An entity name passed to **decltype**, as mentioned above, produces the type of the entity. If an entity name is enclosed in an additional set of parentheses, however, **decltype** interprets its argument as an expression, and its result incorporates the value category:

```
int i;
decltype(i)   l = i; // -> int
decltype((i)) m = i; // -> int&
```

Similarly, for all other *lvalue* expressions, the result of **decltype** will be an *lvalue* reference:

```
int* pi = &i;
decltype(*pi) j = *pi; // -> int&
decltype(++i) k = ++i; // -> int&
```

Finally, the value category of the expression will be an *xvalue* if it is a cast to or a function returning an *rvalue* reference:

```
int i;
decltype(static_cast<int&&>(i)) j = static_cast<int&&>(i); // -> int&&
int&& g();
decltype(g()) k = g();                                     // -> int&&
```

Much like the **sizeof** operator (which is also resolved at compile time), the expression operand of **decltype** is not evaluated:

```
void test1()
{
    int i = 0;
    decltype(i++) j;  // equivalent to int j;
    assert(i == 0);   // The expression i++ was not evaluated.
}
```

Note that the choice of using the postfix increment is significant; the prefix increment yields a different type:

```
void test2()
{
    int i = 0;
    int m = 1;
    decltype(++i) k = m; // equivalent to int& k = m;
    assert(i == 0);      // The expression ++i was not evaluated.
}
```

Use Cases

Avoiding unnecessary use of explicit typenames

Consider two logically equivalent ways of declaring a vector of iterators into a list of `Widgets`: