Introduction to Inheritance

Introduction

It's impossible to learn C++ or any other object-oriented language without encountering *inheritance*, a mechanism that lets different classes share implementation and interface design. Inheritance, however, has evolved greatly since it was first introduced to C++, and consequently C++ supports several different inheritance schemes. Although it is fascinating material, a full treatment of all of these inheritance schemes is far beyond the scope of this class.

This handout attempts get you up to speed on various inheritance patterns rather than focusing on some of the language complexities and nuances of C++ inheritance – that's a topic for Wednesday's lecture. When necessary, however, we will discuss relevant C++ syntax, specifically the `virtual` and `protected` keywords and the `= 0` notation.

Inheritance of Implementation and Interface

C++ started off as a language called "C with Classes," so named because it was essentially the C programming language with support for classes and object-oriented programming. C++ is the more modern incarnation of C with Classes, so most (but not all) of the features of C with Classes also appear in C++.

The inheritance introduced in C with Classes allows you to define new classes in terms of older ones. For example, suppose you are using a third-party library that exports a `Printer` class, as shown below:

```
class Printer
{
public:
    /* Constructor, destructor, etc. */

    void setFont(const string &fontName, int size);
    void setColor(const string &color);
    void printDocument(const string& document);
private:
    /* Implementation details */
};
```

This `Printer` class exports several formatting functions, plus `printDocument`, which accepts a string of the document to print. Let's assume that `printDocument` is implemented synchronously – that is, `printDocument` will not return until the document has finished printing. In some cases this behavior is fine, but in others it's simply not acceptable. For example, suppose you're writing database software for a large library and want to give users the option to print out call numbers. Chances are that people using your software will print call numbers for multiple books and will be irritated if they have to sit and wait for their documents to finish printing before continuing their search. To address this problem, you decide to add a new feature to the printer that lets the users enqueue several documents and print them in a single batch job. That way, users searching for books can enqueue call numbers without long pauses.

However, you don't want to force users to queue up documents and then do a batch print job at the end – after all, maybe they're just looking for one book – so you want to retain all of the original features of the `Printer` class. How can you elegantly solve this problem in software?

Let's consider the above problem from a programming perspective. The important points are:

- We are provided the `Printer` class from an external source, so we cannot modify the `Printer` interface.
- We want to preserve all of the existing functionality from the `Printer` class.
- We want to extend the `Printer` class to include extra functionality.

This is an ideal spot to use inheritance. We have an existing class that contains most of our needed functionality, but we'd like to add some extra features. This is the inheritance that C with Classes initially supported and is the simplest form of C++ inheritance.

Let's define a `BatchPrinter` class that supports two new functions, `enqueueDocument` and `printAllDocuments`, on top of all of the regular `Printer` functionality. In C++, we write this as

```
class BatchPrinter: public Printer // Inherit from Printer
{
public:
    void enqueueDocument(const string &document);
    void printAllDocuments();
private:
    queue<string> documents; // Document queue
};
```

Here, the class declaration `class BatchPrinter: public Printer` indicates that the new class `BatchPrinter` inherits the functionality of the `Printer` class. Although we haven't explicitly provided the `printDocument` or `setFont` functions, since those functions are defined in `Printer`, they are also part of `BatchPrinter`. For example:

```
BatchPrinter myPrinter;
myPrinter.setColor("Red");                      // Inherited from Printer
myPrinter.printDocument("This is a document!"); // Same
myPrinter.enqueueDocument("Print this one later."); // Defined in BatchPrinter
myPrinter.printAllDocuments();                      // Same
```

While the `BatchPrinter` can do everything that a `Printer` can do, the converse is not true – a `Printer` cannot call `enqueueDocument` or `printAllDocuments`, since we did not modify the `Printer` class interface.

