The Biggest Changes in C++11

(and Why You Should Care)

June 20, 2011 by Danny Kalev

http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/

It's been 13 years since the first iteration of the C++ language. Danny Kalev, a former member of the C++ standards committee, explains how the programming language has been improved and how it can help you write better code.

Bjarne Stroustrup, the creator of C++, said recently that C++11 "feels like a new language — the pieces just fit together better." Indeed, core C++11 has changed significantly. It now supports lambda expressions, automatic type deduction of objects, uniform initialization syntax, delegating constructors, deleted and defaulted function declarations, `nullptr`, and most importantly, rvalue references — a feature that augurs a paradigm shift in how one conceives and handles objects. And that's just a sample.

The C++11 Standard Library was also revamped with new algorithms, new container classes, atomic operations, type traits, regular expressions, new smart pointers, `async()` facility, and of course a multithreading library.

A complete list of the new core and library features of C++11 is available here.

After the approval of the C++ standard in 1998, two committee members prophesied that the next C++ standard would "certainly" include a built-in garbage collector (GC), and that it probably wouldn't support multithreading because of the technical complexities involved in defining a portable threading model. Thirteen years later, the new C++ standard, C++11, is almost complete. Guess what? It lacks a GC but it does include a state-of–the-art threading library.

In this article I explain the biggest changes in the language, and why they are such a big deal. As you'll see, threading libraries are not the only change. The new standard builds on the decades of expertise and makes C++ even more relevant. As Rogers Cadenhead points out, "That's pretty amazing for something as old as disco, Pet Rocks, and Olympic swimmers with chest hair."

First, let's look at some of the prominent C++11 core-language features.

Lambda Expressions

A lambda expression lets you define functions locally, at the place of the call, thereby eliminating much of the tedium and security risks that function objects incur. A lambda expression has the form:

[capture](parameters)->return-type {body}

The `[]` construct inside a function call's argument list indicates the beginning of a lambda expression. Let's see a lambda example.

Suppose you want to count how many uppercase letters a string contains. Using `for_each()` to traverses a char array, the following lambda expression determines whether each letter is in uppercase. For every uppercase letter it finds, the lambda expression increments `Uppercase`, a variable defined outside the lambda expression:

```
int main()
{
   char s[]="Hello World!";
   int Uppercase = 0; //modified by the lambda
   for_each(s, s+sizeof(s), [&Uppercase] (char c) {
      if (isupper(c))
         Uppercase++;
   });
 cout<< Uppercase<<" uppercase letters in: "<< s<<endl;
}
```

It's as if you defined a function whose body is placed inside another function call. The ampersand in *[&Uppercase]* means that the lambda body gets a reference to*Uppercase* so it can modify it. Without the ampersand, *Uppercase* would be passed by value. C++11 lambdas include constructs for member functions as well.

Automatic Type Deduction and `decltype`

In C++03, you must specify the type of an object when you declare it. Yet in many cases, an object's declaration includes an initializer. C++11 takes advantage of this, letting you declare objects without specifying their types:

```
auto x=0; //x has type int because 0 is int
auto c='a'; //char
auto d=0.5; //double
auto national_debt=14400000000000LL;//long long
```

Automatic type deduction is chiefly useful when the type of the object is verbose or when it's automatically generated (in templates). Consider:

```
void func(const vector<int> &vi)
{
   vector<int>::const_iterator ci=vi.begin();
}
```

Instead, you can declare the iterator like this:

```
auto ci=vi.begin();
```

> *The keyword `auto` isn't new; it actually dates back the pre-ANSI C era. However, C++11 has changed its meaning; `auto` no longer designates an object with automatic storage type. Rather, it declares an object whose type is deducible from its initializer. The old meaning of `auto` was removed from C++11 to avoid confusion.*

C++11 offers a similar mechanism for capturing the type of an object or an expression. The new operator `decltype` takes an expression and "returns" its type:

```
const vector<int> vi;
typedef decltype (vi.begin()) CIT;
CIT another_const_iterator;
```

Uniform Initialization Syntax

C++ has at least four different initialization notations, some of which overlap.

Parenthesized initialization looks like this:

```
std::string s("hello");
int m=int(); //default initialization
```

You can also use the `=` notation for the same purpose in certain cases:

```
std::string s="hello";
int x=5;
```

For POD aggregates, you use braces:

```
int arr[4]={0,1,2,3};
struct tm today={0};
```

Finally, constructors use member initializers:

```
struct S
{
   int x;
   S(): x(0) {}
};
```

This proliferation is a fertile source for confusion, not only among novices. Worse yet, in C++03 you can't initialize POD array members and POD arrays allocated using `new[]`. C++11 cleans up this mess with a uniform brace notation:

```
class C
{
   int a;
   int b;
public:
   C(int i, int j);
};

C c {0,0}; //C++11 only. Equivalent to: C c(0,0);

int* a = new int[3] { 1, 2, 0 }; /C++11 only

class X
{
   int a[4];
public:
   X() : a{1,2,3,4} {} //C++11, member array initializer
};
```

With respect to containers, you can say goodbye to a long list of `push_back()`calls. In C++11 you can initialize containers intuitively:

```
// C++11 container initializer
vector<string> vs={ "first", "second", "third"};
map singers =
  { {"Lady Gaga", "+1 (212) 555-7890"},
    {"Beyonce Knowles", "+1 (212) 555-0987"}};
```

Similarly, C++11 supports in-class initialization of data members:

```
class C
{
   int a=7; //C++11 only
public:
   C();
};
```

Deleted and Defaulted Functions

A function in the form:

```
struct A
{
   A()=default; //C++11
   virtual ~A()=default; //C++11
};
```

is called a *defaulted function*. The `=default;` part instructs the compiler to generate the default implementation for the function. Defaulted functions have two advantages: They are more efficient than manual implementations, and they rid the programmer from the chore of defining those functions manually.

The opposite of a defaulted function is a *deleted function*:

```
int func()=delete;
```

Deleted functions are useful for preventing object copying, among the rest. Recall that C++ automatically declares a copy constructor and an assignment operator for classes. To disable copying, declare these two special member functions `=delete`:

```
struct NoCopy
{
   NoCopy & operator =( const NoCopy & ) = delete;
   NoCopy ( const NoCopy & ) = delete;
};
NoCopy a;
NoCopy b(a); //compilation error, copy ctor is deleted
```

nullptr

At last, C++ has a keyword that designates a null pointer constant. `nullptr`replaces the bug-prone `NULL` macro and the literal 0 that have been used as null pointer substitutes for many years. `nullptr` is strongly-typed:

```
void f(int); //#1
void f(char *);//#2
//C++03
f(0); //which f is called?
//C++11
f(nullptr) //unambiguous, calls #2
```

`nullptr` is applicable to all pointer categories, including function pointers and pointers to members:

```
const char *pc=str.c_str(); //data pointers
if (pc!=nullptr)
   cout<<pc<<endl;
int (A::*pmf)()=nullptr; //pointer to member function
void (*pmf)()=nullptr; //pointer to function
```

Delegating Constructors

In C++11 a constructor may call another constructor of the same class:

```
class M //C++11 delegating constructors
{
   int x, y;
   char *p;
public:
   M(int v) : x(v), y(0), p(new char [MAX]) {} //#1 target
   M(): M(0) {cout<<"delegating ctor"<<endl;} //#2 delegating
};
```

Constructor #2, the delegating constructor, invokes the *target constructor* #1.

Rvalue References

Reference types in C++03 can only bind to *lvalues*. C++11 introduces a new category of reference types called *rvalue references*. Rvalue references can bind to *rvalues*, e.g. temporary objects and literals.

The primary reason for adding *rvalue* references is *move semantics*. Unlike traditional copying, moving means that a target object *pilfers* the resources of the source object, leaving the source in

an “empty” state. In certain cases where making a copy of an object is both expensive and unnecessary, a move operation can be used instead. To appreciate the performance gains of move semantics, consider string swapping. A naive implementation would look like this:

```
void naiveswap(string &a, string & b)
{
   string temp = a;
   a=b;
   b=temp;
}
```

This is expensive. Copying a string entails the allocation of raw memory and copying the characters from the source to the target. In contrast, moving strings merely swaps two data members, without allocating memory, copying char arrays and deleting memory:

```
void moveswapstr(string& empty, string & filled)
{
//  pseudo code, but you get the idea
   size_t sz=empty.size();
   const char *p= empty.data();
//  move filled's resources to empty
   empty.setsize(filled.size());
   empty.setdata(filled.data());
//  filled becomes empty
   filled.setsize(sz);
   filled.setdata(p);
}
```

If you’re implementing a class that supports moving, you can declare a move constructor and a move assignment operator like this:

```
class Movable
{
   Movable (Movable&&); //move constructor
   Movable&& operator=(Movable&&); //move assignment operator
};
```

The C++11 Standard Library uses move semantics extensively. Many algorithms and containers are now move-optimized.

