Appearing and Disappearing consts in C++

Scott Meyers

February 12, 2011

If you write "int i;" in C++, i's type seems obvious: int. If you write "const int i;", i's type seems equally obviously to be const int. Often, these types are exactly what they seem to be, but sometimes they're not. Under certain circumstances, C++ treats non-const types as const, and under others it treats const types as non-const. Understanding when consts appear and disappear lends insight into the behavior of C++, and that makes it possible to use the language more effectively.

This article examines various aspects of type declaration and deduction in both current standard C++ as well as the forthcoming revised standard (C++0x), with an eye towards helping developers understand how and why the effective type of a variable can be different from what's "obvious." Readers will find the article most useful if they are familiar with the basic rules for template argument deduction, are aware of the difference between lvalues and rvalues, and have had some exposure to the new C++0x features lvalue and rvalue references, auto variables, and lambda expressions.

Given

```
int i;
```

what is the type of i? No, this is not a trick question, the type is int. But suppose we put i into a struct (or a class – it makes no difference):

```
struct S {
    int i;
};
```

What is the type of S::i? Still int, right? Right. It's not quite the same kind of int as the one outside the struct, because taking the address of S::i gives you a member pointer (i.e., an int S::*) instead of an int*, but S::i's type is still int.

Now suppose we have a pointer to this struct:

```
S *ps;
```

What is the type of ps->i? (The fact that ps doesn't point to anything in this example is not relevant. The expression ps->i has a type, even if executing the code would yield undefined behavior, e.g., a crash.)

This is still not a trick question. The type of ps->i is int. But what if we have a pointer to const?

```
const S* pcs;
```

What is the type of pcs->i? Now things are murkier. pcs->i refers to the int i in S, and we just agreed that the type of S::i is int. But when S::i is accessed through pcs, S becomes a const struct, and S::i is thus const. So one can argue that the type of pcs->i should be const int, not int.

In current C++ (i.e., the language defined by the international standard adopted in 1998 and slightly revised in 2003, henceforth known as C++98), the type of pcs->i is const int, and that's pretty much the end of the story. In “new” C++ (i.e., the language defined by the international standard poised to be adopted this year and commonly referred to as C++0x), the type of pcs->i is also const int, which, given the importance of backwards compatibility, is reassuring. But the story is more nuanced.

decltype

C++0x introduces decltype, which is a way to refer to the type of an expression during compilation. Given the earlier declaration for i, for example, decltype(i) is int. But what happens when we apply decltype to pcs->i? The standardization committee could have simply decreed that it's const int and been done with it, but the logic that says the type is int (i.e., the type of S::i, because, after all, we're still referring to i in S) isn't unreasonable, and it turns out that being able to query the declared type of a variable, independent of the expression used to access it, is useful. Besides, this is C++, where, given a choice between reasonable alternatives, the language often sits back and lets you choose. That's the case here.

The way you choose is quintessentially C++. If you apply decltype to the name of a variable, you get the declared type of that variable. For example, this declares variables x and y of the same types as i and S::i:

```
decltype(i) x;            // x is same type as i, i.e., int
decltype(pcs->i) y;       // y is same type as S::i, i.e., int
```

On the other hand, if you apply decltype to an expression that is not the name of a variable, you get the effective type of that expression. If the expression corresponds to an lvalue, decltype always returns an lvalue reference. I'll show an example in a minute, but first I have to mention that while “i” is the name of a variable, “(i)” is an expression that is not the name of a variable. In other words, tossing parentheses around a variable yields an expression that has the same runtime value as the variable, but during compilation, it's just an expression – not the name of a variable. This is important, because it means that if you want to know the type of S::i when accessed through pcs, decltype will give it to you if you ask for it with parentheses around pcs->i:

```
decltype((pcs->i))        // type returned by decltype is const int& (see also discussion below)
```

Contrast this with the example above without the extra parentheses:

```
decltype(pcs->i)          // type returned by decltype is int
```

As I mentioned, when you ask decltype for the type of an expression that is not the name of a variable, it returns an lvalue reference if the type is an lvalue. That's why decltype((pcs->i)) returns const int&: pcs->i is an lvalue. If you had a function that returned a const int, that type would be an rvalue, and decltype would indicate that by not reference-qualifying the type it returns for a call to that function:

```
const int f();            // function returning a const int (i.e., an rvalue)
decltype(f()) x = 0;      // type returned by decltype for a call to f is const int (not const int&),
                                   // so this declares x of type const int and initializes it to 0
```

Unless you're a heavy-duty library writer, you probably won't need to worry about the subtle behavioral details of decltype. However, it's certainly worth knowing that there may be a difference between a variable's declared type and the type of an expression used to access that variable, and it's equally worth knowing that decltype can help you distinguish them.

