Topics in Inheritance

Introduction

C++ is a fully-featured object-oriented language that supports a wide variety of inheritance schemes. Consequently, C++ inheritance is complicated and at times entirely counterintuitive. While a full treatment of C++ inheritance is far beyond the scope of this class, there are several important topics in inheritance that you will almost certainly encounter in your C++ career. This handout discusses important topics in C++ inheritance and serves as a launching point for a deeper exploration of C++ inheritance.

Name Resolution and Inheritance

Consider the following code snippet:

```
const int x = 0;
class MyClass
{
public:
    void doSomething()
    {
        int x = 137;
        cout << x << endl;
    }
private:
    int x;
};
```

Here, there are three variables named `x` – a global constant, a data member of `MyClass`, and a local variable of `doSomething`. When you write `cout << x << endl`, the C++ compiler determines that you're referring to the local variable `x` rather than the other `x` variables. This process is known as *name resolution*, the translation of function, variable, and class names into the objects they represent.

In many ways, C++'s name resolution system is wonderful, but when working with inheritance the name resolution system sometimes behaves counterintuitively. Let's consider two classes, `CardGame` and `PokerGame`, which are defined below:

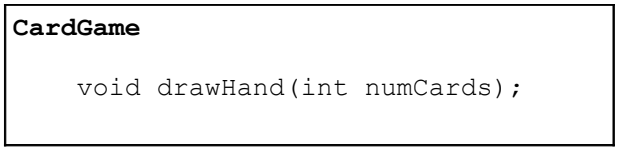
```
class CardGame
{
public:
    /* Draws the specified number of cards into a new hand. */
    void drawHand(int numCards);
private:
    /* Implementation details... */
};
```

```
class PokerGame: public CardGame
{
public:
    /* Draws five cards, since it's a poker hand. */
    void drawHand();
};
```

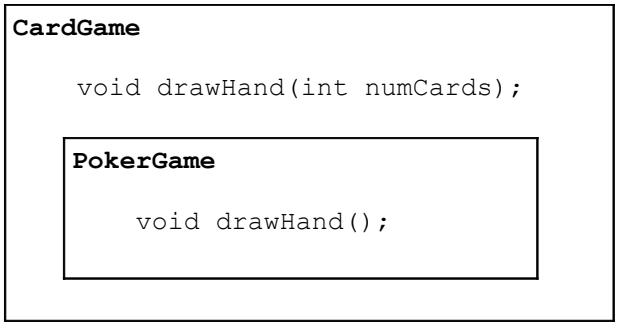
Nothing about this setup seems unreasonable – the `CardGame` superclass has generic functionality to draw cards while the `PokerGame` subclass exports a convenience `drawHand` function that always draws five cards. By itself there's nothing wrong with this code, and the above declarations will compile and link, but if we try to use a `PokerGame` object we will get compiler errors when using the original version of `drawHand`. Suppose we write the following code:

```
PokerGame game;
game.drawHand(NUM_START_CARDS);
```

Here, we'd expect that since `CardGame` exports a function named `drawHand` that takes an `int` parameter, the above code will be legal. Unfortunately, it's not, and we'll get a compiler error telling us that the `drawHand` function does not accept any parameters. To understand why, we'll have to consider how C++ scoping rules work with parent and derived classes. When you create a class like the above `CardGame` class, C++ provides a new layer of scoping for that class. Thus, in the above example, the scope might look something like this:



When you create a derived class, C++ nests the derived class's scope inside the parent class's scope. That way, when referring to a function or variable of the parent class, C++ can find it one layer out of the derived class's scope. The scoping picture of the `PokerGame` class thus looks something like this:



When C++ looks for the object a name refers to, it starts in the current scope and steps outward until it finds any object that has that name. Normally, this is exactly what we want, but when working with function overloads it poses a problem. When we called `game.drawHand(NUM_START_CARDS)`, the compiler searched for a function called `drawHand` inside the `PokerGame` scope. Since `PokerGame` defines a `drawHand` function, the compiler halted its search without ever leaving the `PokerGame` scope and concluded that the only `drawHand` function must be the zero-parameter `drawHand` function. Although the original `drawHand` function still exists, the compiler didn't know to look for it.

