const

Introduction

The `const` keyword is one of C++'s most ubiquitous and unusual features. Normally, C++ compilers will let you get away with almost anything, but when it comes to `const` even lax compilers will complain over seemingly tiny errors. Also, unlike most C++ keywords, `const` is a semantic distinction that exists only at compile-time and does not change how your program executes. Furthermore, programs tend to either completely ignore the `const` keyword or to use it in practically every other line. In this handout, we'll explore `const` in a variety of contexts and will cover some of the subtler points of `const`.

A Word on Error Messages

The error messages you'll get if you accidentally break `const`ness can be intimidating and confusing. Commonly you'll get unusual errors about conversions losing qualifiers or l-values specifying `const` objects. These are the telltale signs that you've accidentally tried to modify a `const` variable. Also, the errors for `const` can be completely incomprehensible when working with template classes and the STL. As with all STL errors, you'll need to practice a good deal before you'll be able to understand these error messages.

`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`, you'd get a compile-time error since you would be modifying a `const` variable.

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:

```
int length = myVector.size();
for(int i = 0; i < length; i++)
    /* ...Do something that doesn't modify the length of the vector... */
```

Here, we compute `length` only once since we know at compile-time that our operation isn't going to modify the contents of `myVector`. Since we don't have the overhead of a call to `myVector.size()` every iteration, on some compilers this loop can be about ten percent faster than if we had simply written the conventional for loop.

Because the length of the `vector` is a constant, we know at compile-time that the variable `length` should never change. We can therefore mark it `const` to have the compiler enforce that it *must* never change, as shown here:

```
const int length = myVector.size();
for(int i = 0; i < length; i++)
    /* ...Do something that doesn't modify the length of the vector... */
```

This code works identically to the original, except that we've explicitly announced that we will not change `length`. If we accidentally try to modify `length`, we'll get a compile-time error directing us to the offending line, rather than a runtime bug. In essence, the compiler will help us debug our code by converting a potential runtime bug into an easily fixable compile-time error. For example, suppose we want to perform some special handling if the `vector` is exactly ten elements long. If we write `if(length = 10)` instead of `if(length >= 10)`, if `length` isn't declared `const`, our code will always execute exactly ten times and we'll get strange behavior where an if statement executes more frequently than expected. Since assignments-in-if-statements are perfectly legal C++ code, the C++ compiler won't warn us of this fact and we will have to debug the program by hand. However, if we mark `length` `const`, the compiler will generate an error when compiling the statement `if(length = 10)` because it contains an assignment to a `const` variable. No longer will we have to track down errant runtime behavior – for once it appears that the C++ compiler is helping us write good code!

`const` and Pointers

The `const` keyword is useful in many circumstances, but, like all other features of C++, has its share of quirks and oddities. Perhaps the most persistent source of confusion when working with `const` arises in the context of `const` 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 GlobalCppString = "This is a string!";
```

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

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

This syntax is partially correct. If you were ever to write `GlobalString[0] = 'X'`, rather than getting segmentation faults at runtime (see the C strings handout for more info), you'd get a compiler error that would direct you to the line where you tried to modify the global constant. But unfortunately this variable declaration isn't quite right. Suppose, for example, that you write the following code:

```
GlobalString = "Reassigned!";
```

Here, you're reassigning `GlobalString` to point to the string literal "Reassigned!" Note that you aren't modifying the contents of the character sequence `GlobalString` is pointing to – instead you're changing *what character sequence `GlobalString` points to*. In other words, you're modifying the *pointer*, not the *pointee*, so the above line will compile correctly and other code that references `GlobalString` will suddenly begin using the string "Reassigned!" instead of "This is a string!" as you would hope.

When working with `const`, C++ distinguishes between two similar-sounding entities: a *pointer-to-`const`* and a *`const` pointer*. A pointer-to-`const` is a pointer like `GlobalString` 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, `myConstPointer` can't be reassigned, but you are free to modify the value it points to.

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 `GlobalString` behave as a true C string constant, we'd need to make the pointer both a `const` pointer and a pointer-to-`const`. This is totally legal in C++, and the result is a `const` pointer-to-`const`. The syntax looks like this:

```
const char * const GlobalString = "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 "`GlobalString` 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**

`const` Objects

So far, all of the `const` cases we've dealt with have concerned primitive types. What happens when we mix `const` with objects?

