The C++ 'const' Declaration: Why & How

The 'const' system is one of the really messy features of C++.

It is simple in concept: variables declared with ‘const’ added become constants and cannot be altered by the program. However it is also used to bodge in a substitute for one of the missing features of C++ and there it gets horridly complicated and sometimes frustratingly restrictive. The following attempts to explain how 'const' is used and why it exists.

Simple Use of ‘const’

The simplest use is to declare a named constant. This was available in the ancestor of C++, C.

To do this, one declares a constant as if it was a variable but add ‘const’ before it. One has to initialise it immediately in the constructor because, of course, one cannot set the value later as that would be altering it. For example,

```
const int Constant1=96;
```

will create an integer constant, unimaginatively called ‘`Constant1`’, with the value 96.

Such constants are useful for parameters which are used in the program but do not need to be changed after the program is compiled. It has an advantage for programmers over the C preprocessor ‘`#define`’ command in that it is understood & used by the compiler itself, not just substituted into the program text by the preprocessor before reaching the main compiler, so error messages are much more helpful.

It also works with pointers but one has to be careful where ‘`const`’ is put as that determines whether the pointer or what it points to is constant. For example,

```
const int * Constant2
```

declares that `Constant2` is a variable pointer to a constant integer and

```
int const * Constant2
```

is an alternative syntax which does the same, whereas

```
int * const Constant3
```

declares that `Constant3` is constant pointer to a variable integer and

```
int const * const Constant4
```

declares that Constant4 is constant pointer to a constant integer. Basically ‘const’ applies to whatever is on its immediate left (other than if there is nothing there in which case it applies to whatever is its immediate right).

Use of ‘const’ in Functions Return Values

Of the possible combinations of pointers and ‘`const`’, the constant pointer to a variable is useful for storage that can be changed in value but not moved in memory.

Even more useful is a pointer (constant or otherwise) to a ‘`const`’ value. This is useful for returning constant strings and arrays from functions which, because they are implemented as pointers, the program could otherwise try to alter and crash. Instead of a difficult to track down crash, the attempt to alter unalterable values will be detected during compilation.

For example, if a function which returns a fixed ‘Some text’ string is written like

```
char *Function1()
{ return “Some text”;}
```

then the program could crash if it accidentally tried to alter the value doing

```
Function1()[1]=’a’;
```

whereas the compiler would have spotted the error if the original function had been written

```
const char *Function1()
{ return "Some text";}
```

because the compiler would then know that the value was unalterable. (Of course, the compiler could theoretically have worked that out anyway but C is not that clever.)

Where it Gets Messy - in Parameter Passing

When a subroutine or function is called with parameters, variables passed as the parameters might be read from in order to transfer data into the subroutine/function, written to in order to transfer data back to the calling program or both to do both. Some languages enable one to specify this directly, such as having ‘`in:`’, ‘`out:`’ & ‘`inout:`’ parameter types, whereas in C one has to work at a lower level and specify the method for passing the variables choosing one that also allows the desired data transfer direction.

For example, a subroutine like

```
void Subroutine1(int Parameter1)
{ printf("%d",Parameter1);}
```

accepts the parameter passed to it in the default C & C++ way - which is a copy. Therefore the subroutine can read the value of the variable passed to it but not alter it because any alterations it makes are only made to the copy and are lost when the subroutine ends. E.g.

```
void Subroutine2(int Parameter1)
{ Parameter1=96;}
```

would leave the variable it was called with unchanged not set to 96.

The addition of an ‘`&`’ to the parameter name in C++ (which was a very confusing choice of symbol because an ‘`&`’ in front of variables elsewhere in C generates pointers!) causes the actual variable itself, rather than a copy, to be used as the parameter in the subroutine and therefore can be written to thereby passing data back out the subroutine. Therefore

```
void Subroutine3(int &Parameter1)
{ Parameter1=96;}
```

would set the variable it was called with to 96. This method of passing a variable as itself rather than a copy is called a ‘reference’ in C++.

That way of passing variables was a C++ addition to C. To pass an alterable variable in original C, a rather involved method was used. This involved using a pointer to the variable as the parameter then altering what it pointed to was used. For example

```
void Subroutine4(int *Parameter1)
{ *Parameter1=96;}
```

works but requires the every use of the variable in the called routine altered like that and the calling routine also altered to pass a pointer to the variable. It is rather cumbersome.

