C++ Primer, Fifth Edition

Workarounds for C++11 Features
not Implemented prior to Visual Studio 2012

Barbara E. Moo
bmoo@att.net

August 16, 2012

The programs in *C++ Primer* use various features that were introduced into C++ as part of the 2011 ISO Standard. In many cases (e.g., `auto` and `decltype`, list initialization, and the range `for` statement) these new features represent easier ways of doing things that can be done using pre-C++11 features. In other cases (e.g., variadic templates, rvalue references, and move operations) there is no simple workaround.

This document outlines the C++11 features used in the distributed code, and, where possible, suggests workarounds. The code in this distribution uses these workarounds to replace C++11 features. We tested this code using the Visual Studio 2010 compiler, but earlier versions of the compiler should be able to compile this code as well.

A Note about C++11 Library Facilities Some of the C++11 library facilities were first defined as part of a draft standard known as `TR1`, which stands for *Technical Report 1*. The `TR1` facilities include smart pointers (Chapter 12), the unordered containers (Chapter 11), `tuple` (Chapter 17), regular expressions (Chapter 17), random numbers (Chapter 17), `function` (Chapter 14), and `bind` (Chapter 10). Microsoft introduced support for some of the TR1 libraries in Visual Studio 2008, which was released in November, 2007.

This code assumes that these libraries are supported by your compiler. For most of these facilities there is no obvious substitute, so if you are using an earlier compiler, then you will not be able to compile the code that uses these libraries. The files that use `TR1` libraries are:

```
10\bind2.cc          11\unorderedWordCount.cc  12\allocSP.cc
12\TextQuery.cc      12\unique.cc              12\UP.cc
12\weak.cc           12\QueryResult.h          12\StrBlob.h
12\TextQuery.h       14\calc.cc                15\Basket.cc
15\Basket_main.cc    15\Query.cc               15\vecBasket.cc
15\Basket.h          15\Query.h                16\multiset.cc
16\Sales_data.cc     16\SP.cc                  16\Blob.h
16\Sales_data.h      17\findbook.cc            17\game.cc
17\normalInts.cc     17\rand1.cc               17\rand2.cc
17\rand4.cc          17\rand6.cc               17\seed.cc
17\tuple.cc          17\ourRandom.h            19\calc.cc
19\memFN-bind.cc
```

C++11 Features and Workarounds

`auto` and `decltype` Replace these type specifiers by the actual type used. Note that the Visual Studio 2010 compiler and GCC 4.4 releases support `auto`

and `decltype`. The MS compiler turns on these features by default. To use them with GCC, you must specify `-std=c++0x` as a compiler option to turn on C++11 features.

`begin` and `end` library functions Assuming `arr` is the name of a built-in array, you can obtain the pointers corresponding to those returned by `begin` and `end` as follows:

```
T *b = begin(arr), *e = end(arr);  // C++11 code
T *b = arr, *e = arr + sizeof(arr)/sizeof(*arr); // workaround
```

`cbegin` and `cend` container members Use `begin` and `end` instead and declare variables that hold the results using the container's `const_iterator` type. For example, assuming `vi` is a `vector<int>`:

```
auto it = vi.cbegin(); // C++11 code

// comparable pre C++11 code
vector<int>::const_iterator it = vi.begin();
```

`constexpr` variables Use `const` instead. Note that 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 that 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 pre C++11 compilers don't allow in-class initializers, it is important to remember that the default constructor should explicitly initialize every member of built-in type.

Containers of containers When the element type of a container is itself a container you must separate the closing angle brackets by a space, (e.g. `> >` not `>>`):

```
vector<vector<int>> vi;  // ok under C++11
vector<vector<int> > vi; // space required in pre C++11 compilers
```

`= 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__` There is no direct substitute for this feature.

`forward` There is no direct substitute in pre C++11.

`function`, `bind`, `mem_fn` See above; these facilities are part of the `TR1` library.

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:

```
// C++11 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. As noted above, you can use `sizeof` to calculate these pointers:

```
// C++11 code
if (expected != actual)
    error_msg({"functionX", expected, actual});
// workaround version
if (expected != actual) {
    const string arr[] =
              {"functionX", expected, actual};
    error_msg(arr, arr + sizeof(arr)/sizeof(*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(temp, temp + sizeof(temp)/sizeof(*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 directly initialize (i.e., without an equal sign) a variable of built-in type. You must omit the curly braces:

   ```
   int i{0};     // ok in C++11; narrowing conversions not allowed
   int i(0);     // ok in pre C++11 and C++11, narrowing conversions allowed
   int i = {0}; // ok in pre C++11 and C++11, narrowing conversions allowed
   ```

   Note that pre C++11 forms of initialization 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. In this case, you can use a strategy as described in step 1 above.

Lambda expressions Lambdas are a short-hand way of defining function object classes (see Section 14.8). You can replace a lambda by the corresponding class:

```
// lambda version: get an iterator to the first element whose size() is >= sz
void f(vector<string> words) {
    auto wc = find_if(words.begin(), words.end(),
            [sz](const string &a)
                { return a.size() >= sz; });
}
```

```
// pre C++11 version uses a function object class
class SizeComp {
public:
    SizeComp(size_t n): sz(n) { }
    bool operator()(const string &s) const
         { return s.size() >= sz; }
private:
    size_t sz;  // a data member for each variable captured by value
};

void f(vector<string> words) {
    vector<string>::iterator wc =
        find_if(words.begin(), words.end(), SizeComp(sz));
}
```

`lround` function Directory `17` contains a header named `ourRandom.h` that provides a definition for `lround`. This function can be simulated as follows:

```
long n = lround(d); // C++11 version
long n = long( (d>=0) ? d + 0.5 : d - 0.5 ); // workaround
```

`make_shared` Use `new` to initialize a `shared_ptr`:

```
auto p = make_shared(string("hi")); // C++11 code
shared_ptr<string> p(new string("hi")); // TR1 code
```

`noexcept` exception specification For nonthrowing functions, use `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.

`nullptr` Use a literal 0 or `NULL` preprocessor macro defined in `cstdlib`:

```
T *elements = nullptr;  // C++11 code
T *elements = 0;        // pre C++11 code
T *elements = NULL;     // equivalent
```

Rvalue references and `move` There is no direct workaround for these features. Because moving rather than copying an object is usually an optimization, few programs *require* these features.

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.

Regular expressions, random numbers, and `tuple` See above; these classes are part of the `TR1` library. Note that GCC does not implement the regular expression library, and there is no workaround.

Range-based `for` statement Use a traditional `for` loop:

```
string s = "some values";
for (auto c : s)     // C++11, process every character in s
    // statement

// comparable loop using a traditional for
for (string::iterator it = s.begin(); it != s.end(); ++it) {
    char c = *it;
    // statement
}
```

Scoped `enums` There is no direct substitute.

`shared_ptr` and `weak_ptr` See above; these classes are part of the `TR1` library. Note that `unique_ptr` is not part of the `TR1` library.

`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.

`string` numeric conversions In both the GCC and MS distributions, directory `9` contains a header named `StrConvs.h` that implements `to_string`. For the GCC distribution that header also implements `stod`. The MS compiler includes `stod` as part of its `TR1` support.

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.

Trailing return type There is no direct substitute.

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

Unordered containers and `hash<T>` See above; these classes are part of the `TR1` library.

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.