In the above scenario, `Printer` is called a *base class* of `BatchPrinter`, which is a *derived class*. In computer science jargon, the relationship between a derived class and its base class is the *is-a* relationship. That is, a `BatchPrinter` *is-a* `Printer` because everything the `Printer` can do, the `BatchPrinter` can do as well. The converse is not true, though, since a `Printer` is not necessarily a `BatchPrinter`.

Note that it is completely legal to have several classes inherit from a single base class. Thus, if we wanted to develop another printer that supported graphics printing in addition to text, we could write the following class definition:

```
class GraphicsPrinter: public Printer
{
public:
    /* Constructor, destructor, etc. */

    void printPicture(const Picture& picture); // For some Picture class
private:
    /* Implementation details */
};
```

Now, `GraphicsPrinter` can do everything a regular `Printer` can do, but can also print `Picture` objects. Again, `GraphicsPrinter` *is-a* `Printer`, but not vice-versa. Similarly, `GraphicsPrinter` is not a `BatchPrinter`. Although they are both derived classes of `Printer`, they have nothing else in common.

Runtime Costs of Basic Inheritance

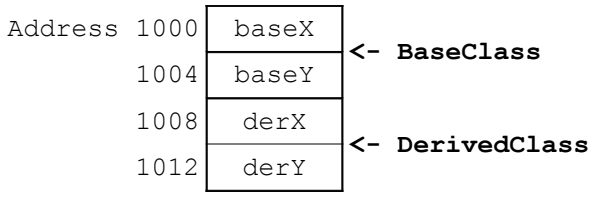
The inheritance scheme outlined above incurs no runtime penalties. Programs using this type of inheritance will be just as fast as programs not using inheritance.

In memory, a derived class is simply a base class object with its extra data members tacked on the end. For example, suppose you have the following classes:

```
class BaseClass
{
private:
    int baseX, baseY;
};

class DerivedClass
{
private:
   int derX, derY;
};
```

Then, in memory, a `DerivedClass` object looks like this:



Inheritance of Interface

The inheritance strategy outlined above works quite well in a wide number of circumstances and is still used widely in modern C++.* However, this version of inheritance simply isn't powerful enough to address several real-world problems.

---

* In fact, a recently-developed programming technique called *policy classes* is rooted in this form of simple inheritance.

Let's return to the `Printer` class. `Printer` exports a `printDocument` member function that accepts a `string` parameter, then sends the string to the printer. One of our other derived classes, `GraphicsPrinter`, has a `printPicture` member function that accepts some sort of `Picture` object, then sends the picture to the printer. What if we want to print a document containing a mix of text and pictures – for example, this handout? We'd then need to introduce yet another subclass of `Printer`, perhaps a `MixedTextPrinter`, that supports a `printMixedText` member function. This approach has several problems. First, each printer can only print out one type of document. That is, a `MixedTextPrinter` cannot print out pictures, nor a `GraphicsPrinter` a mixed-text document. We could eliminate this problem by writing a single `MixedTextAndGraphicsPrinter` class, but this too has its problems if we then introduce another type of object to print (say, a high-resolution photo). This leads to the second problem, a lack of extensibility. For any new type of object we want to print, we need to introduce another member function or class capable of handling that object. In our case this is inconvenient and does not scale well. We need to pick another plan of attack.

To solve this problem, let's consider exactly how a printer works. Many printers are programmed to accept as input a grid of dots representing the picture to print. That is, whether you're printing text or a three-dimensional pie chart, the input to the printer is a grid of pixels representing the dots making up the image. Provided that we can transform an object in memory into a mess of pixels, we can send it to the printer. Consider the following class definition for a `GenericBatchPrinter` object:

```
class GenericBatchPrinter
{
public:
    /* Constructor, destructor, etc. */

    void enqueueDocument( /* What goes here? */ );
    void printAllDocuments();
private:
    /* Implementation details */
};
```

