IBM XL C/C++ Compiler for AIX

Benefits of C++11 (formerly C++0x) – Part 1

By: Nemanja Ivanovic
Hubert S. Tong

Level: Intermediate

May 2012

Contents

Before you start

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

- Explain the following C++11 features and show their use in short code snippets
1. Variadic templates
2. Inline namespaces
3. Automatic type deduction
4. Trailing return types
5. Rvalue references (and reference collapsing)
6. decltype
7. static assertions
8. Scoped enumerations
- Put all of this together to implement a capture of up to 5 elements (Part 2)
- Refine the capture to an arbitrary number of elements (Part 3)

- Total time: 45 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

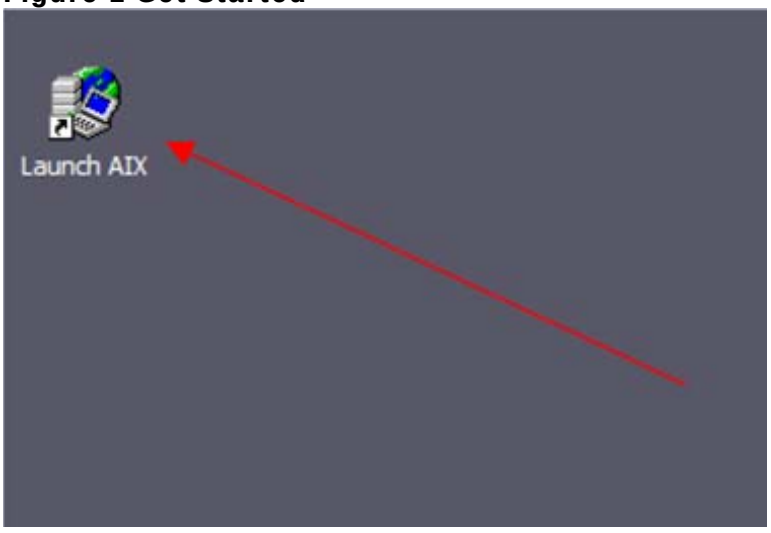
In order to successfully complete this exercise, the reader needs to understand the following compiler options:

- `-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)

Getting Started

Start the Terminal Emulator to AIX System

Figure 1 Get Started

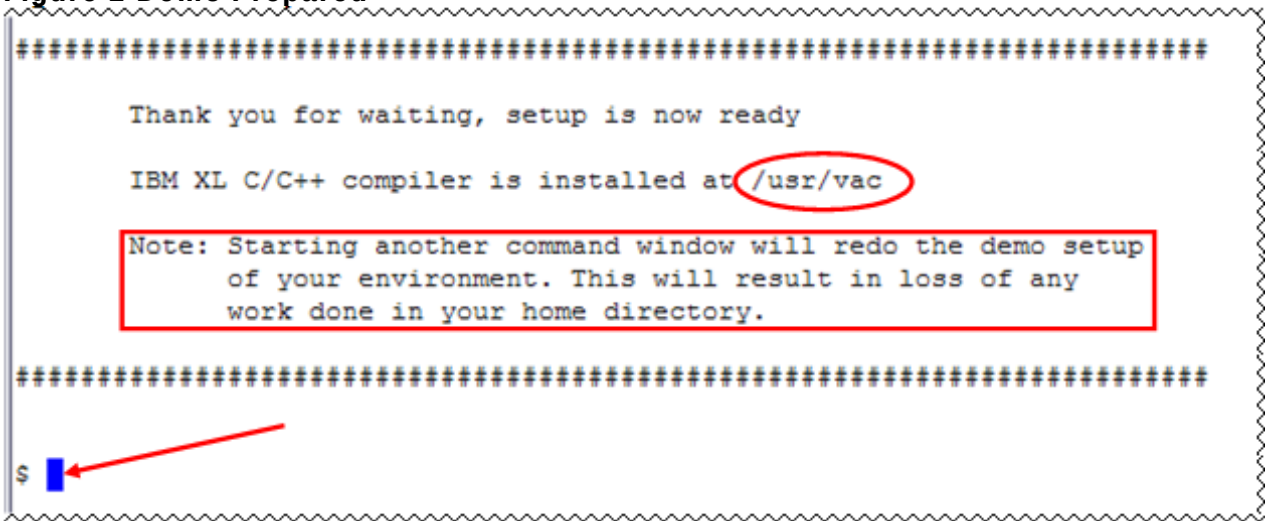


Double click the “Launch AIX” icon on the desktop (See Figure 1) to start the character terminal to AIX system.

Get Started with Debugging Optimized Code

Successful login will result with user presented with a menu of demo hosted on the server. Type 22 and press Enter to select “Benefits of C++11 – Part 1” demo.

