Chapter 10: Refining Abstractions

In the previous chapter, we explored the class concept and saw how to use classes to model an interface paired with an implementation. You learned how to realize the idealized versions of abstraction and encapsulation using the `public` and `private` keywords, as well as how to use constructors to enforce class invariants. However, our tour of classes has just begun, and there are many nuances of class design we have yet to address. For example, since class clients cannot look at the class implementation, how can they tell which parts of the public interface are designed to read the class's state and which parts will write it? How can you more accurately control how constructors initialize data? And how can you share data across all instances of a class? These questions are all essentially variants on a common theme: how can we refine our abstractions to make them more precise?

This chapter explores some of C++'s language features that allow you as a programmer to more clearly communicate your intentions when designing classes. The tools you will learn in this chapter will follow you through the rest of your programming career, and appreciating exactly where each is applicable will give you a significant advantage when designing software.

Parameterizing Classes with Templates

One of the most important lessons an upstart computer scientist or software engineer can learn is *decomposition* or *factoring* – breaking problems down into smaller and smaller pieces and solving each subproblem separately. At the heart of decomposition is the concept of *generality* – code should avoid overspecializing on a single problem and should be robust enough to adapt to other situations. Take as an example the STL. Rather than specializing the STL container classes on a single type, the authors decided to parameterize the containers over the types they store. This means that the code written for the `vector` class can be used to store almost any type, and the `map` can use arbitrary types as key/value pairs. Similarly, the STL algorithms were designed to operate on all types of iterators rather than on specific container classes, making them flexible and adaptable.

The STL is an excellent example of how versatile, flexible, and powerful C++ templates can be. In C++ a *template* is just that – a code pattern that can be instantiated to produce a type or function that works on an arbitrary type. Up to this point you've primarily been a client of template code, and now it's time to gear up to write your own templates. In this section we'll cover the basics of templates and give a quick tour of how template classes operate under the hood. We will make extensive use of templates later in this text and especially in the extended examples, and hopefully by the time you've finished reading this book you'll have an appreciation for just how versatile templates can be.

Class Templates

In C++, a *class template* is a class that, like the STL `vector` or `map`, is parameterized over some number of types. In a sense, a class template is a class with a hole in it. When a client uses a template class, she fills in these holes to yield a complete type. You have already seen this with the STL containers: you cannot create a variable of type `vector` or `map`, though you *can* create a variable of type `vector<int>` or `map<string, string>`.

Class templates are most commonly used to create types that represent particular data structures. For example, the `vector` class template is an implementation of a linear sequence using a dynamically-allocated array as an implementation. The operations that maintain the dynamic array are more or less independent of the type of elements in that array. By writing `vector` as a class template rather than a concrete class, the designers of the STL make it possible to use linear sequences of arbitrary C++ types.

Of course, not all classes should be written as class templates. For example, the `FMRadio` class from the previous chapter is an unlikely candidate for a class template because it does not hold a collection of data that could be of arbitrary type. Although `FMRadio` does hold multiple pieces of data (notably the radio's presets), those presets are always radio frequencies, which we've encoded with `double`s. It would not make sense for the `FMRadio`'s presets to be stored as `vector<int>`s, nor as `string`s. As a general rule, most classes don't need to be written as class templates.

Defining a Class Template

Once you've decided that the class you're writing is best parameterized over some arbitrary type, you can indicate to C++ that you're defining a template class by using the `template` keyword and specifying what types the template should be parameterized over. For example, suppose that we want to define our own version of the `pair` struct used by the STL. If we want to call this struct `MyPair` and have it be parameterized over two types, we can write the following:

```
template <typename FirstType, typename SecondType> struct MyPair
{
    FirstType first;
    SecondType second;
};
```

Here, the syntax `template <typename FirstType, typename SecondType>` indicates to C++ that what follows is a class template that is parameterized over two types, one called `FirstType` and one called `SecondType`. In many ways, type arguments to a class template are similar to regular arguments to C++ functions. For example, the actual names of the parameters are unimportant as far as clients are concerned, much in the same way that the actual names of parameters to functions are unimportant. The above definition is functionally equivalent to this one below:

```
template <typename One, typename Two> struct MyPair
{
    One first;
    Two second;
};
```

Within the body of the class template, we can use the names `One` and `Two` (or `FirstType` and `SecondType`) to refer to the types that the client specifies when she instantiates `MyPair`, much in the same way that parameters inside a function correspond to the values passed into the function by its caller.

In this above example, we used the `typename` keyword to introduce a type argument to a class template. If you work on other C++ code bases, you might see the above class template written as follows:

```
template <class FirstType, class SecondType> struct MyPair
{
    FirstType first;
    SecondType second;
};
```

In this instance, `typename` and `class` are completely equivalent to one another. However, I find the use of `class` misleading because it incorrectly implies that the parameter must be a class type. This is not the case – you can still instantiate templates that are parameterized using class with primitive types like int or double. From here on out, we will use typename instead of class.*

To create an instance of MyPair specialized over some particular types, we specify the name of the class template, followed by the type arguments surrounded by angle brackets. For example:

```
MyPair<int, string> one; // A pair of an int and a string.
one.first = 137;
one.second = "Templates are cool!";
```

This syntax should hopefully be familiar from the STL.

Classes and structs are closely related to one another, so unsurprisingly the syntax for declaring a template class is similar to that for a template struct. Let's suppose that we want to convert our MyPair struct into a class with full encapsulation (i.e. with accessor methods and constructors instead of exposed data members). Then we would begin by declaring MyPair as

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    /* ... */

private:
    FirstType first;
    SecondType second;
};
```

Now, what sorts of functions should we define for our MyPair class? Ideally, we'd like to have some way of accessing the elements stored in the pair, so we'll define a pair of functions getFirst and setFirst along with an equivalent getSecond and setSecond. This is shown here:

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};
```

Notice that we're using the template arguments FirstType and SecondType to stand for whatever types the client parameterizes MyPair over. We don't need to indicate that FirstType and SecondType are at all different from other types like int or string, since the C++ compiler already knows that from the template declaration. In fact, with a few minor restrictions, once you've defined a template argument, you can use it anywhere that an actual type could be used and C++ will understand what you mean.

Now that we've declared these functions, we should go about implementing them in the intuitive way. If MyPair were not a template class, we could write the following:

* You can only substitute class for typename in this instance – it's illegal to declare a regular C++ class using the typename keyword.

```
    FirstType MyPair::getFirst() // Problem: Not legal syntax
    {
        return first;
    }
```

The problem with this above code is that MyPair is a class *template*, not an actual class. If we don't tell C++ that we're trying to implement a member function for a class template, the compiler won't understand what we mean. Thus the proper way to implement this member function is

```
    template <typename FirstType, typename SecondType>
        FirstType MyPair<FirstType, SecondType>::getFirst()
    {
        return first;
    }
```

Here, we've explicitly prefaced the implementation of getFirst with a template declaration and we've marked that the member function we're implementing is for MyPair<FirstType, SecondType>. The template declaration is necessary for C++ to figure out what FirstType and SecondType mean here, since without this information the compiler would think that FirstType and SecondType were actual types instead of placeholders for types. That we've mentioned this function is available inside MyPair<FirstType, SecondType> instead of just MyPair is also mandatory since there is no real MyPair class – after all, MyPair is a class *template*, not an actual class.

The other member functions can be implemented similarly. For example, here's an implementation of setSecond:

```
    template <typename FirstType, typename SecondType>
        void MyPair<FirstType, SecondType>::setSecond(SecondType newValue)
    {
        second = newValue;
    }
```

When implementing member functions for template classes, you do *not* need to repeat the template definition if you define the function inside the body of the template class. Thus the following code is perfectly legal:

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst()
    {
        return first;
    }
    void setFirst(FirstType newValue)
    {
        first = newValue;
    }

    SecondType getSecond()
    {
        return second;
    }
    void setSecond(SecondType newValue)
    {
        second = newValue;
    }
private:
    FirstType first;
    SecondType second;
};
```

The reason for this is that inside of the class template, the compiler already knows that `FirstType` and `SecondType` are templates, and it's not necessary to remind it.

Now, let's suppose that we want to define a member function called `swap` which accepts as input a reference to another `MyPair` class, then swaps the elements in that `MyPair` with the elements in the receiver object. Then we can prototype the function like this:

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst()
    {
        return first;
    }
    void setFirst(FirstType newValue)
    {
        first = newValue;
    }

    SecondType getSecond()
    {
        return second;
    }
    void setSecond(SecondType newValue)
    {
        second = newValue;
    }

    void swap(MyPair& other);

private:
    FirstType first;
    SecondType second;
};
```

Even though MyPair is a template class parameterized over two arguments, inside the body of the MyPair template class definition we can use the name MyPair without mentioning that it's a MyPair<FirstType, SecondType>. This is perfectly legal C++ and will come up more when we begin discussing copying behavior in a few chapters. The actual implementation of swap is left as an exercise.

.h and .cpp files for template classes

When writing a C++ class, you normally partition the class into two files: a .h file containing the declaration and a .cpp file containing the implementation. The C++ compiler can then compile the code contained in the .cpp file and then link it into the rest of the program when needed. When writing a template class, however, breaking up the definition like this will cause linker errors. The reason is that C++ templates are just that – they're *templates* for C++ code. Whenever you write code that instantiates a template class, C++ generates code for the particular instance of the class by replacing all references to the template parameters with the arguments to the template. For example, with the MyPair class defined above, if we create a MyPair<int, string>, the compiler will generate code internally that looks like this:

```
class MyPair<int, string>
{
public:
    int getFirst();
    void setFirst(int newValue);

    string getSecond();
    void setSecond(string newValue);
private:
   int first;
   string second;
}

int MyPair<int, string>::getFirst()
{
    return first;
}

void MyPair<int, string>::setFirst(int newValue)
{
    first = newValue;
}

string MyPair<int, string>::getSecond()
{
    return second;
}

void MyPair<int, string>::setSecond(string newValue)
{
    second = newValue;
}
```

At this point, compilation continues as usual.

But what would happen if the compiler didn't have access to the implementation of the MyPair class? That is, let's suppose that we've created a header file, my-pair.h, that contains only the class declaration for MyPair, as shown here:

File: *`my-pair.h`*

```
#ifndef MyPair_Included // Include guard prevents multiple inclusions
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

#endif
```

Suppose that we have a file that `#include`s the `my-pair.h` file and then tries to use the `MyPair` class. Since all that the compiler has seen of `MyPair` is the above class definition, the compiler will only generate the following code for `MyPair`:

```
class MyPair<int, string>
{
public:
    int getFirst();
    void setFirst(int newValue);

    string getSecond();
    void setSecond(string newValue);
private:
   int first;
   string second;
}
```

Notice that while all the member functions of `MyPair<int, string>` have been *prototyped*, they haven't been *implemented* because the compiler didn't have access to the implementations of each of these member functions. In other words, if a template class is instantiated and the compiler hasn't seen implementations of its member functions, the resulting template class will have no code for its member functions. This means that the program won't link, and our template class is now useless.

When writing a template class for use in multiple files, the entire class definition, including implementations of member functions, must be visible in the header file. One way of doing this is to create a .h file for the template class that contains both the class definition and implementation without creating a matching .cpp file. This is the approach adopted by the C++ standard library; if you open up any of the headers for the STL, you'll find the complete (and cryptic) implementations of all of the functions and classes exported by those headers.

To give a concrete example of this approach, here's what the `my-pair.h` header file might look like if it contained both the class and its implementation:

***File**: `my-pair.h`*

```
/* This method of packaging the .h/.cpp pair puts the entire class definition and
 * implementation into the .h file.  There is no .cpp file for this header.
 */

#ifndef MyPair_Included
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue)
{
    first = newValue;
}

template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond()
{
    return second;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond(SecondType newValue)
{
    second = newValue;
}

#endif
```

Putting the class and its definition inside the header file is a valid way to prevent linker errors, but it seems to violate the principle of separation of interface and implementation. After all, the reason we have both .h and .cpp files is to hide a class implementation in a file that clients never have to look at. Unfortunately, barring some particularly unsightly and hacky abuses of the preprocessor, you will need to structure template code in this manner.

The Two Meanings of `typename`

One of the more unfortunate quirks of the C++ language is the dual meaning of the `typename` keyword. As mentioned previously, when defining a template class, you can the `typename` keyword to declare type parameters for the template class. However, there is another use of the `typename` keyword that can easily catch you off guard unless you're on the lookout for it. Suppose, for example, that we wish to implement a class akin to the STL `stack` which represents a LIFO container. Because the abstract notion of a stack only concerns the *ordering* of the elements in the container rather than the *type* or *contents* of the elements in the container, we should probably consider implementing the stack as a template class. Here is one possible interface for such a class, which we'll call `Stack` to differentiate it from the STL `stack`:

```
template <typename T> class Stack
{
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();
};
```

There are many ways that we could implement this `Stack` class: we could use dynamically-allocated arrays, or the STL `vector` or `deque` containers. Of these three choices, the `vector` and `deque` are certainly simpler than using dynamically-allocated arrays. Moreover, since all of the additions and deletions from a stack occur at the end of the container, the `deque` is probably a more suitable container with which we could implement our `Stack`. We'll therefore implement the `Stack` using a `deque`, as shown here:

```
template <typename T> class Stack
{
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();

private:
    deque<T> elems;
};
```

Notice that we've used the template parameter `T` to parameterize the `deque`. This is perfectly valid, and is quite common when implementing template classes.

Given this implementation strategy, we can implement each of the member functions as follows. Make sure that you can read this code; it's fairly template-dense.

```
template <typename T> void Stack<T>::push(T value)
{
    elems.push_front(value);
}

template <typename T> T Stack<T>::pop()
{
    T result = elems.front();
    elems.pop_front();
    return result;
}

template <typename T> size_t Stack<T>::size()
{
    return elems.size();
}

template <typename T> bool Stack<T>::empty()
{
    return elems.empty();
}
```

This is a perfectly reasonable implementation of a stack, and in fact the STL `stack` implementation is very similar to this one.