C++11 Standard Library

C++ underwent a major facelift in 2003 in the form of the Library Technical Report 1 (TR1). TR1 included new container classes (`unordered_set`, `unordered_map`,`unordered_multiset`, and `unordered_multimap`) and several new libraries for regular expressions, tuples, function object wrapper and more. With the approval of C++11, TR1 is officially incorporated into standard C++ standard, along with new libraries that have been added since TR1. Here are some of the C++11 Standard Library features:

Threading Library

Unquestionably, the most important addition to C++11 from a programmer's perspective is concurrency. C++11 has a thread class that represents an execution thread, promises and futures, which are objects that are used for synchronization in a concurrent environment, the async() function template for launching concurrent tasks, and the thread_local storage type for declaring thread-unique data. For a quick tour of the C++11 threading library, read Anthony Williams' Simpler Multithreading in C++0x.

New Smart Pointer Classes

C++98 defined only one smart pointer class, `auto_ptr`, which is now deprecated. C++11 includes new smart pointer classes: shared_ptr and the recently-added unique_ptr. Both are compatible with other Standard Library components, so you can safely store these smart pointers in standard containers and manipulate them with standard algorithms.

New C++ Algorithms

The C++11 Standard Library defines new algorithms that mimic the set theory operations `all_of()`, `any_of()` and `none_of()`. The following listing applies the predicate `ispositive()` to the range `[first, first+n)` and uses `all_of()`,`any_of()` and `none_of()` to examine the range's properties:

```
#include <algorithm>
//C++11 code
//are all of the elements positive?
all_of(first, first+n, ispositive()); //false
//is there at least one positive element?
any_of(first, first+n, ispositive());//true
// are none of the elements positive?
none_of(first, first+n, ispositive()); //false
```

A new category of `copy_n` algorithms is also available. Using `copy_n()`, copying an array of 5 elements to another array is a cinch:

```
#include <algorithm>
int source[5]={0,12,34,50,80};
int target[5];
//copy 5 elements from source to target
copy_n(source,5,target);
```

The algorithm `iota()` creates a range of sequentially increasing values, as if by assigning an initial value to `*first`, then incrementing that value using prefix ++. In the following listing, `iota()` assigns the consecutive values {10,11,12,13,14} to the array `arr`, and {'a', 'b', 'c'} to the char array `c`.

```
#include <numeric>
int a[5]={0};
char c[3]={0};
iota(a, a+5, 10); //changes a to {10,11,12,13,14}
iota(c, c+3, 'a'); //{'a','b','c'}
```

C++11 still lacks a few useful libraries such as an XML API, sockets, GUI, reflection — and yes, a proper automated garbage collector. However, it does offer plenty of new features that will make C++ more secure, efficient (yes, even more efficient than it has been thus far! See Google's benchmark tests), and easier to learn and use.
If the changes in C++11 seem overwhelming, don't be alarmed. Take the time to digest these changes gradually. At the end of this process you will probably agree with Stroustrup:
C++11 *does* feel like a new language — a much better one.

Get to Know the New C++11 Initialization Forms

By Danny Kalev

Date: Mar 28, 2012

http://www.informit.com/articles/printerfriendly/1852519

Initializing your objects, arrays, and containers is much easier in C++11 than it used to be in C++03. Danny Kalev explains how to use the new brace-initialization notation, class member initializers, and initialization lists to write better and shorter code, without compromising code safety or efficiency.

Initialization in C++03 is tricky, to say the least, with four different initialization notations and far too many arbitrary restrictions and loopholes:

- No way to initialize a member array
- No convenient form of initializing containers
- No way to initialize dynamically allocated POD types

C++11 fixes these problems with new facilities for initializing objects. In this article, I present the new C++11 brace-initialization notation, discuss class member initialization, and explain how to initialize containers by using initializer lists.

C++03 Initialization: A Tale of Four Initialization Notations

To appreciate the new initialization syntax of C++11, let's look at the C++03 initialization Babel first. C++03 has various categories of initialization:

- **Initialization of fundamental types.** The initialization of fundamental types uses the equal sign (=):

```
int n=0;
void*p=0;
char c='A';
```

- **Initialization of data members in a class and objects.** Classes with a user-defined constructor require a constructor's member initialization list (*mem-init* for short) for their data members. An object's initializers are enclosed in parentheses in the object's declaration:

```
//C++03 initialization of classes and objects
struct S1
```

```
{
  explicit S1(int n, int m) : x(n), y(m){} //mem-init
private:
  int x, y;
};
S1 s(0,1); //object initializers enclosed in parentheses
S1 s2={0,1}; //compilation error
```

- **Initialization of aggregates.** Aggregate initialization requires braces, with the exception of string literals that may also appear between a pair of double quotes ([dp][dp]):

```
//C++03: POD arrays and structs are aggregates
int c1[2]={0,2};
char c2[]="message";
//or you can use the more verbose form:
char c3[]={'m','e','s','s','a','g','e','\0'};
struct S
{
  int a,b;
};
S s={0,1};
```

You can use parentheses to initialize fundamental types as well. The parentheses in this case are interchangeable with the equal sign notation:

```
int n(0); //same as int n=0;
double d(0.5);
```

C++03 Initialization: Arbitrary Restrictions and Loopholes

To add insult to injury, C++03 imposes arbitrary restrictions in some cases, such as the inability to initialize member arrays:

```
class C
{
  int x[100];
  C(); //no proper way to initialize x
};
```

Similarly, you can't initialize a dynamically allocated POD array:

```
char *buff=new char[1024]; //no proper way to initialize the elements of buff
```

Finally, there's no easy way to initialize the elements of a Standard Library container. For instance, if you want to initialize a vector of strings, you'd normally use a sequence of `push_back()` calls like this:

```
vector <string> vs;
vs.push_back("alpha");
vs.push_back("beta");
vs.push_back("gamma");
```

To conclude, C++03 initialization is a mess. Let's see how C++11 tackles these problems with its new and uniform initialization notation.

Introducing C++11 Brace-Initialization

C++11 attempts to overcome the problems of C++03 initialization by introducing a universal initialization notation that applies to every type—whether a POD variable, a class object with a user-defined constructor, a POD array, a dynamically allocated array, or even a Standard Library container. The universal initializer is called a *brace-init*. It looks like this:

```
//C++11 brace-init
int a{0};
string s{"hello"};
string s2{s}; //copy construction
vector <string> vs{"alpha", "beta", "gamma"};
map<string, string> stars
 { {"Superman", "+1 (212) 545-7890"},
  {"Batman", "+1 (212) 545-0987"}};
double *pd= new double [3] {0.5, 1.2, 12.99};
class C
{
   int x[4];
public:
   C(): x{0,1,2,3} {}
};
```

Notice that unlike the traditional aggregate initializer of C and C++03, which uses braces after an equal sign (`={}`), the C++11 brace-init consists of a pair of braces (without the equal sign) in which the initializer(s) will be enclosed. An empty pair of braces indicates default initialization. Default initialization of POD types usually means initialization to binary zeros, whereas for non-POD types default initialization means default construction:

```
//C++11: default initialization using {}
int n{}; //zero initialization: n is initialized to 0
int *p{}; //initialized to nullptr
double d{}; //initialized to 0.0
char s[12]{}; //all 12 chars are initialized to '\0'
string s{}; //same as: string s;
char *p=new char [5]{}; // all five chars are initialized to '\0'
```

Class Member Initialization

C++11 pulls another rabbit out of its hat with class member initializers. Perhaps an example will best illustrate these:

```
class C
{
   int x=7; //class member initializer
public:
   C();
};
```

The data member *x* is automatically initialized to 7 in every instance of class `C`. In former dialects of C++, you would use the more cumbersome mem-init notation for the same purpose:

```
class C
{
   int x;
public:
   C() : x(7) {}
};
```

C++11 class member initializers are mostly a matter of convenience. They provide an overt and simplified form of initializing data members. But class member initializers also let you perform a few tricks that have hitherto been impossible. For example, you can use a class member initializer to initialize a member array:

```
class C
{
   int y[5] {1,2,3,4};
public:
   C();
};
```

Notice that a class member initializer can consist of any valid initialization expression, whether that's the traditional equal sign, a pair of parentheses, or the new brace-init:

```
class C
{
   string s("abc");
   double d=0;
   char * p {nullptr};
   int y[5] {1,2,3,4};
public:
   C();
};
```

Regardless of the initialization form used, the compiler conceptually transforms every class member initializer into a corresponding mem-init. Thus, class C above is semantically equivalent to the following class:

```
class C2
{
   string s;
   double d;
   char * p;
   int y[5];
public:
   C() : s("abc"), d(0.0), p(nullptr), y{1,2,3,4} {}
};
```

Bear in mind that if the same data member has both a class member initializer and a mem-init in the constructor, the latter takes precedence. In fact, you can take advantage of this behavior by specifying a default value for a member in the form of a class member initializer that will be used if the constructor doesn't have an explicit mem-init for that member. Otherwise, the constructor's mem-init will take effect, overriding the class member initializer. This technique is useful in classes that have multiple constructors:

```
class C
{
   int x=7; //class member initializer
   C(); //x is initialized to 7 when the default ctor is invoked
   C(int y) : x(y) {} //overrides the class member initializer
};
C c; //c.x = 7
```

```
C c2(5); //c.x = 5
```

Initializer Lists and Sequence Constructors

An *initializer list* lets you use a sequence of values wherever an initializer can appear. For example, you can initialize a vector in C++11 like this:

```
vector<int> vi {1,2,3,4,5,6};
vector<double> vd {0.5, 1.33, 2.66};
```