But where does ‘`const`’ come into this? Well, there is a second common use for passing data by reference or pointer instead of as a copy. That is when copying the variable would waste too much memory or take too long. This is particularly likely with large & compound user-defined variable types (‘structures’ in C & ‘classes’ in C++). So a subroutine declared

```
void Subroutine4(big_structure_type &Parameter1);
```

might being using ‘`&`’ because it is going to alter the variable passed to it or it might just be to save copying time and there is no way to tell which it is if the function is compiled in someone else’s library. This could be a risk if one needs to trust the subroutine not to alter the variable.

To solve this, ‘`const`’ can be used in the parameter list. E.g.

```
void Subroutine4(big_structure_type const &Parameter1);
```

which will cause the variable to be passed without copying but stop it from then being altered. This is messy because it is essentially making an in-only variable passing method from a both-ways variable passing method which was itself made from an in-only variable passing method just to trick the compiler into doing some optimization.

Ideally, the programmer should not need control this detail of specifying exactly how it variables are passed, just say which direction the information goes and leave the compiler to optimize it automatically, but C was designed for raw low-level programming on far less powerful computers than are standard these days so the programmer has to do it explicitly.

Messier Still - in the Object Oriented Programming

In Object Oriented Programming, calling a ‘method’ (the Object Oriented name for a function) of an object gives an extra complication. As well as the variables in the parameter list, the method has access to the member variables of the object itself which are always passed directly not as copies. For example a trivial class, ‘`Class1`’, defined as

```
class Class1
{ void Method1();
  int MemberVariable1;}
```

has no explicit parameters at all to ‘`Method1`’ but calling it in an object in this class might alter ‘`MemberVariable1`’ of that object if ‘`Method1`’ happened to be, for example,

```
void Class1::Method1()
{ MemberVariable1=MemberVariable1+1;}
```

The solution to that is to put ‘`const`’ after the parameter list like

```
class Class2
{ void Method1() const;
  int MemberVariable1;}
```

which will ban `Method1` in `Class2` from being anything which can attempt to alter any member variables in the object.

Of course one sometimes needs to combine some of these different uses of ‘`const`’ which can get confusing as in

```
const int*const Method3(const int*const&)const;
```

where the 5 uses ‘`const`’ respectively mean that the variable pointed to by the returned pointer & the returned pointer itself won’t be alterable and that the method does not alter the variable pointed to by the given pointer, the given pointer itself & the object of which it is a method!.

Inconveniences of ‘const’

Besides the confusingness of the ‘`const`’ syntax, there are some useful things which the system prevents programs doing.

One in particular annoys me because my programs often needed to be optimized for speed. This is that a method which is declared ‘`const`’ cannot even make changes to the hidden parts of its object that would not make any changes that would be apparent from the outside. This includes storing intermediary results of long calculations which would save processing time in subsequent calls to the class’s methods. Instead it either has to pass such intermediary results back to the calling routine to store and pass back next time (messy) or recalculate from scratch next time (inefficient). In later versions of C++, the ‘`mutable`’ keyword was added which enables ‘`const`’ to be overridden for this purpose but it totally relies on trusting the programmer to only use it for that purpose so, if you have to write a program using someone else's class which uses ‘`mutable`’ then you cannot guarantee that ‘`mutable`’ things will really be constant which renders ‘`const`’ virtually useless.

One cannot simply avoid using ‘`const`’ on class methods because ‘`const`’ is infectious. An object which has been made ‘`const`’, for example by being passed as a parameter in the ‘`const &`’ way, can only have those of its methods that are explicitly declared ‘`const`’ called (because C++’s calling system is too simple to work out which methods not explicitly declared ‘`const`’ don’t actually change anything). Therefore class methods that don’t change the object are best declared ‘`const`’ so that they are not prevented from being called when an object of the class has somehow acquired ‘`const`’ status. In later versions of C++, an object or variable which has been declared ‘`const`’ can be converted to changeable by use of ‘`const_cast`’ which is a similar bodge to ‘`mutable`’ and using it likewise renders ‘`const`’ virtually useless.

By Andrew Hardwick.

Written 2001.
Converted to HTML & augmented 2002/3/13.
Updated 2002/8/9, 2004/7/19, 2005/6/9 & 2006/5/22.
Spelling corrections 2008/3/4.
Grammar corrections & clarifications 2011/6/16.
Typo corrections 2011/12/29, 2012/4/4 & 2015/11/12.
Available on-line at http://duramecho.com.