IBM XL C/C++ Compiler for AIX

Benefits of C++11 (formerly C++0x) – Part 2

By: Nemanja Ivanovic
Hubert S. Tong

Level: Intermediate

May 2012

Contents

About this series

Walk through this scenario and others online as part of the IBM XL C/C++ Compiler for AIX.

About this Tutorial

This tutorial introduces a few of the key features of the C++11 (ISO/IEC 14882:2011) standard and how you can make the most of them with the IBM XL C/C++ Compiler V12.1 for AIX.

Objectives

- Use the knowledge of C++11 features presented in Part 1 of this demo to implement a capture of up to 5 elements

- Total time: 30 minutes

Prerequisites

- Basic Unix skills
- Basic Source code compile and build experience
- Basic knowledge of C++ programming language
- Basic knowledge of C++ template metaprogramming

System Requirements

http://www.ibm.com/software/awdtools/xlcpp/aix/sysreq/

Glossary

IBM XL C/C++ Compiler: IBM® XL C and C++ compilers offer advanced compiler and optimization technologies and are built on a common code base for easier porting of your applications between platforms. They comply with the latest C/C++ international standards and industry specifications and support a large array of common language features.

Introduction

This tutorial is designed to cover the basic concepts behind each of the forementioned features of the C++ language in order to show the reader the rationale behind the feature and what its benefits are. Once these basic concepts are covered, they are put together into a small library that provides a clean interface for functionality that would be messy at best without the use of the features. Furthermore, the library is implemented with only standard C++ and is therefore fully portable across platforms and even compliant compilers. Readers that are familiar with individual features are encouraged to skip to the last portion of this tutorial in which a detailed explanation of the library and its interface is provided.

Compiler Options Needed

In order to successfully complete this exercise, the reader needs to understand the following compiler options:

- `-qlanglvl=extended0x` (enables the XLC/C++ implementation of C++0x)
- `-qattr=full` (produces a listing that includes the attribute component, the listing file has the same name as the source file but with a `.lst` extension)
- `-o` (specifies the name to give the resulting executable)
- `-D` (defines a macro whose name is provided immediately following the option without any spaces)
- `-q64` (produces a 64-bit object file or executable)

Start the Terminal Emulator to AIX System

Figure 1 Get Started

Double click the “Launch AIX” icon on the desktop (See Figure 1) to start the character terminal to AIX system.

Get Started with Debugging Optimized Code

Successful login will result with user being presented with a menu of demo hosted on the server. Type 23 and press Enter to select “Benefits for C++11 – Part 2” demo.

Figure 2 Demo Prepared

```
################################################################################

       Thank you for waiting, setup is now ready

       IBM XL C/C++ compiler is installed at /usr/vac

       Note: Starting another command window will redo the demo setup
             of your environment. This will result in loss of any
             work done in your home directory.

################################################################################

$
```

On the terminal window you will see important information and directory path to compiler install directory (See Figure 3 Demo Prepared *oval red*).

Note: Starting another command window will start the demonstration setup of your environment. This will result in loss of any work done in your home directory (See Figure 2 Demo Prepared *rectangle red*). This will impact any progress you have made on demo steps going forward.

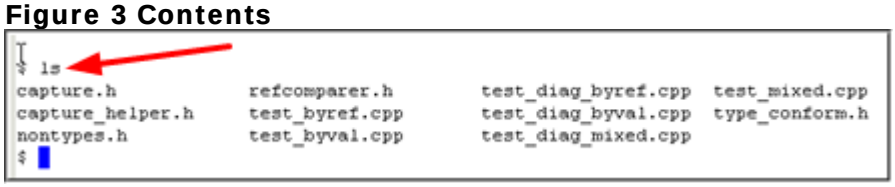
This demo does not require more than one terminal window. However, if you prefer more than one terminal window then you may open them before going forward.

Terminal window is now ready for commands (See Figure 2 Demo Prepared *arrow*). Your home directory contains necessary source code to perform the tutorial. Type ls command to see the directory content (See Figure 3 Contents).