You may include as many initializers as you like between the braces. Although superficially this new syntax seems identical to the brace-init notation we discussed earlier, behind the scenes it's a different story. C++11 furnishes every STL container with a new constructor type called a *sequence constructor*. A sequence constructor intercepts initializers in the form of `{x,y...}`. To make this machinery work, C++11 introduced another secret ingredient: an auxiliary class template called `std::initializer_list<T>`. When the compiler sees an initializer list, say `{0.5, 1.33, 2.66}`, it transforms the values in that list into an array of `T` with *n* elements (*n* is the number of values enclosed in braces) and uses that array to populate an implicitly generated `initializer_list<T>` object. The class template `initializer_list` has three member functions that access the array:

```
template<class E> class initializer_list
{
   //implementation (a pair of pointers or a pointer + length)
public:
   constexpr initializer_list(const E*, const E*); // [first,last)
   constexpr initializer_list(const E*, int); // [first, first+length)
   constexpr int size() const; // no. of elements
   constexpr const T* begin() const; // first element
   constexpr const T* end() const; // one more than the last element
};
```

To better understand how the compiler handles initializer lists of containers, let's dissect a concrete example. Suppose your code contains the following declaration of a vector:

```
vector<double> vd {0.5, 1.33, 2.66};
```

The compiler detects the initializer list `{0.5, 1.33, 2.66}` and performs the following steps:

1. Detect the type of the values in the initializer list. In the case of `{0.5, 1.33, 2.66}`, the type is `double`.

2. Copy the values from the list into an array of three `doubles`.
3. Construct an `initializer_list<double>` object that "wraps" the array created in the preceding step.
4. Pass the `initializer_list<double>` object by reference to `vd`'s sequence constructor. The constructor in turn allocates *n* elements in the vector object, initializing them with the values of the array.

It's hard to imagine that so much is going on behind the scenes every time you initialize an STL container with a pair of braces! The good news is that you don't have to do anything for this magic to happen—it just works. Of course, you still need a C++11-compliant compiler as well as a C++11-compliant Standard Library to use initializer lists. Make sure that your target project is built with the appropriate compilation options, too.

In Conclusion

The C++ standards committee invested a lot of time and effort in finding a solution to the limitations of C++03 initialization. It looks like they succeeded. Historically, brace-init, class member initializers, and initializer lists were three independent proposals. Later, they were revised to ensure compatibility and uniformity. Together, these three initialization-related features make C++11 programming simpler and more intuitive. You will surely appreciate them next time you initialize a dynamically allocated array—or, indeed, a vector.

C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator

August 6, 2012 by Danny Kalev

Copy constructors sounds like a topic for an article from 1989. And yet, the changes in the new C++ standard affect the design of a class' special member functions fundamentally. Find out more about the impact of move semantics on objects' behavior and learn how to implement the move constructor and the move assignment operator in C++11.

C++11 is the informal name for ISO/IEC 14882:2011, the new C++ standard that was published in September 2011. It includes theTR1 libraries and a large number of new core features (a detailed discussion about these new C++11 features is available here; also see The Biggest Changes in C++11 (and Why You Should Care)):

- Initializer lists
- Uniform initialization notation
- Lambda functions and expressions
- Strongly-typed enumerations
- Automatic type deduction in declarations
- `__thread_local` storage class
- Control and query of object alignment
- Static assertions
- Type `long long`
- Variadic templates

Important as these features may be, the defining feature of C++11 is *rvalue references*.

The Right Time for Rvalue References

Rvalue references are a new category of reference variables that can bind to*rvalues*. Rvalues are slippery entities, such as temporaries and literal values; up until now, you haven't been able to bind these safely to reference variables.

Technically, an rvalue is an unnamed value that exists only during the evaluation of an expression. For example, the following expression produces an rvalue:

```
x+(y*z); // A C++ expression that produces a temporary
```

C++ creates a temporary (an *rvalue*) that stores the result of `y*z`, and then adds it to `x`. Conceptually, this *rvalue* evaporates by the time you reach the semicolon at the end of the full expression.

A declaration of an *rvalue* reference looks like this:

```
std::string&& rrstr; //C++11 rvalue reference variable
```

The traditional reference variables of C++ e.g.,

```
std::string& ref;
```

are now called *lvalue references*.

Rvalue references occur almost anywhere, even if you don't use them directly in your code. They affect the semantics and lifetime of objects in C++11. To see how exactly, it's time to talk about *move semantics*.

Get to Know Move Semantics

Hitherto, copying has been the only means for transferring a state from one object to another (an object's state is the collective set of its non-static data members' values). Formally, copying causes a target object `t` to end up with the same state as the source `s`, without modifying `s`. For example, when you copy a string `s1` to`s2`, the result is two identical strings with the same state as `s1`.

And yet, in many real-world scenarios, you don't copy objects but *move* them. When my landlord cashes my rent check, he moves money from my account into his. Similarly, removing the SIM card from your mobile phone and installing it in another mobile is a move operation, and so are cutting-and-pasting icons on your desktop, or borrowing a book from a library.

Notwithstanding the conceptual difference between copying and moving, there's a practical difference too: Move operations tend to be faster than copying because they transfer an existing resource to a new destination, whereas copying requires the creation of a new resource from scratch. The efficiency of moving can be witnessed among the rest in functions that return objects by value. Consider:

```
string func()
{
   string s;
   //do something with s
   return s;
```

```
}
string mystr=func();
```

When `func()` returns, C++ constructs a temporary copy of `s` on the caller's stack memory. Next, `s` is destroyed and the temporary is used for copy-constructing `mystr`. After that, the temporary itself is destroyed. Moving achieves the same effect without so many copies and destructor calls along the way.

Moving a string is almost free; it merely assigns the values of the source's data members to the corresponding data members of the target. In contrast, copying a string requires the allocation of dynamic memory and copying the characters from the source.

Move Special Member Functions

C++11 introduces two new special member functions: the *move constructor* and the *move assignment operator*. They are an addition to the fabulous four you know so well:

- Default constructor
- Copy constructor
- Copy assignment operator
- Destructor

If a class doesn't have any user-declared special member functions (save a default constructor), C++ declares its remaining five (or six) special member functions implicitly, including a move constructor and a move assignment operator. For example, the following class

```
class S{};
```

doesn't have any user-declared special member functions. Therefore, C++ declares all of its six special member functions implicitly. Under certain conditions, implicitly declared special member functions become implicitly defined as well. The implicitly-defined move special member functions move their sub-objects and data members in a member-wise fashion. Thus, a move constructor invokes its sub-objects' move constructors, recursively. Similarly, a move assignment operator invokes its sub-objects' move assignment operators, recursively.

What happens to a moved-from object? The state of a moved-from object is unspecified. Therefore, always assume that a moved-from object no longer owns any resources, and that its state is similar to that of an empty (as if default-constructed) object. For example, if you move a string `s1` to `s2`, after the move operation the state of `s2` is identical to that of `s1` before the move, whereas `s1` is now an empty (though valid) string object.

Designing a Move Constructor

A move constructor looks like this:

```
C::C(C&& other); //C++11 move constructor
```

It doesn't allocate new resources. Instead, it *pilfers* `other`'s resources and then sets `other` to its default-constructed state.

Let's look at a concrete example. Suppose you're designing a `MemoryPage` class that represents a memory buffer:

```
class MemoryPage
{
   size_t size;
   char * buf;
public:
   explicit MemoryPage(int sz=512):
   size(sz), buf(new char [size]) {}
   ~MemoryPage( delete[] buf;}
   //typical C++03 copy ctor and assignment operator
   MemoryPage(const MemoryPage&);
   MemoryPage& operator=(const MemoryPage&);
};
```

A typical move constructor definition would look like this:

```
//C++11
MemoryPage(MemoryPage&& other): size(0), buf(nullptr)
{
   // pilfer other's resource
   size=other.size;
   buf=other.buf;
   // reset other
   other.size=0;
   other.buf=nullptr;
}
```

The move constructor is much faster than a copy constructor because it doesn't allocate memory nor does it copy memory buffers.

Designing a Move Assignment Operator

A move assignment operator has the following signature:

```
C& C::operator=(C&& other);//C++11 move assignment operator
```

A move assignment operator is similar to a copy constructor except that before pilfering the source object, it releases any resources that its object may own. The move assignment operator performs four logical steps:

- Release any resources that `*this` currently owns.
- Pilfer `other`'s resource.
- Set `other` to a default state.
- Return `*this`.

Here's a definition of `MemoryPage`'s move assignment operator:

```
//C++11
MemoryPage& MemoryPage::operator=(MemoryPage&& other)
{
   if (this!=&other)
   {
      // release the current object's resources
      delete[] buf;
      size=0;
      // pilfer other's resource
      size=other.size;
      buf=other.buf;
      // reset other
      other.size=0;
      other.buf=nullptr;
   }
   return *this;
}
```

Overloading Functions

The overload resolution rules of C++11 were modified to support rvalue references. Standard Library functions such as `vector::push_back()` now define two overloaded versions: one that takes `const T&` for lvalue arguments as before, and a new one that takes a parameter of type `T&&` for rvalue arguments. The following program populates a vector with `MemoryPage` objects using two `push_back`() calls:

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

```
int main()
{
   vector<MemoryPage> vm;
   vm.push_back(MemoryPage(1024));
   vm.push_back(MemoryPage(2048));
}
```

Both `push_back()` calls resolve as `push_back(T&&)` because their arguments are rvalues. `push_back(T&&)` moves the resources from the argument into `vector`'s internal `MemoryPage` objects using `MemoryPage`'s move constructor. In older versions of C++, the same program *would* generate copies of the argument since the copy constructor of `MemoryPage` would be called instead.