Now, suppose that we're interested in extending the functionality of the `Stack` so that class clients can iterate over the elements of the `Stack` in the order that they will be removed. For example, if we push the elements 1, 2, 3, 4, 5 onto the stack, the iteration would visit the elements in the order 5, 4, 3, 2, 1. This functionality is usually not found on a `Stack`, but is useful for debugging (e.g. printing out the contents of the stack) or modifying the elements of the `Stack` after they've already been inserted. To do this, we'll need to add `begin()` and `end()` functions to the `Stack` class that return iterators over the underlying `deque`. Because the internal `deque` is a `deque<T>`, these iterators have type `deque<T>::iterator`. Consequently, you might think that we would update the interface as follows:

```
template <typename T> class Stack
{
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();

    deque<T>::iterator begin(); // Problem: Illegal syntax.
    deque<T>::iterator end();   // Problem: Illegal syntax.

private:
    deque<T> elems;
};
```

This code is perfectly well-intentioned, but unfortunately is not legal C++ code. The problem has to do with the fact that `deque<T>` is a *dependent type*, a type that “depends” on a template parameter. Intuitively, this is because `deque<T>` isn't a concrete type – it's a pattern that says “once you give me the type `T`, I'll give you back a `deque` of `T`'s”. Due to a somewhat arcane restriction in the C++ language, if you try to access a type nested inside of a dependent type inside of a template class (for example, trying to use the `iterator` type nested inside `deque<T>`), you must preface that type with the `typename` keyword. The correct version of the `Stack` class is as follows:

```
template <typename T> class Stack
{
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();

    typename deque<T>::iterator begin(); // Now correct
    typename deque<T>::iterator end();   // Now correct

private:
    deque<T> elems;
};
```

This syntactic oddity is one of the truly embarrassing parts of C++. There is no high-level reason why `typename` should be necessary, and its existence is a perpetual source of confusion and frustration among new C++ programmers. I wholeheartedly wish that I could give you a nice clean explanation as to why `typename` is necessary, but the real answer is highly technical and in many ways unsatisfactory. Of course, this doesn't excuse you from having to put the `typename` keyword in when it's necessary, and you'll have to make sure to use it where appropriate. The good news is that `typename` is unnecessary in most circumstances. You only need to use the `typename` keyword when accessing a type nested inside of a dependent type. From a practical standpoint, this means that if you want to look up a type nested inside of a type that's either a template parameter or is parameterized over a template parameter, you must preface the type with the `typename` keyword. In the examples used in the upcoming chapters, this will only occur when looking up `iterator`s inside of STL containers that themselves are parameterized over a template argument, such as a `deque<T>::iterator` or a `vector<T>::iterator`.

To complete the above example, the implementation of the `begin` and `end` functions are shown here:

```
template <typename T> typename deque<T>::iterator Stack<T>::begin()
{
    return elems.begin();
}

template <typename T> typename deque<T>::iterator Stack<T>::end()
{
    return elems.end();
}
```

These functions might be the densest pieces of code you've encountered so far. The code `template <typename T>` declares that the member function implementation is an implementation of a template class's member function. `typename deque<T>::iterator` is the return type of the function, and `Stack<T>::begin()` is the name of the member function and the (empty) parameter list. When writing template classes, code like this is fairly ubiquitous, but with practice you'll be able to read this code much more easily.

Clarifying Interfaces with `const`

At its core, C++ is a language based on modifying program state. `int`s get incremented in `for` loops; `vector`s have innumerable calls to `clear`, `resize`, and `push_back`; and console I/O overwrites variables with values read directly from the user.

Consider the following function:

```
void LoadFileContents(string filename, vector<string>& out)
{
    ifstream input(filename.c_str()); // Open the file
    out.clear();

    string line;
    while (getline(cin, line))
        out.push_back(line);
}
```

This function takes in a string containing the name of a file, then reads the contents of the file into a `vector<string>` specified as a reference parameter. Because this function writes the result to an *existing* `vector` rather than creating a new `vector` for output, we say that the function has *side effects*. Side effects are extremely common in C++ code, and in fact without side effects C++ programs would be very difficult to write. However, when working with increasing large software systems, side effects can be dangerous. As mentioned last chapter, a single incorrect bit can take down an entire software system. Consequently, you must be very careful when designing functions with side effects so that the scope of what those side effects can modify is minimized. To see exactly why this is, let's consider the extreme case. Suppose every function in a program is allowed to modify *any* piece of data in the program. That is, whenever a function is called, the values of all variables in all functions might be changed. What would this mean for programming? Certainly, it would be much more difficult to reason about how programs operate. Consider, for example, the following loop:

```
for (size_t k = 0; k < 100; ++k)
    MyFunction();
```

Here, we iterate over the first one hundred integers, calling some function called `MyFunction`. What will this program do? Certainly it depends on the implementation of `MyFunction`, but a reasonable programmer would probably infer that `MyFunction` will be called exactly one hundred times. But this is making the reasonable assumption that because `MyFunction` isn't passed `k` as a parameter, it has no way of modifying the local variable `k` in the calling function. However, we're assuming that functions are allowed to modify *any* data in the program. Given this assumption, there's no reason that the `MyFunction` function couldn't change the value of `k` whenever it's called. It might, for example, set `k` to be 0 on every iteration, meaning that the loop will never terminate (see if you can convince yourself why this is). Similarly, the function might increment `k` by one every time it's called, causing the loop to execute half as many times as it should (since `k` will take on values 0, 2, 4, 8, ... instead of 0, 1, 2, 3, ...). Without looking at the implementation of `MyFunction`, there would be no way to know exactly what will happen to `k`. Trying to infer what the program will do by looking at its complete source code would be substantially more complicated, and building programs more than a few hundred lines of code would quickly become difficult or impossible.

Hopefully the above example has convinced you that allowing functions to make arbitrary changes to program state is not a viable option. Fortunately, C++ is specifically designed to allow programmers to constrain where data can be modified. Many of the programming concepts we've explored so far revolve around this idea. For example:

- **Avoiding global variables**. You have probably been hammered repeatedly with the idea that global variables can be hazardous. Global variables make programs significantly harder to maintain because globals can be modified by any function, at any time, for any reason. This means that if a program encounters an error because a global variable has an incorrect value, it is difficult to track down exactly where in the program that variable received the incorrect value. By using local variables instead of globals, it is easier to track down exactly where errors occur by following which functions have access to those variables.

- **Marking data members `private`**. We initially explored encapsulation from a theoretical perspective as a means for separating implementation from interface. However, encapsulation also helps control where side effects can occur in a program. If a class's data members are marked `private`, then any changes to those data members must result from the class's public interface. Verifying that the class's interface is implemented correctly can therefore increase confidence that data members aren't mercilessly clobbered.

- **Decomposing large functions**. Besides making code cleaner, more maintainable, and easier to follow, decomposition minimizes the amount of code that has access to each local variable. If a task is well-decomposed, then each function will have access only to a small number of variables and thus cannot affect much program state.

Each of these programming patterns ensure that data can only be modified in places where a programmer has explicitly granted particular functions access to that data. However, C++ provides an even stronger mechanism for preventing unexpected side effects – the `const` keyword. You have already seen `const` in the context of global constants, but the `const` keyword has many other uses. This section introduces the mechanics of `const` (for example, where `const` can be used and what it means in these contexts) and how to use it properly in C++ code.

`const` Variables

So far, you've only seen `const` in the context of global constants. For example, given the following global declaration:

```
const int MyConstant = 137;
```

Whenever you refer to the value `MyConstant` in code, the compiler knows that you're talking about the value 137. If later in your program you were to write `MyConstant = 42`, the complier would flag the line as an error because code to this effect modifies a value you explicitly indicated should never be modified. However, `const` is not limited to global constants. You can also declare *local* variables `const` to indicate that their values should never change. Consider the following code snippet:

```
for (set<int>::iterator itr = mySet.lower_bound(42);
     itr != mySet.upper_bound(137); ++itr)
{
    /* ... manipulate *itr ... */
}
```

This code iterates over all of the values in an STL `set` whose values are in the range [42, 137].[*] However, this code is not nearly as efficient as it could be. Because C++ evaluates the looping condition of a `for` loop on each iteration, the program will evaluate the statement `itr != mySet.upper_bound(137)` once per loop iteration, so the program will recompute `mySet.upper_bound(137)` multiple times. Although the

* If you're a bit rusty on the `upper_bound` and `lower_bound` functions, refer back to the chapter on STL associative containers.

STL set is highly optimized and the upper_bound function is particularly fast (on a set with *n* elements, upper_bound runs in time proportional to $\log_2 n$), if there are many elements in the range [42, 137] the overhead of multiple calls to upper_bound may be noticeable. To fix this, we might consider computing mySet.upper_bound exactly once, storing the value somewhere, and then referencing the precomputed value inside the for loop. Here's one possible implementation:

```
set<int>::iterator stop = mySet.upper_bound(137);
for (set<int>::iterator itr = mySet.lower_bound(42); itr != stop; ++itr)
{
    /* ... manipulate *itr ... */
}
```

This version of the loop will run much faster than its previous incarnation. However, this new version of the loop now depends on the fact that stop holds the value of mySet.upper_bound(137) throughout the loop. If we accidentally overwrite stop, we'll end up iterating the wrong number of times. In other words, the variable stop isn't really a variable – it shouldn't *vary* – but instead should be a constant. To indicate to C++ that the value of stop shouldn't change, we can mark the stop variable const. This prevents us from changing the value of stop, and will cause a compile-time error if we try to do so. The updated code is shown here:

```
const set<int>::iterator stop = mySet.upper_bound(137);
for (set<int>::iterator itr = mySet.lower_bound(42); itr != stop; ++itr)
{
    /* ... manipulate *itr ... */
}
```

This is your first glimpse of a const local variable. const local variables are similar to global constants: they must be initialized to a value, their values can't change during the course of execution, etc. In fact, the only difference between a const local variable and a global constant is *scope*. Global constants are globally visible and persist throughout the course of a program, while const local variables are created and destroyed like regular local variables.

const Objects

The main idea behind const is to let programmers communicate that the values of certain variables should not change during program execution. When working with primitive types, the meaning of "should not change" is fairly clear: an int changes if it is incremented, decremented or overwritten; a bool changes if it flips from true to false; etc. However, when working with variables of class type, our notion of "should not change" becomes substantially more nuanced. To give you a sense for why this is, let's consider a const string, a C++ string whose contents cannot be modified. We can declare a const string as we would any other const variable. For example:

```
const string myString = "This is a constant string!";
```

Note that, like all const variables, we are still allowed to assign the string an initial value.

Because the string is const, we're not allowed to modify its contents, but we can still perform some basic operations on it. For example, here's some code that prints out the contents of a const string:

```
const string myString = "This is a constant string!";
for(size_t i = 0; i < myString.length(); ++i)
    cout << myString[i] << endl;
```

To us humans, the above code seems completely fine and indeed it is legal C++ code. But how does the compiler know that the length function doesn't modify the contents of the string? This question may seem silly – of *course* the length function won't change the length of the string – but this is only obvious because we humans have a gut feeling about how a function called length should behave. The compiler, on the other hand, knows nothing of natural language, and could care less whether the function were named “length” or “zyzzyzplyx.” This raises a natural question: given an arbitrary class, how can the compiler tell which member functions might modify the receiver object and which ones cannot? To answer this question, let's look at the prototype for the string member function length:*

```
class string
{
public:
    size_t length() const;

    /* ... etc. ... */
};
```

Note that there is a const after the member function declaration. This is another use of the const keyword that indicates that the member function does not modify any of the class's instance variables. That is, when calling a const member function, you're guaranteed that the object's contents cannot change. (This isn't *technically* true, as you'll see later, but it's a perfectly valid way of thinking about const functions).

When working with const objects, you are only allowed to call member functions on that object that have been explicitly marked const. That is, even if you have a function that doesn't modify the object, unless you tell the compiler that the member function is const, the compiler will treat it as a non-const function. This may seem like a nuisance, but has the advantage that it forces you to decide whether or not a member function should be const before you begin implementing it. That is, the constness of a member function is an *interface* design decision, not an *implementation* design decision.

To see how const member functions work in practice, let's consider a simple Point class that stores a point in two-dimensional space. Using the getter/setter paradigm, we end up with this class definition:

```
class Point
{
public:
    Point(double x, double y);

    double getX();
    double getY();

    void setX(double newX);
    void setY(double newY);

private:
    double x, y;
};
```

Let's take a minute to think about which of these functions should be const and which should not be. Clearly, the setX and setY functions should not be const, since these operations by their very nature modify the receiver object. But what about getX and getY? Neither of these functions should modify the receiver object, since they're designed to let clients query the object's internal state. We should therefore

* The actual implementation of the string class looks very different from this because string is a class template rather than an actual class. For our discussion, though, this simplification is perfectly valid.

mark these functions `const` to indicate that they cannot modify the object. This gives us the following definition of `Point`:

```
class Point
{
public:
    Point(double x, double y);

    double getX() const;
    double getY() const;

    void setX(double newX);
    void setY(double newY);

private:
    double x, y;
};
```

There's only one function we've ignored so far – the `Point` constructor. However, in C++ it's illegal to mark a constructor `const`, since the typical operation of a constructor runs contrary to the notion of `const`. Take a minute to think about why this is; you'll be a better C++ coder for it!

Now that we've marked the `getX` and `getY` functions `const`, we can think about how we might go about implementing these functions. You might think that we would implement them just as we would regular member functions, and you would *almost* be right. However, the fact that the function is `const` is part of that function's signature, and so in the implementation of the `getX` and `getY` functions we will need to explicitly indicate that those member functions are `const`. Here is one possible implementation of `getX`; similar code can be written for `getY`.

```
double Point::getX() const
{
     return x;
}
```

Forgetting to add this `const` can be a source of much frustration because the C++ treats `getX()` and `getX() const` as two different functions. We will discuss why this is later in this chapter.

In a `const` member function, all the class's instance variables are treated as `const`. You can read their values, but must not modify them. Similarly, inside a `const` member function, you cannot call other non-`const` member functions. The reason for this is straightforward: because non-`const` member functions can modify the receiver object, if a `const` member function could invoke a non-`const` function, then the `const` function might indirectly modify the receiver object. But beyond these restrictions, `const` member functions can do anything that regular member functions can. Suppose, for example, that we wish to update the `Point` class to support a member function called `distanceToOrigin` which returns the distance between the receiver object and the point (0, 0). Because this function shouldn't modify the receiver object, we'll mark it `const`, as shown here:

```
    class Point
    {
    public:
        Point(double x, double y);

        double getX() const;
        double getY() const;

        void setX(double newX);
        void setY(double newY);

        double distanceToOrigin() const;

    private:
        double x, y;
    };
```

Mathematically, the distance between a point and the origin is defined as $\sqrt{x^2+y^2}$. Using the sqrt function from the <cmath> header file, we can implement the distanceToOrigin function as follows:

```
void Point::distanceToOrigin() const
{
     double dx = getX();   // Legal!  getX is const.
     double dy = y;        // Legal!  Reading an instance variable.
     dx *= dx;             // Legal!  We're modifying dx, which isn't an
                           //         instance variable.
     dy *= dy;             // Legal!  Same reason as above.
     return sqrt(dx + dy); // Legal!  sqrt is a free function that can't
                           //         modify the current object.
}
```

Although this function is marked const, we have substantial leeway with what we can do in the implementation. We can call the getX function, since it too is marked const. We can also read the value of y and store it in another variable because this doesn't change its value. Additionally, we can change the values of the local variables dx and dy, since doing so doesn't change any of the receiver object's data members. Remember, const member functions guarantee that the *receiver object* doesn't change, not that the function doesn't change the values of any variables. Finally, we can call free functions, since those functions don't have access to the class's data members and therefore cannot modify the receiver.

const References

Throughout this text we've used pass-by-reference by default when passing heavy objects like vectors and maps as parameters to functions. This improves program efficiency by avoiding expensive copy operations. Unfortunately, though, using pass-by-reference in this way makes it more difficult to reason about a function's behavior. For example, suppose you see the following function prototype:

```
    void DoSomething(vector<int>& vec);
```

You know that this function accepts a vector<int> by reference, but it's not clear *why*. Does DoSomething modify the contents of the vector<int>, or is it just accepting by reference to avoid making a deep copy of the vector? Without knowing which of the two meanings of pass-by-reference the function writer intended, you should be wary about passing any important data into this function. Otherwise, you might end up losing important data as the function destructively modifies the parameter.

