C++ Theory and Practice
Mixing `const` with Type Names
The C/C++ Users Journal
December, 1996
Copyright © 1996 by Dan Saks

Last month, I attempted to clarify what I believe are some common misconceptions about the meaning of the `const` qualifier in C++ and C (see "C++ Theory and Practice: `const` as a Promise", *CUJ*, November, 1996). This month I have some more thoughts about `const`, particularly regarding the way it interacts with type names in declarations.

Over the past few months, I've received some feedback about my past articles on C++ declarations (see "The C++ Column That Needs a Name: Understanding C++ Declarations", *CUJ*, December 1995 through "C++ Theory and Practice: Declarators, Finale", *CUJ*, October, 1996). Most of it was positive, thank you. However, one recent letter suggested that I did my readers a disservice by suggesting that parsing C++ declarations is easier than most people realize. The author suggested that the better way to cope with complex declarations is to compose them from simpler typedefs.

I agree that typedefs are a good tool for simplifying declarations, and you should use them. What I want to show this month is that even when you use typedefs, you still can't get away with ignoring the underlying structure of C++ declarations. This is particularly true when you combine typedef names with cv-qualifiers (`const` and `volatile`) in declarations.

`typedef` vs. `#define`

Beginning C and C++ programmers often confuse the behavior of `typedef` and `#define`. In many situations, using a typedef name appears to have the same effect as using a parameterless macro name. For example, you can define a symbolic type such as handle as a macro:

```
#define handle int  // a macro
```

or as a typedef name:

```
typedef int handle; // a typedef name
```

Either way, when you use handle in a declaration such as

```
handle open(const char *);
```

it appears that the compiler substitutes `int` for handle. It's hard to see how the macro differs from the typedef name.

Here's the classic example that exposes the difference. Consider the meaning of

```
#define string char *   // a macro
```

as opposed to

```
typedef char *string;   // a typedef name
```

when used in the declaration

```
string s, t;
```

When `string` is a macro, the preprocessor transforms the declarations for `s` and `t` into

```
char *s, t;
```

which declares `s` as “pointer to `char`” and `t` as just “`char`”. Whoops. That’s probably not the intended result. When `string` is a typedef name, the compiler applies the entire type “pointer to `char`” to both `s` and `t` during the semantic analysis of the declaration. That probably is the intended result.

So typedef names do indeed come in handy in simplifying complex declarations, and they do eliminate many of the surprises you can get with macros. But typedef names have their own little surprises, particularly in how they combine with the cv-qualifiers `const` and `volatile`.

For example, given

```
typedef char *string;
```

the declaration

```
const string s;
```

declares `s` as a “`const` pointer to `char`”, not as a “pointer to `const char`”. In other words, the declaration for `s` is equivalent to

```
char *const s;
```

not

```
const char *s;
```

Here’s how it happens. In the declaration

```
const string s;
```

`const` and `string` are decl-specifiers. As I explained some months ago, the order of the decl-specifiers in a declaration doesn’t matter, at least not as far as a compiler is concerned. (See "The C++ Column That Needs a Name: Understanding C++ Declarations, *CUJ*, December 1995.) Thus,

```
const string s;
```

is equivalent to

```
string const s;
```

It's `s` that is `const`, and `s` is a string, which (in this example) is first and foremost a pointer. Therefore it's the pointer that's `const`, not what it points to.

By the way, this problem is not limited to just the C part of C++. It rears its head again in the context of template type parameters. For example, consider the template definition

```
template <class T>
class X
    {
    ...
    const T m;
    ...
    };
```

If you instantiate the template using `X<char *>`, then member `m` in that instantiation behaves as if declared as

```
char *const m;
```

rather than

```
const char *m;
```

Let's Talk Style

This raises an interesting style issue. Bobby Schmidt and I have been banging this one back and forth for a while. Bobby came to a conclusion a while ago, but I'm still waffling about what to do.

Some months ago I suggested to Bobby that, since people tend to want to substitute typedef names into declarations as if they were macros, maybe we should start to encourage people to place `const` as the rightmost specifier when it appears in a decl-specifier sequence. That is, although most people are accustomed to reading and writing

```
const string s;     // (1)
```

maybe we should write it as

```
string const s;     // (2)
```

and urge others to do the same.

The advantage of (2) is that, if you mentally substitute `char *` for `string` (as if it were a macro), then it becomes

```
char *const s;
```

which is the correct interpretation. Substituting `char *` for `string` in (1) becomes

```
const char *s;
```

which is incorrect. ("Incorrect" in the sense that your perception would be at odds with the compiler's, in which case, you lose.)

