Inheritance Topics

Introduction

C++ inheritance is an enormously powerful mechanism 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.

Overloading versus Overriding

A great source of C++ confusion is the difference between function *overloads* and function *overrides*. A function *overload* is a function with the same name as another function but with different parameters. So, for example, the following two functions are overloads of one another:

```
void MyFunction();
void MyFunction(int x);
```

On the other hand, you *override* a function by marking it virtual and then redefining it in a derived class. For example, given the following two class definitions:

```
class BaseClass
{
    public:
        virtual void fn();
};

class DerivedClass: public BaseClass
{
    public:
        virtual void fn();
};
```

The function `DerivedClass::fn` overrides the function `BaseClass::fn`.

When working with virtual functions and runtime polymorphism, it's essential to remember that overloading and overriding are *not* the same. To override a function, you must provide another function with the same parameters and return type as the original function. Otherwise, you're providing an overload, and C++ will treat your function as a new function introduced in a derived class rather than a function overriding parent class member function. Consequently, C++ will use the superclass version of the function instead of the subclass version, which will cause no shortage of debugging headaches.

For those of you with experience in Java, you may have heard of the Java `@Override` directive that causes a compile-time error if the specified function isn't an override of a base class function. Unfortunately, C++ does not have this functionality, so you will have to manually verify that all base

class functions are indeed overridden. While this might seem simple at first, C++ is ripe with situations where something that looks like a function override is instead a function overload. For example, consider these two classes:

```
class Base
{
    public:
        virtual double doSomething(const string &str, int paramTwo) const;
};

class Derived: public Base
{
    public:
        virtual double doSomething(const string &str, int paramTwo);
};
```

While you might think that `Derived` has overridden `Base`'s `doSomething` function, unfortunately `Derived`'s version of `doSomething` is an overload since we have forgetten to mark it `const`. Unless `doSomething` is a purely virtual function in `Base`, you won't get any compile time errors and will have to use nonsensical runtime behavior to make the diagnosis.

Name Resolution and C++ Classes

Name resolution is the process by which the C++ compiler figures out which constructs you're referring to when using variable, class, or function names. For example, consider the following code:

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

Consider the line `cout << x << endl;` During name resolution, C++ will determine which version of `x` to print out (e.g. is it the global constant, the data member, or the local variable?).

In many ways, C++'s name resolution system is wonderful, but when working with classes, the name resolution system can result in unusual and undesired behavior. Let's consider two classes, `CardGame` and `PokerGame`, which are defined below:

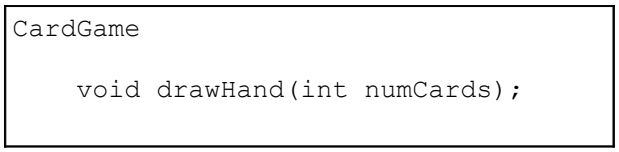
```
class CardGame
{
    public:
        /* Draws the specified number of cards into a new hand. */
        void drawHand(int numCards);
    private:
        /* Extra data */
};
```

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

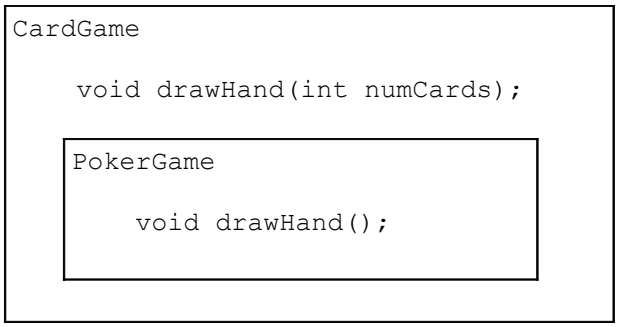
Nothing about this setup seems unreasonable – the `CardGame` superclass has generic functionality to draw cards while the `PokerGame` subclass exports a convenience `drawHand` function that always draws five cards. By itself there's nothing wrong with this code, and the above declarations will compile and link, but if we try to use a `PokerGame` object we will start to 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 would be legal. Unfortunately, it's not, and we'll get a compiler error telling us that `drawHand` 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 that encapsulates 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++ simply needs to step 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 progressively steps outward until it finds the object. Normally, this is exactly what we want, but when working with function overloads it poses a problem because C++ scoping rules dictate that the compiler must stop looking for functions after it finds *any* function whose name matches the provided name, even if it doesn't have enough parameters. 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. Since the zero-parameter function doesn't accept any parameters, it reported a compiler error.