Type Deduction

You might think that this distinction between a variable's declared type and the type of an expression used to access it is new to C++0x, but it fundamentally exists in C++98, too. Consider what happens during type deduction for function templates:

```
template<typename T>
void f(T p);

int i;

const int ci = 0;

const int *pci = &i;

f(i);                    // calls f<int>, i.e., T is int

f(ci);                   // also calls f<int>, i.e., T is int

f(*pci);                 // calls f<int> once again, i.e., T is still int
```

Top-level consts are stripped off during template type deduction[‡], so the type parameter T is deduced to be int for both ints and const ints, both for variables and expressions, and this is true for both C++98 and C++0x. That is, when you pass an expression of a particular type to a template, the type seen (i.e., deduced) by the template may be different from the type of the expression.

Type deduction for auto variables in C++0x is essentially the same as for template parameters. (As far as I know, the only difference between the two is that the type of auto variables may be deduced from initializer lists, while the types of template parameters may not be.) Each of the following declarations therefore declare variables of type int (never const int):

```
auto a1 = i;

auto a2 = ci;

auto a3 = *pci;

auto a4 = pcs->i;
```

During type deduction for template parameters and auto variables, only *top-level* consts are removed. Given a function template taking a pointer or reference parameter, the constness of whatever is pointed or referred to is retained:

‡ Top-level volatiles are treated the same way as top-level consts, but to avoid having to repeatedly say "top-level consts or volatiles," I'll restrict my comments to consts and rely on your understanding that C++ treats volatiles the same way.

```
template<typename T>
void f(T& p);

int i;

const int ci = 0;

const int *pci = &i;

f(i);                   // as before, calls f<int>, i.e., T is int

f(ci);                  // now calls f<const int>, i.e., T is const int

f(*pci);                // also calls f<const int>, i.e., T is const int
```

This behavior is old news, applying as it does to both C++98 and C++0x. The corresponding behavior for auto variables is, of course, new to C++0x:

```
auto& a1 = i;           // a1 is of type int&

auto& a2 = ci;          // a2 is of type const int&

auto& a3 = *pci;        // a3 is also of type const int&

auto& a4 = pcs->i;      // a4 is of type const int&, too
```

Lambda Expressions

Among the snazzier new features of C++0x is lambda expressions. Lambda expressions generate function objects. When generated from lambdas, such objects are known as *closures*, and the classes from which such objects are generated are known as *closure types*. When a local variable is captured by value in a lambda, the closure type generated for that lambda contains a data member corresponding to the local variable, and the data member has the type of the variable.

If you're not familiar with C++0x lambdas, what I just wrote is gobbledygook, I know, but hold on, I'll do my best to clear things up. What I want you to notice at this point is that I said that the type of a data member in a class has the same type as a local variable. That means we have to know what "the same type as a local variable" means. If it were drop-dead simple, I wouldn't be writing about it.

But first let me try to summarize the relationship among lambdas, closures, and closure types. Consider this function, where I've highlighted the lambda expression:

```
void f(const std::vector<int>& v)
{
    int offset = 4;
    std::for_each(v.cbegin(), v.cend(), [=](int i) { std::cout << i + offset; });
}
```

The lambda causes the compiler to generate a class that looks something like this:

```
class SomeMagicNameYouNeedNotKnow {
public:
    SomeMagicNameYouNeedNotKnow(int offset): m_offset(offset) {};

    void operator()(int i) const { std::cout << i + m_offset; }

private:
    int m_offset;
};
```

This class is the *closure type*. Inside f, the call to std::for_each is treated as if an object of this type – the *closure* – had been used in place of the lambda expression. That is, as if the call to std::for_each had been written this way:

```
    std::for_each(v.cbegin(), v.cend(), SomeMagicNameYouNeedNotKnow(offset));
```