Command:

```
ls
```

Figure 3 Contents



Using C++11 Features in a Mini-Library

Now that we have covered the above features, let's put everything together into a small library that makes use of each of the feature.
A few notes about the library:

- It implements a generic class that captures an arbitrary set of arguments by reference, by value or by a mixture of reference and value arguments
- A function pointer can be passed to the object's apply() member function which will execute the passed function on its elements (the captured arguments)
- Objects of the class are typically acquired from helper functions and automatic type deduction is typically used rather than explicitly specifying the type
- Arguments that are lvalues are captured by reference and arguments that are rvalues are captured by value
- If an lvalue needs to be captured by value, a helper function is provided to convert the lvalue to an rvalue (i.e. create a copy of the lvalue)
- There are some limitations built into the library that a subsequent tutorial will eliminate as it expands the functionality and cleans up the code. These limitations are perhaps somewhat artificial and unnecessary, but are meant to illustrate a “first-attempt” at writing such a library.
    - There is a restriction that the apply() member function can only work on captures with up to 5 elements
    - Only function pointers can be passed to apply() - no support for other callable objects
    - The apply() function makes very poor use of variadic templates and as a result, the code is hard to understand, hard to maintain and is prone to bugs
    - Curiously enough, if a function passed to apply() has a more cv-qualified parameter than the corresponding capture element, the call is invalid
    - The print() function also has an unnecessary limitation that it will only print the first 5 elements of the capture
    - Although a capture can be created with any number of elements, access is provided to only the first 5 elements
- Three functional and three diagnostic test cases are provided that illustrate valid and invalid uses of the library

The Library

Perhaps the best way to understand how the library works is by looking at the code one functional unit at a time. We will start by looking at the helper functions, then by looking at the class definition for capture and as needed, we will look at various bits of code that help the implementation.

Interface With The World: capture_helper.h

This header provides just four functions within namespace cptr. These functions create a capture by reference, by value or a mixed one (where some arguments are caught by reference and others by value). They are really meant as the only way for a user to interface with the library. The fourth function allows the user to capture an lvalue by value (i.e. it creates a copy of the lvalue).
Interesting aspects

- The make_capture functions are variadic templates so they can take an arbitrary number of arguments
- These functions' parameter lists include template parameter pack expansions where each parameter is an rvalue reference to the parameter type

- When an lvalue argument of type T is provided, argument deduction will ensure that the respective template argument type is T& and the argument is taken by reference
- When an rvalue argument of type T is provided, argument deduction will deduce the respective template argument type to T and the argument is taken by rvalue reference
- The function will then create a capture object with the deduced template arguments and return it
- The rvalueize function takes an argument by reference and returns a copy of it

The Implementation: capture.h

Since this file contains the bulk of the implementation of the library, we will analyze the code within it in a fair bit of detail.

The Cptr_Type enumeration

Since we want to limit the possible capture types to either “byref”, “byval” and “mixed” to represent capturing arguments by reference, value and mixed respectively, that is encoded in the enumeration. However, we wanted to eliminate meaningless expressions involving enumerators of this enumeration (such as comparisons to values of other types, etc.) as well as to avoid polluting the namespace with what might be common identifiers (byref, byval, mixed). In order to achieve those goals, a scoped enumeration is used, therefore making meaningless expressions invalid as well as requiring that enumerators be qualified.

The Main capture Class Template

The class template has two non-type parameters that must be specified for each instantiation along with a type parameter pack that can contain any number of type arguments. The purpose of the first non-type parameter is to tell the capture how it is to capture its arguments (value/reference/mixed) and the second one is meant to indicate whether one of the arguments already seen is an lvalue. Since at initial instantiation, no previous arguments have been seen, the helper functions instantiate the class template with false. However, as we will see later, as the pack is unrolled, this may change to true.
In order to allow a specialization for captures that have one or more elements as well as a specialization for empty captures, the main template is defined to take any number of template arguments. Finally, the main class template is not meant to ever be instantiated – only the specializations can be instantiated. Therefore, all there is in the class body is a static assertion that will always be thrown if the template is instantiated.

The Empty capture Class Template

We now look at the empty capture class since it is very simple. It exposes the following boolean enumerators:

- some_byref: whether or not any of the arguments encountered while unwinding the template argument pack are reference types
- all_byref: whether the first argument along with the rest of the arguments are reference types (since there are no arguments, this can be set to true)
- iam_byref: whether or not the current argument is a reference type

It also exposes member functions to access the first 5 elements as well as to print them. Of course, there are no elements in the empty capture, so the element access functions return an object of the type “notype” (more on this later) and the print function prints nothing. Further to the above, the capture exposes its size (set to zero since this is an empty capture).

