RETURN

by Dan Saks

Placing const in Declarations

Last month I introduced the `const` qualifier as a way to declare data that you would like your C or C++ compiler to place into ROM. I noted that simply declaring an object `const` isn't enough to get it into ROM. You must also select the appropriate options for your compiler, linker, and other development tools. Moreover, each object declared `const` must satisfy certain semantic constraints before it's even eligible to be placed into ROM.

I'm working my way toward explaining those constraints, but I'm not quite there yet. As with many parts of C and C++, focusing on the semantics of `const` is difficult until you get past the syntax. This month, I'll clarify a few points about the syntax of declarations that relate to using `const`.

Many programmers find `const` confusing, probably because there are so many ways you can place `const` in a declaration. For example, starting with the declaration of a pointer variable:

```
T *p;
```

you can add `const` to produce any of:

```
const T *p;                (1)
T const *p;                (2)
T *const p;                (3)
const T *const p;          (4)
T const *const p;          (5)
```

Not all of these have distinct meanings. Here are some insights that should help you sort them out.

Every object declaration in C and C++ has two principal parts: a sequence of zero or more declaration specifiers, and a sequence of one or more declarators, separated by commas. For example:

```
static unsigned long int  *x[N] ;
declaration specifiers    declarator
```

As with many parts of C and C++, focusing on the semantics of `const` is difficult until you get past the syntax.

A declaration specifier can be any of a number of things. It can be a type specifier such as `int`, `unsigned`, `long`, `double`, or an identifier that names a type. It can be a storage class specifier such as `extern`, `static`, or `register`. It can be a function specifier, such as `inline` or `virtual`.

The draft C and C++ standards use slightly different grammars and terminology to describe this stuff, but the results are effectively the same. (These days, when I talk about C, I'm talking about C9X, the C standard undergoing revision.) What C calls a *declaration-specifier*, C++ calls a *decl-specifier*. C classifies the keyword `typedef` as a storage class, but C++ does not. Both C and C++ recognize the function specifier `inline`, but only C++ recognizes `virtual`.

A declarator is the name being declared, possibly surrounded by operators such as `*`, `[]`, `()`, and (in the case of C++) `&`. In a declarator, `*` means "pointer to," `[]` means "array of," `()` either means "function returning" or serves as grouping, and `&` means "reference to." For example, `*x[N]` is a declarator indicating that `x` is an "array of `N` elements of pointer to..." something. That something is the type specified by the declaration specifiers. Thus:

```
static unsigned long int *x[N];
```

declares `x` as an object of type "array of `N` elements of pointer to unsigned long int." (I didn't say what to make of `static` here, but I will in a moment.) In a declaration as simple as:

```
int n;
```

the declarator is just the identifier `n` without any operators.

A declarator may contain more than one identifier. The declarator `*x[N]` contains two identifiers, `x` and `N`. Only one of those is the one being declared. The other(s) must have been declared previously. The draft C++ standard uses the term *declarator-id* to distinguish the identifier being declared from other identifiers that appear in the declarator. The *declarator-id* in `*x[N]` is `x`. As far as I can tell, the draft C standard has no corresponding terminology.

Some declaration specifiers do not contribute to the type of the *declarator-id*. Rather, they specify other semantic information that applies directly to the *declarator-id*. For example, in:

```
static unsigned long int *x[N];
```

the keyword `static` does not apply to the `unsigned long int` objects that the pointers in `x` point to. Rather, it applies to `x` itself:

```
static unsigned long int        *x [N];
```

This declares that `x` is a static object of type "array of `N` elements of pointer to `unsigned long int`."

The order in which the declaration specifiers appear in a declaration does not matter. At least, it doesn't to compilers. Thus, these declarations are equivalent:

```
static unsigned long int *x[N];
int long static unsigned *x[N];
unsigned int static long *x[N];
```

I think most of us are used to placing a storage class specifier such as `static` as the first (left-most) declaration specifier, but it's just a common convention, not a language requirement.

Okay, so where does `const` fit in all this? Syntactically, `const` is another one of the declaration specifiers. However, it's an unusual specifier in that it can also appear in declarators. For example, in:

```
const T *p;
```

`const` appears as a declaration specifier. In:

```
T *const p;
```

`const` appears in the declarator. Unlike storage class specifiers, such as `extern` or `static`, which apply directly to the *declarator-id* no matter where they appear, `const` modifies the declared type in a way that depends on where it appears. To see how, let's look at each of the five different pointer declarations that I listed at the beginning of this article.

Since the order of the declaration specifiers doesn't matter, these declarations are equivalent:

```
const T *p;          (1)
T const *p;          (2)
```

Both declare that `p` has type "pointer to `const T`." `p` itself is not `const`, but the `T` object that it points to is `const`. Thus, the program can alter the value of `p`, but not the value of `*p`. In contrast:

```
T *const p;          (3)
```

declares that `p` has type "`const` pointer to `T`". Here, `p` is `const` but the `T` object it points to isn't. The program cannot alter `p`, but it can alter `*p`.

Just as (1) and (2) are equivalent, so are:

```
const T *const p;    (4)
T const *const p;    (5)
```

because they differ only in the order of the declaration specifiers. They declare that `p` has type "`const` pointer to `const T`," meaning that both the pointer and what it points to are `const`. The program can alter neither `p` nor `*p`.

Because declarations such as:

```
const T *p;          (1)
T const *p           (2)
```

are equivalent, you should pick one or the other and use it consistently. Almost all C and C++ programmers use (1), if for no other reason than that's what everyone else seems to do. For many years, I also used (1). However, for the past year or so, I've been using (2). Here's why.

When `const` appears in a declarator, it modifies a pointer. Pointer declarations read from right to left. Using declarations of form (2) instead of (1) lets you read the entire declaration, not just the pointer declarators, from right to left. That is:

```
T const *p;          (2)
```

declares `p` as a "pointer to `const T`" and:

```
T *const p;          (3)
```

declares that `p` as a "`const` pointer to `T`." This is the style I will use in upcoming columns.

ERRATA

Last month, I used this declaration as an example:

```
unsigned char two_to_the[]
  = { 1, 2, 4, 8, 16, 32, 128, 256 };
```

The declaration should have been:

```
unsigned char two_to_the[]
  = { 1, 2, 4, 8, 16, 32, 64, 128 };
```

Oh well. At least it wasn't in a mission-critical component. ESP

Dan Saks is the president of Saks & Associates, a C/C++ training and consulting company. He is also a contributing editor for the C/C++ Users Journal. *You can write to him at dsaks@wittenberg.edu.*