Const Correctness

Const correctness is the proper application of the keyword `const` to variables, objects and class definitions. It specifies that something will not change with a usage (we typically refer to this something as "immutable").

All data types (including user-defined) can be declared using `const` to ensure that some programming objects will not change because of a function call

The usual question from those learning the language: 'Why'? There are at least the following good reasons to use const correctness:

1. By ensuring that objects do not change when applying them to functions or by calling certain functions, code can be demonstrably correct. This can save a great deal of time that might be wasted looking for non-existent changes in an object due to being passed to a function.

2. Const correctness more accurately documents the proper use of functions.

3. In certain situations, some compilers can optimize code that uses the `const` qualification.

4. Using const correctness in your code indicates to other, more experienced programmers that you are a cut above your colleagues who do not feel it is necessary. Const correctness is one criterion in deciding how skilled a programmer is. Intermediate and senior programmers use it as much as possible and doing so puts the junior programmer a step ahead.

Defining Constants

The simplest application of const is when it is used to define constants.

```
int const MAX_SIZE = 5;
```

This establishes a constant within the scope. `MAX_SIZE` can be referenced, but it cannot be changed. Put in terms used by compiler designer and writers, `MAX_SIZE` can be used as an **rvalue** (it can appear on the right side of an assignment operator) but not as an **lvalue** (it cannot appear on the left side of an assignment operator). Note that the declaration can be written as:

```
const int MAX_SIZE = 5;
```

Either is acceptable.

In C++, the use of `const` is preferred to the C practice of using the `#define` pre-processor directive:

```
//this is the ‘C’ way
#define MAX_SIZE 5

//In C++ we do it this way:
const int MAX_SIZE = 5;
```

Using the pre-processor, the code is searched for the token that is defined and then performs a text substitution. All instances of `MAX_SIZE` are replaced with 5. By defining the constant, the value is stored in memory much as a variable is. The main reason that C++ approaches it this way is because `#define` offers no type safety – something for which C++ strives. The pre-processor directive is useful in C++ but a more common usage is for things like conditional compilation or compilation guards.

Function Parameters

One of the most common applications of const correctness is when applied to function parameters. Suppose we have a function declared as:

```
void someFunction(MyClass obj);
```

Recall from the discussion of C++ functions that the default semantics for the language include arguments passed by value. This means that, unless declared otherwise, the function will make a copy of the object passed to it.

```
    MyClass o;
    someFunction(o);
```

In this declaration and usage, there is no danger that the object `o` will be changed by the function because the function will make a copy of `o` (using the copy constructor). The down-side of this is that invoking a copy may eat up large amounts of processor time and memory. If the parameter is one of C++’s data types, then this overhead is acceptable, but if an object is passed, the overhead is not acceptable.

As a result, the practice in C++ is to pass pointers or references to objects to functions, so when a programmer encounters code like this:

```
void someFunction(MyClass* obj);
```

…and usage like this:

```
    MyClass o;
    someFunction(&o);
```

…the careful programmer will wonder if the object o will be modified by the call to the function. This is the kind of investigation that can waste a programmer’s valuable time.

Const Qualification of Pointers

Pointers are const qualified in this way:

```
const char* p = “Some text”;
```

This declares a pointer to const character data. The data itself cannot be changed, but the pointer can be reassigned. This is legal:

```
p = “Other words”;
```

…but this is not:

```
p* = ‘Q’;  //compiler error
```

This form of pointer const qualification is most common.

Note that, according to the latest draft ISO specification, this can be used as an equivalent to the above declaration:

```
char const * p = “Some text”;
```

Whether to use one or the other depends on the circumstance. Older compliers (like Visual Studio 2005) may only support the first declaration, while the second may be part of the coding standards established at a particular game studio.

This declaration:

```
char* const q = “Some text”;
```

…defines a const pointer to character data. The pointer cannot be reassigned, but the data can be changed. This is legal:

```
q* = ‘Q’;
```

…but this is not:

```
q = “Other words”; // compiler error
```

This form of qualification is not common and can lead to some very strange results. As a result, you should generally avoid this.