The General capture Class Template

Of course, most of the interesting aspects of the library are implemented as part of this class template specialization. The template parameters for this specialization include the same two non-type parameters along with at least one type parameter.
Internal Data:

- The private data members _data and _cptr contain the first argument and the remaining arguments of the capture respectively
    - Notice that the type of _data is the first template parameter and the type of _cptr is a capture with all arguments but the first one
    - Also, notice that the “lvalue_found” non-type template argument for _cptr is true if it is true for the enclosing capture or if the first argument is an lvalue. Whether the first argument is an lvalue is determined by whether its type is a reference type. This is so because argument deduction in the helper function will deduce an argument type of an lvalue argument to be an lvalue reference type.
- The boolean constants are used to enforce usage constraints for the capture
    - At first, you will notice that the names of boolean constants byval, byref and mixed are the same as the enumerators of Cptr_Type without name clashes
    - If the capture is created with type set to byref, a static assertion ensures that all_lvalues is true (i.e. all arguments passed in are lvalues)
    - On the other hand, if the capture is created with type set to byval, a static assertion ensures that no_lvalues is true (i.e. no arguments passed in are lvalues)
    - Finally, if the type of the capture is mixed, a static assertion ensures that some of the arguments passed in are lvalues and some are lvalues
- Finally, there are some typedefs for easier access to the return types of element access functions

Publicly exposed members

- Similarly to the empty capture, constants describing the capture are exposed (i.e. whether the current/any/all element(s) are captured by value or reference)
- The size is based on the length of the parameter pack. The new operator sizeof...() that takes a parameter pack exists so that the length of the parameter pack can be acquired.
- Also similarly to the empty pack, element access functions are provided that return the element by reference. There are only 5 such functions and logically enough, the N-th element access function of the current capture returns the (N-1)-th element of its capture data member
- The print() function simply emits the first 5 elements to stdout. Notice that it only attempts to print an element if it exist (i.e. it is a real type).
- Only one constructor is exposed and takes all the elements of the capture. Notice that each argument is taken by reference if its corresponding type is a reference type and by value if its corresponding type is a non-reference type. This could be difficult for a user to reliably get right which is why the helper functions take care of this through argument deduction on their arguments.
- Finally, we get to the most interesting, yet most problematic function: apply(). It is a variadic function template that takes one argument – a function pointer. The template arguments are deduced from the parameters of the passed function. Here is how this function achieves its goal:
    - First the number of parameters the passed function takes is checked to ensure it is the same as the number of elements in the capture
    - Next, there are 10 typedef declarations that use a helper template to extract the types of the template arguments at positions 1-5 for both the passed function and the capture
    - Now there are 5 static assertions that ensure that the elements of the capture conform to the parameters of the passed function according to some sort of rules encapsulated in a helper template
    - Finally, perhaps the most problematic aspect of this function
        - Five null function pointers are created with parameter lists acquired from the elements of the capture

        - A switch statement on the size of the capture selects one of those to initialize with the passed function pointer using reinterpret_cast
        - Finally, it returns the result of calling that function on the capture elements
    - The important thing to notice there is that the compiler already has all the capabilities of this function built-in (it can call functions and check the number and types of parameters and arguments)

Helper Header: nontypes.h

This header provides a definition for a class used to denote that an element access function was called that is to return an element passed the end of the capture. For example, calling fourth() on a capture with three elements will return an object of this placeholder type. This is a very simple class that has a private constructor and a public static member function that returns a singleton object of the type.
In addition to defining this placeholder type, a template is provided for checking whether a type is a real type or this placeholder.

Helper Header: refcomparer.h

Simply enough, this file provides templates used to check if a type is a reference and if it is an rvalue reference as well as a template to strip the reference from a type if the second argument is a reference type. The purpose of the match_references template is to ensure that when the reinterpret_cast is done on the function pointer in apply(), the pass-by-value vs. pass-by-reference information in the passed function is not lost.

Helper Header: typeconform.h

This header provides two templates that deal with template argument types. The first one (conform) encapsulates the conformance rules for capture element types and passed function parameter types. You will notice that the rules allowing the function parameter type to be more cv-qualified than the capture element type were missed. The other template (positional_arg) provides access to the types of the argument at the specified position. It works by unwinding its parameter pack and stopping when either the last element is encountered or when the current element is the desired element. In order for it to work correctly, it must be instantiated by the user with the current parameter set to 1 and the desired parameter set to the sought-after element.

Test Cases

Now we turn to the test cases that use the library. There are three test cases that make valid use of the library and three that make invalid use of it. The code for each of the test cases (as well as the library) resides in the directory where all of the files are found and will not be shown in this document.

Test Case test_byref.cpp

This test case starts by creating a capture by reference. It does this by calling the helper function called make_capture_byref() and initializing a variable whose type is automatically deduced. The capture object created captures all of the passed variables by reference. In order to demonstrate this, the test case also defines a function called increment() that takes its first two arguments by reference and the last one by value and it increments all of its parameters. Calling the apply function of the capture with a pointer to this increment() function is shown to have the same results as calling it on the variables themselves.

