Dan Saks

const T vs. T const

In my last column, I discussed one of the reasons why the rules by which a compiler can place data into ROM are a bit more complicated in C++ than they are in C.[1] I have more to say about that subject, but before I do, I'd like to reply to the following query I received through e-mail from Phil Baurer at Komatsu Mining Systems:

"We're having an interesting problem using const with a typedef. I hoped you could comment on this situation. I am wondering if we are bumping into some unknown (by us) rule of the C language.

"We are using the Hitachi C compiler for the Hitachi SH-2 32-bit RISC microcontroller. We thought the following code:

```
typedef void *VP;
const VP vectorTable[]
  = {..<data>..};                (1)
```

should be identical to:

```
const void *vectorTable[]
  = {..<data>..};                (2)
```

"However, the linker places `vectorTable` in (1) into the `CONSTANT` section, but it places `vectorTable` in (2) into the `DATA` section.

"Is this the proper behavior or a bug in the compiler?"

This is proper behavior; it is not a bug. You are indeed bumping into some rules of the C language that you apparently don't know about. Don't feel bad; you're not alone. I believe many other C and C++ programmers are confused about these rules, which is why I'm answering this in my column.

I presented some of these rules in an earlier column.[2] However, in looking back at that column, I don't think I emphasized strongly enough the points which seem to be the source of your confusion. So let me try again.

Although C and C++ read mostly from top-to-bottom and left-to-right, pointer declarations read, in a sense, backwards.

Declarators

Here's the first insight:

Every 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*

A declarator is the name being declared, possibly surrounded by operators such as `*`, `[]`, `()`, and (in the case of C++) `&`. As you already know, the symbol `*` in a declarator means "pointer to" and `[]` means "array of."

Thus, `*x[N]` is a declarator indicating that `x` is an "array of `N` elements of pointer to ..." something, where that something is the type specified in the declaration specifiers. For example,

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

declares `x` as an object of type "array of `N` elements of pointer to unsigned long int." (As explained later, the keyword `static` does not contribute to the type.)

How did I know that `*x[N]` is an "array of ... pointer to ..." rather than a "pointer to an array of ...?" It follows from this rule:

The operators in a declarator group according to the same precedence as they do when they appear in an expression.

For example, if you check the nearest precedence chart for either C or C++, you'll see that `[]` has higher precedence than `*`. Thus the declarator `*x[N]` means that `x` is an array before it's a pointer.

Parentheses serve two roles in declarators: first, as the function call operator, and second, as grouping. As the function call operator, `()` have the same precedence as `[]`. As grouping, `()` have the highest precedence of all.

Most of us place storage class specifiers such as static as the first (leftmost) declaration specifier, but it's just a common convention, not a language requirement.

For example, `*f(int)` is a declarator specifying that `f` is a "function ... returning a pointer ... ." In contrast, `(*f)(int)` specifies that `f` is a "pointer to a function ... ."

A declarator may contain more than one identifier. The declarator `*x[N]` contains two identifiers, `x` and `N`. Only one of those identifiers is the one being declared, and it's called the declarator-id. The other(s), if any, must have been declared previously. For instance, the declarator-id in `*x[N]` is `x`.

A declarator need not contain any operators at all. In a declaration as simple as:

```
int n;
```

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

Declaration specifiers

Some of the declaration specifiers leading up to a declarator can be type specifiers such as `int`, `unsigned`, or an identifier that names a type. They can also be storage class specifiers such as `extern` or `static`. In C++ they can also be function specifiers such as `inline` or `virtual`.

Here's another insight:

Type specifiers contribute to the type of the declarator-id; other specifiers provide non-type information that applies directly to the declarator-id.

For example:

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

declares `x` as a variable of type "array of `N` elements of type pointer to unsigned long int." The keyword `static` specifies that `x` has statically allocated storage.

The examples in your letter lead me to suspect that you may have been tripped up by the fact that:

The keywords `const` and `volatile` are type specifiers.