We are in an interesting situation. If we don't use pass-by-reference on functions that take large objects as parameters, our programs will pay substantial runtime costs unnecessarily. On the other hand, if we do pass large objects by reference, we make it more difficult to reason about exactly what the functions in our program are trying to do. In other words, we can make a tradeoff between *efficiency* and *clarity*. In many cases, this tradeoff is necessary. Clean, straightforward algorithms are often fast and efficient, but more often than not they are slower than their more intricate counterparts. But in this particular arena, there is an easy way to gain the efficiency of pass-by-reference without the associated ambiguity: *`const` references*.

A `const` reference is, in many ways, like a normal reference. `const` references refer to objects and variables declared elsewhere in the program, and any operations performed on the reference are instead performed on the object being referred to. However, unlike regular references, `const` references treat the object they alias as though it were `const`. In other words, `const` references capture the notion of *looking* at an object without being able to *modify* it.

To see how `const` references work in practice, let's consider an example. Suppose that we want to write a function which prints out the contents of a `vector<int>`. Such a function clearly should not modify the `vector`, and so we can prototype this function as follows:

```
void PrintVector(const vector<int>& vec);
```

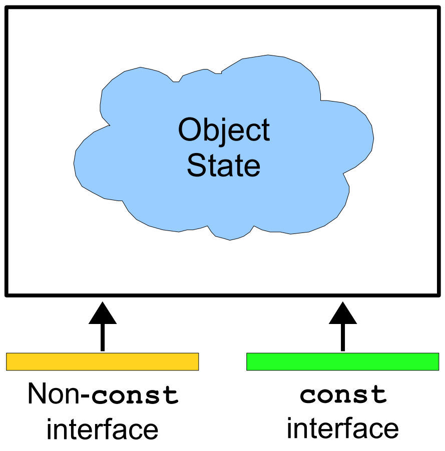
Notice that this function takes in a `const vector<int>&`. This is a `const` reference (also called a *reference-to-`const`*). Inside the `PrintVector` function, the `vec` parameter is treated as though it were `const`, and so we cannot make any changes to it. Thus the following implementation of `PrintVector` is perfectly legal:

```
void PrintVector(const vector<int>& vec)
{
    for (size_t k = 0; k < vec.size(); ++k)
        cout << vec[k] << endl;
}
```

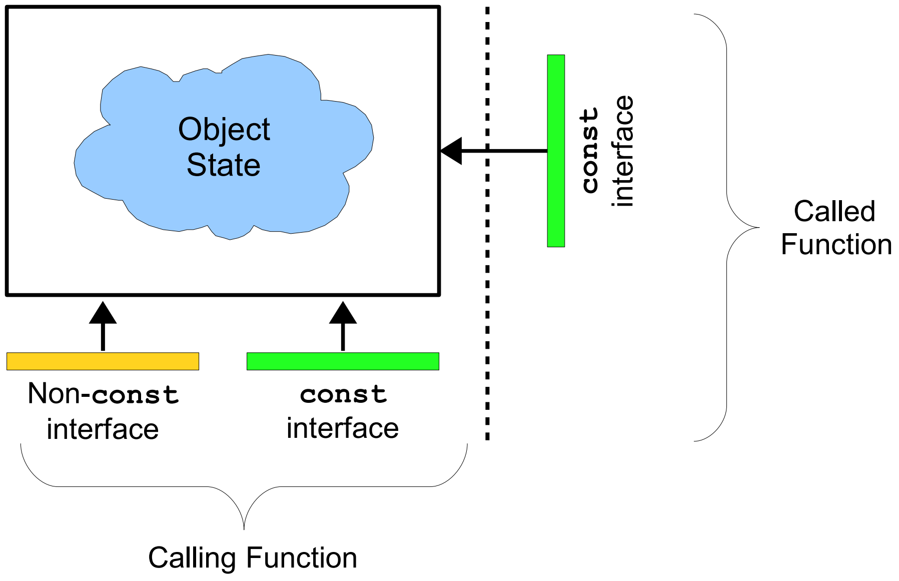
Although the `PrintVector` function takes in a reference to a `const vector<int>`, it is perfectly legal to pass both `const` and non-`const` `vector<int>`s to `PrintVector`. Whether or not the original `vector` is `const`, inside the `PrintVector` function C++ treats the vector as though it were `const`. Thus it's legal (and encouraged) to write code like this:

```
void PrintVector(const vector<int>& vec)
{
    for (size_t k = 0; k < vec.size(); ++k)
        cout << vec[k] << endl;
}

int main()
{
    vector<int> myVector(NUM_INTS);

    PrintVector(myVector);    // Legal!  myVector treated const in PrintVector

    myVector.push_back(137); // Legal!  myVector isn't const out here.
}
```

You might be a bit uneasy with the idea of passing a non-`const` variable into a function that takes a reference-to-`const`. After all, something of type `Type` isn't the same as something of type `const Type`. We can't assign values to `const` objects, nor can we invoke their non-`const` member functions. However, it is perfectly safe to treat a non-`const` object as though it were `const` because the legal operations on a `const` object are a *subset* of the legal operations on a non-`const` object. That is, every object's public interface can be split into two parts, a `const` interface of non-mutating operations and a non-`const` interface of operations which change the object's state. This is shown below:



In this picture, the object's internal state is represented by the fuzzy cloud, with the `const` and non-`const` interfaces each having access to the internals. When an object is non-`const`, it has both interfaces; when `const` it has only the `const` interface. Using this mental model, let's think about what happens when we pass an object by reference-to-`const` into a function. Because the called function takes in a reference-to-`const`, we can treat the function as though it expects only the `const` interface for an object. Graphically:



What does this idea mean for you as a programmer? In particular, when writing functions that need to be able to look at data but not modify it, you should strongly consider using pass-by-reference-to-`const`. This gives you the benefits of pass-by-reference (higher efficiency) with the added guarantee that the parameter won't be destructively modified. Of course, while it's legal to pass non-`const` objects to functions accepting `const` references, you cannot pass `const` objects into functions accepting non-`const` references. The reason for this is simple: if an object is marked `const`, its value cannot be changed. If a `const` object could be passed into a function by non-`const` reference, that function could modify the original object, subverting `const`ness. You can think of `const` as a universal accepter and of non-`const` as the universal donor – you can convert both `const` and non-`const` data to `const` data, but you can't convert `const` data to non-`const` data. Thinking about this using our two-interface analogy, if you have access to a class's non-`const` interface, you can always ignore it and just use the `const` interface. However, if you only have access to the `const` interface, you can't suddenly give yourself access to the non-`const` interface.

Although `const` references behave for the most part like regular references, there is one particularly important behavioral aspect where they diverge. Suppose we are given the following prototype for a function called `DoSomething`, which takes in a reference to an `int`:

```
void DoSomething(int& x);
```

Given this prototype, each of the following calls to `DoSomething` is illegal:

```
DoSomething(137);       // Problem: Cannot pass literal by reference
DoSomething(2.71828);   // Problem: Cannot pass literal by reference

double myDouble;
DoSomething(myDouble); // Problem: int& cannot bind to double
```

Let's examine exactly why each of these three calls fail. In the first case, we tried to pass the integer literal into the `DoSomething` function. This will cause problems if `DoSomething` tries to modify its parameter. Suppose, for example, that `DoSomething` is implemented as follows:

```
void DoSomething(int& x)
{
    x = 0;
}
```

If we pass 137 directly into `DoSomething`, the the line `x = 0` would try to store the value 0 into the integer literal 137. This is clearly nonsensical, and so the compiler disallows it. The second erroneous call to `DoSomething` (where we pass in 2.71828) fails for the same reason. However, what of the third call, `DoSomething(myDouble)`? This fails because `myDouble` is a `double`, not an `int`, and although it's possible to typecast a `double` to an `int` the C++ language explicitly says that this is not acceptable. This may seem harsh, but it allows C++ programs to run extremely efficiently because the compiler can assume that the parameter `x` is bound to an actual `int`, not something implicitly convertible to an `int`.*

However, suppose we change the prototype of `DoSomething` to accept its parameter by `const` reference, as shown here:

```
void DoSomething(const int& x);
```

Then all of the following calls to `DoSomething` are perfectly legal:

```
DoSomething(137);       // Legal
DoSomething(2.71828);   // Legal
```

* I know that this explanation might seem a bit fuzzy, primarily because the main reason is technical and has to do with how `int`s and `double`s are represented in the machine. If you try to execute the machine code instructions to store an integer value into a variable that's declared as a `double`, the `double` will take on a completely meaningless value that has nothing to do with the integer that we intended to store in it. If you're interested in learning more about why this is, consider taking a compilers course or studying an assembly language (MIPS or x86). If you still don't understand why `int&`s can't be bound to `double`s, send me an email and I can try to explain things in more detail.

```
double myDouble;
DoSomething(myDouble);   // Legal!
```

Why the difference? Think about why all of the above examples caused problems when mixed with non-`const` references. In the first case, we might accidentally assign a new value to an integer literal; the second case ran into similar problems. In the third case, due to hardware restrictions, we cannot bind an `int&` to a `double` because writing a value to that `int&` would result in incorrect behavior. All of these cases have to do with the fact that the reference can be used to modify the object it's bound to. But when working with `const` references, none of these problems are possible because the referenced value can't be changed through the reference.

Because normal restrictions on references do not apply to `const` references, you can treat pass-by-reference-to-`const` as a smarter version of pass-by-value. Any value that could be passed by value can be passed by reference-to-`const`, but when using reference-to-`const` objects won't be copied in most cases. We will address this later in this chapter. For now, treating pass-by-reference-to-`const` as a more efficient pass-by-reference will be wise.

`const` and Pointers

The `const` keyword is useful, but has its share of quirks. Perhaps the most persistent source of confusion when working with `const` arises when mixing `const` and pointers. For example, suppose that you want to declare a C string as a global constant. Since to declare a global C++ `string` constant you use the syntax

```
const string kGlobalCppString = "This is a string!";
```

You might assume that to make a global C string constant, the syntax would be:

```
const char* kGlobalStr = "This is a string!"; // Problem: Legal but incorrect
```

This syntax is *partially* correct. If you were ever to write `kGlobalString[0] = 'X'`, rather than getting segmentation faults at runtime (see the C strings chapter for more info), you'd instead get a compiler error that would direct you to the line where you tried to modify the global constant. But unfortunately this variable declaration contains a subtle but crucial mistake. Suppose, for example, that we write the following code:

```
kGlobalString = "Reassigned!";
```

Here, we reassign `kGlobalString` to point to the string literal "Reassigned!" Note that we didn't modify the contents of the character sequence `kGlobalString` points to – instead we changed *what character sequence `kGlobalString` points to*. In other words, we modified the *pointer*, not the *pointee*, and so the above line will compile correctly and other code that references `kGlobalString` will suddenly begin using the string "Reassigned!" instead of "This is a string!" as we would hope.

C++ distinguishes between two similar-sounding entities: a *pointer-to-`const`* and a *`const` pointer*. A pointer-to-`const` is a pointer like `kGlobalString` that points to data that cannot be modified. While you're free to reassign pointers-to-`const`, you cannot change the value of the elements they point to. To declare a pointer-to-`const`, use the syntax **`const Type* myPointer`**, with the `const` on the left of the star. Alternatively, you can declare pointers-to-`const` by writing **`Type const* myPointer`**.

A *`const` pointer*, on the other hand, is a pointer that cannot be assigned to point to a different value. Thus with a `const` pointer, you can modify the *pointee* but not the *pointer*. To declare a `const` pointer, you use the syntax **`Type* const myConstPointer`**, with the `const` on the right side of the star. Here, `myConst-Pointer` can't be reassigned, but you are free to modify the value it points to.

To illustrate by analogy, a pointer-to-`const` is like a telescope – it can look at other objects, and freely change which objects it looks at, but it cannot apply any changes to those objects. A `const` pointer, on the other hand, is like an industrial laser. The laser can be turned on at high power to cut a sheet of metal, or at low power to get a sense of what the metal looks like, but the beam is always pointed at the same place. You wouldn't try to cut a sheet of metal with a telescope, nor would you try to look at an object at a distance by blasting a high-energy laser at it. Remembering whether you want a pointer-to-`const` (look but don't touch) or a `const` pointer (touch, but only touch one thing) will be tricky at first, but will become more natural as you mature as a programmer.

Note that the syntax for a pointer-to-`const` is `const Type * ptr` while the syntax for a `const` pointer is `Type * const ptr`. The only difference is where the `const` is in relation to the star. One trick for remembering which is which is to read the variable declaration from right-to-left. For example, reading `const Type * ptr` backwards says that "`ptr` is a pointer to a `Type` that's `const`," while `Type * const ptr` read backwards is "`ptr` is a `const` pointer to a `Type`."

Returning to the C string example, to make `kGlobalString` behave as a true C string constant, we'd need to make the pointer both a `const` pointer and a pointer-to-`const`. This may seem strange, but is perfectly legal C++. The result is a `const` pointer-to-`const`, a pointer that can only refer to one object and that cannot change the value of that object. Syntactically, this looks as follows:

```
const char * const kGlobalString = "This is a string!";
```

Note that there are *two* `const`s here – one before the star and one after it. Here, the first `const` indicates that you are declaring a pointer-to-`const`, while the second means that the pointer itself is `const`. Using the trick of reading the declaration backwards, here we have "`kGlobalString` is a `const` pointer to a `char` that's `const`." This is the correct way to make the C string completely `const`, although it is admittedly a bit clunky.