For maximum const qualification, you can use:

```
const char* const r = “Some text”;
```

…or (according to the latest C++ draft specification):

```
char const * const r = “Some text”;
```

This defines const pointer to const character data or a pointer that cannot be changed to point to some new text nor can the data be changed.

```
//both of these will generate compiler errors
r* = ‘Q’;
r = “Other words”;
```

This qualification is rarely used.

The most common use of const qualification in pointers is the application of constness to the object that is being pointed to:

```
void someFunction(const MyClass* obj);
```

…and usage like this:

```
    MyClass o;
    someFunction(&o);
```

Using this code, the careful programmer knows that the object `o` will not change during the execution of the function.

Const Qualification of References

The qualification of references is similar to those of pointers but varies in one very important respect: references must be bound to an existing object when declared and they cannot be reassigned.

```
MyClass a;
MyClass b;
MyClass& r = a;
```

If the reference is now assigned to b:

```
r = b;
```

…the code assigns the state of object `b` to `a`. The reference still refers to object `a`.

To declare a reference to a const object, the code looks like this:

```
const MyClass& r = a;
```

…or (according to the latest C++ draft specification):

```
MyClass const & r = a;
```

If we draw a parallel to the const qualification of pointers, we could declare this:

```
MyClass & const r = a; //legal but meaningless code
```

This defines a const reference to a `MyClass` object, which means the reference cannot be reassigned. The problem is that references by their very nature cannot be reassigned so this declaration, while syntactically legal, is semantically nonsensical. It makes no sense to declare const references because they are intrinsically const.

We see the use of references to const objects in arguments to functions:

```
void someFunction(const MyClass& obj);
```

…and usage like this:

```
    MyClass o;
    someFunction(o);
```

Again, the careful programmer can use the function, confident that the object will not change. If const correctness is part of the common practice of a game studio and a programmer sees this:

```
void someFunction(MyClass& obj);
```

Usage like this:

```
    MyClass o;
    someFunction(o);
```

…will give a firm indication to the programmer that the function could modify the object passed into it. Sometimes that is what we want.

The question of whether to use pointers or references is a matter for another article and is not discussed here.

Const Member Functions

When we build a class we can determine if various functions will actually modify the state of the object. If a function does not, it is a good candidate to be declared as a const function. This is a member function that is guaranteed not to change any of the data members of the object (there are a few exceptions to this, but that is an advanced topic). Consider this class declaration:

```
class MyClass
{
public:
    int getIndex() const; //defined as a const function

private:
    int _index;

};
```

We have defined a function that will not change the values of an object of this class. The implementation of this function would look like this:

```
int MyClass::getIndex() const
{
    return _index;
}
//Note that the implementation has to be
//qualified as well as the declaration
```

This function is a simple getter, so it will not change the value of the object. These are excellent candidates for const qualification.

The same does not hold for setter functions:

```
class MyClass
{
public:
    int getIndex() const; //defined as a const function
    void setIndex(int i); //not defined as const
private:
    int _index;

};
```

```
void MyClass::setIndex(int i)
{
    _index = i; //This line would generate a compiler error
}               //if the function was qualified as const.
```

The difference between the two kinds of functions – const and non-const -- comes into play when holding a reference or a pointer to the object. Let us look at might happen in a function that takes a reference to a const object:

```
void someFunction(const MyClass& obj)
{
    int index = obj.getIndex();
    //other code
}
```

This is allowed because the function `getIndex()` is a const function and will not change the value of `obj`. If we try this:

```
void someFunction(const MyClass& obj)
{
    int index = obj.getIndex();
    //other code
    int newIndex = 9;
    obj.setIndex(newIndex); //Compile Error!!!
}
```

`setIndex(int)` is not const and cannot be declared as such because it modifies `obj` and `obj` is declared as const. This guarantees that using const qualification in the signature of the function is honoured.