Somehow we need to tell C++ to search for the other version of `drawHand`. There are two different ways to do this. First, when calling `drawHand`, we could explicitly say that we want to call the version exported by `CardGame`. The syntax looks like this:

```
PokerGame game;
game.CardGame::drawHand(NUM_START_CARDS);
```

While this syntax is legal, it's clunky and forces clients of the `PokerGame` class to know about the `CardGame` base class. Let's consider an alternative solution. We want to tell the compiler to not stop looking for functions named `drawHand` once it's found the one inside of `PokerGame`. To do this, we'll use a `using` declaration to import the `drawHand` function from `CardGame` into the `PokerGame` scope. Up to this point, we've only seen `using` in the context of `using namespace std`, which imports all elements of the standard namespace into the global namespace. However, `using` is a more generic instruction that tells the C++ compiler to treat variables, classes, or functions from a different scope as part of the current scope. Thus, to fix the `PokerGame` example, we'll tell C++ that it should import the function `CardGame::drawHand` into the `PokerGame` scope, as shown here:

```
class PokerGame: public CardGame
{
public:
    void drawHand();
    using CardGame::drawHand; // Import the function name.
};
```

Now, the compiler will find the original version of the function.

Invoking Virtual Member Functions Non-Virtually

From time to time, you will need to be able to explicitly invoke a base class's version of a virtual function. For example, suppose that you're designing a `HybridCar` that's a specialization of `Car`, both of which are defined below:

```
class Car
{
public:
    virtual void applyBrakes();
    virtual void accelerate();
};


class HybridCar: public Car
{
public:
    virtual void applyBrakes();
    virtual void accelerate();
};
```

The `HybridCar` is exactly the same as a regular car, except that whenever a `HybridCar` slows down or speeds up, the `HybridCar` charges and discharges its electric motor to conserve fuel. Here, we have a bit of a problem. Since `HybridCar` does not `applyBrakes` or `accelerate` the same way as a regular `Car`, we need to mark those two functions virtual. However, since `applyBrakes` and `accelerate` are virtual functions, when accessed virtually we'll bypass all of the code for the original `applyBrakes` and

`accelerate` functions even though the `HybridCar` versions of those functions are only minimally different. In effect, while the code is already written, we're unable to access it.

At this stage, we have several options. First, we could simply copy and paste the code from the `Car` class into the `HybridCar` class. This is a bad idea since if we ever change the `Car` class's versions of `applyBrakes` or `accelerate`, `HybridCar`'s behavior will differ from the regular `Car`'s. Second, we could factor out the code for `applyBrakes` and `accelerate` into `protected`, non-virtual functions of the `Car` class. This too is problematic, since we might not have access to the `Car` class – maybe it's being developed by another design team, or perhaps it's complicated legacy code. The third option, however, is to simply have the `HybridCar`'s virtual function call the `Car`'s version of the function, and it's this option that is almost certainly the right choice.

When calling a virtual function through a pointer or reference, C++ ensures that the function call will "fall down" to the most derived class's implementation of that function. However, we can force C++ to call a specific version of a virtual function by calling it using the function's fully-qualified name. For example, consider this version of `applyBrakes`:

```
void HybridCar::applyBrakes()
{
    chargeMotor();
    Car::applyBrakes(); // Call Car's version of applyBrakes, no polymorphism
}
```

The syntax `Car::applyBrakes` instructs C++ to call the `Car` class's version of `applyBrakes`. Even though `applyBrakes` is virtual, since we've used the fully-qualified name, C++ will not use any runtime polymorphism and we are guaranteed to call the correct version of the function. We can write an `accelerate` function for `HybridCar` in a similar fashion.