The following table summarizes what types of pointers you can create with `const`:

Declaration Syntax	**Name**	**Can reassign?**	**Can modify pointee?**
`const Type* myPtr`	Pointer-to-`const`	Yes	**No**
`Type const* myPtr`	Pointer-to-`const`	Yes	**No**
`Type* const myPtr`	`const` pointer	**No**	Yes
`const Type* const myPtr`	`const` pointer-to-`const`	**No**	**No**
`Type const* const myPtr`	`const` pointer-to-`const`	**No**	**No**

As with references and references-to-`const`, it is legal to set a pointer-to-`const` to point to a non-`const` object. This simply means that the object cannot be modified through the pointer-to-`const`.

const_iterator

Suppose you have a function that accepts a vector<string> by reference-to-const and you'd like to print out its contents. You might want to write code that looks like this:

```
void PrintVector(const vector<string>& myVector)
{
    for(vector<string>::iterator itr = myVector.begin(); // Problem
        itr != myVector.end(); ++itr)
        cout << *itr << endl;
}
```

Initially, this code seems perfectly fine, but unfortunately the compiler will give you some positively ferocious errors if you try to compile this code. The problem has to do with a subtlety involving STL iterators and const. Notice that in the first part of the for loop we declare an object of type vector<string>::iterator. Because the vector is const, somehow the compiler has to know that the iterator you're getting to the vector can't modify the vector's contents. Otherwise, we might be able to do something like this:

```
/* Note: This code doesn't compile.  It just shows off what happens if we
 * could get an iterator to a const vector.
 */
void EvilFunction(const vector<string>& myVector)
{
    vector<string>::iterator itr = myVector.begin();

    *itr = 42; // Just modified a const object!
}
```

In other words, if we could get an iterator to iterate over a const vector, that iterator could be used in fiendish and diabolical ways to modify the contents of the vector, something we promised not to do. This raises an interesting issue. Let's reconsider our (currently flawed) implementation of PrintVector:

```
void PrintVector(const vector<string>& myVector)
{
    for(vector<string>::iterator itr = myVector.begin();  // Problem
        itr != myVector.end(); ++itr)
        cout << *itr << endl;
}
```

This code doesn't compile because the for loop tries to get an iterator that traverses the vector. As shown above, given an iterator over a const vector, it's possible to modify the contents of that vector and subvert constness. But in this function we *don't* modify the contents of the vector – we're harmlessly traversing the vector elements and printing its contents! So why does the compiler cryptically complain about our code? The reason is that *constness is conservative*. When the C++ compiler checks your code to ensure that you haven't violated the sanctity of const, its analysis is imprecise. Rather than determining whether or not your code actually modifies a const variable, it checks for syntactic structures which violate const – do you assign a a const variable? Do you invoke a non-const function on a const variable? Do you pass a const variable into a function which takes an argument by non-const reference? Because of this, it is possible to write code that cannot possibly change the value of a const variable but which is still rejected by the compiler. For example, consider the following code snippet:

```
void SubtleFunction(const vector<string>& myVector)
{
    if (myVector.empty())
        myVector.clear(); // Error!  Calls non-const function.
}
```

This function checks to see if the parameter is the empty vector, and, if so, calls `clear` on that vector. Calling `clear` on the empty `vector` does nothing to that `vector`, and so technically speaking this function never changes the value of its parameter. However, the C++ compiler will still reject this code, because you invoked `clear` (a non-`const` member function) on a `const vector`.

Why does the compiler take this approach? The answer is that it is *provably impossible* to build a compiler that can actually determine whether or not a C++ function will change the value of a particular variable. You read that correctly – no compiler, no matter how sophisticated or clever, can correctly determine in all cases whether a C++ program will read or write a particular variable. Because of this, C++'s rules for `const`ness have a margin of error. Some programs that will never change the value of a certain variable will cause compiler errors, but any program that correctly obeys `const` will ensure that `const` variables are never overwritten. This explains why, in our simple `PrintVector` example, the compiler complained. Although we never actually overwrite the elements of the `vector` using our iterator, the fact that someone with an iterator *could* overwrite the elements of the `vector` is enough to cause the compiler to panic.

Because raw iterators don't play nicely with `const` containers, we're going to need to change our code. One idea you may have had would be to mark the iterator `const` to prevent it from overwriting the elements of the `vector`. While well-intentioned, this approach won't work. A `const` iterator is like a `const` pointer – it can't change what element it iterates over, but it can change the value of the elements it iterates over. This is the reverse of what we want – we want an iterator that can't change the values it looks at but can change which elements it iterates over. For this, we can use *`const_iterator`*s. Each STL container that defines an iterator also defines a `const_iterator` that can read the values from the container but not write them. Using a `const_iterator`, we can rewrite our implementation of `PrintVector` as follows:

```
void PrintVector(const vector<string>& myVector)
{
     for(vector<string>::const_iterator itr = myVector.begin(); // Correct!
         itr != myVector.end(); ++itr)
         cout << *itr << endl;
}
```

To maintain `const`ness, you cannot use `const_iterator`s in functions like `insert` or `erase` that modify containers. You can, however, define iterator ranges using `const_iterator`s for algorithms like `binary_search` that don't modify the ranges they apply to.

There is one subtle point we have glossed over in this discussion – how does the `vector` *know* that it should hand back a `const_iterator` when marked `const` and a regular iterator otherwise? That is, how do the `vector`'s `begin` and `end` functions hand back objects of two different types based on whether or not the `vector` is `const`? The answer may surprise you. Here is a (slightly simplified) version of the `vector` interface which showcases the `begin` and `end` functions:

```
template <typename T> class vector
{
public:
    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;

    /* ... etc. ... */
};
```

Notice that there are *two* begin functions – one of which is non-const and returns a regular iterator, and one of which is const and returns a const_iterator. There are similarly two versions end function. This is a technique known as *const-overloading* and allows a function to have two different behaviors based on whether or not an object is const. When a const-overloaded function is invoked, the version of the function is called that matches the constness of the receiver object. For example, if you call begin() on a const vector, it will invoke the const version of begin() and return a const_iterator. If begin() is invoked on a non-const vector, then the non-const version of begin() will be invoked and the function will yield a regular iterator. We will see some examples of const-overloading in upcoming sections.

Limitations of const

Although const is a useful programming construct, certain aspects of const are counterintuitive and can lead to subtle violations of constness. One common problem arises when using pointers in const member functions. Suppose you have the following class that encapsulates a C string:

```
class CString
{
public:
    /* ... other members ...*/
    void constFunction() const;

private:
    char* theString;
};
```

Consider the following legal implementation of constFunction:

```
void CString::constFunction() const
{
    strcpy(theString, "Oh no!");
}
```

Unfortunately, while this code modifies the value of the object pointed to by theString, it is perfectly legal C++ code because it doesn't modify the value of theString – instead, it modifies the value of the elements *pointed at* by myString. In effect, because the member function is declared const, theString acts as a *const pointer* (the point*er* can't change) instead of a *pointer-to-const* (the point*ee* can't change). This raises the issue of the distinction between “bitwise constness” and “semantic constness.” *Bitwise constness*, which is the type enforced by C++, means that const objects are prohibited from making any bitwise changes to themselves. In the above example, since the value of the theString pointer didn't change, C++ considers the constFunction implementation const-correct. However, from the viewpoint of *semantic constness*, const classes should be prohibited from modifying anything that would make the object appear somehow different. With regards to the above scenario with theString, the class isn't semantically const because the object, while const, was able to modify its data.

When working with const it's important to remember that while C++ will enforce bitwise constness, you must take care to ensure that your program is semantically const. From your perspective as a programmer, if you invoke a const member function on an object, you would expect the receiver to be unchanged. If the function isn't semantically const, however, this won't be the case, and a const member function might make significant changes to the object's state.

To demonstrate the difference between bitwise and semantically const code, let's consider another member function of the CString class that simply returns the internally stored string:

```
char* CString::getString() const
{
    return theString;
}
```

Initially, this code looks correct. Since returning theString doesn't modify the receiver object, the function is bitwise const. But this code entirely bypasses constness. Consider, for example, this code:

```
const CString myStr = "This is a C string!";
strcpy(myStr.getString(), "Oh no!");
```

Here, we use the pointer obtained from getString as a parameter to strcpy, overwriting the contents of the string with the string “Oh no!” Although myStr is marked const in this example, we somehow have changed its contents. This entirely defeats the purpose of const and should convey why maintaining semantic constness is a crucial part of good programming practice.

The above implementation of getString is fatally flawed and allows clients to subvert the constness of the receiver object. How can we modify getString so that the above code no longer works? One particularly elegant solution is to modify the signature of getString so that it returns a const char* instead of a raw char*. For example:

```
const char* CString::getString() const
{
    return theString;
}
```

Because the returned C string has been marked const, clients cannot modify any of the characters in the returned sequence. As a general rule of thumb, avoid returning non-const pointers from member functions that are marked const. There are exceptions to this rule, of course, but in most cases const functions should return pointers-to-const.

mutable

Because C++ enforces bitwise constness rather than semantic constness, you might find yourself in a situation where a member function changes an object's bitwise representation while still being semantically const. At first this might seem unusual – how could we possibly leave the object in the same logical state if we change its binary representation? – but such situations can arise in practice. For example, suppose that we want to write a class that represents a grocery list. The class definition is provided here:

```
class GroceryList
{
public:
    GroceryList(const string& filename); // Load from a file.

    /* ... other member functions ... */

    string getItemAt(int index) const;

private:
    vector<string> data;
};
```

The `GroceryList` constructor takes in a filename representing a grocery list (with one element per line), then allows us to look up items in the list using the member function `getItemAt`. Initially, we might want to implement this class as follows:

```
GroceryList::GroceryList(const string& filename)
{
    /* Read in the entire contents of the file and store in the vector. */
    ifstream input(filename.c_str());
    data.insert(data.begin(), istream_iterator<string>(input),
                              istream_iterator<string>());
}

/* Returns the element at the position specified by index. */
string GroceryList::getItemAt(int index) const
{
    return data[index];
}
```

Here, the `GroceryList` constructor takes in the name of a file and reads the contents of that file into a `vector<string>` called `data`. The `getItemAt` member function then accepts an index and returns the corresponding element from the `vector`. While this implementation works correctly, in many cases it is needlessly inefficient. Consider the case where our grocery list is several million lines long (maybe if we're literally trying to find enough food to feed an army), but where we only need to look at the first few elements of the list. With the current implementation of `GroceryList`, the `GroceryList` constructor will read in the entire grocery list file, an operation which undoubtedly will take a long time to finish and dwarfs the small time necessary to retrieve the stored elements. How can we resolve this problem?

There are several strategies we could use to eliminate this inefficiency. Perhaps the easiest approach is to have the constructor open the file, and then to only read in data when it's explicitly requested in the `getItemAt` function. That way, we don't read any data unless it's absolutely necessary. Here is one possible implementation:

```
class GroceryList
{
public:
    GroceryList(const string& filename);

    /* ... other member functions ... */

    string getItemAt(int index); // Problem: No longer const

private:
    vector<string> data;
    ifstream sourceStream;
};

GroceryList::GroceryList(const string& filename)
{
    sourceStream.open(filename.c_str()); // Open the file.
}

string GroceryList::getItemAt(int index)
{
    /* Read in enough data to satisfy the request.  If we've already read it
     * in, this loop will not execute and we won't read any data.
     */
    while(index < data.length())
    {
        string line;
        getline(sourceStream, line);

        data.push_back(line);
    }
    return data[index];
}
```

Unlike our previous implementation, the new `GroceryList` constructor opens the file without reading any data. The new `getItemAt` function is slightly more complicated. Because we no longer read all the data in the constructor, when asked for an element, one of two cases will be true. First, we might have already read in the data for that line, in which case we simply hand back the value stored in the `data` object. Second, we may need to read more data from the file. In this case, we loop reading data until there are enough elements in the `data vector` to satisfy the request, then return the appropriate string.

Although this new implementation is more efficient,[*] the `getItemAt` function can no longer be marked `const` because it modifies both the `data` and `sourceStream` data members. If you'll notice, though, despite the fact that the `getItemAt` function is not bitwise `const`, it is semantically `const`. `GroceryList` is supposed to encapsulate an immutable grocery list, and by shifting the file reading from the constructor to `getItemAt` we have only changed the implementation, not the guarantee that `getItemAt` will not modify the list. We've reached an impasse: the interface for `GroceryList` should not depend on its implementation, and so the `getItemAt` function should be marked `const`. However, we have just produced a perfectly reasonable implementation of `GroceryList` that is not bitwise `const`, meaning that the interface needs to change to accommodate the implementation. Given our two conflicting needs – good interface design and good implementation design – how can we strike a balance?

For situations such as these, where a function is semantically `const` but not bitwise `const`, C++ provides the `mutable` keyword. `mutable` is an attribute that can be applied to data members that indicates that

* The general technique of deferring computations until they are absolutely required is called *lazy evaluation* and is an excellent way to improve program efficiency.

those data members can be modified inside member functions that are marked `const`. Using `mutable`, we can rewrite the `GroceryList` class definition to look like this:

```
class GroceryList
{
public:
    GroceryList(const string& filename); // Load from a file.

    /* ... other member functions ... */

    string getItemAt(int index) const; // Now marked const

private:
    /* These data members now mutable. */
    mutable vector<string> data;
    mutable ifstream sourceStream;
};
```

Because `data` and `sourceStream` are both `mutable`, the new implementation of `getItemAt` can now be marked `const`, as shown above.

`mutable` is a special-purpose keyword that should be used sparingly and with caution. Mutable data members are exempt from the type-checking rules normally applied to `const` and consequently are prone to the same errors as non-`const` variables. Also, once data members have been marked `mutable`, *any* member function can modify them, so be sure to double-check your code for correctness. Most importantly, though, do not use `mutable` to silence compiler warnings and errors unless you're absolutely certain that it's the right thing to do. If you do, you run the risk of having functions marked `const` that are neither bitwise nor semantically `const`, entirely defeating the purpose of the `const` keyword.

`const`-Correctness

> *I still sometimes come across programmers who think `const` isn't worth the trouble. "Aw, `const` is a pain to write everywhere," I've heard some complain. "If I use it in one place, I have to use it all the time. And anyway, other people skip it, and their programs work fine. Some of the libraries that I use aren't `const`-correct either. Is `const` worth it?"*
>
> *We could imagine a similar scene, this time at a rifle range: "Aw, this gun's safety is a pain to set all the time. And anyway, some other people don't use it either, and some of them haven't shot their own feet off..."*
>
> *Safety-incorrect riflemen are not long for this world. Nor are `const`-incorrect programmers, carpenters who don't have time for hard-hats, and electricians who don't have time to identify the live wire. There is no excuse for ignoring the safety mechanisms provided with a product, and there is particularly no excuse for programmers too lazy to write `const`-correct code.*
>
> – Herb Sutter, author of *Exceptional C++* and all-around C++ guru. [Sut98]

Now that you're familiar with the mechanics of `const`, we'll explore how to use `const` correctly in real-world C++ code. In the remainder of this section, we will explore *`const`-correctness*, a system for using `const` to indicate the effects of your functions (or lack thereof). From this point forward, *all* of the code in this book will be `const`-correct and you should make a serious effort to `const`-correct your own code.