Figure 2 Demo Prepared



On the terminal window you will see important information and directory path to compiler install directory (See Figure 2 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
```

```
$ ls
ad_limitation.cpp   packexpand.cpp      scoped_enum.cpp     trailing_bad.cpp
decltype1.cpp       ref_collapse.cpp    static_assert1.cpp  variadic1.cpp
decltype2.cpp       rval1.cpp           static_assert2.cpp
inline_nspaces.cpp  rval2.cpp           trailing.cpp
```

Figure 3 Contents

Introducing C++11 Features

Static Assertions

Simply put, this feature allows the user to put compile-time assertions into their code. The compiler will evaluate the assertion and emit the required message if the assertion fails.

Code Example 1:

```
$ cat static_assert1.cpp

int main(void)
{
    return sizeof(void*);
}

static_assert(sizeof(void*) == 8, "This program must be compiled in 64-bit mode.");

$ xlC -qlanglvl=extended0x -o sa1 static_assert1.cpp
"static_assert1.cpp", line 6.1: 1540-5220 (S) "This program must be compiled in 64-bit mode."

$ xlC -qlanglvl=extended0x -q64 -o sa1 static_assert1.cpp

$ ./sa1
$ echo $?
8
```

The above example shows how a static assertion can be made. The particular example requires the size of a pointer to be 8 bytes and provides a message to be emitted if this isn't the case. It also shows that when compiled in 64-bit mode (with the `-q64` option), the assertion does not fail and there are no messages emitted.

Code Example 2:

```
$ cat static_assert2.cpp

template <class A> struct is_pointer
{
    static const bool value = false;
};
template <class A> struct is_pointer<A*>
{
    static const bool value = true;
};

template <class C> void f(C c)
{
    static_assert(!is_pointer<C>::value, "This function must be called with a non-pointer argument.");
    // ... do something ...
}

int main(void)
{
    int i = 10;
    f(i);
#if TRIGGER
    f(&i);
#endif
}

$ xlC -qlanglvl=extended0x -DTRIGGER -o sa2 static_assert2.cpp
"static_assert2.cpp", line 12.5: 1540-5220 (S) "This function must be called with a non-pointer argument."
"static_assert2.cpp", line 10.25: 1540-0700 (I) The previous message was produced while processing "f<int *>(int *)".
"static_assert2.cpp", line 21.5: 1540-0700 (I) The previous message was produced while processing "main()".

$ xlC -qlanglvl=extended0x -o sa2 static_assert2.cpp # the static assert was not triggered with the non-pointer call
```

The above example shows the use of the feature to make assertions about template arguments. Since template metaprogramming provides an interface for compile-time computation, static assertions are very useful for providing error-checking for template instantiations, specializations, etc.

Note: the definition of `is_pointer` class template does not have specializatons to handle cv-qualified pointers (such as `int *const`), but the object of the exercise is to illustrate static assertions rather than a fully robust implementation of `is_pointer`.

decltype

This feature allows the user to acquire the static type of an object or an expression. There are instances in which deducing the type of an expression is difficult for the user and it may even be impossible in a general sense. When the type of such an expression is required, this feature is an indispensable tool.

Code Example 3:

```
//  A little bit of preprocessor mess for illustration
#if __64BIT__
#if __IBM_DFP__ && __ALTIVEC__
_Decimal64 some_var;
#elif _ARCH_COM || _ARCH_PWR7
long some_var;
#endif
#else
long long some_var;
#if __ALTIVEC__
vector bool int some_var;
#endif
#endif

//  Don't worry about deciphering the preprocessor mess - the compiler is good at this
decltype(some_var) func()
{
    decltype(some_var) var = 11;
    /* do something ... */
    return var;
}

$ xlC -qlanglvl=extended0x -qattr=full -c decltype1.cpp

$ grep ^var decltype1.lst
var                             auto long long in function func

$ xlC -qlanglvl=extended0x -qattr=full -q64 -qdfp -qaltivec -qarch=pwr7 -c decltype1.cpp

$ grep ^var decltype1.lst
var                             auto _Decimal64 in function func
```

The above example illustrates the use of **`decltype`** for simplifying code. The actual preprocessor directives at the top are not relevant, they are just meant to illustrate that sometimes the type of an object or expression is hard to determine. Without **`decltype`**, the preprocessor directives would need to be replicated wherever something is needed that has the same type as our variable "`some_var`". It is obvious from the example that specifying compiler options that change the predefined macros change the declared type of "`some_var`" which in turn changes the type returned by the **`decltype`** expression. Without wrapping function `func()` in any preprocessor directives, its return type is correct.
Important note: **`decltype`** can be used to get the **static** type of an expression. If the expression has a different run-time type, that isn't reflected by **`decltype`**. This is one of the reasons its name was chosen as such – it is the declared type of the expression.

Code Example 4:

```
$ cat decltype2.cpp

#include <iostream>
int calls = 0;
struct S
{
    int operator/ (const S &other);
    bool operator!= (const int &other) { return mem != other; }
    int mem;
};
int S::operator/ (const S &other)
{
    return mem / other.mem;
}