This `GenericBatchPrinter` object exports an `enqueueDocument` function that stores a document in a print queue that can then be printed by calling `printAllDocuments`. There is one major question, though – what should the parameter type be? We can print any type of object we can think of, provided that we can convert it into a grid of pixels. We might be tempted to accept a grid of pixels as a parameter. This, however, has several drawbacks. First, pixel grids take up a huge amount of memory. Color printers usually store color information as quadruples of the cyan, magenta, yellow, and black (CYMK) color components, so a single pixel is usually a four-byte value. If you have a 200 DPI printer and want to print to an 8.5 x 11" page, you'd need to store approximately half a megabyte of graphics information. That's an awful lot of memory, and if you wanted to enqueue a large number of documents this approach might strain or exhaust system resources. Plus, we don't actually need the pixels until we begin printing, and even then we only need pixel information for one document at a time. Second, what if later in design we realize that we need extra information about the print job? For example, suppose we want to implement a printing priority system where more urgent documents print before less important ones. In this case, we'd need to add an extra parameter to `enqueueDocument` and all existing code using `enqueueDocument` would stop working. Finally, this approach exposes too much of the inner workings of `GenericBatchPrinter` to the client. By treating documents as masses of pixels instead of documents, the `GenericBatchPrinter` violates some of the fundamental rules of data abstraction.

Let's review these problems:

- The above approach is needlessly memory-intensive by catering to the lowest common denominator of all possible printable documents.
- The approach limits later extensions by fixing the parameter as an inflexible pixel array.
- The `GenericBatchPrinter` should work on documents, not pixels.

Is there a language feature that would let us solve all of these problems? The answer is yes. What if we simply create an object that looks like this:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    int* convertToPixelArray() const; // int* represents a pixel array
    string getDocumentName() const; // Sample extra information function
private:
    /* Implementation details */
};
```

Using this object, we can write `enqueueDocument` as a function accepting a `Document` object. That way, when the `GenericBatchPrinter` needs to get an array of pixels, it can simply call `convertToPixelArray` on any stored `Document` objects. Similarly, if the `GenericBatchPrinter` needs any extra information, we simply need to add some extra member functions to the `Document` class. While this still requires us to rewrite code to add these member functions, the actual calls to `enqueueDocument` will still work correctly, and the only people who need to modify any code are the `Document` class implementers, not the `Document` class clients.

While this solution might seem elegant, it still has a major problem – how can we write a `Document` class that encompasses all of the possible documents we can try to print? The answer is simple: we can't. Using the language features we've covered so far, it simply isn't possible to solve this problem.

Consider for a minute what form our problem looks like. We need to provide a `Document` object that represents a printable document, but we cannot write a single umbrella class representing every conceivable document. Instead of creating a single `Document` class, what if we could create several different `Document` classes, each of which provided a working implementation of the `convertToPixelArray` and `getDocumentName` functions? That is, we might have a `TextDocument` class that stores a `string` and whose `convertToPixelArray` converts the string into a grid of pixels representing that string. We could also have a `GraphicsDocument` object with member functions like `addCircle` or `addImage` whose `convertToPixelArray` function generates a graphical representation of the stored image. In other words, we want to make the `Document` class represent an *interface* rather than an *implementation*. `Document` should simply outline what member functions are common to other classes like `GraphicsDocument` or `TextDocument` without specifying how those functions should work.

In C++ code, this means that we will rewrite the `Document` class to look like this:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual int* convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    /* Implementation details */
};
```

If you'll notice, we tagged both of the member functions with the `virtual` keyword, and put an `= 0` after each function declaration. What does this strange syntax mean? The `= 0` syntax is an odd bit of C++ syntax that says "this function does not actually exist." In other words, we've prototyped a function that we have no intention of ever writing. Why would we ever want to do this? The reason is simple. Because we've prototyped the function, other pieces of C++ code know what the parameter and return types are for the `convertToPixelArray` and `getDocumentName` functions. However, since there is no meaningful implementation for either of these functions, we add the `= 0` to tell C++ not to expect one.