As I said earlier, `push_back(const T&)` is called when the argument is an lvalue:

```
#include <vector>
using namespace std;
int main()
{
   vector<MemoryPage> vm;
   MemoryPage mp1(1024);//lvalue
   vm.push_back(mp); //push_back(const T&)
}
```

However, you can enforce the selection of `push_back(T&&)` even in this case by casting an lvalue to an rvalue reference using `static_cast`:

```
//calls push_back(T&&)
```

vm.push_back(static_cast<MemoryPage&&>(mp));

Alternatively, use the new standard function `std::move()` for the same purpose:

```
vm.push_back(std::move(mp));//calls push_back(T&&)
```

It may seem as if `push_back(T&&)` is always the best choice because it eliminates unnecessary copies. However, remember that `push_back(T&&)` empties its argument. If you want the argument to retain its state after a `push_back()` call, stick to copy semantics. Generally speaking, don't rush to throw away the copy constructor and the copy assignment operator. In some cases, the same class could be used in a context that requires pure copy semantics, whereas in other contexts move semantics would be preferable.

In Conclusion

C++11 is a different and better C++. Its rvalue references and move-oriented Standard Library eliminate many unnecessary copy operations, thereby improving performance significantly, with minimal, if any, code changes. The move constructor and the move assignment operator are the vehicles of move operations. It takes a while to internalize the principles of move semantics – and to design classes accordingly. However, the benefits are substantial. I would dare predicting that other programming languages will soon find ways to usher-in move semantics too.

C++11 Tutorial: New Constructor Features that Make Object Initialization Faster and Smoother

August 23, 2012 by Danny Kalev

Constructors in C++11 still do what they've always done: initialize an object. However, two new features, namely delegating constructors and class member initializers, make constructors simpler, easier to maintain, and generally more efficient. Learn how to combine these new features in your C++ classes.

C++11 introduced several constructor-related enhancements including:

- Delegating constructors
- Class member initializers

The interaction between these two features enables you to tailor your classes' constructors to achieve safer and simpler initialization. In this article, I demonstrate the use of delegating constructors, explain what class member initializers are, and show how to combine both of them in your classes.

Constructive Talks

Earlier variants of C++ lack a mechanism for calling a constructor from another constructor of the same class (constructors of the same class are known as *sibling constructors*). This limitation is chiefly noticeable in cases where default arguments aren't an option. Consequently, the class's maintainer has to write multiple constructors that repeat similar initialization code segments.

Thus far, the common workaround has been to define a separate initialization member function, and call it from every constructor:

```
class C
{
    void init();//comon initialization tasks
    int s;
    T t; //T has a conversion operator to double
public:
    C(): s(12), t(4.5) {init();}
    C(int i) : s(i), t(4.5) {init();}
    C(T& t1) : s(12), t(t1) {init();}
};
```

This approach has two drawbacks. First, delegating the initialization task to `init()`is inefficient because by the time `init()` executes, the data members of C have already been constructed. In other words, `init()` merely *reassigns* new values to C's data members. It doesn't really initialize them. (By contrast, initialization by means of a member initialization list takes place before the constructor's body executes.)

The second problem is obvious: Every constructor repeats identical bits of initialization code. Not only is it wearisome, it's also a recipe for future maintenance problems. Every time you modify the class's members or its members' initialization protocol, you'd need to update multiple constructors accordingly.

Introducing Target Constructors

C++11 solves this problem by allowing a constructor (known as the *delegating constructor*) to call another sibling constructor (the *target constructor*) from the delegating constructor's member initialization list. For example:

```
class C //a C11 class with a target constructor
{
   int s;
   T t;  //has a conversion operator to double
public:
   //the target constructor
   C(int i, T& t1): s(i), t(t1) {/*more common init code*/}
   //three delegating ctors that call the target ctor
   C(): C(12, 4.5) {/*specific post-init code*/}
   C(int i): C(i, 4.5) {/*specific post-init code*/}
   C(T& t1): C(12, t1) {/*specific post-init code*/}
};
```

The modified class C now defines a fourth constructor: the target constructor. The three delegating constructors call it from their member initialization list to perform the common initialization procedure. Once the target constructor returns, the body of the delegating constructor is executed.

Notice that there are no special syntactic markers for target and delegating constructors. A target constructor is defined as such if another constructor of the same class calls it. Likewise, a delegating constructor is a constructor that calls a sibling constructor in its member initialization list.

From this you can infer that a target constructor can also be a delegating constructor if it calls another target constructor. However, if a constructor delegates to itself directly or indirectly, the

program is ill-formed. The following example demonstrates a target constructor that calls another target constructor:

```
class Z //C++11, a target ctor that's also a delegating ctor
{
   int j;
   bool flag;
public:
   Z(): Z(5) {} //#1
   explicit Z(int n) :Z(n, false) {} //#2
   Z(int n, bool b): j(n), flag(b) {} //#3
};
```

Constructor #1 calls the target constructor #2. Constructor #2 is also a delegating constructor since it calls the target constructor #3.

Technically, the use of default arguments could make two of the three constructors of class Z redundant:

```
class Z //using default arguments to minimize ctors
{
   int j;
   bool flag;
public:
   Z(int n=5, bool b=false): j(n), flag(b) {}
};
```

However, default arguments aren't always desirable (or even possible) for various reasons. Therefore, delegating constructors that call a target constructor are a convenient and more efficient alternative to `init()`.

Class Member Initializers

C++11 introduced another related feature called *class member initializers* that could make the design and implementation of your constructors even simpler.

Note: Class member initializers are also called in-class initializers. I use both terms interchangeably in this article.

In this case too, C++11 follows other programming languages that let you initialize a data member directly in its declaration:

```
class M //C++11
{
   int j=5; //in-class initializer
   bool flag (false); //another in-class initializer
public:
   M();
};
M m1; //m1.j = 5, m1.flag=false
```

Under the hood, the compiler transforms every class member initializer (such as`int j=5;`) into a constructor's member initializer. Therefore, the definition of class M above is semantically equivalent to the following C++03 class definition:

```
class M2
{
   int j;
   bool flag;
public:
   M2(): j(5), flag(false) {}
};
```

If the constructor includes an explicit member initializer for a member that also has an in-class initializer, the constructor's member initializer takes precedence, effectively overriding the in-class initializer for that particular constructor. Consider:

```
class M2
{
   int j=7;
public:
   M2(); //j=7
   M2(int i): j(i){}//overrides j's in-class initializer
};
M2 m2; //j=7,
M2 m3(5); //j=5
```

This property of class member initializers makes them useful for defining a "default" value that constructors may override individually. In other words, a constructor that needs to override the default value for a certain member can use an explicit member initializer for it. Otherwise, the in-class initializer takes effect.

Class member initializers can simplify the design of a class's constructor. You can specify the initializer for a given member directly with its declaration instead of using a target constructor for

other constructors to call. If, however, a certain member requires a more complex initialization procedure (for example, its value depends on other data members' values, or if its value depends on a specific constructor), a combination of traditional constructor member initializers and target constructors does the trick.

Remember that if a target constructor appears in a constructor's member initialization list, that list shall contain nothing else. No other base or member initializers are allowed in this case.

Let's look again at the constructors of class C to see how to combine class member initializers and delegating constructors. Recall that the original constructor set of C looks like this:

```
//a target constructor
C(int i, T& t1): s(i), t(t1) {/*common init code*/}
//three delegating ctors invoke the target
C(): C(12, 4.5) {/*specific post-init code*/}
C(int i): C(i, 4.5) {/*specific post-init code*/}
C(T& t1): C(12, t1) {/*specific post-init code*/}
```

The member s is initialized to 12 in two out of four constructors. You can therefore use a class member initializer for it, while leaving constructor-dependent initializers in member initializer lists or in the single target constructor:

```
class C //C11, combine in-class init with a target ctor
class C //C11, combine in-class init with a target ctor
{
   int s=12;
   T t;
public:
   //a target constructor
   C(int i, T& t1): s(i), t(t1) {}
   C(): t( 4.5) {} //s=12, t=4.5
   C(int i): C(i, 4.5) {}//using a target ctor. s=i, t=4.5
   C( T& t1): t( t1) {} //s=12, t=t1
};
```

Of course, you can use a different combination such as an in-class initializer for the member t instead of s, or use in-class initializers for both. There is no right or wrong here. Rather, the aim is to reduce the amount of repeated initialization code and to minimize the complexity of the constructors while ensuring that data members are initialized properly. Most of all, the aim is to get rid of the `init()`hack.

Apart from making your code more readable and easier to maintain, delegating constructors and class member initializers offer a hidden bonus. They also improve your constructors' performance because they literally initialize data members, as opposed to reassigning a new value to them (which is what `init()`-like member functions actually do).