/*  This is a little messy in the return type since we have not yet seen any objects of types A and B
    which we can just create the expression out of. However, we simply could not do this in a fully
    generic way without decltype. We will see how this can be cleaned up when we look at trailing returns.
*/
template <class A, class B> decltype(*(A*)0 / *(B*)0) safe_divide(A a, B b)
{
    calls++;
    if (b != 0)
        return a / b;
    return 0;
}

int main(void)
{
    S s = {55}, t = {11};
    int i = 10;
    double d = 5.5;
    decltype(safe_divide(i,d)) a = safe_divide(i, d);
    std::cout << "Value of a: " << a << ". Number of calls to safe_divide: " << calls << std::endl;
    decltype(safe_divide(s, t)) b = safe_divide(s, t);
    std::cout << "Value of b: " << b << ". Number of calls to safe_divide: " << calls << std::endl;
}

$ xlC -qlanglvl=extended0x -o decltype2 decltype2.cpp

$ ./decltype2
Value of a: 1.81818. Number of calls to safe_divide: 1
Value of b: 5. Number of calls to safe_divide: 2
```

The above example shows a non-trivial use of the decltype feature. Namely, the return type of function `safe_divide()` cannot be known in general without a facility such as **`decltype`**. There are however some curious oddities in the above example. For example, the expression in **`decltype`** is very odd and messy. The reason this is so is alluded to in the comment – we have to provide an expression for **`decltype`**, but we do not have any objects of the two types. One might think that we can provide an expression such as `A() / B()` in **`decltype`**, but this would require the types `A` and `B` to be default-constructible and this wouldn't be a fully generic solution. We will see how to fix this with trailing returns. Another bit of a nuisance is the verbosity of the declarations of variables `a` and `b` in `main()`, it would be better if we could just tell the compiler more concisely to figure out the type of the variables. Please read on for the solution. The last thing to note about the above example is that the expression passed to **`decltype`** is a so-called “unevaluated operand”. What this means is that the expression will not actually be evaluated so it won't have the side-effects it would if it was actually evaluated. You can see this above since there appear to be four calls to function `safe_divide()`, but the `calls` variable is incremented only twice.

Automatic Type Deduction

Similarly to **`decltype`**, automatic type deduction allows the user to tell the compiler to deduce the type of a declared variable based on an expression. However, automatic type deduction works by providing an initializer for the variable. Then the type deduction and initialization are not split into two separate processes, but are done together. In the previous example, one could simply replace the **`decltype`** expressions in main with the word **`auto`** and the program would work the same way.
Auto type deduction and **`decltype`** are extremely useful for reducing the complexity of declarations.
Simple guidelines for their use:

- Use them whenever the type you are trying to use in a declaration is hard to figure out, unwieldy to specify or redundant; including for example, STL container iterators
- When an initialization can be done, use auto type deduction
- When an initialization cannot be done (typedef's, return types, etc.), use **`decltype`**
- IMPORTANT: Keep in mind that both features work on static types – the run time type of an expression can not be deduced by these features.

Code Example 5: Example of a limitation to compile-time types:

```
$ cat ad_limitation.cpp

struct A { };
struct B : public A { };

int main(void)
{
    A *pa = new B();
    auto wanted_B = *pa;
    auto got_B = *new B();
}

$ xlC -qlanglvl=extended0x -qattr=full ad_limitation.cpp

$ grep -E 'wanted_B|got_B' ad_limitation.lst
got_B                          auto B in function main
wanted_B                       auto A in function main
```

It is apparent from the above example that even though the variable `pa` actually points to an object of the derived type, the compiler still gives the variable `wanted_B` the type of the base class. This is simply because the variable `pa` is declared to be of type pointer-to-A (base class) so the fact that it actually points to an object of the derived type is irrelevant. Even though the variables `wanted_B` and `got_B` are initialized in a similar way, they have different types.

Trailing Return Types

This feature allows the return type of a function to appear after the parameter list, therefore allowing expressions involving parameters as part of the return type. In turn, this frees the programmer from the need to form complex and messy expressions when specifying the return type of a function. Generic functions whose return types depend on their template arguments can truly benefit from the simplified syntax provided by this feature.