What is `const`-correctness?

At a high-level, `const`-correct code is code that clearly indicates which variables and functions cannot modify program state. More concretely, `const`-correctness requires that `const` be applied consistently and pervasively. In particular, `const`-correct code tends to use `const` as follows:

- **Objects are never passed by value**. Any object that would be passed by value is instead passed by reference-to-`const` or pointer-to-`const`.
- **Member functions which do not change state are marked `const`.** Similarly, a function that is not marked `const` should mutate state somehow.
- **Variables which are set but never modified are marked `const`**. Again, a variable not marked `const` should have its value changed at some point.

Let us take some time to explore the ramifications of each of these items individually.

Objects are never passed by value

C++ has three parameter-passing mechanisms – pass-by-value, pass-by-reference, and pass-by-pointer. The first of these requires C++ to make a full copy of the parameter being passed in, while the latter two initialize the parameter by copying a pointer to the object instead of the full object.* When passing primitive types (`int`, `double`, `char*`, etc.) as parameters to a function, the cost of a deep copy is usually negligible, but passing a heavy object like a `string`, `vector`, or `map` can at times be as expensive as the body of the function using the copy. Moreover, when passing objects by value to a function, those objects also need to be cleaned up by their destructors once that function returns. The cost of passing an object by value is thus at least the cost of a call to the class's copy constructor (discussed in a later chapter) and a call to the destructor, whereas passing that same object by reference or by pointer simply costs a single pointer copy.

To avoid incurring the overhead of a full object deep-copy, you should avoid passing objects by value into functions and should instead opt to pass either by reference or by pointer. To be `const`-correct, moreover, you should consider passing the object by reference-to-`const` or by pointer-to-`const` if you don't plan on mutating the object inside the function. In fact, you can treat pass-by-reference-to-`const` or pass-by-pointer-to-`const` as the smarter, faster way of passing an object by value. With both pass-by-value and pass-by-reference-to-`const`, the caller is guaranteed that the object will not change value inside the function call.

There is one difference between pass-by-reference-to-`const` and pass-by-value, though, and that's when using pass-by-value the function gets a fresh object that it is free to destructively modify. When using pass-by-reference-to-`const`, the function cannot mutate the parameter. At times this might be a bit vexing. For example, consider the `ConvertToLowerCase` function we wrote in the earlier chapter on STL algorithms:

```
string ConvertToLowerCase(string toConvert)
{
    transform(toConvert.begin(), toConvert.end(),
              toConvert.begin(),
              ::tolower);
    return toConvert;
}
```

* References are commonly implemented behind-the-scenes in a manner similar to pointers, so passing an object by reference is at least as efficient as passing an object by pointer.

Here, if we simply change the parameter from being passed-by-value to being passed-by-reference-to-const, the code won't compile because we modify the toConvert variable. In situations like these, it is sometimes preferable to use pass-by-value, but alternatively we can rewrite the function as follows:

```
string ConvertToLowerCase(const string& toConvert)
{
    string result = toConvert;
    transform(result.begin(), result.end(), result.begin(), ::tolower);
    return result;
}
```

Here, we simply create a new variable called result, initialize it to the parameter toConvert, then proceed as in the above function.

Member functions which do not change state are `const`

If you'll recall from our earlier discussion of const member functions, when working with const instances of a class, C++ only allows you to invoke member functions which are explicitly marked const. No matter how innocuous a function is, if it isn't explicitly marked const, you cannot invoke it on a const instance of an object. This means that when designing classes, you should take great care to mark const every member function that does not change the state of the object. Is this a lot of work? Absolutely! Does it pay off? Of course!

As an extreme example of why you should always mark nonmutating member functions const, suppose you try to pass a CS106B/X Vector to a function by reference-to-const. Since the Vector is marked as const, you can only call Vector member functions that themselves are const. Unfortunately, *none* of the Vector's member functions are const, so you can't call *any* member functions of a const Vector. A const CS106B/X Vector is effectively a digital brick. As fun as bricks are, from a functional standpoint they're pretty much useless, so do make sure to constify your member functions.

If you take care to const correct all member functions that don't modify state, then your code will have an additional, stronger property: member functions which are *not* marked const are guaranteed to make some sort of change to the receiver's internal state. From an interface perspective this is wonderful – if you want to call a particular function that isn't marked const, you can almost guarantee that it's going to make some form of modification to the receiver object. Thus when you're getting accustomed to a new code base, you can quickly determine what operations on an object modify that object and which just return some sort of internal state.

Variables which are set but never changed are `const`

Variables vary. That's why they're called variables. Constants, on the other hand, do not. Semantically, there is a huge difference between the sorts of operations you can perform on constants and the operations you can perform on variables, and using one where you meant to use the other can cause all sorts of debugging headaches. Using const, we can make explicit the distinction between constant values and true variables, which can make debugging and code maintenance much easier. If a variable is const, you cannot inadvertently pass it by reference or by pointer to a function which subtly modifies it, nor can you accidentally overwrite it with = when you meant to check for equality with ==. Many years after you've marked a variable const, programmers trying to decipher your code will let out a sigh of relief as they realize that they don't need to watch out for subtle operations which overwrite or change its value.

Without getting carried away, you should try to mark as many local variables const as possible. The additional compile-time safety checks and readability will more than compensate for the extra time you spent typing those extra five characters.

Example: CS106B/X Map

As an example of what `const`-correctness looks like in practice, we'll consider how to take a variant of the CS106B/X `Map` class and modify it so that it is `const`-correct. The initial interface looks like this:

```
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size();
    bool isEmpty();

    void put(string key, ValueType value);
    void remove(string key);
    bool containsKey(string key);

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(string key);
    ValueType& operator[](string key);

    void clear();

    void mapAll(void fn(string key, ValueType val));

    template <typename ClientDataType>
    void mapAll(void fn(string key, ValueType val, ClientDataType& data),
                ClientDataType& data);

    Iterator iterator();

private:
    /* ... Implementation specific ... */
};
```

The `operator[]` function shown here is what's called an *overloaded operator* and is the function that lets us write code to the effect of `myMap["Key"] = value` and `value = myMap["Key"]`. We will cover overloaded operators in a later chapter, but for now you can think of it simply as a function that is called whenever the `Map` has the element-selection brackets applied to it.

The first set of changes we should make to the `Map` is to mark all of the public member functions which don't modify state `const`. This results in the following interface:

```
/* Note: Still more changes to make.  Do not use this code as a reference! */
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(string key, ValueType value);
    void remove(string key);
    bool containsKey(string key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(string key) const;
    ValueType& operator[](string key);

    void clear();

    void mapAll(void fn(string key, ValueType val)) const;
    template <typename ClientDataType>
    void mapAll(void fn(string key, ValueType val, ClientDataType& data),
                ClientDataType& data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};
```

The `size`, `isEmpty`, and `containsKey` functions are all `const` because they simply query object properties without changing the `Map`. `get` is also `const` since accessing a key/value pair in the `Map` does not actually modify the underlying state, but `operator[]` should definitely *not* be marked `const` because it may update the container if the specified key does not exist.

The trickier functions to `const`-correct are `mapAll` and `iterator`. Unlike the STL iterators, CS106B/X iterators are read-only and can't modify the underlying container. Handing back an iterator to the `Map` contents therefore cannot change the `Map`'s contents, so we have marked `iterator const`. In addition, since `mapAll` passes its arguments to the callback function by value, there is no way for the callback function to modify the underlying container. It should therefore be marked `const`.

Now that the interface has its member functions `const`-ified, we should make a second pass over the `Map` and replace all instances of pass-by-value with pass-by-reference-to-`const`. In general, objects should never be passed by value and should always be passed either by pointer or reference with the appropriate `const`ness. This eliminates unnecessary copying and can make programs perform asymptotically better. The resulting class looks like this:

```
/* Note: Still more changes to make.  Do not use this code as a reference! */
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(const string& key, const ValueType& value);
    void remove(const string& key);
    bool containsKey(const string& key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(const string& key) const;
    ValueType& operator[](const string& key);

    void clear();

    void mapAll(void (fn)(const string& key, const ValueType& val)) const;
    template <typename ClientDataType>
    void mapAll(void (fn)(const string& key, const ValueType& val,
                          ClientDataType& data),
                ClientDataType &data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};
```

The parameters to `put`, `remove`, `containsKey`, `get`, and `operator[]` have all been updated to use pass-by-reference-to-`const` instead of pass-by-value. The trickier functions to modify are the `mapAll` functions. These functions themselves accept function pointers which initially took their values by value. We have updated them appropriately so that the function pointers accept their arguments by reference-to-`const`, since we assume that the class client will also be `const`-correct. Note that we did *not* mark the `ClientDataType&` parameter to `mapAll` `const`, since the `Map` client may actually want to modify that parameter.

There is one last change to make, and it concerns the `get` function, which currently returns a copy of the value associated with a given key. At a high-level, there is nothing intuitively wrong with returning a copy of the stored value, but from an efficiency standpoint we may end up paying a steep runtime cost by returning the object by value. After all, this requires a full object deep copy, plus a call to the object's destructor once the returned object goes out of scope. Instead, we'll modify the interface such that this function returns the object by reference-to-`const`. This allows the `Map` client to look at the value and, if they choose, copy it, but prevents clients from intrusively modifying the `Map` internals through a `const` function. The final, correct interface for `Map` looks like this:

```
/* const-corrected version of the CS106B/X Map. */
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(const string& key, const ValueType& value);
    void remove(const string& key);
    bool containsKey(const string& key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    const ValueType& get(const string& key) const;
    ValueType& operator[](const string& key);

    void clear();

    void mapAll(void (fn)(const string& key, const ValueType& val)) const;
    template <typename ClientDataType>
    void mapAll(void (fn)(const string& key, const ValueType& val,
                          ClientDataType& data),
                ClientDataType &data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};
```

As an interesting intellectual exercise, compare this code to the original version of the `Map`. The interface declaration is considerably longer than before because of the additional `const`s, but ultimately is more pleasing. Someone unfamiliar with the interface can understand, for example, that the `Map`'s `Iterator` type cannot modify the underlying container (since otherwise the `iterator()` function wouldn't be `const`), and can also note that `mapAll` allows only a read-only map operation over the `Map`. This makes the code more self-documenting, a great boon to programmers responsible for maintaining this code base in the long run.

Why be `const`-correct?

As you can see from the example with the CS106B/X `Map`, making code `const`-correct can be tricky and time-consuming. Indeed, typing out all the requisite `const`s and `&`s can become tedious after a while. So why should you want to be `const`-correct in the first place?

There are multiple reasons why code is better off `const`-correct than non-`const`-correct. Here are a few:

- **Code correctness**. If nothing else, marking code `const` whenever possible reduces the possibility for lurking bugs in your code. Because the compiler can check which regions of the code are and are not mutable, your code is less likely to contain logic errors stemming either from a misuse of an interface or from a buggy implementation of a member function.

- **Code documentation**. const-correct code is self-documenting and clearly indicates to other programmers what it is and is not capable of doing. If you are presented an interface for an entirely foreign class, you may still be able to figure out which methods are safe to call with important data by noting which member functions are const or accept parameters by reference-to-const.

- **Library integration**. The C++ standard libraries and most third-party libraries are fully const-correct and expect that any classes or functions that interface with them to be const-correct as well. Writing code that is not const-correct can prevent you from fully harnessing the full power of some of these libraries.

Optimizing Construction with Member Initializer Lists

We've just concluded a whirlwind tour of const, and now it's time to change gears and talk about an entirely different aspect of class design: the member initializer list.

Normally, when you create a class, you'll initialize all of its data members in the body constructor. However, in some cases you'll need to initialize instance variables before the constructor begins running. Perhaps you'll have a const instance variable that you cannot assign a value, or maybe you have an object as an instance variable where you do not want to use the default constructor. For situations like these, C++ has a construct called the *member initializer list* that you can use to fine-tune the way your data members are set up. This section discusses initializer list syntax, situations where initializer lists are appropriate, and some of the subtleties of initializer lists.

How C++ Constructs Objects

To fully understand why initializer lists exist in the first place, you'll need to understand the way that C++ creates and initializes new objects.

Let's suppose you have the following class:

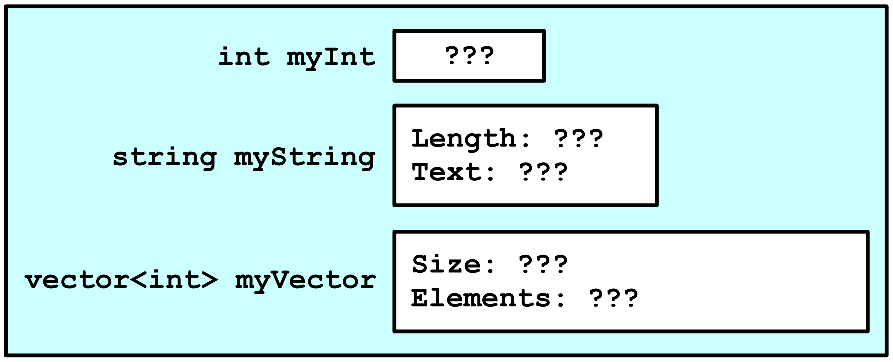
```
class SimpleClass
{
public:
    SimpleClass();

private:
    int myInt;
    string myString;
    vector<int> myVector;
};
```

Let's define the SimpleClass constructor as follows:

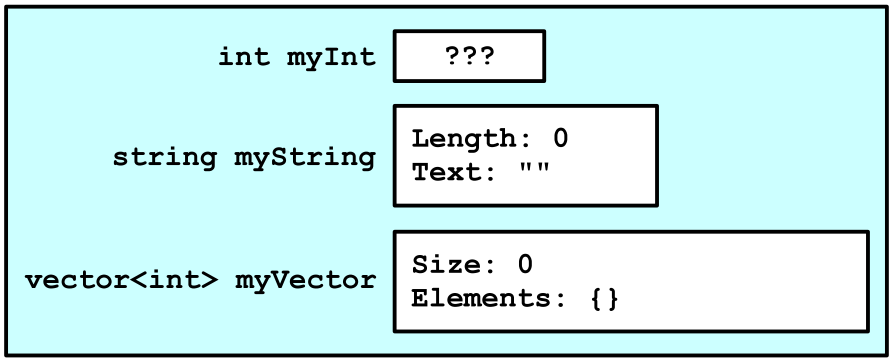
```
SimpleClass::SimpleClass()
{
    myInt = 5;
    myString = "C++!";
    myVector.resize(10);
}
```

What happens when you create a new instance of the class MyClass? It turns out that the simple line of code MyClass mc actually causes a cascade of events that goes on behind the scenes. Let's take a look at what happens, step-by-step.