Therefore, it might be a good idea to go through your existing pre-C++11 code and refactor it with the following guidelines in mind: Simple initializers that occur frequently should become class member initializers in general, whereas ad-hoc initializers should be dealt with by target constructors and member initializer lists.

C++11 Tutorial: Let Your Compiler Detect the Types of Your Objects Automatically

September 27, 2012 by Danny Kalev

Imagine that your compiler could guess the type of the variables you declare as if by magic. In C++11, it's possible — well, almost. The new auto and decltype facilities detect the type of an object automatically, thereby paving the way for cleaner and more intuitive function declaration syntax, while ridding you of unnecessary verbiage and keystrokes. Find out how auto and decltype simplify the design of generic code, improve code readability, and reduce maintenance overhead.

Fact #1: Most declarations of C++ variables include an explicit initializer. Fact #2: The majority of those initializers encode a unique datatype. C++11 took advantage of these facts by introducing two new closely related keywords: `auto` and `decltype`.`auto` lets you declare objects without specifying their types explicitly, while `decltype` captures the type of an object.

Together, `auto` and `decltype` make the design of generic code simpler and more robust while reducing the number of unnecessary keystrokes. Let's look closely at these new C++11 silver bullets.

auto Declarations

As stated above, most of the declarations in a C++ program have an explicit initializer (on a related note, you may find this article about class member initializers useful). Consider the following examples:

```
int x=0;
const double PI=3.14;
string curiosity_greeting (“hello universe”);
bool dirty=true;
char *p=new char [1024];
long long distance=10000000000LL;
int (*pf)() = func;
```

Each of the seven initializers above is associated with a unique datatype. Let's examine some of them:

- The literal 0 is int.
- Decimal digits with a decimal point (e.g. 3.14) are taken to be double.
- The initializer new char [] implies char*.
- An integral literal value with the affix LL (or ll) is long long.

The compiler can extract this information and automatically assign types to the objects you're declaring. For that purpose, C++11 introduced a new syntactic mechanism. A declaration whose type is omitted begins with the new keyword auto. An *auto declaration* must include an initializer with a distinct type (an empty pair of parentheses won't do). The rest is relegated to the compiler. How convenient!

Note: C++ actually has two auto keywords: the old auto and the new auto. The old auto dates back to the late 1960s, serving as a storage class in declarations of variables with automatic storage. Since automatic storage is always implied from the context in C++, the old auto was seldom used, which is why most programmers didn't even know it existed in C++ until recently.C++11 removed the old auto (though it still exists in C) in favor of the new `auto`*, which is used in* `auto` *declarations. Some implementations may require that you turn on C++11 computability mode to support the new* `auto`*. In this article, I refer exclusively to the new* `auto`*.*

Here are the previous seven declarations transformed into auto declarations:

```
//C++11
auto x=0;
const auto PI=3.14;
auto curiosity_greeting= string (“hello universe”);
auto dirty=true;
auto p=new char [1024];
auto distance=10000000000LL;
auto pf = func;
```

In the case of class objects such as `std::string`, you need to include the name of the class in the initializer to disambiguate the code. Otherwise, the quoted literal string "hello universe" would be construed as `const char[15]`.

Const, Arrays, and Reference Variables

If you want to declare cv-qualified objects, you have to provide the const and volatile qualifiers in the auto declaration itself. That's because the compiler can't tell whether the initializer 3.14 is plain double, const double, volatile double, or const volatile double:

```
volatile auto i=5; //volatile int
auto volatile const flag=true; //const volatile bool
auto const y=’a’; //const char
```

To declare an auto array, enclose the initializers in braces. The compiler counts the number of initializers enclosed and uses that information to determine the size (and dimensions) of the array:

```
auto iarr={0,1,2,3}; //int [4]
auto ararr={{0,1},{0,1}}; //int[2][2]
```

Tip: Certain implementations insist that you `#include` the new standard header `<initializer_list>` when you declare an auto array. Technically, the C++11 standard requires that an implementation `#include <initializer_list>` implicitly whenever that header is needed, so you're not supposed to write any `#include <initializer_list>` directives, ever. However, not all implementations are fully compliant in this respect. Therefore, if you encounter compiler errors in array auto declarations, try to `#include <initializer_list>` manually.

Declaring an auto array of pointers to functions (and pointers to member functions) follows the same principles:

```
int func1();
int func2();
auto pf_arr={func1, func2};//array  int (*[2])()
int (A::*pmf) (int)=&A::f; //pointer to member
auto pmf_arr={pmf,pmf}; //array int (A::*[2]) (int)
```

Reference variables are a slightly different case. Unlike with pointers, when the initializer is a reference type, auto still defaults to value semantics:

```
int& f();
auto x=f(); //x is int, not int&
```

To declare a reference variable using auto, add the & to the declaration explicitly:

```
auto& xref=f(); //xref is int&
```

Technically, you can declare multiple entities in a single auto expression:

```
auto x=5, d=0.54, flag=false, arr={1,2,3}, pf=func;
```

Operator decltype

The new operator decltype captures the type of an expression. You can use decltype to store the type of an object, expression, or literal value. In this sense, decltype is the complementary operation of auto – whereas auto instructs the compiler to fill-in the omitted type, decltype "uncovers" the type of an object:

```
auto x=0;  //int
decltype (x)  z=x; //same as auto z=x;
typedef decltype (x) XTYPE; //XTYPE is a synonym for int
```

```
XTYPE y=5;
```

How does it work? The expression `decltype(x)` extracts the type of `x`, which is int. That type is assigned to the new typedef name XTYPE. XTYPE is then used in the declaration of a new variable, `y`, which has the same type as `x`.

Let's look at a more realistic example in which decltype extracts the iterator type from a generic expression. Suppose you want to capture the iterator type returned from a `vector<T>::begin()` call. You can do it like this:

```
vector<Foo> vf;
typedef decltype(vf.begin()) ITER;
for (ITER it=vf.begin(); it<vf.end(); it++)
   cout<<*it<<endl;
```

Notice that decltype takes an expression as its argument. Therefore, you can apply decltype not just to objects and function calls, but also to arithmetic expressions, literal values, and array objects:

```
typedef decltype(12) INT;  //a literal value
float f;
typedef decltype(f) FLOAT; //a variable
typedef decltype (std::vector<int>()) VI; //a temporary
typedef decltype(4.9*0.357) DOUBLE; //a math expression
auto arr={2,4,8};
typedef decltype(arr) int_array;//an array
int_array table;
table[0]=-10;
```

C++11 Style Function Declarations

C++11 introduced a new notation for declaring a function using auto and decltype. Recall that a traditional function declaration looks like this:

```
return_type name(params,...); //old style function declaration
```

In the new function declaration notation, the return type appears after the closing parenthesis of the parameter list and is preceded by a `->` sign. For example:

```
auto newfunc(bool b)->decltype(b)//C++11 style function declaration
{ return b;}
```

`newfunc()` takes a parameter of type bool and returns bool. This style is useful in template functions where the return type depends on the *template-id* (the actual template instance). Thus, template functions can define a generic return type using the new notation and decltype. The actual return type is calculated automatically, depending on the type of the decltype expression:

```
//return type is vector<T>::iterator
template <class T> auto get_end (vector<T>& v) ->decltype(v.end());
{
   return v.end();
}
vector<Foo> vf;
get_end(vf); //returns vector<Foo>::iterator
const vector<Foo> cvf;
get_end(cvf); //returns vector<Foo>::const_iterator
```

The new function declaration notation overcomes parsing ambiguity problems that were associated with the traditional function declaration notation. Additionally, this notation is easier to maintain because the actual return type isn't hardcoded in the source file. As a bonus, it is said to be more readable.

In Conclusion

auto and decltype fill in a gap that, until recently, forced programmers to use hacks and non-standard extensions, such as the GCC-specific typeof() operator. auto doesn't just eliminate unnecessary keystrokes. Rather, it also simplifies complex declarations and improves the design of generic code. decltype complements auto by letting you capture the type of complex expressions, literal values, and function calls without spelling out (or even knowing) their type. Together with the new function declaration notation, auto and decltype make a C++11 programmer's job a tad easier.

C++11 Tutorial: Lambda Expressions — The Nuts and Bolts of Functional Programming

November 1, 2012 by Danny Kalev

One highlight of C++11 is lambda expressions: function-like blocks of executable statements that you can insert where normally a function call would appear. Lambdas are more compact, efficient, and secure than function objects. Danny Kalev shows you how to read lambda expressions and use them in C++11 applications.

Originally, functional programming in C consisted of defining a full-blown functions and calling them from other translation units. Pointers to functions and file-scope(i.e., static) functions were additional means of diversifying function usage in C.

C++ improved matters by adding inline functions, member functions and function objects. However, these improvements, particularly function objects, proved to be labor intensive.

At last, the final evolutionary stage of functional programming has arrived in the form of lambda expressions.