```
$ cat trailing.cpp

int f(int, wchar_t);
double f(wchar_t, int);
long f(int, double);

template <class A, class B> auto forward1(A a, B b) -> decltype(f(a, b));
template <class A, class B> auto forward2(A a, B b) -> decltype(f(f(a, b), f(b, a)));

void g()
{
    forward1(55, L'a');
    forward2(55, L'a');
}

$ xlC -qlanglvl=extended0x -qattr=full -c trailing.cpp

$ grep -E 'forward1<|forward2<' trailing.lst
forward1<int,wchar_t>            extern int (int, wchar_t)
forward2<int,wchar_t>            extern long (int, wchar_t)
```

Code Example 6:

The above example shows how the trailing return types feature can be used to declare return types of functions that are dependent on function parameters in a natural way. There is no need for strange expressions involving casting a zero to a pointer to a type and so on. One important note about this feature is that the auto keyword that appears instead of a return type must be the only type-specifier that appears – the type must be fully specified in the trailing return specifier. The below example shows some examples of what isn't allowed.

Code Example 7:

```
$ cat trailing_bad.cpp

auto &f() -> int;
const auto g() -> int;
struct S
{
    auto mf1() const -> const int; // OK const member function
    auto mf2() const -> int;       // OK const member function
    const auto mf3() const -> int; // bad
};

$ xlC -qlanglvl=extended0x -c trailing_bad.cpp
"trailing_bad.cpp", line 1.6: 1540-2647 (S) Expecting single token "auto" for trailing return type placeholder.
"trailing_bad.cpp", line 2.1: 1540-2647 (S) Expecting single token "auto" for trailing return type placeholder.
"trailing_bad.cpp", line 7.5: 1540-2647 (S) Expecting single token "auto" for trailing return type placeholder.
```

The above example shows that the auto type specifier must be the only token that replaces the return type specifier – the return type must be fully specified in the trailing return.

Scoped Enumerations

Up until C++11, enumerated types had a fundamental issue – lack of type safety. There are situations in which an enumerated type is very useful, but it is difficult to prevent type safety violations.

Code Example 8:

```
$ cat -n scoped_enum.cpp

     1  //  Namespace pollution from enums
     2  enum Colour { red, green, blue, purple };
     3  enum Shape { triangle, square };
     4  #if NS
     5  enum Light { red, amber, green };
     6  #endif
     7
     8  //  Implicit conversion to int
     9  int i = green + 5; // what does green + 5 mean?
    10  void f(int i);
    11  void g()
    12  {
    13      f(red | 7);       // what does red | 7 mean?
    14      f(blue | square); // huh?
    15  }
    16
    17  //  Scoped enums to the rescue
    18  //  No namespace pollution
    19  enum class Colour_s { red_s, green_s, blue_s, purple_s };
    20  enum class Shape_s { triangle_s, square_s };
    21  enum struct Light_s { red_s, amber_s, green_s };
    22  Colour_s cs = red_s; // Doesn't exist at namespace scope
    23
    24  //  No implicit conversion to int
    25  int j = Colour_s::green_s + 5;
    26  void f1(int i);
    27  void g1()
    28  {
    29      f1(Colour_s::red_s | 7);
    30      f1(Colour_s::blue_s | Shape_s::square_s);
    31  }

$ xlC -qlanglvl=extended0x -c -DNS scoped_enum.cpp
"scoped_enum.cpp", line 5.14: 1540-0403 (S) "red" is already defined.
"scoped_enum.cpp", line 2.15: 1540-0425 (I) "red" is defined on line 2 of "scoped_enum.cpp".
"scoped_enum.cpp", line 22.15: 1540-0274 (S) The name lookup for "red_s" did not find a declaration.
"scoped_enum.cpp", line 25.27: 1540-0218 (S) The call does not match any parameter list for "operator+".
"scoped_enum.cpp", line 29.24: 1540-0218 (S) The call does not match any parameter list for "operator|".
"scoped_enum.cpp", line 30.25: 1540-0218 (S) The call does not match any parameter list for "operator|".

$ xlC -qlanglvl=extended0x -c scoped_enum.cpp
"scoped_enum.cpp", line 22.15: 1540-0274 (S) The name lookup for "red_s" did not find a declaration.
"scoped_enum.cpp", line 25.27: 1540-0218 (S) The call does not match any parameter list for "operator+".
"scoped_enum.cpp", line 29.24: 1540-0218 (S) The call does not match any parameter list for "operator|".
"scoped_enum.cpp", line 30.25: 1540-0218 (S) The call does not match any parameter list for "operator|".
```

The above example illustrates a few of the limitations of `enum`s and how scoped `enum`s remove those limitations. For example, enumerators of a scoped `enum` are declared only in the scope of that `enum` and not at in the enclosing scope. All access to enumerators of scoped `enum`s must be done by qualifying the enumerator with the name of the `enum`. Furthermore, there is no implicit conversion from a scoped `enum` to `int`. This prevents their use in expressions that are logically meaningless. Notice that there are no messages for lines 9, 13 and 14; whereas there are messages for the respective scoped `enum` uses on lines 25, 29 and

30. Further to restricting scope of visibility of enumerators, scoped `enum`s allow forward declarations as well as explicitly specifying the underlying type of an `enum`.

Variadic Templates

This feature extends the power of templates by allowing definitions of templates with a varied number of arguments. In a way, it is similar to variable arguments in C/C++ functions, but it facilitates type safety since all types are known when the template is instantiated. The best way to learn about this feature is by seeing it in action, so let's have a look at an example.