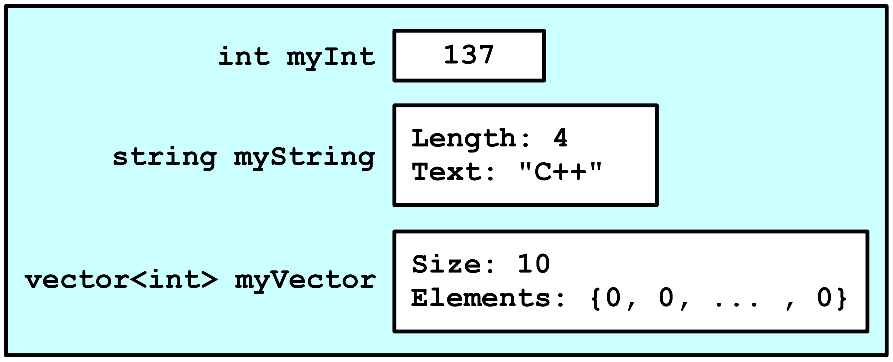
The first step in constructing a C++ object is simply to get enough space to hold all of the object's data members. The memory is not initialized to any particular value, so initially all of your object's data members hold garbage values. In memory, this looks something like this:



As you can see, none of the instance variables have been initialized, so they all contain junk. At this point, C++ calls the default constructor of each instance variable. For primitive types, this leaves the variables unchanged. After this step, our object looks something like this:



Finally, C++ will invoke the object's constructor so you can perform any additional initialization code. Using the constructor defined above, the final version of the new object will look like this:



At this point, our object is fully-constructed and ready to use.

However, there's one thing to consider here. Before we reached the `SimpleClass` constructor, C++ called the default constructor on both `myString` and `myVector`. `myString` was therefore initialized to the empty string, and `myVector` was constructed to hold no elements. However, in the `SimpleClass` constructor, we immediately assigned `myString` to hold "C++!" and resized `myVector` to hold ten elements. This means that we effectively initialized `myString` and `myVector` *twice* – once with their default constructors and once in the `SimpleClass` constructor.*

To improve efficiency and resolve certain other problems which we'll explore later, C++ has a feature called an *initializer list*. An initializer list is simply a series of values that C++ will use instead of the default values to initialize instance variables. For example, in the above example, you can use an initializer list to specify that the variable `myString` should be set to "C++!" before the constructor even begins running.

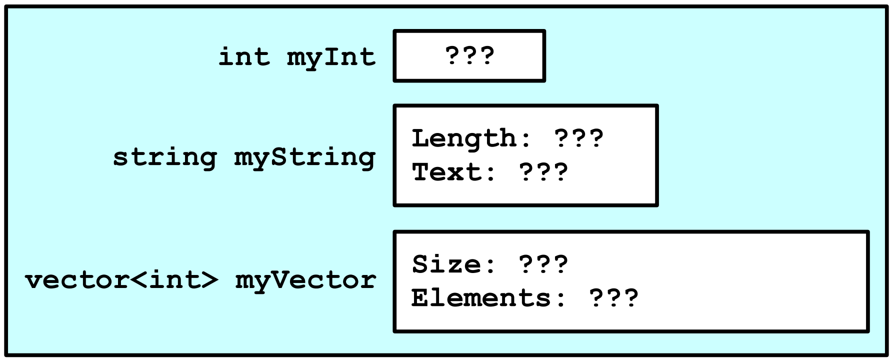
To use an initializer list, you add a colon after the constructor and then list which values to initialize which variables with. For example, here's a modified version of the `SimpleClass` constructor that initializes all the instance variables in an initializer list instead of in the constructor:

```
SimpleClass::SimpleClass() : myInt(5), myString("C++!"), myVector(10)
{
    // Note: Empty constructor
}
```

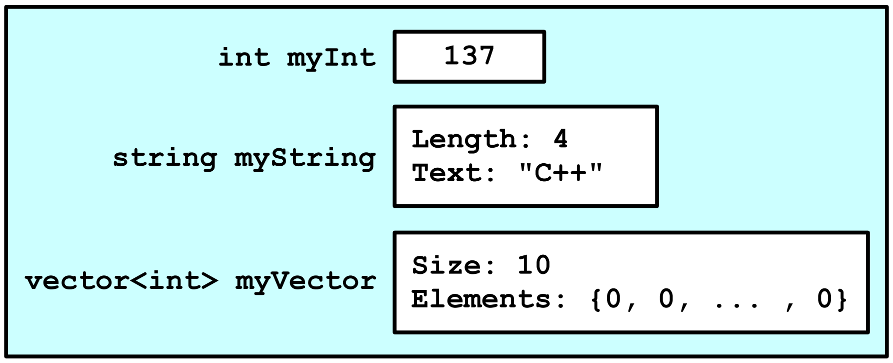
Here, we're telling C++ to initialize the variables `myInt` and `myString` to 5 and "C++!," respectively, before the class constructor is even called. Also, by writing `myVector(10)`, we're telling C++ to invoke the parametrized constructor of `myVector` passing in the value 10, which creates a `vector` with ten elements. This time, when we create a new object of type `myVector`, the creation steps will look like this:

First, as in the previous case, the object is allocated somewhere in memory and all variables have garbage values:

* Technically speaking, the objects are only initialized once, but the runtime efficiency is as though the objects were initialized multiple times. We'll talk about the differences between initialization and assignment in a later chapter.



Next, C++ invokes all of the constructors for the object's data members using the values specified in the initializer list. The object now looks like this:



Finally, C++ invokes the `MyClass` constructor, which does nothing. The final version of the class thus is identical to the above version.

As you can see, the values of the instance variables `myInt`, `myString`, and `myVector` are correctly set before the `SimpleClass` constructor is invoked. This is considerably more efficient than the previous version and will run much faster.

Note that while in this example we used initializer lists to initialize all of the object's instance variables, there is no requirement that you do so. However, in practice it's usually a good idea to set up all variables in an initializer list to make clear what values you want for each of your data members.

Parameters in Initializer Lists

In the above example, the initializer list we used specified constant values for each of the data members. However, it's both legal and useful to initialize data members with expressions instead of literal constants. For example, consider the following class, which encapsulates a C-style string:

```
class CString
{
public:
    CString(const char* input);
    ~CString();

    /* ... */
private:
    char* str;
};
```

Here, the constructor accepts a C-style string, then initializes the class to hold a copy of that string. Assuming that we have a `StringDuplicate` function like the one described in the chapter on C strings, we could write the constructor for `CString` as follows:

```
CString::CString(const char* input) : str(StringDuplicate(input))
{
    // Empty constructor
}
```

Notice that we were able to reference the constructor parameter `input` inside the initializer list. This allows us to use information specific to the current instance of the class to perform initialization and is ubiquitous in C++ code.

In some cases you may have a constructor that accepts a parameter with the same name as a class data member. For example, consider this the `RationalNumber` class, which encapsulates a rational number:

```
class RationalNumber
{
public:
    RationalNumber(int numerator = 0, int denominator = 1);

    /* ... */
private:
    int numerator, denominator;
};
```

The following is a perfectly legal constructor that initializes the data members to the values specified as parameters to the function:

```
RationalNumber::RationalNumber(int numerator, int denominator) :
    numerator(numerator), denominator(denominator)
{
    // Empty constructor
}
```

C++ is smart enough to realize that the syntax `numerator(numerator)` means to initialize the `numerator` data member to the value held by the `numerator` parameter, rather than causing a compile-time error or initializing the `numerator` data member to itself. Code of this form might indicate that you need to rename the parameters to the constructor, but is perfectly legal.

On an unrelated note, notice that in the `RationalNumber` class declaration we specified that the `numerator` and `denominator` parameters to `RationalNumber` were equal to zero and one, respectively. These are default arguments to the constructor and allow us to call the constructor with fewer than two parameters. If we don't specify the parameters, C++ will use these values instead. For example:

```
RationalNumber fiveHalves(5, 2);
RationalNumber three(3); // Calls constructor with arguments (3, 1)
RationalNumber zero; // Calls constructor with arguments (0, 1)
```

You can use default arguments in any function, provided that if a single parameter has a default argument every parameter after it also has a default. Thus the following code is illegal:

```
void DoSomething(int x = 5, int y); // Error – y needs a default
```

While the following is legal:

```
void DoSomething(int x, int y = 5); // Legal
```

When writing functions that take default arguments, you should *only* specify the default arguments in the function prototype, not the function definition. If you don't prototype the function, however, you should specify the defaults in the definition. C++ is very strict about this and even if you specify the same defaults in both the prototype and definition the compiler will complain.

When Initializer Lists are Mandatory

Initializer lists are useful from an efficiency standpoint. However, there are times where initializer lists are the only syntactically legal way to set up your instance variables.

Suppose we'd like to make an object called `Counter` that supports two functions, `increment` and `decrement`, that adjust an internal counter. However, we'd like to add the restriction that the `Counter` can't drop below 0 or exceed a user-defined limit. Thus we'll use a parametrized constructor that accepts an `int` representing the maximum value for the `Counter` and stores it as an instance variable. Since the value of the upper limit will never change, we'll mark it `const` so that we can't accidentally modify it in our code. The class definition for `Counter` thus looks something like this:

```
class Counter
{
public:
    Counter(int maxValue);

    void increment();
    void decrement();
    int getValue() const;

private:
    int value;
    const int maximum;
};
```

Then we'd *like* the constructor to look like this:

```
Counter::Counter(int maxValue)
{
    value = 0;
    maximum = maxValue; // ERROR!
}
```

Unfortunately, the above code isn't valid because in the second line we're assigning a value to a variable marked `const`. Even though we're in the constructor, we still cannot violate the sanctity of `const`ness. To fix this, we'll initialize the value of `maximum` in the initializer list, so that `maximum` will be *initialized* to the value of maxValue, rather than *assigned* the value maxValue. This is a subtle distinction, so make sure to think about it before proceeding.

The correct version of the constructor is thus

```
Counter::Counter(int maxValue) : value(0), maximum(maxValue)
{
    // Empty constructor
}
```

Note that we initialized maximum based on the constructor parameter maxValue. Interestingly, if we had forgotten to initialize maximum in the initializer list, the compiler would have reported an error. In C++, it is *mandatory* to initialize all const primitive-type instance variables in an initializer list. Otherwise, you'd have constants whose values were total garbage.

Another case where initializer lists are mandatory arises when a class contains objects with no legal or meaningful default constructor. Suppose, for example, that you have an object that stores a CS106B/X Set of a custom type customT with comparison callback MyCallback. Since the Set requires you to specify the callback function in the constructor, and since you're always going to use MyCallback as that parameter, you might think that the syntax looks like this:

```
class SetWrapperClass
{
public:
    SetWrapperClass();

private:
    Set<customT> mySet(MyCallback); // Problem: Need a comparison function
};
```

Unfortunately, this isn't legal C++ syntax. However, you can fix this by rewriting the class as

```
class SetWrapperClass
{
public:
    SetWrapperClass();

private:
    Set<customT> mySet; // Note: no parameters specified
};
```

And then initializing mySet in the initializer list as

```
SetWrapperClass::SetWrapperClass() : mySet(MyCallback)
{
    // Yet another empty constructor!
}
```

Now, when the object is created, mySet will have MyCallback passed to its constructor and everything will work out correctly.

Multiple Constructors

If you write a class with multiple constructors (which, after we discuss of copy constructors, will be most of your classes), you'll need to make initializer lists for each of your constructors. That is, an initializer list for one constructor won't invoke if a different constructor is called.

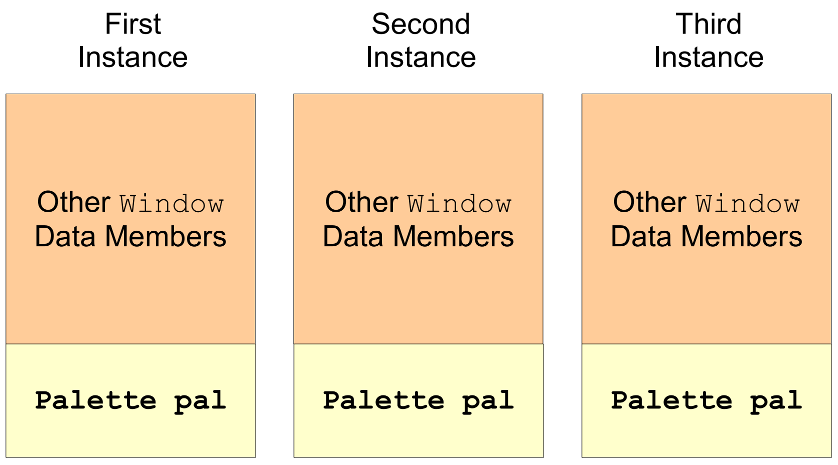
Sharing Information With `static`

Suppose that we're developing a windowed operating system and want to write the code that draws windows on the screen. We decide to create a class `Window` that exports a `drawWindow` function. In order to display the window correctly, `drawWindow` needs access to a `Palette` object that performs primitive rendering operations like drawing lines, arcs, and filled polygons. Assume that we know that the window will always be drawn with the same `Palette`. Given this description, we might initially design `Window` so that it has a `Palette` as a data member, as shown here:

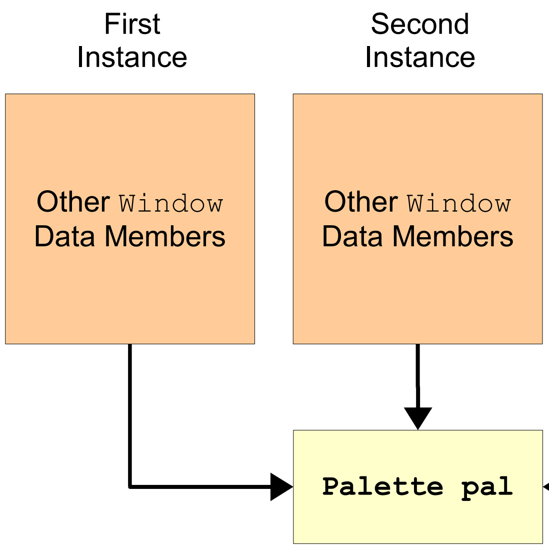
```
class Window
{
public:
    /* ... constructors, destructors, etc. ...*/

    /* All windows can draw themselves. */
    void drawWindow();
private:
    /* ... other data members ... */
    Palette pal;
};
```

Now, every window has its own palette and can draw itself appropriately.