When using the fully-qualified-name syntax, you're allowed to access any superclass's version of the function, not just the direct ancestor. So if `Car` were derived from the even more generic class `FourWheeledVehicle` that itself provides an `applyBrakes` method, we could invoke that version from `HybridCar` by writing `FourWheeledVehicle::applyBrakes()`. However, you cannot use the fully-qualified name syntax outside of the class hierarchy. Although you are calling a specific version of a member function, it is still a member function and thus requires a receiving object.

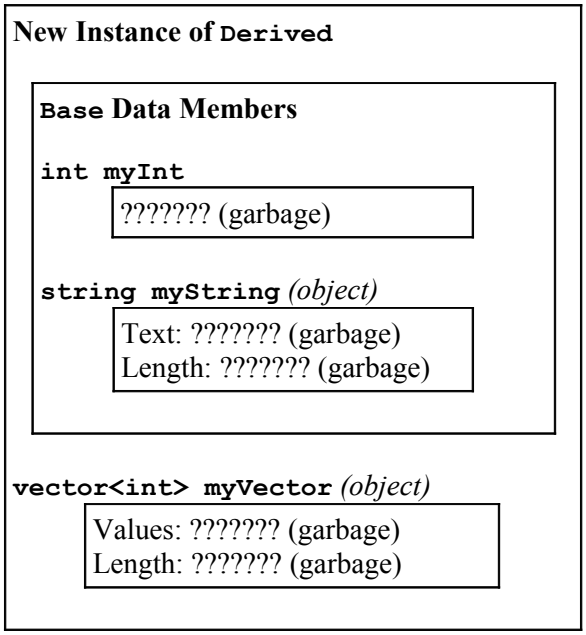
Object Initialization in Derived Classes

Recall from several weeks ago that class construction proceeds in three steps – allocating space to hold the object, calling the constructors of all data members, and invoking the object constructor. While this picture is mostly correct, it omits an important step – initializing any base classes through the base class constructor. Let's suppose we have the following two classes, `Base` and `Derived`:

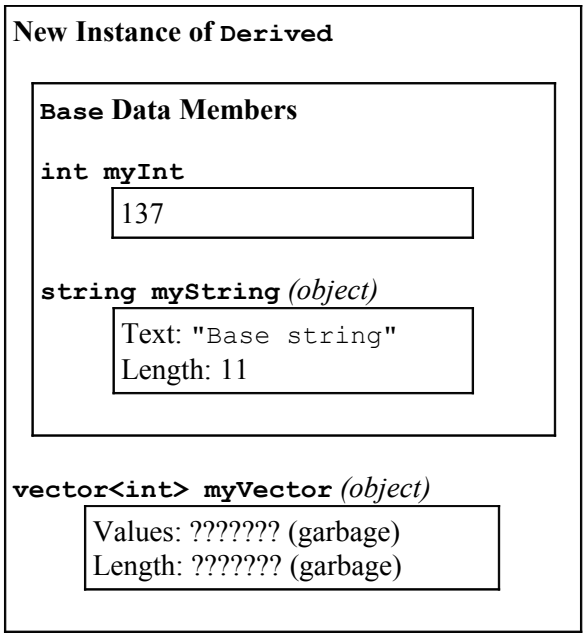
```
class Base
{
    public:
        Base() : myInt(137), myString("Base string!") {}
    private:
        int myInt;
        string myString;
};
```

```
class Derived: public Base
{
private:
    vector<int> myVector;
};
```

When we construct a new `Derived` object, the first step is still to get a block of uninitialized memory with enough space to hold an entire `Derived` object. This memory looks something like this:



At this point, C++ will initialize the `Base` class using its default constructor. Similarly, if `Base` has any parent classes, those parent classes will also be initialized. After this step, the object now looks like this:



From this point forward, construction will proceed as normal.

