C++0x feature support in GCC 4.5

Skill Level: Intermediate

Arpan Sen (arpansen@gmail.com)
Independent author

01 Mar 2011

If you're one of the many users of the GCC C++ compiler, you should be aware of the new features that the upcoming C++0x specification has in store and available in GCC version 4.5.

Introduction

Compiling C++0x code
You can compile C++0x code only by using the `-std=c++0x` or `-std=gnu++0x` command-line options to the g++ compiler.

The GNU Compiler Collection (GCC) is the C++ compiler to the vast majority of us and has taken the lead in supporting features from the upcoming C++0x standard (see Resources for links to more information). This article focuses on a subset of several C++0x features that GCC version 4.5 supports, including static assertions, initializer lists, type narrowing, newer semantics of the auto keyword, lambda functions, and variadic templates. Some of these features—like static assertions—first made their appearance in GCC version 4.3, while lambda functions first appeared with the 4.5 release. If you intend to be one of the early adopters of C++0x, consider getting a copy of the draft standard and download GCC 4.5 (see Resources).

Let's begin by briefly discussing parts of the C++0x standard that are not yet supported in GCC.

Static assertions

Remember those times when your portable code crashed at a customer site, because integer sizes were not the 4 bytes you assumed them to be? The `static_assert` construct in C++0x helps track precisely these kind of problems, except that they are available at compile time and immensely useful when you need to migrate sources to different platforms. Boost libraries (see Resources) have had support for static assertions for some time now, but having static assertions integrated as part of the language core (`static_assert` is a C++0x keyword) implies that headers are no longer needed (see Listing 1).

Listing 1. Learning to use static assertions

```
// no headers
// using static assertion in global scope
static_assert(sizeof(int) == 4, "Integer sizes expected to be 4");

int main()
{
    return 0;
}
```

On my 64-bit enterprise Linux® system, this assertion fails during compilation. Here's the log:

```
g++ 1.cpp --std=c++0x
1.cpp :1:1: error: static assertion failed: " Integer sizes expected to be 4"
```

You can use static assertions in global scope and inside namespaces, function bodies, or class declarations. Listing 2 provides another example where using static assertions helps prevent instantiating a templated class with certain argument types. The trick is to add a static assertion that is bound to fail in a specialized version of the class. While using static assertions, always remember that the expression being checked must be evaluable at compile time.

Listing 2. Using static assertions inside class declarations

```
// using static assertions as part of class declaration
template <typename T1, typename T2>
class T {
        T1 a;
        T2 b;
};

template <typename T1>
class T<T1, bool> {
        static_assert(sizeof(T1) == 0, "T<T1, bool> is not allowed, sorry");
};

int main( )
{
   T<float, bool> f1; // this will fail compilation
   T <float, int> i2; // fine
   …
```

```
}
```

Introducing new character types

Admittedly, C/C++ support for Unicode has always been lacking. Unicode defines character encodings in three distinct sizes—UTF-8, UTF-16, and UTF-32—while the traditional character type is 8 bits. Using `wchar_t` is not an option, because the size of `wchar_t` is defined by implementation.

C++0x introduces two new keywords—`char16_t` and `char32_t`—with guaranteed sizes (16 and 32 bits, respectively—see Listing 3) and are unsigned types. To use these new types, you must include the C++ header `cstdint`, although the standard asks you to include the header `cuchar`.

Listing 3. Using the char16_t and char32_t data types

```
#include <iostream>
#include <cstdint>  // this is the header for char16_t and char32_t
using namespace std;

int main()
{
    char16_t c = 'A';
    cout << sizeof(c) << endl;
    char32_t d = 'b';
    cout << sizeof(d) << endl;
    return 0;
}
```

Sure enough, the output from Listing 3 prints 2 and 4 on the console. Now, try defining an array of the `char32_t` type, as shown below in Listing 4.