Lambda by Example

Note: *According to the C++11 standard, implementations* `#include <initializer_list>` *implicitly when necessary. Therefore, you're not supposed to* `#include` *this header in your programs. However, certain compilers (GCC 4.7 for example) aren't fully compliant with the C++11 standard yet. Therefore, if the new initialization notation causes cryptic compilation errors, add the directive* `#include <initializer_list>` *to your code manually.*

Before discussing the technicalities, let's look at a concrete example. The last line in the following code listing is a lambda expression that screens the elements of a vector according to a certain computational criterion:

```
//C++11
vector <accountant> emps {{"Josh", 2100.0}, {"Kate", 2900.0},
{"Rose",1700.0}};
const auto min_wage = 1600.0;
const auto upper_limit = 1.5*min_wage;
//report which accountant has a salary that is within a specific range
std::find_if(emps.begin(), emps.end(),
```

```
[=](const accountant& a) {return a.salary()>=min_wage && a.salary() <
upper_limit;});
```

As all lambda expressions, ours begins with the *lambda introducer* `[]`. Even without knowing the syntactic rules, you can guess what this lambda expression does: The executable statements between the braces look like an ordinary function body. That's the essence of lambdas; they function (pun unintended) as locally-defined functions. In our example, the lambda expression reports whether the current accountant object in the vector `emps` gets a salary that is both lower than the upper limit and higher than the minimum wage.

Inline computations such as this are ideal candidates for lambda expressions because they consist of only one statement.

Dissecting a Lambda Expression

Now let's dissect the syntax. A typical lambda expression looks like this:

```
[capture clause] (parameters) -> return-type {body}
```

As said earlier, all lambdas begin with a pair of balanced brackets. What's inside the brackets is the optional *capture clause*. I get to that shortly.

The lambda's parameters appear between the parentheses. Although you can omit the parentheses if the lambda takes no parameters, I recommend you leave them, for the sake of clarity.

Lambda expressions can have an explicit return type that's preceded by a `->` sign after the parameter list (find out more about this new functionality in Let Your Compiler Detect the Types of Your Objects Automatically). If the compiler can work out the lambda's return type (as was the case in the first example above), or if the lambda doesn't return anything, you can omit the return type.

Finally, the lambda's body appears inside a pair of braces. It contains zero or more statements, just like an ordinary function.

Back to the `find_if()` call. It includes a lambda expression that takes `constaccountant&`. Where did this parameter come from? Recall that `find_if()` calls its predicate function with an argument of type `*InputIterator`. In our example,`*InputIterator` is `accountant`. Hence, the lambda's parameter is `constaccountant&`. The `find_if()` algorithm invokes the lambda expression for every `accountant` in `emps`.

Since our lambda's body consists of the following Boolean expression:

```
{return a.salary()>= min_wage && a.salary() < upper_limit;}
```

The compiler figures out that the lambda's return type is `bool`. However, you may specify the return type explicitly, like this:

```
[=](const accountant& a)->bool
{return a.salary()>= min_wage && a.salary() < upper_limit;});
```

Capture Lists

Unlike an ordinary function, which can only access its parameters and local variables, a lambda expression can also access variables from the enclosing scope(s). Such a lambda is said to have *external references*. In our example, the lambda accesses, or *captures*, two variables from its enclosing scope: `min_wage`and `upper_limit`.

There are two ways to capture variables with external references:

- Capture by copy
- Capture by reference

The capture mechanism is important because it affects how the lambda expression manipulates variables with external references.

At this stage I'm compelled to divulge another behind-the-scenes secret. Conceptually, the compiler transforms every lambda expression you write into a function object, according to the following guidelines:

- The lambda's parameter list becomes the parameter list of the overloaded`operator()`.
- The lambda's body morphs into the body of the overloaded `operator()`.
- The captured variables become data members of the said function object.

The compiler-generated function object is called the *closure object*. The lambda's capture clause thus defines which data members the closure will have, and what their types will be. A variable captured by copy becomes a data member that is a copy of the corresponding variable from the enclosing scope. Similarly, a variable captured by reference becomes a reference variable that is bound to the corresponding variable from the enclosing scope.

A *default capture* specifies the mechanism by which all of the variables from the enclosing scope are captured. A default capture by copy looks like this:

```
[=] //capture all of the variables from the enclosing scope by value
```

A default capture by reference looks like this:

```
[&]//capture all of the variables from the enclosing scope by reference
```

There is also a third capture form that I will not discuss here for the sake of brevity. It's used in lambdas defined inside a member function:

```
[this]//capture all of the data members of the enclosing class
```

You can also specify the capture mechanism for individual variables. In the following example `min_wage` is captured by copy and `upper_limit` by reference:

```
[min_wage, &upper_limit](const accountant& a)->bool
{return a.salary()>= min_wage && a.salary() < upper_limit;});
```

Finally, a lambda with an empty capture clause is one with no external references. It accesses only variables that are local to the lambda:

```
[] (int i, int j) {return i*j;}
```

Let's look at a few examples of lambdas with various capture clauses:

```
vector<int> v1={0,12,4}, v2={10,12,14,16};
      //read about the new C++11 initialization notation
[&v1](int k) {vi.push_back(k); };  //capture v1 by reference
[&]  (int m) {v1.push_back(m); v2.push_back(m) };//capture vi and vi2 by ref
[v1]() //capture v1 by copy
{for_each(auto y=v1.begin(), y!=v1.end(), y++) {cout<<y<<" ";}};
```

Name Me a Closure

In most cases, lambda expressions are ad-hoc blocks of statements that execute only once. You can't call them later because they have no names. However, C++11 lets you store lambda expressions in named variables in the same manner you name ordinary variables and functions. Here's an example:

```
auto factorial = [](int i, int j) {return i * j;};
```

This `auto`-declaration defines a *closure type* named `factorial` that you can call later instead of typing the entire lambda expression (a closure type is in fact a compiler-generated function class):

```
int arr{1,2,3,4,5,6,7,8,9,10,11,12};
long res = std::accumulate(arr, arr+12, 1, factorial);
cout<<"12!="<<res<<endl; // 479001600
```

On every iteration, the `factorial` closure multiplies the current element's value and the value that it has accumulated thus far. The result is the factorial of 12. Without a lambda, you'd need to define a separate function such like this one:

```
inline int factorial (int n, int m)
{
   return n*m;
}
```

Now try to write a factorial function object and see how many more keystrokes that would require compared to the lambda expression!

> ***Tip**: When you define a named closure, the compiler generates a corresponding function class for it. Every time you call the lambda through its named variable, the compiler instantiates a closure object at the place of call. Therefore, named closures are useful for reusable functionality (factorial, absolute value, etc.), whereas unnamed lambdas are more suitable for inline ad-hoc computations.*

In Conclusion

Unquestionably, the rising popularity of functional programming will make lambdas widely-used in new C++ projects. It's true that lambdas don't offer anything you haven't been able to do before with function objects. However, lambdas are more convenient than function objects because the tedium of writing boilerplate code for every function class (a constructor, data members and an overloaded `operator()`among the rest) is relegated to compiler. Additionally, lambdas tend to be more efficient because the compiler is able to optimize them more aggressively than it would a user-declared function or class. Finally, lambdas provide a higher level of security because they let you localize (or even hide) functionality from other clients and modules.

With respect to compiler support, Microsoft's Visual Studion 11 and GCC 4.5 support the most up-to-date specification of lambdas. EDG, Clang, and Intel's compilers also support slightly outdated versions of the lambda proposal.

Using constexpr to Improve Security, Performance and Encapsulation in C++

December 19, 2012 by Danny Kalev

`constexpr` is a new C++11 keyword that rids you of the need to create macros and hardcoded literals. It also guarantees, under certain conditions, that objects undergo static initialization. Danny Kalev shows how to embed `constexpr` in C++ applications to define constant expressions that might not be so constant otherwise.

The new C++11 keyword `constexpr` controls the evaluation time of an expression. By enforcing compile-time evaluation of its expression, `constexpr` lets you define true constant expressions that are crucial for time-critical applications, system programming, templates, and generally speaking, in any code that relies on compile-time constants.

NOTE: I use the phrases compile-time evaluation and static initialization in this article interchangeably, as well as dynamic initialization and runtime evaluation, respectively.

Timing is Everything

Before discussing `constexpr`, I need to clarify the difference between traditional`const` and the new `constexpr`.

As we all know, `const` guarantees that a program doesn't change a variable's value. However, `const` doesn't guarantee which type of initialization the variable undergoes. For instance, a `const` variable initialized by a function call requiresdynamic initialization (the function is called at runtime; consequently, the constant is initialized at runtime). With certain functions, there's no justification for this restriction because the compiler can evaluate the function call statically, effectively replacing the call with a constant value. Consider:

```
const int mx = numeric_limits<int>::max(); // dynamic initialization
```

The function `max()` merely returns a literal value. However, because the initializer is a function call, `mx` undergoes dynamic initialization. Therefore, you can't use it as a *constant expression*:

```
int arr[mx]; //compilation error: "constant expression required"
```

A similar surprise occurs when the initializer of a class's `const static` data member comes "too late":

```
struct S
{
   static const int sz;
};
const int page_sz = 4 * S::sz; //OK, but dynamic initialization
const int S::sz = 256; //OK, but too late
```

Here the problem is that the initializer of `S::sz` appears after the initialization of <`page_sz`. Consequently, `page_sz` undergoes dynamic initialization. That isn't just slower than static initialization; it also disqualifies `S::sz` from being used as a*constant integral expression,* as the following example shows:

```
enum PAGE
{
   Regular=page_sz, //compilation error: “constant expression required”
   Large=page_sz*2 //compilation error: “constant expression required”
};
```

Problems with Pre-C++11 Workarounds

In pre-C++11 code, the common workaround for the late function call problem is a macro. However, macros are usually a bad choice for various reasons, including the lack of type-safety and debugging difficulties. C++ programmers thus far were forced to choose between code safety (i.e., calling a function and thereby sacrificing efficiency) and performance (i.e., using type-unsafe macros).