Notice that during this process, C++ invoked the default constructor for `Base`. If you'll remember from several weeks ago, this is the same behavior C++ uses to initialize data members. In many circumstances, you won't want to use a base class's default constructor, and as with data members, you can invoke a specific base class constructor using the initializer list. Let's consider a slightly modified version of `Base`, shown below, that does not have a default constructor:

```
class Base
{
public:
    explicit Base(int val) : myInt(val), myString("Base string") {}
private:
    int myInt;
    string myString;
};
```

Now, if we were to use the above definition of `Derived`, we'd get compile-time errors since there is no default constructor available for `Base`. To fix this, we'll change the `Derived` class's constructor so that it invokes the `Base` constructor, passing in the value 0:

```
class Derived: public Base
{
public:
    Derived() : Base(0) {}
private:
    vector<int> myVector;
};
```

A subtle but important point to note is that during object construction, the base class is constructed before its derived class's data members have been initialized. This means that if you were to somehow access data members or member functions of a derived class from inside the base class constructor, you'd run into trouble as you used completely garbage values (or even garbage objects) in your code. To prevent you from accidentally accessing this nonsense data, inside the body of a class constructor, C++ will disable virtual function calls. To see this in action, consider the following code:

```
class Base
{
public:
    Base()
    {
        fn();
    }
    virtual void fn()
    {
        cout << "Base" << endl;
    }
};
```

```
class Derived: public Base
{
public:
    virtual void fn()
    {
        cout << "Derived" << endl;
    }
};
```

Here, the `Base` constructor invokes its virtual function `fn`. While normally you would expect that this would invoke `Derived`'s version of `fn`, since we're inside the body of the `Base` constructor, the code will execute `Base`'s version of `fn`, which prints out "Base." This is an important safety feature. If C++ were to call `fn` virtually, it might invoke a derived class's version of `fn` which could access uninitialized variables. Although this is admittedly counterintuitive, the alternative is much worse.

Copy Constructors and Assignment Operators for Derived Classes

Copy constructors and assignment operators are complicated beasts that are even more perilous when mixed with inheritance. Not only must you remember to copy over all data members while tracking several other edge cases, but you'll need to explicitly copy over base class data as part of the initialization or assignment.

Consider the following base class, which has a well-defined copy constructor and assignment operator:

```
class Base
{
public:
    Base();
    Base(const Base &other);
    Base& operator= (const Base &other);
    virtual ~Base();
private:
    /* Could be anything. */
};
```

Consider the following derived class:

```
class Derived: public Base
{
public:
    Derived();
    Derived(const Derived &other);
    Derived& operator= (const Derived &other);
    virtual ~Derived();
private:
    char *theString; // Store a C string
    void copyOther(const Derived &other);
    void clear();
};
```

We might write the following code for the `Derived` assignment operator and copy constructor:

```
/* Generic "copy other" member function. */
void Derived::copyOther(const Derived &other)
{
    theString = new char[strlen(other.theString) + 1];
    strcpy(theString, other.theString);
}

/* Clear-out member function. */
void Derived::clear()
{
    delete [] theString;
    theString = NULL;
}

/* Copy constructor. */
Derived::Derived(const Derived &other) // ERROR!
{
    copyOther(other);
}

/* Assignment operator. */
Derived& Derived::operator= (const Derived &other) // ERROR!
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}
```

Initially, it seems like this code will work perfectly, but, alas, it is seriously flawed. During this copy operation, we have not copied over the data from `other`'s base class into the current object's base class. As a result, we'll end up with half-copied data, where the data specific to `Derived` is correctly cloned but `Base`'s data hasn't changed. Consequently, we have a partially-copied object, which will almost certainly crash at some point down the line.

When writing assignment operators and copy constructors for derived classes, you must make sure to manually invoke the assignment operators and copy constructors for base classes to guarantee that the object is fully-copied. Fortunately, this is not a particularly difficult task. Let's first focus on the copy constructor. Somehow, we need to tell the base class that it should initialize itself as a copy of `other`'s base class. Since `other` is a class of type `Base` since it inherits from `Base`, we can pass `other` as a parameter to `Base`'s copy constructor inside the initializer list. Thus, the new version of the `Derived` copy constructor looks like this:

```
/* Copy constructor. */
Derived::Derived(const Derived &other) : Base(other) // CORRECT
{
    copyOther(other);
}
```

