RETURN

Dan Saks

What const Really Means

In my last installment, I explained the basic syntax of object declarations in C and C++ in terms of declaration specifiers and declarators. I used these terms to explain how the `const` qualifier fits into that syntax. Along the way, I dropped some hints about how placing `const` in a declaration affects the declaration's meaning. This month, I'll explain those effects more precisely.

Last time, I wrote that these declarations are equivalent:

```
const T *p;
T const *p;
```

and that (unlike just about everybody else) I prefer the latter form. Both declare that `p` has type "pointer to `const T`." `p` 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`.

That last sentence is an oversimplification. As such, it's probably misleading. A subtle but extremely important distinction exists between the expression that you use to designate (refer to) an object and the object itself. Now, I'm about to tell you something you probably already know, but please don't get impatient and wander off. I'm going to use terminology that's more precise than you're probably accustomed to using. I need that precise terminology to clarify the statement that I said was misleading.

For example, given:

```
char c = ' ';
char *pc = &c;
```

then `c` and `*pc` are two different expressions that designate the same object. An expression such as:

```
*pc = 'x';
```

alters the `char` object that `*pc` designates. In so doing, the assignment alters the `char` object that `c` designates because `c` and `*pc` designate the same object. (`c` now has the value "`x`.")

Of course, you already knew that. However, we rarely use phrases such as "it alters the `char` object that `c` designates." We usually just say "it alters `c`." Most of the time, the subtle distinction doesn't matter. In the following discussion, it does.

The expression you use to refer to an object differs from the object itself. This subtle distinction usually doesn't matter, but sometimes it does.

Returning to my earlier example:

```
T const *p;
```

declares that `p` has type "pointer to `const T`." Thus, the expression `*p` has type "`const T`." In standardese, we say that `*p` has a `const`-qualified type. Although the `const` qualifier appears in `p`'s declaration, it applies only to what `p` points to, not to `p` itself. If `p` were declared as "`const` pointer to ..." then `p` would have a `const`-qualified type. However, `p` is just plain "pointer to ...," so it has an unqualified type.

Here again is the statement that I said was misleading: given the above declaration for `p`, the program can alter the value of `p`, but not the value of `*p`. To see why it's misleading, let's consider an example in which `T` is `int`.

The declarations:

```
int x[N];
int const *p = x;
```

compile without error. The second one initializes `p` to point to the first element of `x`. Thereafter, the `const`-qualified expression `*p` designates the same object as the unqualified expression `x[0]`.

Normally, you don't see those declarations for `x` and `p` in the same scope. You're more likely to see `p` as a formal parameter, as in:

```
int x[N];
int f(int const *p);
```

Then a call such as `f(x)` initializes `p` with `x`.

In any event, the program cannot use `*p` to alter the object:

```
*p = 1;    // error
```

However, the program can still use `x[0]` to alter that same object:

```
x[0] = 1;    // ok
```

By changing the value of `x[0]`, the program also changes the value of `*p`.

The expression `p` has an unqualified type. Therefore, the program can change the value of the object that `p` designates. After:

```
++p;
```

`*p` designates the same object as `x[1]`. The program still can't use `*p` to change the object, but now changing the value of `x[1]` effectively changes the value of `*p`.

Thus, a program can alter the value of `*p` after all. It just can't use `*p` to do it. Rather, it must use an unqualified expression such as `x[i]`.

Here, then, is a more precise statement of what a declaration such as:

```
T const *p;
```

means. A program can use the expression `p` to alter the value of the pointer object that `p` designates, but it can't use the expression `*p` to alter the value of any objects that `*p` might designate. If the program has another expression `e` of unqualified type that designates an object that `*p` also designates, the program can still use `e` to change that object.

Thus, a program might be able to change an object right out from under a `const`-qualified expression.

If that's so, then what good is the `const` qualifier? Does it really do anything? Fortunately, it does. Even though the `const` qualifier doesn't have as much oomph as you might have expected, it still has enough so that you can do some useful things with it.

C and C++ programs can convert a pointer of one type into a pointer of a compatible type that uses the `const` qualifier differently. Such a conversion is a qualification conversion. However, both languages restrict qualification conversions in a way that allows for a consistent notion of objects that are truly read-only and can reside in ROM.

Let's look at a slight variation on the previous example:

```
int n = 0;
int const *p = &n; // ok
```

Here, `n` designates a modifiable `int` object. At this point, the compiler knows that code later in the program is allowed to alter that `int` object. Therefore, it cannot place the object in ROM.

The expression `&n` has type "pointer to `int`." The declaration for `p` converts `&n` to type "pointer to `const int`," adding a `const` qualifier in the process. This is a valid qualification conversion. This conversion in no way invalidates `n`'s declaration. The program can still use `n` to alter the `int` object, even if it can't use `*p` for the same purpose.

In contrast, consider:

```
int const n = 10;
int *p = &n;     // error
```

Here, `n` designates a non-modifiable `int` object. If we expect the compiler to be able to place `const` objects in ROM, the compiler must be allowed to presume that no code in the program will alter this `int` object. It must also defend its presumption as follows.

The expression `&n` has type "pointer to `const int`." The declaration for `p` converts `&n` to type "pointer to `int`" by stripping away a `const` qualifier. This is not a valid qualification conversion. It opens the door for an expression such as `+++p` to modify an object previously defined as `const`.

I suggest that you "think of `const` as a promise." (See my article "C++

What good is the const qualifier? Does it really do anything? Fortunately, it does. Even though the const qualifier doesn't have as much oomph as you might have expected, it still has enough so that you can do some useful things with it.

Theory and Practice: `const` as a Promise," *The C/C++ Users Journal*, November 1996.) Making a promise about your own work doesn't mean that others who came before you are bound (retroactively) by the same promise. However, once you make a promise about your own work, you must not entrust that work to others unless they agree to uphold the promise as well. Otherwise, your promise means nothing.

Let's apply this reasoning to another example. Consider:

```
int m = 0;
int *const p = &m;
int *q = p;
```

Here, `m` designates a non-`const` `int` object. The declaration makes no promises that the program won't alter `m`.

The expression `p` has type "`const` pointer to (non-`const`) `int`." `p` has a `const`-qualified type. The program promises that once it initializes `p`, it won't change `p`. However, `*p` has an unqualified type, so the program can still use `*p` to alter an `int` object. The expression `*p` designates the same object as `m`, and that object is non-`const`, so `p`'s declaration does not violate any promises made by `m`'s declaration.

The expression `q` has type "(non-`const`) pointer to (non-`const`) `int`." `q`'s declaration makes no promises about either `q` or `*q`. The declaration initializes `q` by copying `p`. `p` is indeed `const`, but this initialization doesn't alter `p`; it just copies `p`. Afterwards, `q` points to the same object as `p`, but that object is non-`const`. Again, `q`'s declaration violates no promises in any prior declarations.

As I mentioned in an earlier column, placing data in ROM is just one of several uses for the `const` qualifier. Using the `const` qualifier in a declaration does not assure that the declared object will actually wind up in ROM. The declaration must satisfy other semantic constraints. In the coming months, I'll elaborate those constraints and explain other uses for the `const` qualifier.

Thanks to my brother Joel for a very helpful critique of this column.

A little help, please

In the short time that I've been writing this column, I've already received several requests from readers looking for books on embedded programming techniques. If you know of any such books, whether good or bad, please send me the titles so I can pass them on to others. I'd also welcome any short comments on the book(s) that you'd care to share. **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 *(Plum Hall, 1991). You can write to him at dsaks@wittenberg.edu.*