Let us first 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(int i = 0; i < myString.length(); i++)
    cout << myString[i] << endl;
```

To us as humans, the above code seems completely fine and indeed it is legal C++ code. However, this code is surprisingly subtle for one key reason – how does the compiler know that the `length` function doesn't modify the contents of the `string`? This question generalizes to a larger question: given an arbitrary class, how can the compiler tell which member functions modify the class and which ones don't?

To answer this question, let's look at the prototype for the `string` member function `length`:

```
size_type length() const;
```

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 will not change.*

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.

For example, let's consider a `Point` class that simply stores a point in two-dimensional space:

```
class Point
{
    public:
        double getX();
        double getY();

        void setX(double newX);
        void setY(double newY);
    private:
        double x, y;
};
```

Consider the following implementation of `getX`:

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

Since this function doesn't modify the `Point` object in any way, we should change the prototype in the class definition to read `double getX() const` so we can call this function on `const Point` objects. Similarly, we need to add a `const` marker to the function definition, as shown here:

---

* In some cases it is possible for a `const` member function to modify its receiving object, as you'll see later in this handout. However, it's perfectly reasonable to view `const` member functions this way.

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

Forgetting to add this `const` can be a source of much frustration because the C++ considers `getX()` and `getX() const` two different functions.

In a `const` member function, all the class's instance variables are treated as `const`. Thus you can read their values but not modify them. Additionally, inside a `const` member function, you cannot call other non-`const` member functions, since they might modify the contents of the current object. Beyond these restrictions, `const` member functions can do anything that regular member functions can do. Consider, for example, the following implementation of a `distanceToOrigin` function for the `Point` class:

```
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!
     return sqrt(dx + dy); // Legal!  sqrt is a free function that can't
                           //         modify the current object.
}
```

`const` References

Throughout this course we've used pass-by-reference in order to avoid copying objects between function calls. However, pass-by-reference can lead to some ambiguity. 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`?

To remove this ambiguity, we can use `const` references. A `const` reference is like a normal reference except that the original object is treated as though it were `const`. For example, consider this rewritten function prototype:

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

Because the parameter is a `const` reference, the `DoSomething` function cannot modify the `vector`.

You are allowed to pass both `const` and non-`const` variables to functions accepting `const` references. Whether or not the original variable is `const`, inside the function call it is treated as though it were. Thus it's legal (and encouraged) to write code like this:

```
/* Since we're not changing vec, we marked it const in this function. */
void PrintVector(const vector<int> &vec)
{
    copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}

int main()
{
    vector<int> myVector(NUM_INTS);
    PrintVector(myVector); // Becomes const once inside PrintVector
    myVector.push_back(137); // Legal here, myVector isn't const.
}
```

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. 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.

`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();  // ERROR!
         itr != myVector.end(); ++itr)
         cout << *itr << endl;
}
```

Unfortunately, this code will generate a compile-time error. The problem is in the first part of the for loop where 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:

```
vector<string>::iterator itr = myVector.begin(); // Assume myVector is const
*itr = 42; // Just modified a const object!
```

Your initial thought might be to declare the iterator `const` to indicate that you won't modify what the iterator's pointing to. This won't work either, though, since a `const` iterator is like a `const` pointer – while you can't modify which element the iterator is iterating over, you can change the value of the element referenced by the iterator.