Somehow we need to tell C++ that we want it to continue searching 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 admittedly clunky and forces clients of the `PokerGame` class to know about the `CardGame` superclass. Let's consider an alternative, more elegant solution. What we want to do is tell the compiler that it should 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, you'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` can be used in other places and instructs 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.
};
```

This will correctly solve the problem and the compiler will find the original version of the function.

Invoking Virtual Member Functions Non-Virtually

Normally, you'll use virtual functions to override older, less specific versions of a function inside a derived class. However, from time to time you'll need to be able to invoke a superclass's version of a virtual function. For example, suppose that you're designing a `HybridCar` class that's a specialization of the `Car` class, 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, it seems as though 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` we'll also have to change the code inside `HybridCar` or we'll end up with a `HybridCar` whose behavior differs from the regular `Car`. 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 virtual function call the superclass'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. By doing so, we're telling C++ that we want the version specific to a certain class, not the version specific to the current class. Using this, we can write a possible implementation of `HybridCar::applyBrakes`, as shown here:

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

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()`.

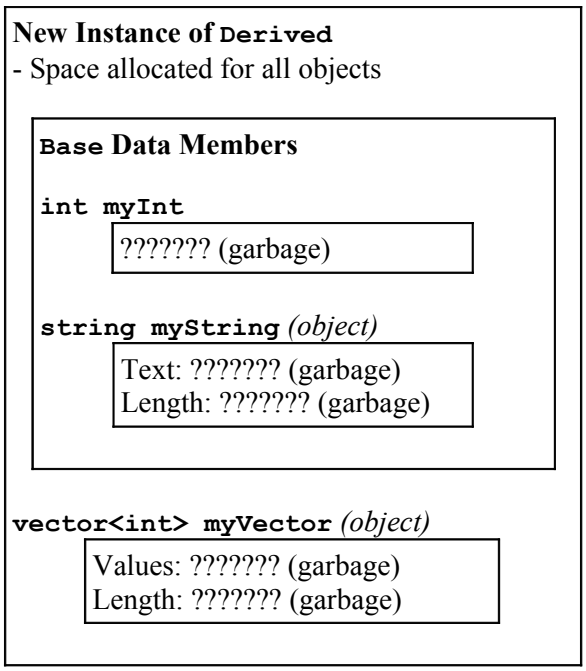
Object Initialization in Derived Classes

Recall from several weeks ago that when C++ constructs a new instance of a class, it does so 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`:

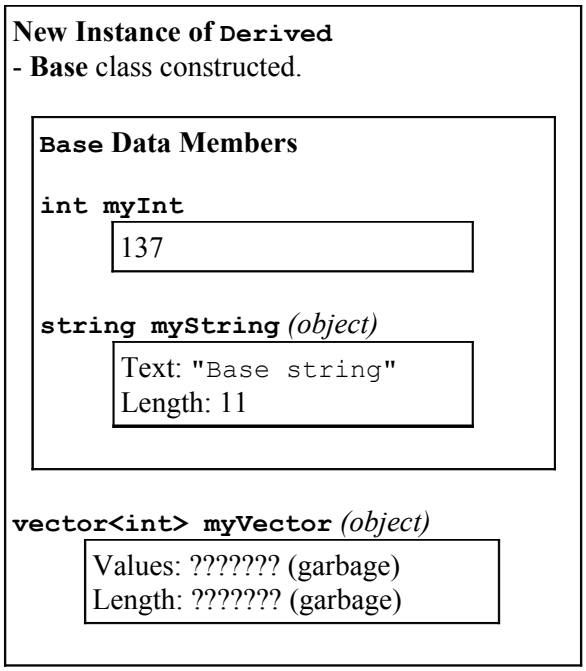
```
class Base
{
    public:
        Base() : myInt(137), myString("Base string!") {}
    private:
        int myInt;
        string myString;
};

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

When we construct a new instance of a `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, rather than initializing the `Derived` class's data members, C++ will instead initialize the `Base` class using its default constructor. Similarly, if `Base` has any parent classes, those parent classes would also be initialized. After this step, the object now looks something 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. Like the default constructors for data members, however, in many circumstances you won't want to use a base class's default constructor. Fortunately, just as you can invoke a data member's constructor in an initializer list, you can invoke a custom constructor for a base class 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 of any parent classes to ensure 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 to be 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);
}
```

As for the assignment operator, the code we have so far correctly clears out the `Derived` part of the `Derived` class, but leaves the `Base` portion untouched. But since the `Base` class knows best how to copy its own data members, 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 =`, leading to infinite recursion and a stack-heap collision.

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 superclasses, which fortunately 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 of the object pointed at by `one`. Instead, it will only hold a copy of the `Base` part of the `Derived` object.

To avoid slicing, you must remember that 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;
Derived myDerived;
myBaseVector.push_back(myDerived);
```

Here, since `myDerived` is an object of type `Derived` which is derived from `Base`, the line `myBaseVector.push_back(myDerived)` will compile without errors, but will slice the `myDerived` object and only copy over the `Base` portion of `myDerived`. You must be vigilant to avoid slicing, or you'll run into some incredibly subtle yet catastrophically dangerous bugs.

More to Explore

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

1. **Multiple Inheritance**: One of C++'s most complicated and most contested feature is multiple inheritance, the ability to have a class inherit from two or more superclasses. Used correctly, multiple inheritance can actually be quite useful, especially if some of the superclasses are purely abstract base classes. Otherwise, you'll run into disambiguation issues, `virtual` inheritance, and the infamous "deadly diamond of death." Nearly all professional C++ programmers advise against multiple inheritance, but it's important to know about because it can be a useful tool.
2. **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 it's possible to use something called *private*

inheritance by writing `private` and then the class name. Private inheritance is fundamentally different from public inheritance and means that all public data members and member functions of the base class become private members of the derived class. Private inheritance is useful from time to time, so consult a reference for more information.

3. **Private Virtual Functions**: It's possible to declare a virtual function `private`, indicating that any derived class can provide an implementation for the function although the derived classes are not permitted to call the function themselves. As with all the above features, it's worth knowing about private virtual functions, but they're only occasionally useful.
4. **`dynamic_cast`**: C++ has four "casting operators," operators that perform typecasts from one type to another. One of the most interesting casts is `dynamic_cast`, which casts a pointer from a base type to a derived type and returns `NULL` if the cast can't be completed. `dynamic_cast` is related to C++'s Runtime Type Identification (RTTI) system, so be sure to consult a reference for more information.
5. **Protected Constructors**: Classes can declare constructors `protected` so that only class instances and derived classes can access them. This is a useful technique for preventing instantiation of abstract base classes.

Practice Problems

1. Consider four classes `A`, `B`, `C`, and `D` such that `A` is the parent of `B`, which is the parent of `C`, which is the parent of `D`. Each class exports a function `virtual void fn()`. Write a member function `showcase` for `D` that calls each class's version of the `fn` function.
2. 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 =`?)*
3. 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.
4. 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?