Our interest here is in the relationship between the local variable offset and the corresponding data member in the closure type, which I've called m_offset. (Compilers can use whatever names they like for the data members in a closure type, so don't read anything into my use of m_offset.) Given that offset is of type int, it's not surprising that m_offset's type is also int, but notice that, in accord with the rules for generating closure types from lambdas, operator() in the closure type is declared const. That means that in the body of operator() – which is the same as the body of the lambda – m_offset is const. Practically speaking, m_offset's type is const int, and if you try to modify it inside the lambda, your compilers will exchange cross words with you:

```
    std::for_each(v.cbegin(), v.cend(), [=](int i) { std::cout << i + offset++; });          // error!
```

If you really want to modify your copy of offset (i.e., you really want to modify m_offset), you'll need to use a mutable lambda[*]. Such lambdas generate closure types where the operator() function is not const:

```
    std::for_each(v.cbegin(), v.cend(), [=](int i) mutable { std::cout << i + offset++; });    // okay
```

When you capture a copy of a variable, const is effectively slapped onto the copy's type unless you preemptively slap a mutable on the lambda first.

In the function f, offset is never modified, so you might well decide to declare it const yourself. This is laudable practice, and, as one might hope, it has no effect on the non-mutable lambda:

```
    void f(const std::vector<int>& v)
    {
        const int offset = 4;
        std::for_each(v.cbegin(), v.cend(), [=](int i) { std::cout << i + offset; });      // fine
    }
```

Alas, it's a different story with the mutable lambda:

```
    void f(const std::vector<int>& v)
    {
        const int offset = 4;
```

[*] Either that or cast the constness of your copy of offset away each time you want to modify it. Using a mutable lambda is easier and yields more comprehensible code.

```
    std::for_each(v.cbegin(), v.cend(), [=](int i) mutable { std::cout << i + offset++; });        // error!
}
```

"Error? Error?!," you say? Yes, error. This code won't compile. That's because when you capture a const local variable by value, the copy of the variable in the closure type is also const: *the constness of the local variable is copied to the type of the copy in the closure.* The closure type for that last lambda is essentially this:

```
class SomeMagicNameYouNeedNotKnow {
    const int m_offset;

public:
    SomeMagicNameYouNeedNotKnow(int p_offset): m_offset(p_offset){};
    void operator()(int i) { std::cout << i + m_offset++; }                                // error!
};
```

This makes clear why you can't do an increment on m_offset. Furthermore, the fact that m_offset is defined to be const (i.e., is declared const at its point of definition) means that you can't even cast away its constness and rely on it being modified, because the result of attempting to modify the value of a object defined to be const is undefined if the constness of that object is cast away.

One might wonder why, when deducing the type of a data member for a closure type, the top-level constness of the source type is retained, even though it's ignored during type deduction for templates and autos. One might also wonder why there is no way to override this behavior, i.e., to tell compilers to ignore top-level consts when creating data members in closure types for captured-by-value local variables. I, myself, have wondered these things, but I have been unable to find out the official reasoning behind them. However, it's been observed that if code like this were valid,

```
void f(const std::vector<int>& v)
{
    const int offset = 4;
    std::for_each( v.cbegin(), v.cend(),
                   [=](int i) mutable { std::cout << i + offset++; });          // not legal, but suppose
}                                                                               // it were
```

a casual reader might mistakenly conclude that the const local variable offset was being modified, even though it would actually be the closure's copy of offset that was being operated on.

An analogous concern could be the reason why operator() in a closure class generated from a non-mutable lambda is const: to prevent people from thinking that the lambda is modifying a local variable when it is actually changing the value of its copy of that variable:

```
void f(const std::vector<int>& v)
{
    int offset = 4;                                                             // now not const
    std::for_each( v.cbegin(), v.cend(),
                   [=](int i) { std::cout << i + offset++; });                  // now not mutable, yet
}                                                                               // the code is still invalid
```

If this code compiled, it would not modify the local variable offset, but it's not difficult to imagine that somebody reading it might think otherwise.

I'm don't personally find such reasoning persuasive, because the author of the lambda has expressly requested that the resulting closure contain a *copy* of offset, and I think we can expect readers of this code to take note of that, but what I think isn't important. What's important is the rules governing the treatment of types when variables used in lambdas are captured by copy, and the rules are as I've described them. Fortunately, the use cases for lambdas that modify their local state are rather restricted, so the chances of your bumping up against C++'s treatment of such types is fairly small.

Summary

In the following, the italicized terms are of my own making; there is nothing standard about them.