Listing 4. Declaring and initializing char32_t array

```
#include <iostream>
#include <cstdintglt;
using namespace std;

int main()
{
    char32_t e[ ] = "Hello World";
    cout << sizeof(e) << endl;
    return 0;
}
```

Here's the compiler log:

```
g++ 1.cpp --std=c++0x
1.cpp: In function 'int main()':
1.cpp:7:8: error: int-array initialized from non-wide string
```

What's the issue here? Well, for starters, "Hello World" is a regular 8-bit-per-character string and assigning it to an array of 32-bits per character is invalid. You create string literals for these new types by prefixing `u` or `U` to the string, depending on the type. Look at Listing 5.

Listing 5. Initializing char16_t and char32_t string literals

```
#include <iostream>
#include <cstdint>
using namespace std;

int main()
{
    char16_t f[ ] = u"Hello World";  // prefix the string with u for char16_t
    cout << sizeof(f) << endl;
    char32_t e[ ] = U"Hello World"; // prefix the string with U for char32_t
    cout << sizeof(e) << endl;
    return 0;
}
```

The expected console output was `24` and `48`, and g++ did the job nicely.

Welcome to the new auto syntax: type deduction from initializer expressions

This one is a lifesaver of sorts, with the compiler deducing the proper type for a variable from its initializer expression. So, all the previous declarations, like `class1::class3::struct1::enum2 e`, are now history. The `auto` keyword has completely different semantics with C++0x. Listing 6 provides the first example.

Listing 6. Using auto for automatic type inference

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    auto *num1 = new int(7);  // type for num1 is int*
    const auto num2 = 3.1415;  // type for num2 is double

    // now for some serious business
    map<int, string> map1;
    map1.insert(make_pair<int, string> (7, "8"));
    auto mapit1 = map1.find(7);  // type for mapit1 is std::map::iterator
    cout << mapit1->second << endl;
    … // continue coding
}
```

The type for `num1` is an integer pointer; `num2` is a float and `mapit1is`

`std::map::iterator`. Multiple declarations are allowed with `auto`, but all deductions must lead to the same type. Declarations are processed left to right. Listing 7 below shows code that uses `auto` and does not work.

Listing 7. Erroneous use of the auto keyword

```
int main( )
{
   auto i = 9, j = 8.2; // error – i and j should be same type
   auto k = &k; // dumb error – can’t declare and use in initializer
   …
}
```

Here's the compiler log for Listing 7:

```
1.cpp: In function 'int main()':
1.cpp:3:8: error: inconsistent deduction for 'auto': 'int' and then 'double'
1.cpp:4:8: error: variable 'auto k' with 'auto' type used in its own initializer
```

There are other subtleties with `auto`. For example, using it as a storage class specifier would not work with C++0x. The following code snippet works nicely if you remove the `–std=c++0x` from the g++ command line:

```
auto int variable1 = 8;
```

Here's the g++ `--std=c++0x` log:

```
1.cpp: In function 'int main()':
1.cpp:28:14: error: two or more data types in declaration of 'variable1'
```

Of initializer lists and type narrowing

Remember the good old `int integer_array1 [ ] = {1, 2, 3, 4, 5};` construct from classic C/C++? The list defined using `{}`—that's an initializer list. Except that the language did not provide semantics for a broader-based use of this construct. C++0x defines additional usage rules for the initializer lists:

- Initializer lists in variable definitions
- Initializer lists in new expressions
- Can be used as a function argument and/or function return statement
- Allowed as a subscript expression

- Allowed as an argument to constructor invocation
- Type narrowing is not allowed

Before moving on to initializer list examples and typical usage, let's look at what is meant by *type narrowing*. Consider the code in Listing 8.

Listing 8. Initializing an integer array with float values

```
int main( )
{
   int nasty[ ] = {8, 99, 2.3, 4.0, 5};
   // …
   return 0;
}
```

When compiled with the g++ `–Wall` option, the compiler did not emit any warning for the above code despite the very ugly `double` to `integer` type conversion happening. This is an example of type narrowing, and thankfully, the C++0x standard does not allow for it in your code. Here's the log when you compile the code with g++ `–std=c++0x`:

```
1.cpp: In function 'int main()':
1.cpp:14:34: error: narrowing conversion of
    '2.2999999999999998223643160599749535322189331054 7e+0'
    from 'double' to 'int' inside { }
1.cpp:14:34: error: narrowing conversion of '4.0e+0'
    from 'double' to 'int' inside { }
```

Let's move on to brighter things. The declarations in Listing 9 are now allowed.

Listing 9. Using initializer lists with STL containers

```
// Initializer list used with variable definition
std::vector<double> doubles = {2.3, 4.511, 1.23, 0.99};

// Initializer list used with new
std::list<double> *d2 = new std::list<double> {1.2, 1.3};

// Initialize a map
std::map<string, int> = { {“key1”, 1}, {“key2”, 2} };
```

It's okay to use initializer lists to initialize scalar variables, and the usual rule of type narrowing applies in such cases. If a variable is initialized with an empty initializer list, the object is value initialized. The output from Listing 10 is `2 0 a A both x3 and x4 are null`.

Listing 10. Using initializer lists for scalar initialization

```
int main( )
{
    int x{2};
    double x2{};
    char* x3{};
    int* x4 = {};
    char c1 = {'a'} ;
    char c2 = char{'A'} ;
    cout << x << " " << x2 << " "
        << c1 << " " << c2 << endl;
    if (x3 == NULL && x4 == NULL)
        cout << "both x3 and x4 are null\n";
    // int y{2.3}; # don't try this error due to type narrowing
    return 0;
}
```

Note that `int y(2.3)` is allowed in C++0x; no type narrowing is assumed, and `y` equals 2 while `int y{2.3}` is plain error. Given this kind of semantics, using initializer lists should be preferred whenever possible. The C++0x standard defines a new class type (appropriately) called `initializer_list` that you can use to pass arguments to functions and constructors. Also, function return statements are allowed to return `initializer_list`; however, you must include the header `initializer_list` to use this class type. Listing 11 provides sample uses of `initializer_list`.

Listing 11. Using initializer lists as function arguments and return types

```
#include <initializer_list>

// argument to function
void func1(std::initializer_list<int>);

// function returning initializer list
std::initializer_list<double> func2 (double);
```

Listing 12 shows how you traverse an initializer list. Note that the function can only access the list as an immutable sequence—that is, trying to modify the contents of the list results in an error.

Listing 12. Accessing the contents of an initializer list

```
#include <initializer_list>
using namespace std;
void display (initializer_list<int> arguments) {
    for (auto p= arguments.begin(); p!= arguments.end(); ++p) {
            // *p = *p * 2; # Not allowed to modify data
            cout << *p << "\n";
    }
}

int main( )
{
    display( {3, 77, 8, 1, 9} );
    return 0;
}
```

Using auto with initializer list

What is the type for `x1` in the Listing 13 code?

Listing 13. Using automatic type deduction with initializer lists

```
int main( )
{
    auto x1 = {2, 4};
    auto z2 = {3, 2.3} ; // go figure
    ...
    return 0 ;
}
```

If you have not guessed it already, the type for `x1` is `std::initializer_list<int>`. As for `z2`, g++ throws an error, complaining quite rightly of type narrowing. If you still don't believe in the type for `x1`, just type in Listing 14.

Listing 14. Using automatic type deduction with initializer lists

```
#include <iostream>
#include <initializer_list>
using namespace std;

template <typename T>
void display() {}

template <>
void display<std::initializer_list<int>> () { cout << "Hurray!\n"; }

int main()
{
   auto x = {2, 3};
   display<decltype(x)> ();
   return 0;
}
```

The output from Listing 14 is `Hurray!`. The template specialization for `display` settles the debate over the alleged type of `x1`. This also brings us to yet another C++0x construct: the `decltype`.

Understanding decltype

C++ has never had an easy mechanism for querying the type of a variable or an expression. GCC provides an extension called `typeof` (see Resources), but that's non-standard. Enter the `decltype` operator from C++0x, which returns the type of a variable or expression. If template programming is your thing, this probably is something that should make you an immediate C++0x convert. Listing 15 applies the

`decltype` operator on a variable and shows the usage of the operator with an expression.

Listing 15. Applying a decltype operator to an expression