To understand the `virtual` keyword, consider this `TextDocument` class outlined below:

```
class TextDocument: public Document // Inherit from Document
{
public:
    /* Constructor, destructor, etc. */

    int* convertToPixelArray() const; // Has an actual implementation
    string getDocumentName() const;   // Has an actual implementation

    void setText(const string &text); // Text-specific formatting functions
    void setFont(const string &font);
    void setSize(int size);
private:
    /* Implementation details */
};
```

This `TextDocument` class inherits from `Document`, but unlike `Document`, its `convertToPixelArray` and `getDocumentName` functions have actual implementations (not included in this handout). Thus we can write code like this:

```
TextDocument myDocument;
int *array = myDocument.convertToPixelArray();
```

The `virtual` keyword comes into play when we start accessing `TextDocument` objects through pointers. Consider the following code snippet:

```
TextDocument *myDocument = new TextDocument;
int *array = myDocument->convertToPixelArray();
```

This code should be somewhat clear – we allocate a new `TextDocument` object, then store it in a pointer called `myDocument`, then call the `convertToPixelArray` function on this new object. However, consider this code snippet below:

```
Document *myDocument = new TextDocument; // Note: pointer is a Document *
int *array = myDocument->convertToPixelArray();
```

This code looks similar to the above code but represents a fundamentally different operation. In the first line, we allocate a new `TextDocument` object, but store it in a pointer of type `Document *`. Initially, this might seem nonsensical – pointers of type `Document *` should only be able to point to objects of type `Document`. However, because `TextDocument` is a derived class of `Document`, *`TextDocument` is-a `Document`*. The is-a relation applies literally here – since `TextDocument` is-a `Document`, we can point to objects of type `TextDocument` using pointers of type `Document *`.

Even if we can point to objects of type `TextDocument` with objects of type `Document *`, why is the line `myDocument->convertToPixelArray()` legal? As mentioned earlier, the `Document` class definition states that `convertToPixelArray` does not actually exist as a function, so this code should not compile. This is where the `virtual` keyword comes in. Since we marked `convertToPixelArray` `virtual`, when we call the `convertToPixelArray` function through a `Document *` object, C++ will call the function named `convertToPixelArray` for the class that's *actually being pointed at*, not the `convertToPixelArray` function defined for objects of the type of the pointer. In this case, since our `Document *` is pointing at a `TextDocument`, the call to `convertToPixelArray` will call the `TextDocument`'s version of `convertToPixelArray`.

The above approach to the problem is known as *polymorphism*. We define a base class (in this case `Document`) that exports several functions marked `virtual`. In our program, we pass around pointers to objects of this base class, which may in fact be pointing to a base class object or to some derived class. Whenever we make member function calls to the virtual functions of the base class, C++ figures out at runtime what the version of the function to call.

Let's return to our `GenericBatchPrinter` class, which now in its final form looks something like this:

```
class GenericBatchPrinter
{
public:
    /* Constructor, destructor, etc. */

    void enqueueDocument(Document *doc);
    void printAllDocuments();
private:
    queue<Document *> documents;
};
```

Our `enqueueDocument` function now accepts a `Document *`, and its private data members include an STL `queue` of `Document *`s. We can now implement `enqueueDocument` and `printAllDocuments` using code like this:

```
void GenericBatchPrinter::enqueueDocument(Document *doc)
{
    documents.push(doc); // Recall STL queue uses push instead of enqueue
}
```

```
void GenericBatchPrinter::printAllDocuments()
{
    /* Print all enqueued documents */
    while(!documents.empty())
    {
        Document *nextDocument = documents.front();
        documents.pop(); // Recall STL queue requires explicit pop operation

        cout << "Printing document: ";
        cout << nextDocument->getDocumentName() << endl;
        sendToPrinter(nextDocument->convertToPixelArray());
        delete nextDocument; // Assume it was allocated with new
    }
}
```

