding versus overloading.

Overloading a method or function allows you to create functions of the same name that take different arguments. Overriding a method allows to replace an inherited method with a different implementation under the same name. Most often, the overridden method will have the same number and types of arguments, since it is intended to be a matching replacement. What happens when it doesn't? For example, let's say we added the some **promote** methods to the **Employee** class:

```
void promote(int additionalDaysOff);
void promote(double percentRaise);
```

This method is overloaded and it chooses between the two available versions depending on whether called with an **int** or a **double**. Usually, **Boss** inherits both of these versions. Now, let's say **Boss** wants to introduce its own version of **promote**, this one taking a **string** which identifies a new title for the **Boss**:

```
void promote(const string& newTitle);
```

You might like/think/hope that the Boss would now have all three versions of **promote**, but that isn't the way it works. In this case, the **Boss**'s override of **promote** completely shadows all previous versions of **promote**, no matter what the arguments are. If we want **Boss** to have versions of **promote** that take **int** and **double**, we would need to redefine them in the **Boss** class and just provide a wrapper that calls the **inherited** version, something like this:

```
void promote(int additionalDaysOff) { Employee::promote(additionalDaysOff);}
void promote(double percentRaise) { Employee::promote(percentRaise);}
```

Seems a little awkward, but that's C++ for ya. The idea is to avoid nasty surprises where you end up getting a different inherited version when the subclass was trying to replace all of the parent's implementation of that method. (There is also a solution involving the **using** directive.)

Multi-Methods

A **multi-method** is a method that is chosen according to the dynamic type of more than one object. They tend to be useful when dealing with interactions between two objects. As we have seen, **`virtual`** functions allow the run-time resolution of a method based on the dynamic type of the **receiving** object. However, a parameter to a function can only be matched according to its **`static`** type. This limits us to determining which method to call to the dynamic type of one object (the object upon which the method is invoked). C++ has no built-in support for multi-methods. Other languages, such as CLOS, do have support for these. Assume we have an intersect method which tells us if two shapes intersect:

```
class Shape {
   ...
   virtual bool intersect(const Shape *s) = 0;
   ...
};

class Rectangle : public Shape {
   ...
   virtual bool intersect(const Shape *s);
   ...
};

class Circle : public Shape {
   ...
   virtual bool intersect(const Shape* s);
   ...
};
```

It doesn't make sense to see if a **`Rectangle`** or a **`Circle`** intersects a **`Shape`**. So we immediately see the need to provide more specialized methods:

```
class Shape {
   ...
   virtual bool intersect(const Shape *s) = 0;
   virtual bool intersect(const Circle *c) = 0;
   virtual bool intersect(const Rectangle *r) = 0;
   ...
};

class Rectangle : public Shape {
   ...
   virtual bool intersect(const Shape *s);     // illustrated below
   virtual bool intersect(const Circle *c);    // Checks circle/rectangle
   virtual bool intersect(const Rectangle *r); // Checks rectangle/rectangle
   ...
};

class Circle : public Shape {
   ...
   virtual bool intersect(const Shape *s); // wrapper implementation below
   virtual bool intersect(const Circle *c); // Checks circle/circle
   virtual bool intersect(const Rectangle *r); // Checks rectangle/circle
   ...
```

```
};
```

Note that we have to declare many methods that should never be called in order to prevent the hiding of methods through overloading. If we allocate a **Circle** and a **Rectangle** whose **static** types are **Shape ***, we're in for a few surprises:

```
Shape* circle = new Circle(...);       // upcast to Shape
Shape* rectangle = new Rectangle(...); // upcast to Shape
circle->intersect(rectangle);          // Calls Circle::intersect(Shape*)
rectangle->intersect(circle);          // Calls Rectangle::intersect(Shape*)
```

We're getting one level of reification through the use of **virtual** functions, but we need two levels of reification. Short of using type fields or another similar mechanism, we need to make two virtual function calls in order to get two levels of reification. This is called **double dispatch**, and is a nice way to simulate multi-methods in C++.

We must change the methods which get called via the first virtual function call to make another virtual function call (they used to do nothing). For example we would need to write this:

```
bool Circle::intersect(Shape *shape)
{
   return shape->intersect(this); // "this" is a Circle *, not a Shape *
}
```

We would have to make a similar change to the **Rectangle::intersect(Shape *)** method.

Let's trace a call to **intersect** when we call it with a **Circle** and a **Rectangle**. We allocate our two shapes, both of which are bound to variables with a **static** type of **Shape *.**

```
Shape* circle = new Circle(...);       // upcast to Shape *
Shape* rectangle = new Rectangle(...); // upcast to Shape *
```

We call the intersect method, which is reified to **Circle::intersect(Shape *)**

```
circle->intersect(rectangle); // calls Circle::intersect(Shape *)
```

The **Circle::intersect(Shape *shape)** method then executes:

```
return shape->intersect(this);
```

The dynamic type of **shape** is **Rectangle ***, and the static type of "**this**" is **Circle***. We then call **Rectangle::intersect(Circle *)** and have thus done two levels of reification. Using double dispatch is reasonably efficient, but it requires you to write a lot of methods.