```
T1 x;
T2 y;
typedef T3 decltype(x+y);
T3 z ;
```

Lambda functions

If there's a prize for hype associated with a specific new feature of the C++0x standard, lambda functions take the crown. *Lambda functions* are anonymous functions, which means that you don't have to define a typical C/C++ function to get the job done. Perhaps the most common usage of lambdas is with STL `sort`. So far, to use custom comparison functions, the standard practice has been to define your own function object and then define the operator ( ) appropriately. Consider a vector of five strings; you want to sort them on the basis of increasing string length. Listing 16 shows the old approach.

Listing 16. Using automatic type deduction with initializer lists

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct compare {
    bool operator()(const string& s1, const string& s2) {
        return s1.size() < s2.size();
    }
};

int main()
{
    vector<string> vs = {"This", "is", "a", "C++0x", "exercise"};
    std::sort(vs.begin(), vs.end(), compare());

    for (auto ivs = vs.begin(); ivs != vs.end(); ++ivs)
        cout << *ivs << endl;
    return 0;
}
```

Listing 17 shows the new lambda-based approach.

Listing 17. Defining a lambda function for custom sorting

```
#include <iostream>
#include <string>
```

```
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<string> vs = {"This", "is", "a", "C++0x", "exercise"};
    std::sort(vs.begin(), vs.end(),
       [ ](const string& s1, const string& s2) {
          return s1.size() < s2.size();
       }
    )

    for (auto ivs = vs.begin(); ivs != vs.end(); ++ivs)
        cout << *ivs << endl;
    return 0;
}
```

The output from both programs (Listing 16 and Listing 17) is `a is This C++0x exercise`. A quick glance at Listing 18 reveals how lambda functions are defined. It looks much the same as you would define a regular function except for that `[ ]` thing, right? Well, yes and no. You have to ask some questions before you can drill down to the answers:

- Whatever happened to the return type of lambda functions?
- Can these functions access variables defined outside its scope? For example, can the lambda function in Listing 17 access `vs`?

On to the answers. You do not need to provide the return type of a lambda function; it is deduced from the return statement. In Listing 17, the return type of the lambda function is a Boolean. For the second question, just try the silly code provided in Listing 18.

Listing 18. Accessing out of scope variables inside lambda functions

```
int main()
{
    vector<string> vs = {"This", "is", "a", "C++0x", "exercise"};
    std::sort(vs.begin(), vs.end(),
       [ ](const string& s1, const string& s2) {
          cout << vs.size( ) << endl;
          return s1.size() < s2.size();
       }
    )
   …
}
```

So, you're trying to print the size of the vector every time `sort` calls the lambda. Here's what g++ has reported during compilation:

```
1.cpp: In lambda function:
1.cpp:12:9: error: 'vs' is not captured
```

There are two ways to "fix" this problem—actually, there are four ways. You can use the square brackets ([ ]) that you encountered earlier to pass arguments to the lambda function. If you choose to pass a variable by reference, prefix it with &, and put it inside the [ ]; for variable passing by value, prefix it with the equal sign (=). Listing 19 has vs being passed by reference.

Listing 19. Passing variables by reference to the lambda function

```
int main()
{
    vector<string> vs = {"This", "is", "a", "C++0x", "exercise"};
    std::sort(vs.begin(), vs.end(),
       [ &vs ](const string& s1, const string& s2) {
          cout << vs.size( ) << endl;
          return s1.size() < s2.size();
       }
    )
   …
}
```

This time, the compiler nicely complied. (For clarity, try this code with [=vs].) So, what's the third way to pass the vector to the lambda? Well, just use [&]. This syntax implies that all local variables will be passed by reference to the lambda function. Technically speaking, you can also use [=] and pass all local variables by value to the function: This is the fourth way to do things. However, for performance reasons, this approach is not recommended.

Variadic templates

How do you define a templated class or a function with a variable number of arguments, each with a potentially different type? C has support for defining functions with a variable number of arguments using va_list, and C++ never improved on that. Until now, that is. C++0x allows you to define functions and classes with variable numbers of arguments, and support is now the same in GCC. Listing 20 shows the syntax.