The test case then illustrates wrapping the make_capture_byref() function by defining a rather contrived example that makes an adjustment to its argument if it does not conform to some requirements and then creates the capture by calling the make_capture_byref() function. In this particular example, the wrapper takes a container and the volume of the container that is occupied. Then if the volume occupied is greater than the actual volume of the container, the occupied volume argument is adjusted.
Finally the test case illustrates capturing lvalue arguments by value through using the appropriate make function and the rvalueize() function to convert the lvalues to rvalues. Notice that when modifications are made to the elements of a capture made by reference, they are reflected in the original objects.

Test Case test_byval.cpp

This test case illustrates creating a capture by value rather than by reference. Since capture objects that capture elements by value must be created with rvalues, the rvaluize() helper function is used to capture current values of objects that are lvalues. To illustrate that the arguments are indeed captured by value and not by reference, a function is applied to the capture that takes some of its arguments by reference and modifies them. This of course modifies the capture elements, but not the original lvalues that the capture was created from.

Test Case test_mixed.cpp

Finally, this test case shows how to create a capture with some arguments captured by reference and others by value. All arguments passed in as lvalues are captured by reference and those passed in as rvalues are captured by value. As always, applying a function to the capture can modify its elements, but only modifying those that are captured by reference modify the external state.

Diagnostic Test Cases:

test_diag_byval.cpp, test_diag_byref.cpp, test_diag_byref.cpp

These test cases include invalid uses of the capture class. They are meant to be compiled with the respective macros enabled to trigger one of the 5 messages. To define the macros, we use the -D compiler option. Since the capture class makes use of static assertions to provide more directed messages to users, these diagnostic test cases will trigger their emission.

```
$ xlC -qlanglvl=extended0x -o no_macros test_diag_byval.cpp # Without macros, successful compilation and execution
$ ./no_macros
Before applying any transformations to the capture, here's what it looks like:
1 10 100 1000 55.7
Values of i (1), j(1) and k(1).

$ xlC -qlanglvl=extended0x -DTEST1 -o macro1 test_diag_byval.cpp # Function passed to apply() has too few arguments
"capture.h", line 84.13: 1540-5220 (S) "The number of parameters that the function takes must be the same as the number of elements in the capture."

$ xlC -qlanglvl=extended0x -DTEST2 -o macro2 test_diag_byval.cpp # Function passed to apply() has too many arguments
"capture.h", line 84.13: 1540-5220 (S) "The number of parameters that the function takes must be the same as the number of elements in the capture."

$ xlC -qlanglvl=extended0x -DTEST3 -o macro3 test_diag_byval.cpp # Parameter 2 of passed function does not conform
"capture.h", line 104.13: 1540-5220 (S) "Parameter 2 of the passed function does not conform to the second element in the capture."

$ xlC -qlanglvl=extended0x -DTEST4 -o macro4 test_diag_byval.cpp # Parameter 3 of passed function does not conform
"capture.h", line 106.13: 1540-5220 (S) "Parameter 3 of the passed function does not conform to the third element in the capture."

$ xlC -qlanglvl=extended0x -DTEST5 -o macro5 test_diag_byval.cpp # Attempt to pass an lvalue to make_capture_byval()
"capture.h", line 59.5: 1540-5220 (S) "In order to instantiate a capture with its ref-type field set to byval, all elements must be rvalues."
```

Diagnostic Output 1 (informational messages omitted for brevity)

