

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. 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. Worse, the errors for `const` can be completely incomprehensible when working with template classes and the STL. As with 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're computing `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, this loop is about ten percent faster than if we had simply written the conventional for loop.

Since 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 here...
```

This code works identically to the original, except that we've explicitly announced that we plan not to change `length`. This means that if we accidentally try to modify `length`, we'll get a compile-time error rather than a runtime bug. 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`, we'll get unusual runtime behavior. However, if `length` is `const`, the compiler will complain about assignment to a `const` variable. For once it appears that the C++ compiler is helping us write good code!

const and Pointers

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`. Alternatively, you can declare *pointers-to-const* by writing `Type const *myPointer`.

On the other hand, a *const pointer* 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`. 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? |
|---------------------------------------|------------------------|---------------|---------------------|
| <code>const Type *myPtr</code> | Pointer-to-const | Yes | No |
| <code>Type const *myPtr</code> | Pointer-to-const | Yes | No |
| <code>Type * const myPtr</code> | const pointer | No | Yes |
| <code>const Type * const myPtr</code> | const pointer-to-const | No | No |
| <code>Type const * const myPtr</code> | 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;
```

There is one big question here – how does the compiler know that the `length` function doesn't modify the contents of the `string`? This question is actually quite important and 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:

```
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 class. 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 accepted 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 you 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, you 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 constness, you cannot use const_iterators in functions like insert or erase that modify containers. You can, however, define iterator ranges using const_iterators 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 iterators and const versions that return const_iterators. When two member functions have the same name but differ in their constness, C++ will call the version of the function that has the same constness 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 that C++ expert Scott Meyers calls the distinction between “bitwise `constness`” versus “semantic `constness`.” Bitwise `constness`, 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 `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 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.

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 that code in other modules 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`.

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 calling object is `const`. (*Hint: Write two different functions named `isConst`*)
4. Why is the STL `map`'s bracket operator not marked `const`?
5. What is the difference between a `const vector<int>` and a `vector<const int>`?
6. In this handout we talked about functions that are bitwise `const` but not semantically `const`. Give an example of a function that's semantically `const` but not bitwise `const`.