C++ Primer, Fifth Edition

Workarounds for C++11 Features not Implemented in Visual Studio 2012

Barbara E. Moo
bmoo@att.net

August 9, 2012

Some C++11 features used in *C++ Primer* are not yet implemented in the latest Microsoft compiler (Visual Studio 2012). This document outlines these unimplemented features and, where possible, suggests workarounds.

We used the release candidate, Version 17.00.50522.1, to test the code that is included on the book's website. When Visual Studio is generally available (expected to be sometime in mid-September) we will verify the code and make any updates as appropriate.

`constexpr` variables Use `const` instead. Note, the compiler will not verify that the initializer is a constant expression.

`constexpr` functions Explicitly write out the code that would have been inside the function. Note, preprocessor macros are a reasonable substitute for `constexpr` functions. However, as with `const` variables, the compiler will not verify whether the macro returns a constant expression.

`constexpr` constructors and member functions There is no workaround.

`= default` Explicitly define the member. Because the compiler does not yet implement in-class initializers, it is important to remember that the default constructor should explicitly initialize every member of built-in type.

`= delete` For a member function, declare the member as `private` and do not supply a definition for that member.

Delegating and inherited constructors Write the corresponding definitions directly.

`__func__` Use the Microsoft nonstandard CPP variable, `__FUNCTION__`, which, like `__func__`, holds the name of the current function.

Using `function` with pointers to member There is a known bug that causes the compiler to incorrectly reject some uses of pointers to members with the library `function` template. Use `mem_fn` to generate a callable object. For example, assuming that `Lshift` is a member of a class named `ShiftOps`:

```
// compiler incorrectly rejects this code
function<int (ShiftOps*, int, int)> memp =
                          &ShiftOps::Lshift;
// equivalent code that does compile
function<int (ShiftOps*, int, int)> memp =
                          mem_fn(&ShiftOps::Lshift);
```

In-class initializers Explicitly supply the initializer in the constructor initializer list of every constructor that would otherwise use the in-class initializer. Doing so is particularly important for members of built-in type.

`initializer_list<T>` In place of an `initializer_list<T>` parameter, use a library container or an array. For example, we can rewrite the `error_msg` function from page 221 to take pointers to elements of an array:

```
// original code
void error_msg(initializer_list<string>);
// workaround version
void error_msg(const string*, const string*);
```

Users will have to put arguments in a local array and pass pointers to that array. The Microsoft library defines the `begin` and `end` functions, which we can use to calculate these pointers:

```
// original code
if (expected != actual)
 error_msg({"functionX", expected, actual});
// workaround version
if (expected != actual) {
    const string arr[] =
                {"functionX", expected, actual};
    error_msg(begin(arr), end(arr));
}
```

List initialization:

1. If you need to supply a list of element initializers for a container, such as `vector`, define and initialize an array with the same elements and then initialize the container by copying the elements from the array:

   ```
   // desired initialization
   vector<int> v{1,2,3,4,5,6,7};
   // equivalent effect
   const int temp[] = {1,2,3,4,5,6,7};
   vector<int> v(begin(temp), end(temp));
   ```

2. If you need to list initialize a return value, use a local variable of the return type to hold whatever values you want to return and then return that local variable. For example, we can rewrite the program on page 226 as follows:

   ```
   vector<string> retVals; // local variable that we'll return
   if (expected.empty())
       return retVals;  // return an empty vector
   else if (expected == actual) {
       retVals.push_back("functionX"); // build the vector
       retVals.push_back("okay");
       return retVals;                 // and return it
   } else // ...
   ```

3. There is no substitute for using curly braces to initialize a variable of built-in type. You must omit the curly braces, in which case narrowing conversions will be allowed.

4. To pass arguments to use to construct an object of class type, use parentheses rather than curly braces:

   ```
   vector<string> v1 = {10, "hi"}; // ten elements each has value hi
   vector<string> v2(10, "hi");    // equivalent declaration
   ```

   As described on page 99, if you're using curly braces to provide element initializers, you cannot substitute parentheses, but should use the strategy described in step 1 above.

`lround` function `Version_test.h` provides a definition for `lround`. This function can be simulated as follows:

```
long n = lround(d); // original version
long n = long( (d>=0) ? d + 0.5 : d - 0.5 ); // workaround
```

`noexcept` exception specification Nonthrowing functions can be designated as `throw()` in place of `noexcept` without an operand. However, there may be subtle differences in what happens if a function designated as `throw()` (rather than as `noexcept()` throws. There is no direct substitute for a `noexcept` specification that takes an operand.

`noexcept` operator There is no direct subsitute for the `noexcept` operator.

Reference qualified member functions There is no direct workaround for this feature. Because reference qualifiers are generally used as an optimization to trigger move semantics, few programs *require* this feature.

`sizeof` data member Instead of taking the size directly from the class, e.g. `Sales_data::revenue`, take the size from an object of the class type, e.g. `Sales_data().revenue`. Because the operand of `sizeof` is not evaluated, this technique does not incur the time overhead that would be needed to construct the class object.

Template default arguments for function templates Define a set of overloaded templates. For example:

```
template <typename T, typename F> //  user supplied comparison
int compare(const T &v1, const T &v2, F f);
template <typename T>  //  use less<T>
int compare(const T &v1, const T &v2);
```

Template type alias There is no direct workaround for this feature.

`unions` with class-type members that have constructors or copy-control members. There is no direct workaround for this missing feature.

Variadic templates There is no direct substitute for variadic templates. Some applications can simulate variadic templates by defining a collection of overloaded functions taking different numbers of parameters. Whether this approach works depends on the details of the application.