```
$ cat variadic1.cpp

#include <iostream>
#include <typeinfo>
template <class C> struct remove_reference { typedef C type; };
template <class C> struct remove_reference<C&> { typedef C type; };
template <class C> struct remove_reference<C&&> { typedef C type; };

template <class Head, class ... Tail>
struct sum {
    typedef decltype (*(typename remove_reference<Head>::type *)0 + *(typename sum<Tail...>::type *)0) type;
};
template <class Head>
struct sum<Head> { typedef typename remove_reference<Head>::type type; };

template <class Head> Head f(Head o)
{
    return o;
}

template <class Head, class ... Tail>
typename sum<Head, Tail...>::type f(Head h, Tail ... t)
{
    return h + f(t...);
}

int main(void)
{
    int i = 11;
    std::cout << "Type of f('a', 12): " << typeid(f('a', 12)).name()
              << ". Result: " << f('a', 12) << std::endl;
    std::cout << "Type of f('a', 12, 3.0): " << typeid(f('a', 12, 3.0)).name()
              << ". Result: " << f('a', 12, 3.0) << std::endl;
    std::cout << "Type of f('a', 12, 3.0, 4.8L): " << typeid(f('a', 12, 3.0, 4.8L)).name()
              << ". Result: " << f('a', 12, 3.0, 4.8L) << std::endl;
    std::cout << "Type of f('a', 12, 3.0, 4.8L, i): " << typeid(f('a', 12, 3.0, 4.8L, i)).name()
              << ". Result: " << f('a', 12, 3.0, 4.8L, i) << std::endl;
    return 0;
}

$ xlC -qlanglvl=extended0x -o variadic1 variadic1.cpp

$ ./variadic1
Type of f('a', 12): int. Result: 109
Type of f('a', 12, 3.0): double. Result: 112
Type of f('a', 12, 3.0, 4.8L): long double. Result: 116.8
Type of f('a', 12, 3.0, 4.8L, i): long double. Result: 127.8
```

Code Example 9:

The above example is rather complex, so we will analyze it one step at a time. First of all, let's look at the variadic templates in the example, starting with function `f()`. Its declaration shows the syntax for declaring variadic templates – the template parameter name is preceded by an ellipsis which indicates a “template parameter pack”. Such a pack can contain any number of type arguments at instantiation (including zero). In order to use the pack, it usually needs to be expanded (by, for example, moving the ellipsis to the right side of the pack name). When a template parameter pack is expanded into a list of function arguments,

the name declared by the expansion is a function parameter pack, which may in turn be expanded to access individual arguments. In contrast to variable function arguments from C, the type of each parameter in the pack is known and does not require casting.
Typical use of variadic templates is akin to recursion: for functions, a base overload is provided as well as a template for general overloads; for classes, a template for the general case is provided along with a specialization for the base case. The recursion-like structure is clear within the function template – there is a call to the function with the same name (of course, this isn't a recursive call, but a call to an overload with one fewer argument). Although less obvious, a similar structure exists in the class template: something is done with the first argument and another instantiation with one less argument is used.

But what of the strange structure for the return type of the function template? Would it not be easier to just use decltype and trailing returns? Something like:
`template /*...*/ auto f(Head h, Tail ... t) -> decltype(h + f(t...))`
The problem with this structure is that the template currently being defined is not part of the overload set yet, so the only function available to decltype for the call to f(t...) is the base case, so if the Tail parameter pack contains more than one element, there is simply no overload to call. That is why the workaround with a variadic class template is required. Notice that type promotion is thereby preserved.
Now we need to look at why the remove_reference template is needed. In truth, for this example, it is not needed. However, if one of the types in the parameter pack was a reference type, there would be a compile-time failure. This is simply because the decltype expression in the variadic class template would attempt to use a type that is a pointer to a reference – which of course isn't allowed. Lastly, the strange type in the final specialization of remove_reference is an rvalue reference and is covered later in this tutorial.

Let's now summarize variadic templates since this is quite a bit of information to take in from an example.

1. This feature allows definitions of templates with any number of parameters (the number is known at instantiation time, but not at definition time).
2. To declare a variadic template, prefix a template parameter name with an ellipsis (…)
3. When using a parameter pack, it typically needs to be expanded. The expansion consists of a pattern followed by an ellipsis. The following are valid expansions:
4. `template <class ... Args> struct A : public Args... //` (possibly) multiple inheritance
5. `func(static_cast<int>(args)...) //` assuming that “args” is an argument pack, it will call function “func” with all of the arguments statically cast to int
6. `func(static_cast<Args1>(args2)...) //` assuming that Args1 is a parameter pack and args2 is an argument pack corresponding to another parameter pack, will call function “func” with each argument in the list args2 converted to the corresponding type in Args1 (see example)
7. Typically, variadic templates are used by specifying a base specialization/overload and a general template that unwinds the parameter pack
8. The number of parameters in the pack is available from a `sizeof...(Pack)` expression (where Pack is a parameter pack).

Code Example 10:

```
$ cat packexpand.cpp
template <class ... Args> void tempfunc1(Args... args);

template <class ... Args1> struct A
{
    template <class ... Args2> static void tempfunc2(Args2 ... args2)
    {
        tempfunc1(static_cast<Args1>(args2)...);
    }
};

void f()
{
    A<int, double, char, int>::tempfunc2(4.0, 'a', 33, 24.00);
}

$ xlC -qlanglvl=extended0x -qattr=full -c packexpand.cpp

$ grep ^tempfunc[1-2]\< packexpand.lst
tempfunc2<double,char,int,double> static void (double, char, int, double) in class template specialization A<int,double,char,int>
tempfunc1<int,double,char,int>   extern void (int, double, char, int)
```

This example illustrates a pack expansion that has a slightly more complex form than just the name of the parameter pack followed by an ellipsis.

Inline Namespaces

This feature provides better support for versioning of libraries by allowing namespaces to be declared with the “inline” specification whereby the members of the namespace appear as if they are members of the enclosing namespace. A typical use example for this is if a library is to be shipped with two versions: the two versions can be in two separate namespaces one of which would be declared as an “inline namespace”.

```
$ cat -n inline_nspaces.cpp

     1  namespace mylib
     2  {
     3  #if NEW
     4      inline
     5  #endif
     6      namespace mynewlib { template<class C> struct some_struct { typedef double type; }; }
     7  #ifndef NEW
     8      inline
     9  #endif
    10      namespace myoldlib { template<class C> struct some_struct { typedef int type; }; }
    11
    12      /*  This can't be done with a using declaration */
    13      template<> struct some_struct<int*> { typedef void *type; };
    14  }
    15  namespace mylib
    16  {
    17      namespace mynewlib { template<class C> struct another_struct { typedef char type; }; }
    18      namespace myoldlib { template<class C> struct another_struct { typedef long type; }; }
    19
    20      template<> struct another_struct<some_struct<int>> { typedef some_struct<int>::type type; };
    21  }
    22
    23  struct A { };
    24  using namespace mylib;
    25  some_struct<int>::type f1() { return A(); }
    26  some_struct<int*>::type f2() { return A(); }
    27  another_struct<int>::type f3() { return A(); }
    28  another_struct<some_struct<int>>::type f4() { return A(); }

$ xlC -qlanglvl=extended0x -c inline_nspaces.cpp
"inline_nspaces.cpp", line 25.38: 1540-2467 (S) A return value of type "int" cannot be initialized with a prvalue of type "A".
"inline_nspaces.cpp", line 26.39: 1540-2467 (S) A return value of type "void *" cannot be initialized with a prvalue of type "A".
"inline_nspaces.cpp", line 27.41: 1540-2467 (S) A return value of type "long" cannot be initialized with a prvalue of type "A".
"inline_nspaces.cpp", line 28.54: 1540-2467 (S) A return value of type "int" cannot be initialized with a prvalue of type "A".

$ xlC -qlanglvl=extended0x -DNEW -c inline_nspaces.cpp
"inline_nspaces.cpp", line 25.38: 1540-2467 (S) A return value of type "double" cannot be initialized with a prvalue of type "A".
"inline_nspaces.cpp", line 26.39: 1540-2467 (S) A return value of type "void *" cannot be initialized with a prvalue of type "A".
"inline_nspaces.cpp", line 27.41: 1540-2467 (S) A return value of type "char" cannot be initialized with a prvalue of type "A".
"inline_nspaces.cpp", line 28.54: 1540-2467 (S) A return value of type "double" cannot be initialized with a prvalue of type "A".
```

Code Example 11

The above example illustrates the key aspects of the feature. It is clear from the example that elements of an inline namespace appear as if they are elements of the enclosing namespace. Furthermore, since the enclosed namespace is already declared to be inline, all subsequent extensions of the namespace appear as if they are members of the enclosing namespace. Unlike with using declarations, templates in an inline namespace can be explicitly specialized in the enclosing namespace. Finally, it is up to the library architect to decide which namespace is inline and therefore the default.

There is also a small bonus feature included in the above example. Perhaps you spotted that instantiating and specializing a template with a template argument no longer requires that the two “>” symbols be separated by a space. This was an unnecessary nuisance in C++2003 that has been removed in C++2011.

Rvalue References

The new C++ standard includes a new type of reference, the aptly named “rvalue reference”. This is a reference that can be bound to an rvalue (previously only **const** references could bind to rvalues, and even then, only copyable rvalues). But if there was a way of binding a reference to an rvalue before, why don't we just use that method? Why do we need another type of reference?
The problem is that the distinction of whether the rvalue to which the reference is bound is **`const`** or non-**`const`** is lost. Furthermore, there is no mechanism to ensure that a reference must bind to an rvalue. The solution is the rvalue reference. This type of reference is denoted by the “`&&`” symbol and can bind only to an rvalue. Let's explore the uses of rvalue references through some examples.