To fix this problem, all STL containers define a special iterator called a `const_iterator` that is capable of reading the values of a container but not modifying them. Thus the proper version of the above code is:

```
void PrintVector(const vector<string> &mySet)
{
     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.

One interesting point about the difference between the `iterator` and `const_iterator` is that all STL containers define two different `begin` and `end` functions – non-`const` versions that return `iterator`s and `const` versions that return `const_iterator`s. When two member functions have the same name but differ in their `const`ness, C++ will call the version of the function that has the same `const`ness as the receiver object. That is, a non-`const` `vector` will always call the non-`const` version of `begin`, while a `const` `vector` will always call the `const` version of `begin`. This is sometimes known as "`const` overloading."

Limitations of `const`

Although `const` is a useful programming construct, it is imperfect. 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 totally legal since we didn't modify the value of `theString` – instead, we changed the value of the elements it pointed at. In effect, because the member function is declared `const`, `theString` acts as a `const` pointer instead of a pointer-to-`const`.

This raises the issue of the distinction between "bitwise `const`ness" versus "semantic `const`ness." *Bitwise `const`ness*, which is the type enforced by C++, means that `const` classes are prohibited from making any bitwise changes to themselves. In the above example, since the value of `theString` didn't change (because we didn't reassign it), C++ considers it `const`-correct. However, from the viewpoint of *semantic `const`ness*, `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 `const`ness, you must take care to ensure that your program is semantically `const`. From your perspective as a programmer, if you call a function that's marked `const`, you would expect that it cannot modify whatever class it was working on. If the function isn't semantically `const`, however, you'll run into problems where code that shouldn't be modifying an object somehow leaves the object in a different 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 the string doesn't modify the string's contents, we've marked the function `const`, and, indeed, the function is bitwise `const`. However, our code has a major flaw. Consider the following code:

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

Here, we're using the pointer get obtained from `getString` as a parameter to `strcpy`. After the `strcpy` completes, `myStr`'s internal string will contain "Oh no!" instead of "This is a C string!" We've modified a `const` object using only `const` member functions, something that defeats the purpose of `const`.

Somehow we need to change the code to prevent this from happening. The problem is that the pointer returned by `getString` is not itself `const`, so it's legal to use it with functions like `strcpy`. To resolve this problem, we can simply change the return value of `getString` from `char *` to `const char *`. This approach solves the problem because it's now illegal to modify the string pointed at by the return value. In general, when returning pointers or references to an object's internal data, you should make sure to mark them `const` when appropriate.

The above example illustrates that making semantically-`const` code can be difficult. However, the benefits of semantically-`const` code are noticeable – your code will be more readable and less error-prone.

`mutable`

Because C++ enforces bitwise `const`ness rather than semantic `const`ness, you might find yourself in a situation where a member function changes an object's bitwise representation without modifying its semantic value. 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>()); // See note*
}

/* 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:

---

* If you're still a bit shaky on iterator adapters, the final two arguments to the `insert` function define a range spanning the entire source file `input`. See the handout on iterator adapters for more information.

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

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

    string getItemAt(int index); // NOTE: not 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);
        /* ... some sort of error-checking ... */
        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. 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 those data members can be modified inside member functions that are marked `const`. We can thus rewrite the `GroceryList` class definition to look like this:

---

* The general technique of deferring computations until they are absolutely required is called *lazy evaluation* and is an excellent way to improve program efficiency. Consider taking CS242 or CS258 if you're interested in learning more about lazy evaluation.

```
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 functions 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.

A Word on Pervasiveness

In general, code you'll write will either be completely `const`-correct (that is, member functions will be marked `const` when appropriate, parameters will be passed as reference-to-`const` for efficiency reasons, etc.) or it will be completely non-`const`-correct. Unlike other C++ language features, if you use `const` even once in your code, you implicitly require code in other modules to be `const`-correct. For example, suppose you try to pass a CS106 `Vector` to a function as a 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` CS106 `Vector` is effectively a digital brick.

You can think of all C++ code as belonging to either the "`const`-correct camp" or the "non-`const`-correct camp." Since almost all professional C++ libraries are `const`-correct, you should expect to write `const`-correct code. For the purposes of this class, we will always work with `const`-correct code because `const`-correctness is an important part of C++. However, in CS106B/X, you should almost certainly *not* use `const`. The CS106 libraries aren't `const`-correct, and odds are that even a single `const` will cause a cascade of carnage that will completely cripple your compiler.

As a general rule, you should always write `const`-correct code unless you know in advance that none of the libraries you're using will be `const`-correct.

Practice Problems

Here are some good practice problems to get you thinking about `const`. I recommend you play around with them a bit to get a feel for how `const` works.

1. Modify the class interface of the CS106 `Vector` so that it's `const`-correct. *(Hint: You'll need to define two versions of `operator []`)*
2. What does the following line of code mean?
   ```
   const char * const myFunction(const string &input) const;
   ```
3. Write a class with a member function `isConst` that returns whether the receiving object is `const`. *(Hint: Write two different functions named `isConst`)*
4. 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`?
5. 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 **`?
6. In C++, it is legal to convert a pointer of type `T *` to a pointer of type `const T *`, since the `const T *`'s restrictions on what can be done with the pointee are a subset of the behaviors allowed by a variable of type `T *`. It is not legal, however, to convert a pointer of type `T **` to a pointer of type `const T **`, because doing so would open a hole in the type system that would allow you to modify a variable of type `const T`. Show why this is the case.