There's nothing fundamentally wrong with this setup, but it contains a small flaw. Let's suppose that we have three different window objects. In memory, those objects would look like this:



Since `pal` is a data member of `Window`, every `Window` has its own `Palette`. There might be an arbitrary number of windows on screen at any time, but there's only one screen and it doesn't make sense for every window to have its own palette. After all, each window is likely to use a similar set of colors as those used by every other window, and it seems more reasonable for every window to share a single palette, as shown here:


First Instance
Second Instance
Third Instance
Other Window Data Members
Other Window Data Members
Other Window Data Members
Palette pal

How can we model this in code? Using the techniques so far, we have few options. First, we could create a global `Palette` object, then have each `Window` use this global `Palette`. This is a particularly bad choice for two reasons:

- **It uses global variables**. Independently of any other strengths and weaknesses of this approach, global variables are a big programming no-no. Globals can be accessed and modified anywhere in the program, making debugging difficult should problems arise. It is also possible to inadvertently reference global variables inside of unrelated functions, leading to subtle bugs that can take down the entire program.

- **It lacks encapsulation**. Because the `Palette` is a global variable, other parts of the program can modify the `Window` `Palette` without going through the `Window` class. This leads to the same sorts of problems possible with public data members: class invariants breaking unexpectedly, code written with one version of `Window` breaking when the `Window` is updated, etc.

Second, we could have each `Window` object contain a *pointer* to a `Palette` object, then pass a shared `Palette` as a parameter to each instance of `Window`. For example, we could design the class like this:

```
class Window
{
public:
    Window(Palette* p, /* ... */);
    /* ... other constructors, destructors, etc. ...*/

    /* All windows can draw themselves. */
    void drawWindow();

private:
    /* ... other data members ... */
    Palette* pal;
};
```

This allows us to share a single `Palette` across multiple `Window`s and looks remarkably like the above diagram. However, this approach has its weaknesses:

- **It complicates `Window` creation**. Let's think about how we would go about creating `Window`s with this setup. Before creating our first `Window`, we'd need to create a `Palette` to associate with it, as shown here:

  ```
  Palette* p = new Palette;
  Window*  w = new Window(p, /* ... */);
  ```

  If later we want to create more `Window`s, we'd need to keep track of the original `Palette` we used so that we can provide it as a parameter to the `Window` constructor. This means that any part of the program responsible for `Window` management needs to know about the shared `Palette`.

- **It violates encapsulation**. Clients of `Window` shouldn't have to know how `Window`s are implemented, and by requiring `Window` users to explicitly manage the shared `Palette` we're exposing too much detail about the `Window` class. This approach also locks us in to a fixed implementation. For example, what if we want to switch from `Palette` to a `TurboPalette` that draws twice as quickly? With the current approach all `Window` clients would need to upgrade their code to match the new implementation.

- **It complicates resource management**. Who is responsible for cleaning up the `Window` `Palette` at the end of the program? `Window` clients shouldn't have to, since the `Palette` really belongs to the `Window` class. But no particular `Window` owns the `Palette`, since each instance of `Window` shares a single `Palette`. There are systems we could use to make cleanup work correctly (see the later extended example on smart pointers for one possibility), but they increase program complexity.

Both of these approaches have their individual strengths, but have drawbacks that outweigh their benefits. Let's review exactly what we're trying to do. We'd like to have a single `Palette` that's shared across multiple different `Window`s. Moreover, we'd like this `Palette` to obey all of the rules normally applicable to class design: it should be encapsulated and it should be managed by the class rather than clients. Using the techniques we've covered so far it is difficult to construct a solution with these properties. For a clean solution, we'll need to introduce a new language feature: *static data members*.

Static Data Members

Static data members are data members associated with a class as a whole rather than a particular instance of that class. In the above example with `Window` and `Palette`, the window `Palette` is associated with *`Window`s in general* rather than any one specific `Window` object and is an ideal candidate for a static data member.

In many ways static data members behave similarly to regular data members. For example, if a class has a private static data member, only member functions of the class can access that variable. However, static data members behave differently from other data members because there is only one copy of each static data member. Each instance of a class containing a static data member shares the same version of that data member. That is, if a single class instance changes a static data member, the change affects all instances of that class.

The syntax for declaring static data members is slightly more complicated than for declaring nonstatic data members. There are two steps: *declaration* and *definition*. For example, if we want to create a static `Palette` object inside of `Window`, we could *declare* the variable as shown here:

```
class Window
{
public:
    /* ... constructors, destructors, etc. ...*/

    /* All windows can draw themselves. */
    void drawWindow();

private:
    /* ... other data members ... */
    static Palette sharedPal;
};
```

Here, `sharedPal` is declared as a static data member using the `static` keyword. But while we've *declared* `sharedPal` as a static data member, we haven't *defined* `sharedPal` yet. Much in the same way that functions are separated into prototypes (declarations) and implementations (definitions), static data members have to be both declared inside the class in which they reside and defined inside the .cpp file associated with that class. For the above example, inside the .cpp file for the `Window` class, we would write

```
Palette Window::sharedPal;
```

There are two important points to note here. First, when defining the `static` variable, we must use the fully-qualified name (`Window::sharedPal`) instead of just its local name (`sharedPal`). Second, we do *not* repeat the `static` keyword during the variable declaration – otherwise, the compiler will think we're doing something completely different (see the “More to Explore” section). You may have noticed that even though `Window::sharedPal` is private we're still allowed to use it outside the class. This is only legal during definition, and outside of this one context it is illegal to use `Window::sharedPal` outside of the `Window` class.

In some circumstances you may want to create a class containing a static data member where the data member needs to take on an initial value. For example, if we want to create a class containing an `int` as a static data member, we would probably want to initialize the `int` to a particular value. Given the following class declaration:

```
class MyClass
{
public:
    void doSomething();

private:
    static int myStaticData;
};
```

It is perfectly legal to initialize `myStaticData` as follows:

```
int MyClass::myStaticData = 137;
```

As you'd expect, this means that `myStaticData` initially holds the value 137.

Although the syntax for creating a static data member can be intimidating, once initialized static data members look just like regular data members. For example, consider the following member function:

```
void MyClass::doSomething()
{
    ++myStaticData; // Modifies myStaticData for all classes
}
```

Nothing here seems all that out-of-the-ordinary and this code will work just fine. Note, however, that modifications to `myStaticData` are visible to all other instances of `MyClass`.

Let's consider another example where static data members can be useful. Suppose that you're debugging the windowing code from before and you're pretty sure that you've forgotten to `delete` all instances of `Window` that you've allocated with `new`. Since C++ won't give you any warnings about this, you'll need to do the instance counting yourself. The number of active instances of a class is class-specific information that doesn't pertain to any specific instance of the object, and this is the perfect spot to use static data members. To handle our instance counting, we'll modify the `Window` definition as follows:

```
class Window
{
public:
    /* ... constructors, destructors, etc. ...*/

    void drawWindow();

private:
    /* ... other data members ... */
    static Palette sharedPal;
    static int numInstances;
};
```

We'll also define the variable outside the class as

```
int Window::numInstances = 0;
```

We know that whenever we create a new instance of a class, the class's constructor will be called. This means that if we increment `numInstances` inside the `Window` constructor, we'll correctly track the number of times the a `Window` has been created. Thus, we'll rewrite the `Window` constructor as follows:

```
Window::Window(/* ... */)
{
    /* ... All older initialization code ... */
    ++numInstances;
}
```

Similarly, we'll decrement `numInstances` in the `Window` destructor. We'll also have the destructor print out a message if this is the last remaining instance so we can see how many instances are left:

```
Window::~Window()
{
    /* ... All older cleanup code ... */
    --numInstances;
    if(numInstances == 0)
        cout << "No more Windows!*" << endl;
}
```

* This is not meant as a slight to Microsoft.

Static Member Functions

Inside of member functions, a special variable called `this` acts as a pointer to the current object. Whenever you access a class's instance variables inside a member function, you're really accessing the instance variables of the `this` pointer. For example, given the following `Point` class:

```
class Point
{
public:
    Point(int xLoc, int yLoc);
    int getX() const;
    int getY() const;

private:
    int x, y;
};
```

If we implement the `Point` constructor as follows:

```
Point::Point(int xLoc, int yLoc)
{
    x = xLoc;
    y = yLoc;
}
```

This code is equivalent to

```
Point::Point(int xLoc, int yLoc)
{
    this->x = xLoc;
    this->y = yLoc;
}
```

How does C++ know what value `this` refers to? The answer is subtle but important. Suppose that we have a `Point` object called `pt` and that we write the following code:

```
int x = pt.getX();
```

The C++ compiler converts this into code along the lines of

```
int x = Point::getX(&pt);
```

Where `Point::getX` is prototyped as

```
int Point::getX(Point *const this);
```

This is not legal C++ code, but illustrates what's going on behind the scenes whenever you call a member function.

The mechanism behind member function calls should rarely be of interest to you as a programmer. However, the fact that an *N*-argument member function is really an (*N*+1)-argument free function can cause problems in a few places. For example, suppose that we want to provide a comparison function for `Point`s that looks like this:

```
class Point
{
public:
    Point(int xLoc, int yLoc);
    int getX() const;
    int getY() const;

    bool compareTwoPoints(const Point& one, const Point& two) const;

private:
    int x, y;
};

bool Point::compareTwoPoints(const Point& one, const Point& two) const
{
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}
```

If you have a `vector<Point>` that you'd like to pass to the STL `sort` algorithm, you'll run into trouble if you try to use this syntax:

```
sort(myVector.begin(), myVector.end(), &Point::compareTwoPoints); // Problem
```

The problem is that `sort` expects a comparison function that takes two parameters and returns a `bool`. However, `Point::compareTwoPoints` takes *three* parameters: two points to compare and an invisible "`this`" pointer. Thus the above code will generate an error.

If you want to define a comparison or predicate function inside of a class, you'll want that member function to not have an invisible `this`. What does this mean from a practical standpoint? A member function without a `this` pointer does not have a receiver object, and thus can only operate on its parameters and any static data members of the class it's declared in (since that data is particular to the class rather than any particular instance). Functions of this sort are called *static member functions* and can be created using the `static` keyword. In the above example with `Point`, we could create the `Point` comparison function as a `static` member function using the following syntax:

```
class Point
{
public:
    Point(int xLoc, int yLoc);

    int getX() const;
    int getY() const;

    static bool compareTwoPoints(const Point& one, const Point& two);

private:
    int x, y;
};

bool Point::compareTwoPoints(const Point& one, const Point& two) const
{
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}
```

Now, the above call to `sort` will work since `compareTwoPoints` would no longer have a `this` pointer.

Unlike static data members, when writing static member functions you do not need to separate the code out into a separate declaration and definition. You may want to do so anyway, though.

Let's return to our earlier example about tracking the number of `Window` instances currently in use. While it's nice that the destructor prints out a message when the last instance has been cleaned up, we'd prefer a more robust model where we can check how many more copies of the class exist. This function is not specific to a particular class instance, so we'll make this function static. We'll call this function `getRemainingInstances` and implement it as shown here:

```
class Window
{
public:
    /* ... constructors, destructors, etc. ...*/
    void drawWindow();

    static int getRemainingInstances();

private:
    /* ... other data members ... */
    static Palette sharedPal;
    static int numInstances;
};

Palette Window::sharedPal;
int Window::numInstances = 0;

int Window::getRemainingInstances()
{
    return numInstances;
}
```

As with static data, note that when defining static member functions, you omit the `static` keyword. Only put `static` inside the class declaration.

You can invoke static member functions either using the familiar `object.method` syntax, or you can use the fully qualified name of the function. For example, with the above example, we could check how many remaining instances there were of the `MyClass` class by calling `getRemainingInstances` as follows:

```
cout << Window::getRemainingInstances() << endl;
```

`const` and `static`

Unfortunately, the `const` and `static` keywords do not always interact intuitively. One of the biggest issues to be aware of is that `const` member functions can modify `static` data. For example, consider the following class:

```
class ConstStaticClass
{
public:
    void constFn() const;

private:
    static int staticData;
};
int ConstStaticClass::staticData = 0;
```

Then the following implementation of `constFn` is completely valid:

```
void ConstStaticClass::constFn() const
{
    ++staticData;
}
```

Although the above implementation of `constFn` increments a static data member, the above code will compile and run without any problems. The reason is that the code doesn't modify the receiver object. Static data members are not associated with a particular class instance, so modifications to static data members do not change the state of any one instance of the class.

Additionally, since `static` member functions don't have a `this` pointer, they cannot be declared `const`. In the case of `getNumInstances`, this means that although the function doesn't modify any class data, we still cannot mark it `const`.

Integral Class Constants

There is one other topic concerning the interaction of `const` and `static`: class constants. Suppose we want to make a constant variable accessible only in the context of a class. What we want is a variable that'-s `const`, so it's immutable, and `static`, so that all copies of the class share the data. It's legal to declare these variables like this:

```
class ClassConstantExample
{
public:
    /* Omitted. */

private:
    static const int MyConstant;
};

const int ClassConstantExample::MyConstant = 137;
```

Note the `const` in the definition of `ClassConstantExample::MyConstant`.

However, since the double declaration/definition can be a bit tedious, C++ has a built-in shorthand you can use when declaring class constants of integral types. That is, if you have a `static const int` or a `stat-ic const char`, you can condense the definition and declaration into a single statement by writing;

```
class ClassConstantExample
{
public:
    /* Omitted. */

private:
    static const int MyConstant = 137; // Condense into a single line
};
```

This shorthand is common in professional code. Be careful when using the shorthand, though, because some older compilers won't correctly interpret it. Also, be aware that this only works with *integral types*, so you cannot initialize a `static const double` or `static const float` this way.

Integrating Seamlessly with Conversion Constructors

When designing classes, you might find that certain data types can logically be converted into objects of the type you're creating. For example, when designing the C++ `string` class, you might note that `char *` C strings could have a defined conversion to `string` objects. In these situations, it may be useful to define *implicit conversions* between the two types. To define implicit conversions, C++ uses *conversion constructors*, constructors that accept a single parameter and initialize an object to be a copy of that parameter.

While useful, conversion constructors have several major idiosyncrasies, especially when C++ interprets normal constructors as conversion constructors. This section explores implicit type conversions, conversion constructors, and how to prevent coding errors stemming from inadvertent conversion constructors.

Implicit Conversions

In C++, an *implicit conversion* is a conversion from one type to another that doesn't require an explicit typecast. Perhaps the simplest example is the following conversion from an `int` to a `double`:

```
double myDouble = 137 + 2.71828;
```

Here, even though 137 is an `int` while 2.71828 is a `double`, C++ will implicitly convert it to a `double` so the operation can proceed smoothly.