```
$ cat -n rval1.cpp
     1  #include <iostream>
     2  using std::cout;
     3  using std::endl;
     4  struct S
     5  {
     6      S(int i) : mem(i) { }
     7      operator int() { cout << "Called operator int()" << endl; return mem; }
     8      operator int&() { cout << "Called operator int&()" << endl; return mem; }
     9      operator int&&() { cout << "Called operator int&&()" << endl; return int(mem); } // Temporary rvalue
    10      operator char()  { cout << "Called operator char()" << endl; return mem; }
    11      private:
    12          int mem;
    13  };
    14  S s = 10;
    15  int i = 11;
    16
    17  /*  const references */
    18  const int &cri1 = s;  // Can bind to an lvalue or rvalue, calls operator int& (lvalue)
    19  const int &cri2 = i;  // Clearly can bind to an lvalue
    20  const int &cri3 = 10; // Clearly can bind to an rvalue
    21  const int &&cri4 = s; // Can only bind to rvalue, calls operator int&&
    22
    23  /*  non-const references */
    24  int &lri1 = s;        // Can only bind to lvalue, calls operator int&
    25  int &lri2 = i;
    26  int &&rri1 = s;       // Can only bind to rvalue, calls operator int&&
    27  int &&rri2 = 10;      // Binds to an rvalue
    28  char &&rrc1 = s;      // Binds to rvalue, calls operator char (rvalue)
    29
    30  /*  Perhaps surprising */
    31  int &lri3 = rri1;     // OK - rri1 is an lvalue even though it's an rvalue reference
    32
    33  void f(const int &, int line) { cout << "Called f(const int&) at line " << line << endl; }
    34  void f(int &, int line) { cout << "Called f(int&) at line " << line << endl; }
    35  void f(int &&, int line) { cout << "Called f(int&&) at line " << line << endl; }
    36  void f(const int &&, int line) { cout << "Called f(const int&&) at line " << line << endl; }
    37
    38  int main(void)
    39  {
    40      f(cri1, __LINE__);
    41      f(lri1, __LINE__);
    42      f(10, __LINE__);
    43
    44      cout << endl << "Perhaps some surprising results" << endl;
    45      f(cri4, __LINE__);
    46      f(rri1, __LINE__);
    47      f(s.operator int&&(), __LINE__);
    48      return 0;
    49  }
```

Code Example 12

The above example illustrates a number of rules governing rvalues, how they are used and how they bind. It is perhaps somewhat surprising that there are no ambiguities in the above code: all initializations and function calls have a best match. Let's compile and execute this program and analyze the results.
Compile and execute the above example

```
$ xlC -qlanglvl=extended0x -o rval1 rval1.cpp
"rval1.cpp", line 9.74: 1540-1103 (W) The address of a local variable or temporary is used in a return expression.

$ ./rval1
Called operator int&()
Called operator int&&()
Called operator int&()
Called operator int&&()
Called operator char()
Called f(const int&) at line 40
Called f(int&) at line 41
Called f(int&&) at line 42

Perhaps some surprising results
Called f(const int&) at line 45
Called f(int&) at line 46
Called operator int&&()
Called f(int&&) at line 47
```

- The compiler warning is emitted since we are creating a temporary in the conversion operator and returning that temporary by reference. Not a concern for this discussion.
- Rules governing rvalue references
  - An rvalue reference can only bind to an rvalue (prvalue or xvalue)
  - An rvalue reference is an lvalue
  - The result of calling a function that returns an rvalue reference is an xvalue
  - The result of calling a function that returns an lvalue reference is still an lvalue
  - The result of calling a function that returns by value is a prvalue
  - The new value categories are: xvalue (for expiring value) and prvalue (for pure rvalue).
- Output from the program
  - First line is emitted from initializing the reference on line 18. Not surprisingly, the best match for the conversion function is operator int& since the call is an lvalue
  - Second line is emitted from initializing the reference on line 21. Although both operator int() and operator int&&() are valid choices, the latter is tried first in this context
  - Line 24 causes the next line of output. There is only one valid choice of conversion function for this initialization – the only one that returns an lvalue.
  - Line 26 again causes a call to operator int&&() since it is a better choice for initializing an rvalue reference than is operator int().
  - The next line of output is caused by line 28. The only conversion function that converts struct S to char returns a prvalue and the rvalue reference binds to it
  - The next 3 lines of output are exactly what one might expect, a const lvalue argument matches the const lvalue reference parameter, a non-const lvalue argument matches the non-const lvalue reference parameter and a prvalue argument matches the rvalue reference parameter.
  - However, the following 4 function calls may be somewhat unexpected:
    - The calls on lines 45 and 46 pass rvalue reference arguments and those arguments actually match the respective lvalue reference parameter. This is so because an rvalue reference is an lvalue.
    - A seemingly analogous call on line 47 passes the result of calling a function that returns an rvalue reference but the argument matches the rvalue reference pa-

rameter. This is so because the result of the function call in the argument is an xvalue which can bind to an rvalue reference

Code Example 13: Move Constructors