The above code is not particularly complex. The `enqueueDocument` function simply takes a new `Document *` and enqueues it in the document queue, and the `printAllDocuments` function continuously dequeues documents, converts them to pixel arrays, then sends them to the printer. But while this above code might seem simple, it's actually working some wonders behind the scenes. Notice that when we call `nextDocument->getDocumentName()` and `nextDocument->convertToPixelArray()`, the object pointed at by `nextDocument` could be of any type derived from `Document`. That is, the above code will work whether we've enqueued `TextDocument`s, `GraphicsDocument`s, or even `MixedTextDocument`s. Moreover, the `GenericBatchPrinter` class does not even need to know of the existence of these types of documents; as long as `GenericBatchPrinter` knows the generic `Document` interface, C++ can determine which functions to call.

Virtual Functions, Pure Virtual Functions, and Abstract Classes

In the above example with the `Document` class, we defined `Document` as

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual int* convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    /* Implementation details */
};
```

Here, all of the `Document` member functions are marked `virtual` and have the `= 0` syntax to indicate that the functions are not actually defined. Functions marked `virtual` with `= 0` are called *pure virtual functions* and represent functions that exist solely to define how other pieces of C++ code should interact with derived classes.*

---

* Those of you familiar with inheritance in other languages like Java might wonder why C++ uses the awkward `= 0` syntax instead of a clearer keyword like `abstract` or `pure`. The reason was mostly political. Bjarne Stroustrup introduced pure virtual functions to the C++ language several weeks before the planned release of the next set of revisions to C++. Adding a new keyword would have delayed the next language release, so to ensure that C++ had support for pure virtual functions, he chose the `= 0` syntax.

Classes that contain pure virtual functions are called *abstract classes*. Because abstract classes contain code for which there is no implementation, it is illegal to directly instantiate abstract classes. In the case of our document example, this means that both of the following are illegal:

```
Document myDocument; // Error!
Document *myDocument = new Document; // Error!
```

Of course, it's still legal to declare `Document *` variables, since those are pointers to abstract classes rather than abstract classes themselves.

A derived class whose base class is abstract may or may not implement all of the pure virtual functions defined in the base class. If the derived class does implement each function, then the derived class is non-abstract (unless, of course, it introduces its own pure virtual functions). Otherwise, if there is at least one pure virtual function declared in the base class and not defined in the derived class, the derived class itself will be an abstract class.

There is no requirement that functions marked `virtual` be pure virtual functions. That is, you can provide `virtual` functions that have implementations. For example, consider the following class representing a roller-blader:

```
class RollerBlader
{
public:
    /* Constructor, destructor, etc. */

    virtual void slowDown(); // Virtual, not pure virtual
private:
    /* Implementation details */
};

void RollerBlader::slowDown() // Implementation doesn't have virtual keyword
{
    applyBrakes();
}
```

Here, `slowDown` is implemented as a virtual function that is not pure virtual. In the implementation of `slowDown`, you do not repeat the `virtual` keyword, and for all intents and purposes treat `slowDown` as a regular C++ function. Now, suppose we write a `InexperiencedRollerBlader` class, as shown here:

```
class InexperiencedRollerBlader: public RollerBlader
{
public:
    /* Constructor, destructor, etc. */

    void slowDown();
private:
    /* Implementation details */
};

void InexperiencedRollerBlader::slowDown()
{
   fallDown();
}
```

This `InexperiencedRollerBlader` class provides its own implementation of `slowDown` that calls some `fallDown` function.* Now, consider the following code snippet:

```
RollerBlader *blader = new RollerBlader;
blader->slowDown();