Well, being the persuasive fellow that I am, Bobby bought the argument right away. You may have noticed that, for some time now, he has been writing declarations such as

```
char const *s
```

and

```
int const N = 10;
```

rather than

```
const char *s
```

and

```
const int N = 10;
```

I, on the other hand, have not been able to persuade myself to follow suit.

In addition to making it easier to accurately substitute typedef names into declarations, placing the const on the right seems to have another nice property — it lets you read entire declarations, not just pointer operators, from right to left. For example,

```
T *const p;
```

declares `p` as a "`const` pointer to `T`", which mentions `const`, the pointer operator(`*`) and `T` as they appear from right to left in the declaration. Similarly,

```
T const *p;
```

declares `p` as a "pointer to `const T`", which again mentions the `*`, `const` and `T` as they appear from right to left in the declaration.

In contrast, you can't read the more common style

```
const T *p;
```

strictly from right to left or from left to right. I don't think I've ever heard anyone say `p` is a "pointer to `T const`". I think the only way you can read this strictly from right to left is as `p` is a "pointer to a `T` that's `const`". But I don't think anyone says that either. Just about everyone reads it as `p` is a "pointer to `const T`".

If you buy this argument about reading from right to left, then maybe it follows that you should write the entire sequence of decl-specifiers in reverse order. For example, you should declare what most people call an "`unsigned long int`" variable by writing

```
int long unsigned uli;
```

Yes, the order doesn't matter to a compiler, but I doubt many people would be willing to start doing this just to be "consistent".

Bobby said he doesn't find the ability to read declarations strictly from right to left to be a very compelling reason for the `const` on the right, and that's cool, because I don't either. He started placing the `const` on the right because he likes the way substituting the type name as if it were a macro makes the type come out right (as in the previous example using `string` as an alias for `char *`). He's seen programmers make mistakes with `const` and `typedef`, and this style helps avoid many of those mistakes.

I've been reluctant to start putting the `const` on the right because I'm not sure I want to encourage people to think of typedef names as macros. A macro is a sequence of source text; a typedef name is a symbol representing a bundle of compile-time type attributes. The compiler processes them differently, and I'm not sure that programmers should ever think of typedef names as macros, particularly because it's easy to misinterpret the typedef-as-macro conceptual model. But maybe it all depends on how you present the model.

For example, consider

```
typedef char *(X::*pmf)();
```

when used in

```
pmf const f;
```

If you think the typedef-as-macro model suggests that you obtain the type of `f` by just appending `const` to the end of the typedef, you will be disappointed in the results. This yields

```
char *(X::*)() const;
```

which has type "pointer to member of `X` with type `const` member function with no parameters returning pointer to `char`". And that's wrong.

The correct interpretation comes from substituting `const f` together in place of the typedef name in the original typedef declaration, and then dropping the keyword `typedef`. This yields

```
char *(X::*const f)();
```

which means `f` has type "`const` pointer to member of `X` with type function with no parameters returning pointer to `char`".

I really do believe the key insight in understanding C++ (and C) declarations is in understanding the separate sub-structures of decl-specifier sequences and declarators. You have no choice but to read declarators inside-out according to the precedence rules. The order of the decl-specifiers does not matter to the compiler. But, since C and C++ are biased toward the English language, which reads words from left to right, most programmers put the decl-specifiers in what they believe is the natural order from left to right.

What's natural? Probably what you first saw others do. And you almost always see other people put `const` to the left of the type name.

But, cv-qualifiers (`const` and `volatile`) are unique among the decl-specifiers in that they can appear in a declarator as well. They are the only decl-specifiers that you substitute along with the type name when applying the typedef-as-macro conceptual model. For example, when you use

```
typedef char *string;
```

in

```
extern const string s;
```

you take `const s` and substitute them together in place of `string` in the typedef declaration. Drop the `typedef` and that leaves you with

```
char *const s;
```

Now substitute this back in place of `const string s` in the original declaration for `s`, and you get

```
extern char *const s;
```

which is the correct interpretation of the original declaration for `s`, above.

The advantage of placing the `const` as the rightmost decl-specifier, as in

```
extern string const s;
```

is that it lets you think of `string` as a macro, `const s` as the entire macro parameter, and `extern` as merely the surrounding context of the “macro call”. And maybe that’s a good conceptual model.

I must confess, I really am thinking out loud here. I have rewritten parts of this article several times because I’ve actually changed my perception of the problem as I’ve been writing. (Writing is very good at clarifying things that way.) I started out thinking that I was going to suggest continuing to place `const` on the left, but now I find myself leaning toward putting it on the right (as Bobby does).

Well, maybe I just talked (or rather, wrote) myself into it after all.