When C++ performs implicit conversions, it does not “magically” figure out how to transform one data type into another. Rather, it creates a temporary object of the correct type that's initialized to the value of the implicitly converted object. Thus the above code is equivalent to

```
double temp = (double)myInt;
double myDouble = temp + 2.71828;
```

It's important to remember that when using implicit conversions you are creating temporary objects. With primitive types this is hardly noticeable, but makes a difference when working with classes. For example, consider the following code:

```
string myString = "This ";
string myOtherString = myString + "is a string";
```

Note that in the second line, we're adding a C++ `string` to a C `char *` string. Thus C++ will implicitly convert “is a string” into a C++ `string` by storing it in a temporary object. The above code, therefore, is equivalent to

```
string myString = "This ";
string tempStr = "is a string";
string myOtherString = myString + tempStr;
```

Notice that in both of the above examples, at some point C++ needed a way to initialize a temporary object to be equal to an existing object of a different type. In the first example, we made a temporary `double` that was equal to an `int`, and in the second, a temporary `string` equal to a `char *`.* When C++ performs these conversions, it uses a special function called a *conversion constructor* to initialize the new object. Conversion constructors are simply class constructors that accept a single parameter and initialize the new object to a copy of the parameter. In the `double` example, the newly-created `double` had the same value as the `int` parameter. With the C++ `string`, the temporary `string` was equivalent to the C string.

C++ will invoke conversion constructors whenever an object of one type is used in an expression where an object of a different type is expected. Thus, if you pass a `char *` to a function accepting a C++ `string`, the `string` will be initialized to the `char *` in its conversion constructor. Similarly, if you have a function like this one:

```
string MyFunction()
{
    return "This is a string!";
}
```

The temporary object created for the return value will be initialized to the C string "This is a string!" using the conversion constructor.

Writing Conversion Constructors

To see how to write conversion constructors, we'll use the example of a `CString` class that's essentially our own version of the C++ `string` class. Internally, `CString` stores the string as a C string called `theString`. Since we'd like to define an implicit conversion from `char *` to `CString`, we'll declare a conversion constructor, as shown below:

```
class CString
{
public:
    CString(const char* other);
    /* Other member functions. */

private:
    char* theString;
};
```

Then we'd implement the conversion constructor as

```
CString::CString(const char* other)
{
    /* Allocate space and copy over the string. */
    theString = new char[strlen(other) + 1];
    strcpy(theString, other);
}
```

Now, whenever we have a `char *` C string, we can implicitly convert it to a `CString`.

* Technically speaking, this isn't quite what happens, since there's a special form of the + operator that works on a mix of C strings and C++ strings. However, for this purposes of this discussion, we can safely ignore this.

In the above case, we defined an implicit conversion from `char *` C strings to our special class `CString`. However, it's possible to define a second conversion from a C++ `string` to our new `CString` class. In fact, C++ allows you to provide conversion constructors for any number of different types that may or may not be primitive types.

Here's a modified `CString` interface that provides two conversion constructors from `string` and `char *`:

```
class CString
{
public:
    CString(const string& other);
    CString(const char*   other);
    /* ... other member functions... */

private:
    char *theString;
};
```

A Word on Readability

When designing classes with conversion constructors, it's easy to get carried away by adding too many implicit conversions. For example, suppose that for the `CString` class we want to define a conversion constructor that converts `int`s to their string representations. This is completely legal, but can result in confusing or unreadable code. For example, if there's an implicit conversion from `int`s to `CString`s, then we can write code like this:

```
CString myStr = myInt + 137;
```

The resulting `CString` would then hold a string version of the value of `myInt + 137`, not the string composed of the concatenation of the value of `myInt` and the string "137." This can be a bit confusing and can lead to counterintuitive code. Worse, since C++ does not normally define implicit conversions between numeric and string types, people unfamiliar with the `CString` implementation might get confused by lines assigning `int`s to `CString`s.

In general, when working with conversion constructors, make sure that the conversion is intuitive and consistent with major C++ conventions. If not, consider using non-constructor member functions. For example, if we would like to be able to convert `int` values into their string representations, we might want to make a global function `intToString` that performs the conversion. This way, someone reading the code could explicitly see that we're converting an `int` to a `CString`.

Problems with Conversion Constructors

While conversion constructors are quite useful in a wide number of circumstances, the fact that C++ automatically treats all single-parameter constructors as conversion constructors can lead to convoluted or nonsensical code.

One of my favorite examples of "conversion-constructors-gone-wrong" comes from an older version of the CS106B/X ADT class libraries. Originally, the CS106B/X `Vector` was defined as

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 10); // Hint about the size of the Vector

    /* ... */
};
```

Nothing seems all that out-of-the-ordinary here – we have a `Vector` template class that lets you give the class a hint about the number of elements you will be storing in it. However, because the constructor accepts a single parameter, C++ will interpret it as a conversion constructor and thus will let us implicitly convert from `int`s to `Vector`s. This can lead to some very strange behavior. For example, given the above class definition, consider the following code:

```
Vector<int> myVector = 137;
```

This code, while nonsensical, is legal and equivalent to `Vector<int> myVector(137)`. Fortunately, this probably won't cause any problems at runtime – it just doesn't make sense in code.

However, suppose we have the following code:

```
void DoSomething(Vector<int>& myVector)
{
    myVector = NULL;
}
```

This code is totally legal even though it makes no logical sense. Since `NULL` is `#define`d to be 0, The above code will create a new `Vector<int>` initialized with the parameter 0 and then assign it to `myVector`. In other words, the above code is equivalent to

```
void DoSomething(Vector<int>& myVector)
{
    Vector<int> tempVector(0);
    myVector = tempVector;
}
```

`tempVector` is empty when it's created, so when we assign `tempVector` to `myVector`, we'll set `myVector` to the empty vector. Thus the nonsensical line `myVector = 0` is effectively an obfuscated call to `myVector.clear()`.

This is a quintessential example of why conversion constructors can be dangerous. When writing single-argument constructors, you run the risk of letting C++ interpret your constructor as a conversion constructor.

explicit

To prevent problems like the one described above, C++ provides the `explicit` keyword to indicate that a constructor must not be interpreted as a conversion constructor. If a constructor is marked `explicit`, it indicates that the constructor should not be considered for the purposes of implicit conversions. For example, let's look at the current version of the CS106B/X `Vector`, which has its constructor marked `explicit`:

```
template <typename ElemType> class Vector
{
public:
    explicit Vector(int sizeHint = 10); // Hint the size of the Vector

    /* ... */
};
```

Now, if we write code like

```
Vector<int> myVector = 10;
```

We'll get a compile-time error since there's no implicit conversion from `int` to `Vector<int>`. However, we can still write

```
Vector<int> myVector(10);
```

Which is what we were trying to accomplish in the first place. Similarly, we eliminate the `myVector = 0` error, and a whole host of other nasty problems.

When designing classes, if you have a single-argument constructor that is not intended as a conversion function, you *must* mark it `explicit` to avoid running into the "implicit conversion" trap. While indeed this is more work for you as an implementer, it will make your code safer and more stable.

Chapter Summary

- Templates can be used to define a family of abstractions that depend on an arbitrary type.
- The `typename` keyword is used to declare parameters to a template class.
- A template class's interface and implementation should be put into the .h file and no .cpp file should be created for the class.
- The `typename` keyword is also used in front of types nested inside of dependent types.
- Marking a variable `const` prevents its value from being changed after the variable is initialized.
- A `const` member function cannot modify any of the class's data members.
- `const` member functions clarify interfaces by indicating which member functions read values and which member functions write values.
- `const` can have different meanings when applied to pointers based on where the `const` occurs.
- C++ enforces bitwise `const`ness; it is up to you to ensure that your classes are semantically `const`.
- The `mutable` keyword allows you to write semantically `const` functions which are not bitwise `const`.
- Member initializer lists initialize data members to particular values before the constructor begins running.

- The `static` keyword allows you to indicate that certain data is shared across all instances of a class.
- `static` data members must be declared in the .h file and defined in the .cpp file.
- `static` member functions are functions associated with a class as a whole, rather than a particular instance of a class.
- `static` member functions are invoked by writing `ClassName::functionName()`.
- Integral class constants can be initialized in the body of the class and do not need to be separately defined.
- Conversion constructors allow classes to be initialized to values of a different type.
- The `explicit` keyword prevents accidental implicit conversions from occurring.

Practice Problems

1. How do you declare a class template?
2. How do you implement member functions for a class template?
3. Is there a difference between the `typename` and `class` keywords when declaring template arguments?
4. When is it necessary to preface a type with the `typename` keyword in a class template?
5. The following line of code declares a member function inside a class:

   ```
   const char * const MyFunction(const string& input) const;
   ```

   Explain what each `const` in this statement means.
6. What is `const`-overloading?
7. What is the difference between semantic `const`ness and bitwise `const`ness?
8. What is the difference between a `const` pointer and a pointer-to-`const`?
9. How are `const` references different from regular references?
10. What does the `mutable` keyword do?
11. What are the steps involved in class construction? In what order do they execute?
12. How do you declare an initializer list?
13. What is `static` data and how does it differ from regular member data?
14. What are the two steps required to add `static` data to a class?

15. What is a static member function? How do you call a static member function?

16. What is a conversion constructor?

17. Explain what the `explicit` keyword does.

18. The STL `map`'s bracket operator accepts a key and returns a reference to the value associated with that key. If the key is not found, the `map` will insert a new key/value pair so that the returned reference is valid. Is this function bitwise `const`? Semantically `const`?

19. When working with pointers to pointers, `const` can become considerably trickier to read. For example, a `const int * const * const` is a `const` pointer to a `const` pointer to a `const int`, so neither the pointer, its pointee, or its pointee's pointee can be changed. What is an `int * const *`? How about an `int ** const **`?

20. The CS106B/X `Vector` has the following interface:

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size();
    bool isEmpty();

    ElemType getAt(int index);
    void setAt(int index, ElemType value);

    ElemType& operator[](int index);

    void add(ElemType elem);
    void insertAt(int index, ElemType elem);
    void removeAt(int index);

    void clear();

    void mapAll(void fn(ElemType elem));
    template <typename ClientDataType>
        void mapAll(void fn(ElemType elem, ClientDataType & data),
                    ClientDataType & data);

    Iterator iterator();
};
```

Modify this interface so that it is `const`-correct. *(Hint: You may need to `const`-overload some of these functions)*

21. Modify the Snake simulation code from the earlier extended example so that it is `const`-correct.

22. Explain each of the steps involved in object construction. Why do they occur in the order they do? Why are each of them necessary?

23. Why must a function with a single parameter with default value must have default values specified for each parameter afterwards?

24. NASA is currently working on Project Constellation, which aims to resume the lunar landings and ultimately to land astronauts on Mars. The spacecraft under development consists of two parts – an orbital module called Orion and a landing vehicle called Altair. During a lunar mission, the Orion vehicle will orbit the Moon while the Altair vehicle descends to the surface. The Orion vehicle is designed such that it does not necessarily have to have an Altair landing module and consequently can be used for low Earth orbit missions in addition to lunar journeys. You have been hired to develop the systems software for the spacecraft. Because software correctness and safety are critically important, you want to design the system such that the compiler will alert you to as many potential software problems as possible.

    Suppose that we have two classes, one called `OrionModule` and one called `AltairModule`. Since every Altair landing vehicle is associated with a single `OrionModule`, you want to define the `AltairModule` class such that it stores a pointer to its `OrionModule`. The `AltairModule` class should be allowed to modify the `OrionModule` it points to (since it needs to be able to dock/undock and possibly to borrow CPU power for critical landing maneuvers), but it should under no circumstance be allowed to change which `OrionModule` it's associated with. Here is a skeleton implementation of the `AltairModule` class:

```
class AltairModule
{
public:
    /* Constructor accepts an OrionModule representing the Orion
     * spacecraft this Altair is associated with, then sets up
     * parentModule to point to that OrionModule.
     */
    AltairModule(OrionModule* owner);

          /* ... */
      private:
          OrionModule* parentModule;
      };
```

    Given the above description about what the `AltairModule` should be able to do with its owner `OrionModule`, appropriately insert `const` into the definition of the `parentModule` member variable. Then, implement the constructor `AltairModule` such that the `parentModule` variable is initialized to point to the `owner` parameter.

25. Explain why `static` member functions cannot be marked `const`.

26. Write a class `UniquelyIdentified` such that each instance of the class has a unique ID number determined by taking the ID number of the previous instance and adding one. The first instance should have ID number 1. Thus the third instance of the class will have ID 3, the ninety-sixth instance 96, etc. Also write a `const`-correct member function `getUniqueID` that returns the class's unique ID. Don't worry about reusing older IDs if their objects go out of scope.

27. The C header file `<cstdlib>` exports two functions for random number generation – `srand`, which seeds the randomizer, and `rand`, which generates a pseudorandom `int` between 0 and the constant `RAND_MAX`. To make the pseudorandom values of `rand` appear truly random, you can seed the randomizer using the value returned by the `time` function exported from `<ctime>`. The syntax is `srand((unsigned int)time(NULL))`. Write a class `RandomGenerator` that exports a function `next` that returns a random `double` in the range [0, 1). When created, the `RandomGenerator` class should seed the randomizer with `srand` only if a previous instance of `RandomGenerator` hasn't already seeded it.

28. Does it make sense to initialize static data members in a member initializer list? Explain why or why not.

29. Should you ever mark static data members `mutable`? Why or why not?

These practice problems concern a `RationalNumber` class that encapsulates a rational number (that is, a number expressible as the quotient of two integers). `RationalNumber` is declared as follows:

```
class RationalNumber
{
public:
    RationalNumber(int num = 0, int denom = 1) :
        numerator(num), denominator(denom) {}

    double getValue() const
    {
        return static_cast<double>(numerator) / denominator;
    }

    void setNumerator(int value)
    {
        numerator = value;
    }
    void setDenominator(int value)
    {
        denominator = value;
    }

private:
    int numerator, denominator;
};
```

The constructor to `RationalNumber` accepts two parameters that have default values. This means that if you omit one or more of the parameters to `RationalNumber`, they'll be filled in using the defaults. Thus all three of the following lines of code are legal:

```
RationalNumber zero; // Value is 0 / 1 = 0
RationalNumber five(5); // Value is 5 / 1 = 5
RationalNumber piApprox(355, 113); // Value is 355/113 = 3.1415929203...
```

30. Explain why the `RationalNumber` constructor is a conversion constructor.

31. Write a `RealNumber` class that encapsulates a real number (any number on the number line). It should have a conversion constructor that accepts a `double` and a default constructor that sets the value to zero. *(Note: You only need to write one constructor. Use `RationalNumber` as an example)*

32. Write a conversion constructor that converts `RationalNumber`s into `RealNumber`s.

33. If a constructor has two or more arguments and no default values, can it be a conversion constructor?

34. C++ will apply at most one implicit type conversion at a time. That is, if you define three types `A`, `B`, and `C` such that `A` is implicitly convertible to `B` and `B` is implicitly convertible to `C`, C++ will not automatically convert objects of type `A` to objects of type `C`. Give an reason for why this might be. *(Hint: Add another implicit conversion between these types)*