For example, the const in:

```
const void *vectorTable[]
  = {..<data>..};                (2)
```

does not apply directly to `vectorTable`; it applies directly to `void`. This declaration declares `vectorTable` as a variable of type "array of pointer to const void." It appears that you were expecting it to be "const array of pointer to void."

Here's yet another important insight:

The order in which the declaration specifiers appear in a declaration doesn't matter.

Thus, for example,

```
const VP vectorTable[]
```

is equivalent to:

```
VP const vectorTable[]
```

and

```
const void *vectorTable[]
```

is equivalent to:

```
void const *vectorTable[]
```

Most of us place storage class specifiers such as `static` as the first (leftmost) declaration specifier, but it's just a common convention, not a language requirement.

The declaration specifiers `const` and `volatile` are unusual in that:

The only declaration specifiers that can also appear in declarators are `const` and `volatile`.

For example, the const in:

```
void *const vectorTable[]
```

appears in the declarator. In this case, you cannot rearrange the order of the keywords. For example:

```
*const void vectorTable[]
```

is an error.

A clarifying style

As I explained earlier, the order of the declaration specifiers doesn't matter to the compiler. Therefore, these declarations are equivalent:

```
const void *vectorTable[]          (3)

void const *vectorTable[]          (4)
```

Almost all C and C++ programmers prefer to write const and volatile to the left of the other type specifiers, as in (3). I prefer to write const and volatile to the right, as in (4), and I recommend it. Strongly.

Although C and C++ read mostly from top-to-bottom and left-to-right, pointer declarations read, in a sense, backwards. That is, pointer declarations read from right-to-left. By placing const to the right of the other type specifiers, you can read pointer declarations strictly from right-to-left and get const to come out in the "right" places. For example:

```
T const *p;
```

declares `p` as a "pointer to a const T," which is exactly what it is. Also:

```
T *const p;
```

declares `p` as a "const pointer to a T," which is also the correct interpretation.

Writing const to the right of the

Recognizing the boundary between the last declaration specifier and the declarator is one of the keys to understanding declarations.

other declaration specifiers actually makes it easier to see the effect of combining const with a typedef name. Using the original example in the letter:

```
typedef void *VP;
const VP vectorTable[];
```

One interpretation is to replace VP as follows:

```
const VP vectorTable[]
      \_____/
const void *vectorTable[]
```

which makes it appear that `vectorTable` has type "array of pointer to const void." This is wrong! The correct interpretation is to replace VP as:

```
const VP vectorTable[]
      \_____/
void *const vectorTable[]
```

That is, `vectorTable` type "array of const pointer to void," but it's not at all obvious.

Writing const as the rightmost declaration specifier makes it easier to see the correct interpretation:

```
VP const vectorTable[]
   \_____/
void *const vectorTable[]
```

Now, I realize that I'm recommending a style that hardly anyone uses. Just about everyone who uses const places it to the left. However, given how few C and C++ programmers really understand what they're doing when it comes to using const in declarations, "everyone else does it" is hardly an argument in favor of the currently popular style. Why not buck the trend and try using a clearer style?

As long as I'm on a roll here, I might as well get in my digs in on a related style point. Although most C programmers seem to have remained unsullied by this, many C++ programmers have acquired the most unfortunate habit of writing:

```
const int* p;
```

rather than:

```
const int *p;
```

That is, they use spacing to join the * with the declaration specifiers rather than with the declarator. I really believe C++ programmers do themselves and each other a disservice when they write declarations in this style. Sure, the spacing makes no difference to the compiler, but putting the space after the * leaves many people with a false impression about the underlying structure of declarations. Recognizing the boundary between the last declaration specifier and the declarator is one of the keys to understanding declarations. Breaking up declarators with spaces this way only confuses the situation.

I hope I've answered your question and clarified some issues. **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. *He served for many years as secretary of the C++ standards committee and remains an active member. With Thomas Plum, he wrote* C++ Programming Guidelines. *You can write to him at dsaks@wittenberg.edu.*

References

1. "Static vs. Dynamic Initialization," December 1998, p. 19.
2. "Placing `const` in Declarations," June 1998, p. 19.