With respect to a `const static` class member, the common workaround is to move the initializer into the class body:

```
struct S
{
   static const int sz=256;
};
const int max_sz = 4 * S::sz; // static initialization
enum PAGE
{
   Regular=page_size, //OK
   Large=page_size*2 //OK
};
```

However, moving the initializer into the class body isn't always an option (for example, if `S` is a third-party class).

Another problem with `const` is that not all programmers are aware of its subtle rules of static and dynamic initialization. After all, the compiler usually doesn't tell you which type of initialization it uses for a `const` variable. For example, `mx` seems to be a constant expression when it isn't.

The aim of `constexpr` is to simplify the rules of creating constant expressions by guaranteeing that expressions declared `constexpr` undergo static initialization when certain conditions are fulfilled.

Constant Expression Functions

Let's look at `numeric_limits<int>::max()` again. My implementation defines this function like this:

```
#define INT_MAX (2147483647)
class numeric_limits<int>
{
public:
   inline static inline int max () { return INT_MAX; }
};
```

Technically, a call to `max()` could make a perfect constant expression because the function consists of a single return statement that returns a literal value. C++11 lets you do exactly that by turning `max()` into a *constant expression function*. A constant expression function is one that fulfills the following requirements:

- It returns a value (i.e., it isn't `void`).
- Its body consists of a single statement of the form

  ```
  return exp;
  ```

  where `exp` is a constant expression.

- The function is declared `constexpr`.

Let's examine a few examples of constant expression functions:

```
constexpr int max()
{
   return INT_MAX;
} //OK
constexpr long long_max()
{
   return 2147483647;
} //OK
```

```
constexpr bool get_val()
{
   bool res=false;
   return res;
} //error, body isn't just a return statement
```

Put differently, a `constexpr` function is a named constant expression with parameters. It's meant to replace macros and hardcoded literals without sacrificing performance or type safety.

`constexpr` functions guarantee compile-time evaluation so long as their arguments are constant expressions, too. However, if any of the arguments isn't a constant expression, the `constexpr` function may be evaluated dynamically. Consider:

```
constexpr int square(int x)
{
   return x * x;
} //OK, compile time evaluation only if x is a constant expression
const int res=square(5); //compile-time evaluation of square(5)
int  y=getval();
int n=square(y);//dynamic evaluation of square(y)
```

The automatic defaulting to dynamic initialization lets you define a single`constexpr` function that accepts both constant expressions and non-constant expressions. You should also note that, unlike ordinary functions, you can't call a constant expression function before it's defined.

Constant Expression Data

A *constant-expression value* is a variable or data member declared `constexpr`. It must be initialized with a constant expression or an rvalue constructed by a constant expression constructor with constant expression arguments (I discuss constant expression constructors shortly).
A `constexpr` value behaves as if it was declared `const`, except that it requires initialization before use and its initializer must be a constant expression. Consequently, a `constexpr` variable can always be used as part of another constant expression. For example:

```
struct S
{
private:
   static constexpr int sz; // constexpr variable
public:
   constexpr int two(); //constexpr function
};
```

```
constexpr int S::sz = 256;
enum DataPacket
{
   Small=S::two(), //error. S::two() called before it was defined
   Big=1024
};
constexpr int S::two() { return sz*2; }
constexptr S s;
int arr[s.two()]; //OK, s.two() called after its definition
```

Constant Expression Constructors

By default, an object with a nontrivial constructor undergoes dynamic initialization. However, under certain conditions, C++11 lets you declare a class's constructor`constexpr`. A `constexpr` constructor allows the compiler to initialize the object at compile-time, provided that the constructor's arguments are all constant expressions.

Formally, a *constant expression constructor* is one that meets the following criteria:

- It's declared `constexpr` explicitly.
- It can have a member initialization list involving only potentially constant expressions (if the expressions used aren't constant expressions then the initialization of that object will be dynamic).
- Its body must be empty.

An object of a user-defined type constructed with a constant expression constructor and constant expression arguments is called a *user-defined literal*. Consider the following class `complex`:

```
struct complex
{
   //a constant expression constructor
    constexpr complex(double r, double i) : re(r), im(i) { }//empty body
  //constant expression functions
   constexpr double real() { return re;}
   constexpr double imag() { return im;}
private:
   double re;
   double im;
};
constexpr complex COMP(0.0, 1.0); // creates a literal complex
```

As you can see, a constant expression constructor is a private case of a constant expression function except that it doesn't have a return value (because constructors don't return values). Typically, the memory layout of `COMP` is similar to that of an array of two `double`s. One of the advantages of user-defined literals with a small memory footprint is that an implementation can store them in the system's ROM. Without a `constexpr` constructor, the object would require dynamic initialization and therefore wouldn't be ROM-able.

As with ordinary constant expression functions, the `constexpr` constructor may accept arguments that aren't constant expressions. In such cases, the initialization is dynamic:

```
double x = 1.0;
constexpr complex cx1(x, 0); // error: x isn't a constant expression
const complex cx2(x, 1); //OK, dynamic initialization
constexpr double xx = COMP.real(); // OK
constexpr double imaglval=COMP.imag(); //OK, static init
complex cx3(2, 4.6); //dynamic initialization
```

Notice that if the initializer is a constant that isn't declared `constexpr`, the implementation is free to choose between static and dynamic initialization. However, if the initializer is declared `constexpr`, the object undergoes static initialization.

In Conclusion

`constexpr` is an effective tool for ensuring compile-time evaluation of function calls, objects and variables. Compile-time evaluation of expressions often leads to more efficient code and enables the compiler to store the result in the system's ROM. Additionally, `constexpr` can be used wherever a constant expression is required, such as the size of arrays and bit-fields, as an enumerator's initializer, or in the forming of another constant expression. With respect to compiler support, GCC 4.6, Intel's C++ 13, IBM's XLC++ 12.1, and Clang 3.1 already support`constexpr`.

Closer to Perfection: Get to Know C++11 Scoped and Based Enum Types

February 6, 2013 by Danny Kalev

C++ enum types pack a set of related constants in an intuitive and efficient user-defined type. Can you ask for more? With two new C++11 enhancements, namely scoped enums and based enums, the answer is "yes." Find out all about the recent facelift that C++11 enums underwent and learn how to refactor your code to benefit from the new enum features – without sacrificing performance or backward compatibility.

Enums are one of my favorite C++ features. They exemplify the notion of an efficient user-defined type without the heavy machinery of virtual functions, constructors, etc. (Compare C++ enums to other programming languages that still insist on using a full-blown class instead, and you'll see what I mean.)

Yet, traditional enum types aren't flawless. Throughout the past three decades, some of their limitations have irritated programmers and compilers alike. Three of the most frequently cited drawbacks of traditional enums are:

- Enumerator names are visible from their enum's enclosing scope.
- You cannot override the default underlying type of an enum programmatically.
- Enumerators convert to `int` automatically, no questions asked.

C++11 addresses these issues with revamped enumerations that give you tighter control over the scope, size, and implicit conversions of enum types. Let's look at these new features more closely, and examine how they can improve both our code quality and frustration level.

A Matter of Scope

Programmers often discover the hard way that enumerators in the same scope must all be distinct from each other and from other variable names. Otherwise, they might clash, as the following example shows:

```
enum Color {
   Bronze,
   Silver,
   Gold
};
enum Bullion
{
```

```
   Silver, //conflicts with Color's Silver
   Gold, //conflicts with Color's Gold Platinum
};
```

The enumerators `Silver` and `Gold` belong to different enum types and have different values. However, they clash because they're visible from their enclosing scope (unlike, say, members of a class). The common workaround, namely appending the enum name to every enumerator, evokes memories from the pre-namespace era:

```
enum Color
{
   BronzeColor,
   SilverColor,
   GoldColor
};
enum Bullion
{
   SilverBullion,
   GoldBullion,
   PlatinumBullion
};
```

However, cumbersome names aren't to everyone's taste. Besides, name conflicts could still occur if you use third-party code.

Declaring enum types in namespaces offers only a partial solution to the problem. A typical project often declares all of its constants and enum types under the same namespace; and frankly, who uses namespaces these days anyway?