The code we have so far for the assignment operator correctly clears out the `Derived` part of the `Derived` class, but leaves the `Base` portion untouched. We can explicitly invoke `Base`'s assignment operator and have `Base` do its own copying work. The code for this is a bit odd and is shown below:

```
/* Assignment operator. */
Derived& Derived::operator= (const Derived &other)
{
    if(this != &other)
    {
        clear();
        Base::operator= (other); // Invoke the assignment operator from Base.
        copyOther(other);
    }
    return *this;
}
```

Here we've inserted a call to `Base`'s assignment operator using the full name of the `operator =` function. This is one of the rare situations where you will need to use the fully-qualified name of an overloaded operator function, since if you simply write `*this = other` you will call `Derived`'s version of `operator =`, causing infinite recursion.

All of the above discussion has assumed that your classes require their own assignment operator and copy constructor. However, if your derived class does not contain any data members that require manual copying and assignment (for example, a derived class that simply holds an `int`), none of the above code will be necessary. C++'s default assignment operator and copy constructor automatically invoke the assignment operator and copy constructor of any base classes, which is exactly what you'd want it to do.

Slicing

In the above example, we encountered problems when we copied over the the data of the `Derived` class but not the `Base` class. However, there's a far more serious problem we can run into called *slicing* where we copy only the base class of an object while leaving its derived classes unmodified.

Suppose we have two `Base *` pointers called `one` and `two` that point to objects either of type `Base` or of type `Derived`. What happens if we write code like `*one = *two`? Here, we're copying the value of the object pointed at by `two` into the variable pointed at by `one`. While at first glance this might seem harmless, the above statement is one of the most potentially dangerous mistakes you can make when working with C++ inheritance. The problem is that this line expands into a call to

```
one->operator =(*two);
```

Note that the version of `operator =` we're calling here is the one defined in `Base`, not `Derived`, so this line will only copy over the `Base` portion of `two` into the `Base` portion of `one`, resulting in half-formed objects that are almost certainly not in the correct state and may be entirely corrupted.

Slicing can be even more insidious in scenarios like this one:

```
void DoSomething(Base baseObject)
{
    // Do something
}

Derived myDerived
DoSomething(myDerived);
```

Recall that the parameter `baseObject` will be initialized using the `Base` copy constructor, not the `Derived` copy constructor, so the object in `DoSomething` will not be a correct copy `myDerived`. Instead, it will only hold a copy of the `Base` part of the `myDerived` object.

You should almost never assign a base class object the value of a derived class. The second you do, you will almost certainly cause runtime errors as your code tries to use incompletely-formed objects. While it may sound simple to follow this rule, at many times it might not be clear that you're slicing an object. For example, consider this code snippet:

```
vector<Base> myBaseVector;
Base *myBasePtr = someFunction();
myBaseVector.push_back(*myBasePtr);
```

Here, the object pointed at by `myBasePtr` could be of type `Base` or any type inheriting from `Base`. When we call `myBaseVector.push_back(*myBasePtr)`, there is a good chance that we will slice the object pointed at by `myBasePtr`, storing only its `Base` component in the `vector` and dropping the rest. You'll need to be extra vigilant when working with derived classes, since it's very easy to generate dangerous, difficult-to-track bugs.

The C++ Casting Operators

One of the most useful features of C++ inheritance is the ability to use an object of one type in place of another. For example, a pointer of type `Derived *` can be used whenever a `Base *` would be expected, and the conversion is automatic. However, in many circumstances, we may want to perform this conversion in reverse. Suppose that we have a pointer that's statically typed as a `Base *`, but we know that the object it points to is actually of typed `Derived *`. How can we use the `Derived` features of the pointee? Because the pointer to the object is a `Base *`, not a `Derived *`, we will have to use a typecast to convert the pointer from the base type to the derived type. Using the typecasts most familiar to us in C++, the code to perform this conversion looks as follows:

```
Base *myBasePtr; // Assume we know that this points to a Derived object.
Derived *myDerivedPtr = (Derived *)myBasePtr;
```