RollerBlader *blader2 = new InexperiencedRollerBlader;
blader2->slowDown();
```

In both cases, we call the `slowDown` function through a pointer of type `RollerBlader *`, so C++ will call the version of `slowDown` for the class that's actually pointed at. In the first case, this will call the `RollerBlader`'s version of `slowDown`, which calls `applyBrakes`. In the second, since `blader2` points to an `InexperiencedRollerBlader`, the `slowDown` call will call `InexperiencedRollerBlader`'s `slowDown` function, which then calls `fallDown`.

When inheriting from non-abstract classes that contain virtual functions, there is no requirement to provide your own implementation of the virtual functions.

A Word of Warning

Consider the following two classes:

```
class NotVirtual
{
public:
    void notAVirtualFunction();
};

class NotVirtualDerived: public NotVirtual
{
public:
    void notAVirtualFunction();
};
```

Here, the base class `NotVirtual` exports a function called `notAVirtualFunction` and its derived class, `NotVirtualDerived`, also provides a `notAVirtualFunction` function. Although these functions have the same name, since `notAVirtualFunction` is not marked `virtual`, the derived class version does *not* replace the base class version. Consider this code snippet:

```
NotVirtual *nv = new NotVirtualDerived;
nv->notAVirtualFunction();
```

Here, since `NotVirtualDerived` is-a `NotVirtual`, the above code will compile. However, since `notAVirtualFunction` is (as its name suggests) not a virtual function, the above code will call the `NotVirtual` version of `notAVirtualFunction`, not `NotVirtualDerived`'s `notAVirtualFunction`.

If you want to let derived classes override functions in a base class, you *must* mark the base class's function `virtual`. Otherwise you'll get some pretty strange runtime behavior when you consistenly invoke a base class's version of the function. Scott Meyers, the highly-esteemed C++ expert, mentions in

---

* Of course, this is not based on personal experience. ☺

his *Effective C++* that you should "Never redefine an inherited non-virtual function."* Doing so is just asking for trouble.

The `protected` Access Specifier

Let's return to the `Document` class from earlier in the handout. Suppose that while designing some of the `Document` subclasses, we note that every single subclass ends up having a `width` and `height` field. To minimize code duplication, we decide to move the `width` and `height` fields from the derived classes into the `Document` base class. Since we don't want `Document` class clients directly accessing these fields, we decide to mark them `private`, as shown here:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual int* convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    int width, height; // Warning: slight problem here
};
```

However, by moving `width` and `height` into the `Document` base class, we've accidentally introduced a problem into our code. Since `width` and `height` are `private`, even though `TextDocument` and the other subclasses inherit from `Document`, the subclasses will not be able to access the `width` and `height` fields. We want the `width` and `height` fields to be accessible only to the derived classes, but not to the outside world. Using only the C++ we've covered up to this point, this is impossible. However, there is a third access specifier beyond `public` and `private` called `protected` that does exactly what we want. Data members and functions marked `protected`, like `private` data members, cannot be accessed except by the class itself. However, unlike `private` variables, `protected` functions and data members are accessible by derived classes.

`protected` is a useful access specifier that in certain circumstances can make your code quite elegant. However, you should be very careful when granting derived classes `protected` access to data members. Like `public` data members, using `protected` data members locks your classes into a single implementation and can make code changes down the line difficult to impossible. Make sure that marking a data member `protected` is truly the right choice before proceeding.

Virtual Destructors

To conclude this discussion on inheritance, we need to cover one last topic – *virtual destructors*. Consider the following two classes:

```
class BaseClass
{
public:
    BaseClass();
    ~BaseClass();
};
```

---

* Taken from *Effective C++, Third Edition* by Scott Meyers. Published by Addison-Wesley, 2005. This book is a veritable gold mine of C++ information and I strongly encourage you to pick up a copy.

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();
private:
   char *myString;
};

DerivedClass::DerivedClass()
{
    myString = new char[128]; // Allocate some memory
}

DerivedClass::~DerivedClass()
{
    delete [] myString; // Deallocate the memory
}
```

Here, we have a trivial constructor and destructor for `BaseClass`. `DerivedClass`, on the other hand, has a constructor and destructor that allocate and deallocate a block of memory. What happens if we write the following code?