Diagnostic Output 2 (informational messages omitted for brevity)

```
$ xlC -qlanglvl=extended0x -o no_macros test_diag_byref.cpp # Without macros, successful compilation and execution
$ ./no_macros
Before applying any transformations to the capture, here's what it looks like:
1 1 1 1 55.7
Values of i (1), j(1) and k(1).

After applying the increment function, here's what the capture looks like:
2 2 1 2 55.7
Values of i (2), j(2) and k(1).

$ xlC -qlanglvl=extended0x -DTEST1 -o macro1 test_diag_byref.cpp # Function passed to apply() has too few arguments
"capture.h", line 84.13: 1540-5220 (S) "The number of parameters that the function takes must be the same as the number of elements in the capture."

$ xlC -qlanglvl=extended0x -DTEST2 -o macro2 test_diag_byref.cpp # Function passed to apply() has too many arguments
"capture.h", line 84.13: 1540-5220 (S) "The number of parameters that the function takes must be the same as the number of elements in the capture."

$ xlC -qlanglvl=extended0x -DTEST3 -o macro3 test_diag_byref.cpp # Parameter 2 of passed function does not conform
"capture.h", line 104.13: 1540-5220 (S) "Parameter 2 of the passed function does not conform to the second element in the capture."

$ xlC -qlanglvl=extended0x -DTEST4 -o macro4 test_diag_byref.cpp # Parameter 3 of passed function does not conform
"capture.h", line 106.13: 1540-5220 (S) "Parameter 3 of the passed function does not conform to the third element in the capture."

$ xlC -qlanglvl=extended0x -DTEST5 -o macro5 test_diag_byref.cpp # Attempt to pass an rvalue to make_capture_byref()
"capture.h", line 56.5: 1540-5220 (S) "In order to instantiate a capture with its ref-type field set to byref, all elements must be lvalues."
```

```
$ xlC -qlanglvl=extended0x -o no_macros test_diag_mixed.cpp # Without macros, successful compilation and execution
$ ./no_macros
Before applying any transformations to the capture, here's what it looks like:
1 5 1 1 5
Values of i (1), j(1) and k(1).

After applying the increment function, here's what the capture looks like:
2 6 1 2 5
Values of i (2), j(1) and k(1).

$ xlC -qlanglvl=extended0x -DTEST1 -o no_macros test_diag_mixed.cpp # All rvalues
"capture.h", line 57.5: 1540-5220 (S) "In order to instantiate a capture with its ref-type field set to mixed, some
     elements must be lvalues and some must be rvalues."

$ xlC -qlanglvl=extended0x -DTEST2 -o no_macros test_diag_mixed.cpp # All lvalues
"capture.h", line 57.5: 1540-5220 (S) "In order to instantiate a capture with its ref-type field set to mixed, some
     elements must be lvalues and some must be rvalues."
```

Diagnostic Output 3 (informational messages omitted for brevity)

Future Direction – Removing Limitations

The limitations built into this library are by no means fundamental to the respective C++11 features, but are merely meant to illustrate how someone learning the new language features may incrementally develop and improve such a library. In a follow-up to this demonstration, we will make better use of language features to remove the above limitations.

- The apply() function
  - Essentially, all the logic within this function that checks the types and numbers of parameters/elements is not needed at all – the compiler already does this on every function call
  - The reinterpret_cast's are also an indication that the library is trying to do what the compiler can already do
  - Finally, it is because we are doing all of this manually that we can only accept function pointers rather than arbitrary callable objects
  - In order to remove all of the above limitations, the apply function will be modified as follows:
    - It will be a regular template (non-variadic) that takes one parameter of arbitrary type which is meant to be the callable object
    - An auxiliary facility will be provided that will recursively unwind the capture to build up its arguments, while keeping the callable object and once the last element of the capture is reached, it will call the callable object with all of its arguments

        - At this point, if overload resolution cannot find a suitable function to call, the compiler will emit errors that the static_assert declarations were trying to mimic
- The syntax provided to access the first five elements is clean, intuitive and concise but some means of accessing the rest of the capture elements needs to be provided
    - The likely solution for this will be a parameterized function element_at whose non-type template argument will specify the element position and it will use an auxiliary template such as positional_arg
- The print() function will be modified to print arbitrary numbers of elements by simply printing its own element and then invoking the print() function of the enclosing class' capture data member

In this exercise you learned how to:

- Use IBM XL C/C++ compiler for AIX to build source code that includes C++11 features
- The basics about a number of useful C++11 features

Conclusion

This tutorial has provided an introduction to C++11 features currently available in version 12.1 of the XLC/C++ compiler. The reader is invited to explore the use of these features further and to exploit them for making their code:

- Perhaps more robust (using static_assert)
- Easier to read and maintain (perhaps using decltype, automatic type deduction and trailing returns)
- Use more versatile generics (with variadic template)
- Faster (implementing move construction and assignment to avoid copies where unnecessary)
- More type safe when using enumerations (scoped enumerations)
- Use more transparent versioning of libraries it provides (with inline namespaces)

Trademarks

Resources

XL C/C++ for AIX, V12.1 Compiler Information Center:
http://pic.dhe.ibm.com/infocenter/comphelp/v121v141/index.jsp

Community Cafe Articles
What is new in XL C/C++ V12.1
http://www.ibm.com/developerworks/rational/library/whats-new-XL-C-Cpp.html

Variadic templates proposal
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2080.pdf

Rvalue references proposal
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2027.html