There is nothing wrong with the above code as written, but it contains two flaws – one minor and one major. The minor flaw concerns the use of the typecast `(Derived *)myBasePtr`. In C++, using a typecast of the form `(Type)` is extremely dangerous because there are only minimal compile-time checks to ensure that the typecast makes any sense. For example, consider the following C++ code:

```
Base *myBasePtr; // Assume we know that this points to a Derived object.
vector<double> *myVectorPtr = (vector<double> *)myBasePtr; // Uh oh!
```

This code is completely nonsensical, since there is no reasonable way that a pointer of type `Base *` can end up pointing to an object of type `vector<double>`. However, because of the explicit pointer-to-pointer typecast, this code is entirely legal. In the above case, it's extremely clear that the conversion we're performing is incorrect, but in others it might be more subtle. Consider the following code:

```
const Base* myBasePtr; // Assume we know that this points to a Derived object.
Derived *myDerivedPtr = (Derived *)myBasePtr;
```

This code again is totally legal and at first glance might seem correct, but unfortunately it contains a serious error. In this example, our initial pointer was a pointer to a `const Base` object, but in the second line we removed that `const`ness with a typecast and the resulting pointer is free to modify the object it points at. We've just subverted `const`, which could lead to a whole host of problems down the line.

The problem with the above style of C++ typecast is that it's just too powerful. If C++ can figure out a way to convert the source object to an object of the target type, it will, even if it's clear from the code that the conversion is an error. To resolve this issue, C++ has four special operators called *casting operators* that you can use to perform safer typecasts. When working with inheritance, two of these casting operators are particularly useful, the first of which is `static_cast`. The `static_cast` operator performs a typecast in the same way that the more familiar C++ typecast does, except that it checks at compile time that the cast "makes sense." More specifically, `static_cast` can be used to perform the following conversions:*

1. Converting between primitive types (e.g. `int` to `float` or `char` to `double`).
2. Converting between pointers or references of a derived type to pointers or references of a base type (e.g. `Derived *` to `Base *`) where the target is at least as `const` as the source.
3. Converting between pointers or references of a base type to pointers or references of a derived type (e.g. `Base *` to `Derived *`) where the target is at least as `const` as the source.

If you'll notice, neither of the errors we made in the previous code snippets are possible with a `static_cast`. We can't convert a `Base *` to a `vector<double> *`, since `Base` and `vector<double>` are not related to each other in an inheritance scheme. Similarly, we cannot convert from a `const Base *` to a `Derived *`, since `Derived *` is less `const` than `const Base *`.

The syntax for the `static_cast` operator looks resembles that of templates and is illustrated below:

```
Base* myBasePtr; // Assume we know this points to a Derived object.
Derived *myDerivedPtr = static_cast<MyDerived *>(myBasePtr);
```

That is, `static_cast`, followed by the type to convert the pointer to, and finally the expression to convert enclosed in parentheses.

Throughout this discussion of typecasts, when converting between pointers of type `Base *` and `Derived *`, we have implicitly assumed that the `Base *` pointer we wanted to convert was pointing to an object of type `Derived`. If this isn't the case, however, the typecast can succeed but lead to a scenario where we have a `Derived *` pointer pointing to a `Base` object, which can cause all sorts of problems at runtime when we try to invoke nonexistent member functions or access data members of the `Derived` class that aren't present in `Base`. The problem is that the `static_cast` operator doesn't check to see that the typecast it's performing makes any sense at runtime. To provide this functionality, you can use another of the C++ casting operators, `dynamic_cast`, which acts like `static_cast` but which performs additional checks before performing the cast. `dynamic_cast`, like `static_cast`, can be used to convert between pointer types related by inheritance (but not to convert between primitive types). However, if the specific typecast requested of `dynamic_cast` is invalid at runtime (e.g. attempting to convert a `Base` object to a `Derived` object), `dynamic_cast` will return a `NULL` pointer instead of a valid pointer to the derived type. For example, consider the following code:

---

* There are several other conversions that you can perform with `static_cast`, especially when working with `void *` pointers, but we will not discuss them here.

```
Base *myBasePtr = new Base;
Derived *myDerivedPtr1 = (Derived *)myBasePtr;
Derived *myDerivedPtr2 = static_cast<Derived *>(myBasePtr);
Derived *myDerivedPtr3 = dynamic_cast<Derived *>(myBasePtr);
```

In this example, we use three different typecasts to convert a pointer that points to an object of type `Base` to a pointer to a `Derived`. In the above example, the first two casts will perform the type conversion, resulting in pointers of type `Derived *` that actually point to a `Base` object, which can be dangerous. However, the final typecast, which uses `dynamic_cast`, will return a `NULL` pointer because the cast cannot succeed.