- An object's *declared type* is the type it has at its point of declaration. This is what we normally think of as its type.
- The *effective type* of an object is the type it has when accessed via a particular expression.
- In C++0x, decltype can be used to distinguish an object's declared and effective types. There is no corresponding standard functionality in C++98, although much of it can be obtained via compiler extensions (e.g., gcc's typeof operator) or libraries such as Boost.Typeof.
- During type deduction for template parameters and C++0x auto variables, top-level consts and volatiles are ignored.
- The declared types of copies of variables captured by value in lambda expressions have the constness of the types of the original variables. In non-mutable lambdas, the effective types of such copies are always const, because the closure class's operator() is a const member function.

Acknowledgments

Walter Bright and Bartosz Milewski provided useful comments on drafts of this article.

[Sidebar] Displaying Types

Regrettably, there's no standard way to display the type of a variable or expression in C++. The closest thing we have is the name member function in the class std::type_info, which is what you get from typeid. Sometimes this works reasonably well,

```
int i;
std::cout << typeid(i).name();         // produces "int" under Visual C++, "i" under gcc.
```

but sometimes it doesn't:

```
const int ci = 0;
std::cout << typeid(ci).name();        // same output as above, i.e., const is omitted
```

You can improve on this by using template technology to pick a type apart, produce string representations of its constituent parts, then reassemble those parts into a displayable form. If you're lucky, you can find somebody who has already done this, as I did when I posted a query about it to the Usenet newsgroup comp.lang.c++. Marc Glisse offered a PrintType template that produces pretty much what you'd expect from the following source code when compiled under gcc. (These examples are taken from Marc's code.)

```
struct Person{};
typedef int (*fun)(double);
std::cout << Printtype<unsigned long*const*volatile*&()>().name();
std::cout << Printtype<void(*&)(int(void),Person&)>().name();
std::cout << Printtype<const Person&(...)>().name();
std::cout << Printtype<long double(volatile int*const,...)>().name();
std::cout << Printtype<int (Person::**)(double)>().name();
std::cout << Printtype<const volatile short Person::*>().name();
```

Marc's approach uses variadic templates, a C++0x feature which, as far as I know, is currently supported only under gcc, but with minor alternation, I expect his code would work as well under any largely conforming C++98 compiler. Information on how to get it is available via the references under Further Information.

Further Information

- C++0x entry at Wikipedia. A nice, readable, seemingly up-to-date overview of language and library features introduced by C++0x.
- *Working Draft, Standard for Programming Language C++*, ISO/IEC JTC1/SC22/WG21 Standardization Committee, Document N3225, 27 November 2010. This is the nearly-final C++0x draft standard. It's not an easy read, but you'll find definitive descriptions of decltype in section 7.1.6.2 (paragraph 4), of auto in section 7.1.6.4, and of lambda expressions in section 5.1.2.
- decltype entry at Wikipedia. A nice, readable, seemingly up-to-date overview of C++0x's decltype feature.
- *Decltype (revision 5)*, Jaako Järki et al, ISO/IEC JTC1/SC22/WG21 Standardization Committee Document N1978, 24 April 2006. Unlike the draft C++ standard, this document provides background and motivation for decltype, but the specification for decltype was modified after this document was written, so the specification for decltype in the draft standard does not exactly match what this document describes.
- Referring to a Type with typeof, GCC Manual. Describes the typeof extension.
- Boost.Typeof, Arkadiy Vertleyb and Peder Holt. Describes the Boost.Typeof library.
- *Deducing the type of variable from its initializer expression*, Jaako Järvi et al, ISO/IEC JTC1/SC22/WG21 Standardization Committee Document N1984, 6 April 2006. This documents gives background and motivation for auto variables. The specification it proposes may not match that in the draft standard in every respect. (I have not compared them, but it's common for details to be tweaked during standardization.)
- *Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4)*, Jaako Järvi et al, ISO/IEC JTC1/SC22/WG21 Standardization Committee Document N2550, 29 February 2008. This is a very brief summary of lambda expressions, but it references earlier standardization documents that provide more detailed background information. The specification for lambdas was revised several times during standardization.
- *Overview of the New C++ (C++0x)*, Scott Meyers, Artima Publishing, 16 August 2010 (most recent revision). Presentation materials from my professional training course in published form. Not as formal or comprehensive as the standardization documents above, but much easier to understand.
- String representation of a type, Usenet newsgroup comp.lang.c++, 28 January 2011 (initial post). Discussion of how to produce displayable representations of types. Includes link to Marc Glisse's solution.