The const qualification of a member function is part of the signature. These are two different functions:

```
class MyClass
{
public:
    double doStuff() const;
    double doStuff();
};
```

The question arises: How can we specify which one to call? This depends on how we refer to the object on which we are calling the function:

```
    MyClass obj;
    const MyClass* p = &obj;

    obj.doStuff(); //non-const version is called
    p -> doStuff(); //const version is called because of
                    //the qualification on the pointer
```

Const Returns from Member Functions

Just as the default semantics for function arguments is pass-by-value, a function return defaults to pass-by-value; the value is copied to the calling function. We can over-ride this by declaring a return of a reference (with the return of a pointer we can mimic return by reference, but it still is a return by value: the address of a variable or object).

If our class has a string data member that holds its name, the code might look like this:

```
//declaration
class MyClass
{
private:
    string _name;
public:
    string getName() const;
};

//definition
string MyClass::getName() const
{
    return _name;
}
```

This has the same kind of inefficiency when we passed objects by value: a copy of the string is made and returned to the calling code. This is made more efficient by returning a reference to the string:

```
//declaration
class MyClass
{
private:
    string _name;
public:
    string& getName() const; //this is a reference now
};

//definition
string& MyClass::getName() const
{
    return _name;
}
```

Unfortunately, this breaks encapsulation because we can modify the object by doing this:

```
MyClass obj; //named, say, ‘Fred’
string& str = obj.getName();
str = “Barney”;
//the name of obj is now ‘Barney’
```

We can enforce the constness of the object while maintaining the efficiency of the return by reference by const qualifying the return:

```
 //declaration
class MyClass
{
private:
    string _name;
public:
    const string& getName() const;
    //this is a reference now

};

//definition
const string& MyClass::getName()const
{
    return _name;
}
```

Now this code does not compile, maintaining the integrity of the object:

```
MyClass obj; //named, say, ‘Fred’
string& str = obj.getName(); //compile error! Needs to be a
                             //reference to a const string
```

Designing Classes and Functions for Const Correctness

When should a programmer start thinking about const correctness in class and function design? Right from the beginning of the design process! When designing a class, the programmer should reflect on every function and consider whether making it const makes sense. The same applies to functions that take pointers or references to objects: is the object modified? If not, the pointer or reference should be to a const object. . In general, const correctness dictates that you should declare all objects using `const` unless they need to be modified.

Retrofitting const correctness into pre-existing classes and functions is a tedious process that will often end in failure, since one added qualification can easily lead to more than one qualification required on a member function. Most attempts to do this end in failure and the result is removing const correctness –this leads to code which will raise suspicion in the careful programmer who decides to use it.

By including const correctness in your design, you are doing a more complete and rigorous analysis of your program. This can never be a waste of time or effort.

Summary of Key Points

1. The keyword const indicates that values in certain situations cannot be changed.

2. Const correctness can enforce demonstrable correctness of software that uses it. It can, possibly, let compilers optimize the generated executable.

3. Const correctness is one of the topics that separate the wannabe programmer from the real deal.

4. Constants can be defined, much like variables, by using the keyword `const`. It is preferred to C's `#define` directive.

5. Function parameters of objects are usually passed as references or pointers to improve efficiency. Qualifying a function parameter as a reference or a pointer to a const object ensures that the function will not change the object.

6. A pointer to a const object is declared as:
   ```
   const MyClass* p;
   ```
   or
   ```
   MyClass const* p;
   ```

7. A const pointer to an object is declared as:
   ```
   MyClass* const p;
   ```

8. A const pointer to a const object is declared as:
   ```
   const MyClass* const p;
   ```
   or
   ```
   MyClass const * const p;
   ```

9. A reference to a const object is declared as:
   ```
   const MyClass& r;
   ```
   or
   ```
   MyClass const& r;
   ```

10. A const reference to an object is a nonsensical notion in C++.

11. Passing a pointer or reference of an object to a function that is not qualified with const is a firm indication that the function alters the object.

12. We can declare member functions of classes to be const. The execution of the function promises not to change the state of the object. Getter functions are ideal candidates to be const member functions.

13. If a class has a const member function that returns a reference to an object that is a data member, the return should be a reference or pointer to a const object.

14. Const correctness should be a primary design concern. Retro-fitting const correctness is, at best, tedious and is usually futile.