```
$ cat rval2.cpp
#include <iostream>
using std::cout;
using std::endl;
struct S
{
    S(int bottom = 0, int top = 1000) : _b(bottom), _t(top), _s((top - bottom)/1000), arr(new int[1000])
    {
        cout << "Called default c'tor." << endl;
        if (_s <= 0)
            _s = 1;
        int val = _b;
        for (int i = 0; i < 1000; i++)
            arr[i] = val += _s;
    }
    S(S &&other) : _b(other._b), _t(other._t), _s(other._s)
    {
        cout << "Called move c'tor." << endl;
        arr = other.arr;
        other.arr = 0;
    }
    S(const S &other) : _b(other._b), _t(other._t), _s(other._s), arr(new int[1000])
    {
        cout << "Called copy c'tor." << endl;
        for (int i = 0; i < _t; i+=_s)
            arr[i] = other.arr[i];
    }
    ~S() { delete[] arr; }
    int *arr;
    private:
        int _b, _t, _s;
};

S &&f() { return S(); }
int main(void)
{
    cout << "Moves:" << endl;
    S s1 (0, 1000000);
    S s2(f());
    cout << endl << "Addr/value in s1.arr before move: " << s1.arr << '/' << s1.arr[0] << endl;
    S s3(static_cast<S&&>(s1));
    cout << "Addr/value in s1.arr after move: " << s1.arr << '/' << s1.arr[0] << endl;

    cout << endl << "Copies:" << endl;
    S s4;
    S s6 = s4;
}

$ xlC -qlanglvl=extended0x -o rval2 rval2.cpp

$ ./rval2
Moves:
Called default c'tor.
Called default c'tor.
Called move c'tor.

Addr/value in s1.arr before move: 20005128/1000
Called move c'tor.
Addr/value in s1.arr after move: 0/0

Copies:
Called default c'tor.
Called copy c'tor.
```

This example illustrates move construction of objects that are potentially expensive to copy. If the user does not need the original object any longer, a new object can be constructed by stealing the original one's resources. For example, the s1 object is invalidated when s3 is created by stealing its resources, making the access to s1.arr[0] after the move unsafe (dereference of a null pointer). This mechanism can obviously be used to implement move

assignment as well. The advantage of move construction and assignment is that there is no longer a need to perform expensive copy operations when swapping objects, passing temporaries around, etc.

Code Example 14: Reference Collapsing

```
$ cat ref_collapse.cpp
typedef int &R1;
typedef R1 &&R2;

typedef int &&R3;
typedef R3 &&R4;
typedef R4 &R5;

int &f1();
R2 f1();

int &&f2();
R4 f2();

int &f3();
R5 f3();

template <class C, class D> void f4(C &&c, D &&d);

void g()
{
    int i;
    f4(i, 5);
}

$ xlC -qlanglvl=extended0x -qattr=full -c ref_collapse.cpp

$ grep ^f[1-4] ref_collapse.lst
f1                              extern int &()
f2                              extern int &&()
f3                              extern int &()
f4                              extern void (C &&, D &&)
f4<int &,int>                   extern void (int &, int &&)
```

In order to make use of rvalue references, a mechanism was added for dealing with situations in which a reference to a reference (an illegal type) might arise. For example, if a typedef declaration declares a reference to an object type, a reference to that type needs to collapse to a reference to the original type. This is of particular importance with template arguments.

The rules for reference collapsing are rather simple:

- Assuming that TL is an lvalue reference to some (possibly cv-qualified) object type T (i.e. T&) and that TR is an rvalue reference to the same object type T (i.e. T&&), the following hold:
  - The type cv TL& is TL
  - The type cv TL&& is TL
  - The type cv TR& is TL
  - The type cv TR&& is TR

Note that the cv-qualification in the original reference type is preserved regardless of the cv-qualification applied on top of it.

These rules are illustrated through the typedef declarations above. The function declarations for functions f1() through f3() do not cause any re-declaration messages since the references collapse. Furthermore, the template argument for class C in function f4() is deduced to int&. Here is why:

There is a special rule which works in conjunction with reference collapsing so that function parameters of the form T &&, where T is a template type parameter, can bind to lvalues:

- The function is called with an lvalue, and the parameter type is a reference type, so it must be an lvalue reference type (int &) in the instantiation for the binding to work
- In order for C&& to collapse to int &, the template argument must be int &

What you have learned

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 the XLC/C++ compiler version 12.1. 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 and 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/n1952.html