When performing downcasts (casts from a base type to a derived type), unless you are absolutely sure that the cast will succeed, you should consider using `dynamic_cast` over a `static_cast` or raw C++ typecast. Because of the extra check at runtime, `dynamic_cast` is slower than the other two casts, but the extra safety is well worth the cost.

There are two more interesting points to take note of when working with `dynamic_cast`. First, you can only use `dynamic_cast` to convert between types if the base class type contains at least one virtual member function. This may seem like a strange requirement, but greatly improves the efficiency of the language as a whole and makes sense when you consider that it's rare to hold a pointer to a `Derived` in a pointer of type `Base` when `Base` isn't polymorphic. The other important note is that if you use `dynamic_cast` to convert between *references* rather than pointers, `dynamic_cast` will throw an object of type `bad_cast` rather than returning a "`NULL` reference" if the cast fails. Consult a reference for more information on `bad_cast`.

More to Explore

C++ inheritance is an enormous topic with all sorts of nuances, far more than we can reasonably cover in a single handout. If you're interested, here are some topics you might want to consider reading into:

1. **Private Inheritance**: Have you ever wondered why when inheriting from a base class you write `public` before the name of the class? It's because there is another form of inheritance called *private inheritance* by writing `private` and then the class name. Private inheritance is fundamentally different from public inheritance and represents the "is-implemented-in-terms-of" relationship instead of "is-a." Private inheritance is uncommon in practice, but you should be aware that it exists.
2. **The Curiously Recurring Template Pattern** (CRTP): Virtual functions make your programs run slightly slower than programs with non-virtual functions because of the extra overhead of the dynamic lookup. In certain situations where you want the benefits of inheritance without the cost of virtual functions, you can use an advanced C++ trick called the *curiously recurring template pattern*, or CRTP. The CRTP is also known as "static interfacing" and is an advanced programming technique, but it might be worth looking into if you plan on pursuing C++ professionally.
3. **`const_cast` and `reinterpret_cast`**: C++ has two other conversion operators, `const_cast`, which can add or remove `const` from a pointer, and `reinterpret_cast`, which performs fundamentally unsafe typecasts (such as converting an `int *` to a `string *`). While the use cases of these operators are far beyond the scope of this class, they do arise in practice and you should be aware of their existences. Consult a reference for more information.

Practice Problems

1. A common mistake is to try to avoid problems with slicing by declaring `operator =` as a virtual function in a base class. Why won't this solve the slicing problem? *(Hint: what is the parameter type to `operator =`?)*
2. Suppose you have two classes, `Base` and `Derived`, where `Derived` inherits from `Base`. Suppose you have an instance of `Base` called `base` and an instance of `Derived` called `derived`. Further suppose that both `Base` and `Derived` have their own assignment operators. Explain why the line `derived = base` won't compile even though `derived` inherits an `operator =` function that takes a `const Base&` as a parameter. *(Hint: think about name resolution.)*
3. Suppose you have three classes, `Base`, `Derived`, and `VeryDerived` where `Derived` inherits from `Base` and `VeryDerived` inherits from `Derived`. Assume all three have copy constructors and assignment operators. Inside the body of `VeryDerived`'s assignment operator, why shouldn't it invoke `Base::operator =` on the other object? What does this tell you about long inheritance chains, copying, and assignment?
4. The C++ casting operators were deliberately designed to take up space to discourage programmers from using typecasts. Explain why the language designers frown upon the use of typecasts.