Listing 20. Variadic function and class template

```
template<typename... Types>
void f(Types... args) // variable number of function arguments
{
}

template<typename... Types>
class c // class with
{
   // member code
};

// Usages
f('a', "hello", 2, 3.1);
```

```
class c<int, double, std::vector<string>> c1;
```

You use the `typename…` to declare variadic templates. C++0x also defines the `sizeof…` operator, which displays the number of arguments. Unfortunately, there is no direct way you can iterate over the arguments. The only way to go about exploring the template is to define a recursive version of the same along with a base case. Not the best of things in life, but that's the way the story is so far. Listing 21 shows how you can print the arguments of a variadic function template.

Listing 21. Displaying the contents of a variadic function template

```
void f() { }

template<typename T, typename... Types>
void f(T a, Types... args)
{
    cout << sizeof...(args) << endl;
    cout << a << endl;
    f(args...);
}

int main()
{
   f("hello", 'a', 8.333);
   return 0;
}
```

Here's the blow-by-blow account of what's happening in Listing 21:

1. The first time around, the templated version of `f` is called, with a string as the first argument type, and the character and double are packed off as the variable size argument.
2. On the first invocation, `sizeof…(args)` displays `2`, because there are only two arguments for the variable `args` list.
3. Observe the syntax `args…`: The … is now on the right of `args`. The second call to `f` will have the variable list containing only `8.333`, while a becomes the first argument.
4. When you run out of arguments, g++ ends up calling `void f() { }` which is the base case for your recursion.

Here's the output from Listing 21:

```
2
hello
1
a
0
```

```
8.333
```

The state of things to come

Future versions of GCC promise to be even more exciting. Some of the features to look forward to include:

- **Garbage collection application binary interface (ABI) support.** The C++0x standard defines interfaces accessible to application developers that allow them to tweak garbage collection. For example, the predefined function `void declare_reachable(void* p);` asks the garbage collector not to reclaim storage pointed to by the pointer `p`.
- **Better support for concurrency.** GCC support for C++0x threads does not work well for all platforms (for example, Cygwin). Thread-specific storage is also a work in progress.

Conclusion

This is an exciting time to be a C++ (or should I say C++0x) developer. The standards committee has been adding features that are morphing C++ into a better platform for (to borrow from the language of Stroustrup [see Resources]) template programming, increased type safety, systems software, and library development. This article provided a brief overview of several features of the language now supported in GCC version 4.5. Refer to the GCC site and the Resources section for further details.

Resources

Learn

- C++0x FAQ: Visit the C++0x FAQ site maintained by Stroustrup.
- C++0x compiler support: Learn more from this comparative study of C++0x compiler support.
- C++0x draft standard: Read the draft copy of the C++0x standard.
- C++ papers: Read the C++ standards committee papers.
- GCC: Visit the GCC home page.
- `typeof`: Learn more about the GCC `typeof` operator.
- Delegating constructors: Read the white paper on delegating constructors.
- AIX and UNIX developerWorks zone: The AIX and UNIX zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX? Visit the New to AIX and UNIX page to learn more.
- Technology bookstore: Browse the technology bookstore for books on this and other technical topics.

Get products and technologies

- GCC binaries: Download the GCC binaries for installation.
- Boost library: Learn more about and download the Boost C++ library.

Discuss

- Follow developerWorks on Twitter.
- Get involved in the My developerWorks community.
- Participate in the AIX and UNIX® forums:
  - AIX Forum
  - AIX Forum for developers
  - Cluster Systems Management
  - Performance Tools Forum
  - Virtualization Forum
  - More AIX and UNIX Forums

About the author

Arpan Sen

Arpan Sen is a lead engineer working on the development of software in the electronic design automation industry. He has worked on several flavors of UNIX, including Solaris, SunOS, HP-UX, and IRIX as well as Linux and Microsoft Windows for several years. He takes a keen interest in software performance-optimization techniques, graph theory, and parallel computing. Arpan holds a post-graduate degree in software systems. You can reach him at arpansen@gmail.com.