```
BaseClass *myClass = new DerivedClass;
delete myClass;
```

Intuitively, you'd think that since `myClass` points to a `DerivedClass` object, the `DerivedClass` destructor would invoke and clean up the dynamically-allocated memory. Unfortunately, this is not the case. Since the `myClass` pointer is statically-typed as a `BaseClass *`, calling `delete myClass` will only invoke the `BaseClass` destructor, leaving `DerivedClass`'s memory orphaned.

C++ has this somewhat strange behavior because we didn't let C++ know that it should check to see if the object pointed at by a `BaseClass *` is really a `DerivedClass`. To fix this, we mark the `BaseClass` destructor `virtual`. Unlike the other virtual functions we've encountered, though, derived class destructors do not replace the base class destructors. Instead, when invoking a destructor virtually, C++ will first call the derived class destructor, then the base class destructor. This might seem strange, but is critically important because it's the job of the base class destructor to clean up the base class. We thus change the two class declarations to look like this:

```
class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass();
};

class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();
private:
   char *myString;
};
```

There is one more point to address here, the *pure virtual destructor*. Because virtual destructors do not act like regular virtual functions, even if you mark a destructor pure virtual, you must provide an implementation. Thus, if we rewrote `BaseClass` to look like

```
class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass() = 0;
};
```

We'd then need to write a trivial implementation for the `BaseClass` destructor, as shown here:

```
BaseClass::~BaseClass()
{
    // Do nothing
}
```

This is an unfortunate language quirk, but you should be aware of it since this will almost certainly come up in the future.

Runtime Costs of Virtual Functions

Whenever you call a virtual function through a pointer to a base class, at runtime, C++ needs to perform a lookup to determine which version of the virtual function to call. This lookup takes time, and consequently code that uses polymorphic classes tends to be a bit slower than normal. Thus, make sure not to haphazardly make all of your classes polymorphic just in case you end up inheriting from them.

Practice Problems

1. In the `GenericBatchPrinter` example from earlier in this handout, why don't we need to worry that the `Document *` pointer from the `queue` points to an object of type `Document`?
2. With the exception of the IOStream library, none of the C++ Standard Library classes are polymorphic. Why might this be? *(Hint: see the previous section)*
3. In the next exercises, we'll explore a set of classes that let you build and modify functions at runtime using tools similar to those in the STL `<functional>` programming library.

   Consider the following abstract class:

   ```
   class Function
   {
   public:
       virtual ~Function() = 0;
       virtual double evaluateAt(double value) = 0;
   };
   ```

   This class exports a single function, `evaluateAt`, that accepts a `double` as a parameter and returns the value of some function evaluated at that point. Write a derived class of `Function`, `SimpleFunction`, whose constructor accepts a regular C++ function that accepts and returns a `double` and whose `evaluateAt` function returns the value of the stored function evaluated at the parameter.
4. The composition of two functions **F** and **G** is defined as **F**(**G**(**x**)) – that is, the function **F** applied to the value of **G** applied to **x**. Write a class `CompositionFunction` whose constructor accepts two `Function *` pointers and whose `evaluateAt` returns the composition of the two functions evaluated at a point.
5. The derivative of a function is the slope of the tangent line to that function at a point. The derivative of a function **F** can be approximated as $\mathbf{F}'(\mathbf{x}) \approx (\mathbf{F}(\mathbf{x} + \Delta\mathbf{x}) - \mathbf{F}(\mathbf{x})) / \Delta\mathbf{x}$ for small values of $\Delta\mathbf{x}$. Write a class `DerivativeFunction` whose constructor accepts a `Function *` pointer and a `double` representing $\Delta\mathbf{x}$ and whose `evaluateAt` approximates the derivative of the stored function using the initial value of $\Delta\mathbf{x}$.