Dan Saks

Top-Level cv-Qualifiers in Function Parameters

For the past few months, I've been discussing the role of cv-qualifiers (the keywords `const` and `volatile`) appearing in function parameter declarations. I've been focusing on how you can overload functions whose parameter types differ only in their use of cv-qualifiers,[1] and why you might want to do so.[2,3]

Although cv-qualifiers that appear in a parameter declaration usually become part of the function's signature, they're not always part of the signature. There are times when C++ ignores cv-qualifiers in parameter declarations as it determines a function's signature. In contrast, C never ignores cv-qualifiers in parameter declarations. This subtle difference between C++ and C is my topic for this month.

Passing by value vs. by address

When a C++ compiler encounters a call to an overloaded function, it selects the function to be called by matching the types of the actual arguments in the call against the types of the formal parameters in one of the function declarations. The compiler's overload resolution process often takes cv-qualifiers into account. For example, given three overloaded functions:

```
void f(T *);
void f(T const *);
void f(T volatile *);
```

and this assortment of pointer variables:

```
T *p;
T const *pc;
T volatile *pv;
```

Although the const and volatile qualifiers usually add valuable compile-time type information to your programs, they don't always live up to all of your expectations.

the expression `f(p)` calls `f(T *)`, `f(pc)` calls `f(T const *)`, and `f(pv)` calls `f(T volatile *)`.

The previous example illustrates overloading with cv-qualifiers using functions that pass parameters by address. Let's see how the behavior changes when functions pass parameters by value rather than by address.

For example, a function `g` declared as:

```
int g(int n);
```

has a parameter `n` passed by value. A call to `g` as in:

```
int k;
...
g(k);
```

copies argument `k` to parameter `n` without altering `k`. Even if you declared `k` as:

```
int const k = 1024;
```

the call `g(k)` would work just as well. You can pass a constant argument as a nonconstant parameter, again because passing by value just makes a copy of the argument. In this example, parameter `n` is nonconstant, so `g` can change `n`'s value. However, changing `n`'s value inside `g` has no effect on `k`'s value because `n` doesn't refer back to `k` in any way.

Changing `g`'s declaration to:

```
int g(int const n);
```

has no effect on code that calls `g`. A call such as `g(k)` still copies argument `k` to parameter `n` without altering `k`. It doesn't matter whether `k` is constant. You can always read the value of a constant; you just can't write to it.

Although declaring parameter `n` as

constant doesn't matter to code that calls `g`, it does matter to code within `g`. When `n` is nonconstant, `g` can use `n` as a read-write variable, just as it can use any nonconstant local variable. When `n` is constant, `g` can't alter the value in `n`. However, `g` can still copy `n` to a nonconstant local variable and perform computations in that local variable, as in:

```
int g(int const n)
  {
  int v = n;

  // can alter v here

  }
```

Declaring a parameter passed by value as constant may affect the function's implementation, but it doesn't affect the function's outward behavior as seen by any caller.

Overloading

The previous discussion raises a number of questions about what it means to declare a pair of functions named `g` as:

```
int g(int n);
int g(int const n);
```

Do you expect `g(3)` to call `g(int)` or `g(int const)`? In other words, should the call choose the `g` that can alter its copy of `3`, or the `g` that cannot alter its copy? Or, is it an error to even declare these functions in the same scope? The two `g`'s declared above exhibit identical outward behavior. Therefore, when a C++ compiler encounters a call to `g`, it has no basis for preferring one `g` over the other.

C++ avoids making the choice by treating both `g`'s as the same `g`. Specifically, the compiler ignores the const qualifier in:

```
int g(int const n);
```

as it determines the function's signature. Thus, the previous function has the same signature as a function declared as:

```
int g(int n);
```

Writing both of these declarations in the same scope of a C++ program is not an error. However, defining both of these functions in the same program is an error, which might not be reported until link time.

Here we see a difference between C and C++. In C, declaring both:

```
int g(int n);
int g(int const n);
```

in the same scope is an error. C never ignores cv-qualifiers in a function parameter declaration. In C, these two `g`'s have different function types. The second declaration provokes a compile-time error because C does not permit function overloading.

Top-level cv-qualifiers

In general, C++ does not include cv-qualifiers in a function's signature when they appear at the "top-level" of a parameter type. Here's a bit of background to help you understand what I mean.

Types in C and C++ can have one or more levels of composition. For example, `p` declared as:

```
T *p;
```

has type "pointer to `T`," which is a type composed of two levels. The first level is "pointer to" and the second level is "`T`." The declaration:

```
T *f(int);
```

declares `f` as a "function returning pointer to `T`." This type has three levels. The first is "function returning," the second is "pointer to," and the third is "`T`."

Different cv-qualifiers can appear at different levels of composition. For example:

```
T *const p;
```

declares `p` with type "constant pointer to `T`." Here, the const qualifier applies only to the first level. In contrast,

```
T volatile *q;
```

declares `q` with type "pointer to volatile `T`." Here, the volatile qualifier applies only to the second level.

In C++, a cv-qualifier that applies to the first level of a type is called a *top-level cv-qualifier*. For example, in:

```
T *const p;
```

the top-level cv-qualifier is `const`, and in:

```
T const *volatile q;
```

the top-level cv-qualifier is `volatile`. On the other hand:

```
T const volatile *q;
```

has no top-level cv-qualifiers. In this case, the cv-qualifiers `const` and `volatile` appear at the second level.

Fundamental types such as `char`, `int`, and `double` have only one level of composition. In a declaration such as:

```
int const n = 10;
```

the top-level cv-qualifier is `const`.

Here's a more precise statement of the way C++ treats cv-qualifiers in parameter types:

The signature of a function includes all cv-qualifiers appearing in that function's parameter types, except for those qualifiers appearing at the top-level of a parameter type.

For example, in:

```
int f(char const *p);
```

the const qualifier is not at the top level in the parameter declaration, so it is part of the function's signature.

On the other hand, in:

```
int f(char *const p);
```

the const qualifier is at the top level, so it is not part of the function's signature. This function has the same signature as:

```
int f(char *p);
```

In a function declared as:

```
int f(char const *const p);
```

the const qualifier to the left of the `*` is not at the top level, so it is part of the function's signature. However, the const qualifier to the right of the `*` is at the top level, so it is not part of the function's signature. Thus, the function declared just above has the same signature as:

```
int f(char const *p);
```

It's important to note that C++ does not ignore top-level cv-qualifiers in object and type declarations. For example, in declaring an object such as:

```
port volatile *const p = ... ;
```

the top-level cv-qualifier is `const`. This is not a parameter declaration, so all cv-qualifiers are significant. The object `p` is indeed constant.

More to come

Although C++ ignores top-level cv-qualifiers in parameter declarations when determining function signatures, it does not ignore those cv-qualifiers entirely. I'll explain what I mean by that in my next column. **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). You can write to him at dsaks@wittenberg.edu.*

References

1. Saks, Dan, "Using `const` and `volatile` in Parameter Types," *Embedded Systems Programming*, September 1999, p. 77.
2. Saks, Dan, "Overloading with `const`," *Embedded Systems Programming*, December 1999, p. 81.
3. Saks, Dan, "More on Overloading with `const`," *Embedded Systems Programming*, January 2000, p. 71.