Enter Scoped Enums

C++11 solves the scoping problem with a new category called *scoped enums*. A scoped enum looks exactly as a traditional enum except that the keyword `class`(or `struct` – the two keywords are interchangeable in this context) appears between the keyword `enum` and the *enum name*, as shown in the following example:

```
enum class  Color //C++11 scoped enum
{
   Bronze,
   Silver,
   Gold
```

```
};
enum struct Bullion //C++11 scoped enum
{
   Silver, //doesn’t conflict with Color’s Silver
   Gold, //ditto
   Platinum
};
```

Enumerators of a scoped enum require a qualified name when referred to from an enclosing scope:

```
Color col1=Bronze; //error, Bronze not in scope
Color col2=Color::Bronze; //OK
if ((col2==Color::Silver) || (col2==Color::Gold))//OK
//…
```

Strong Typing

Traditional enumerators aren’t strongly typed. They automatically convert to `int`:

```
int cointype= BronzeColor; //OK, unscoped enum
```

As opposed to their unscoped counterparts, scoped enums are strongly-typed. Implicit conversions to `int` (or any other integral types) are not permitted. To convert a scoped enumerator to `int`, you have to use an explicit cast:

```
int carcolor=Color::Silver; //error, implicit conversion to int not allowed
int carcolor=static_cast<int>(Color::Silver); //OK
```

One Size Doesn’t Fit All

All enum types are implemented as a built-in integral type known as the *underlying type*. In C, the underlying type is always `int`. In C++03, the underlying type is implementation-defined. An implementation is free to choose an arbitrary integral type (`char`, `short`, `int`, `long`, and their `unsigned` counterparts) as an enum’s underlying type. Furthermore, a C++ implementation is allowed to assign a different underlying type to different enum types. The only requirements in the C++03 standard are that the underlying type is an integral type that can represent all the enumerator values defined in the enumeration, and that the underlying type is not larger than `int`, unless the value of an enumerator cannot fit in an `int` or `unsigned int`.

Thus, a typical C++03 implementation may assign different underlying types to the following enum types:

```
//C++03
enum Bool {False, True}; // fits into char, signed char, unsigned char
enum RecSize {DefaultSize=1000000, LargeSize=2000000}; //underlying type is
int
```

Of course, an implementation may adhere to `int` as the sole underlying type regardless of the values of the enumerators. From my experience, that's what the majority of C++03 compilers in fact do.

The under-specification of an enum's underlying type in C++03 can be a problem if you need platform-independent sizes, e.g., when sending data across an HTTP connection, or when the data has to be stored compactly. In C++03, there's really nothing much that you can do to override the default underlying type – at least not in a standardized manner.

C++11, however, lets you specify the underlying type of an enum explicitly by using an *enum base*. An enum base declaration looks syntactically similar to a base class in a derived class declaration:

```
enum Bool: char {False, True}; //C++11 based enum
```

In the example above, the programmer specified `char` as the underlying type of`Bool`. Consequently, every enumerator of `Bool` occupies only one byte of memory. If the value of an enumerator cannot be represented by the underlying type, a compilation error occurs.

Selecting `char` as the underlying type of an enum type ensures among the rest that enumerators can be transmitted via HTTP without the onerous little-endian versus big endian byte reordering. Additionally, the compact size of one byte (instead of four) per enumerator implies that a large number of enum values can be stored efficiently in a smartphone's memory or a data file.

Of course, this doesn't mean that you should rush to specify `char` as the underlying type of your enum types – quite the contrary. As a rule, let your compiler assign the default underlying type unless you have a good reason to override it. Usually, the default underlying type is also the most efficient datatype for the target hardware.

Combining Features

Based enums aren't scoped by default. They simply have a fixed, user-specified underlying type. If you want the benefits of both scoped enums and based enums, combine the two features, like this:

```
enum class  Bool: char {False, True}; //C++11 scoped and based enum
int x=sizeof (Bool); //x=1
int  y =static_cast<int> (Bool::False); //y=0
++y;
```

Of course, traditional enums are still available in C++11 with the same semantics and scoping rules as before. Therefore, legacy code will not break when you upgrade your compiler. For example, a traditional enumerator will still convert to `int` implicitly even in C++11:

```
enum Direction (Up, Down};
int dir=Down; //OK in C++11 as well
```

Recall that you may use qualified enumerator names even with traditional enums (with scoped enums, you're obliged to use qualified names exclusively):

```
int dir=Direction::Down; //OK, unscoped enum
```

However, it's advisable to go through existing C++ code and refactor it, possibly replacing legacy enums with scoped enums; this shouldn't affect their underlying type. Likewise, if you need a fixed, platform-independent underlying type, add an enum base to your enum declarations.

In Conclusion

C++11 offers two new categories of enum types: scoped enums and based enums. Enumerators of a scoped enum require a qualified name such as `enumname::enumerator` when you refer to them from an enclosing scope. In addition, scoped enums are strongly-typed, so you must use an explicit cast to convert them to `int` if necessary. A based enum lets you specify its underlying type programmatically. With respect to compiler support, GCC 4.4, Intel's C++ 12, MSVC 11, IBM's XLC++ 12.1, Clang 2.9, Embarcadero C++ Builder, and others support the new C++11 enum features.

Use C++11 Inheritance Control Keywords to Prevent Inconsistencies in Class Hierarchies

April 11, 2013 by Danny Kalev

For more than 30 years, C++ got along without inheritance control keywords. It wasn't easy, to say the least. Disabling further derivation of a class was possible but tricky. To prevent users from overriding a virtual function in a derived class you had to lean over backwards. But not any more: Two new context-sensitive keywords make your job a lot easier. Here's how they work.

C++11 adds two inheritance control keywords: `override` and `final`. `override`ensures that an overriding virtual function declared in a derived class has the same signature as that of the base class. `final` blocks further derivation of a class and further overriding of a virtual function. Let's see how these watchdogs can eliminate design and implementation bugs in your class hierarchies.

Virtual Functions and `override`

A derived class can override a member function that was declared virtual in a base class. This is a fundamental aspect of object-oriented design. However, things can go wrong even with such a trivial operation as overriding a function. Two common bugs related to overriding virtual functions are:

- Inadvertent overriding.
- Signature mismatch.

First, let's analyze the *inadvertent overriding* syndrome. You might inadvertently override a virtual function simply by declaring a member function that accidentally has the same name and signature as a base class's virtual member function. Compilers and human readers rarely detect this bug because they usually assume that the new function is meant to override the base class's function:

```
struct A
{
   virtual void func();
};
struct B: A{};
struct F{};
struct D: A, F
{
   void func();//meant to declare a new function but
   //accidentally overrides A::func
```

```
};
```

Reading the code listing above, you can't tell for sure whether the member function`D::func()` overrides `A::func()` deliberately. It could be an accidental overriding that occurred because the parameter lists and the names of both functions are identical by chance.

A *signature mismatch* is a more commonplace scenario. It leads to the accidental creation of a new virtual function (instead of overriding an existing virtual function), as demonstrated in the following example:

```
struct G
{
   virtual void func(int);
};
struct H: G
{
   virtual void func(double); //accidentally creates a new virtual function
};
```

In this case, the programmer intended to override `G::func()` in class `H`. However, because `H::func()` has a different signature, the result is a new virtual function, not an override of a base class function. Not all compilers issue a warning in such cases, and those that do are sometimes configured to suppress this warning.

In C++11, you can eliminate these two bugs by using the new keyword `override`.`override` explicitly states that a function is meant to override a base class's virtual function. More importantly, it checks for signature mismatches between the base class virtual function and the overriding function in the derived classes. If the signatures don't match, the compiler issues an error message.

Let's see how `override` can eliminate the signature mismatch bug:

```
struct G
{
   virtual void func(int);
};
struct H: G
{
   virtual void func(double) override; //compilation error
};
```

When the compiler processes the declaration of `H::func()` it looks for a matching virtual function in a base class. Recall that "matching" in this context means:

- Identical function names.
- A `virtual` specifier in the first base class that declares the function.
- Identical parameter lists, return types (with one exception), cv qualifications etc., in both the base class's function and the derived class's overriding function.

If any of these three conditions isn't met, you get a compilation error. In our example, the parameter lists of the two functions don't match: `G::func()` takes `int` whereas `H::func()` takes `double`. Without the `override` keyword, the compiler would simply assume that the programmer meant to create a new virtual function in `H`.

Preventing the inadvertent overriding bug is trickier. In this case, it's the *lack* of the keyword `override` that should raise your suspicion. If the derived class function is truly meant to override a base class function, it should include an explicit `override`specifier. Otherwise, assume that either `D::func()` is a new virtual function (a comment would be most appreciated in this case!), or that this may well be a bug.

`final` Functions and Classes

The C++11 keyword `final` has two purposes. It prevents inheriting from classes, and it disables the overriding of a virtual function. Let's look at final classes first.

Certain classes that implement system services, infrastructure utilities, encryption etc., are often meant to be *non-subclassable:* The implementers don't want clients to modify those classes by means of deriving new classes from them. Standard Library containers such as `std::vector` and `std::list` are another good example of non-subclassable types. These container classes don't have a virtual destructor or indeed, any virtual member functions.

And yet, every now and then, programmers insist on deriving from `std::vector`without realizing the risks involved. In C++11, non-subclassable types should be declared `final` like this:

```
class TaskManager final{/*..*/};
class PrioritizedTaskManager: public TaskManager {
};  //compilation error: base class TaskManager is final
```

In a similar vein, you can disable further overriding of a virtual function by declaring it `final`. If a derived class attempts to override a `final` function, the compiler issues an error message:

```
struct A
{
```

```
   virtual void func() const;
};
struct B: A
{
   void func() const override final; //OK
};
struct C: B
{
   void func()const; //error, B::func is final
};
```

It doesn't matter whether `C::func()` is declared `override`. Once a virtual function is declared `final`, derived classes cannot override it.

Syntax and Terminology

I have thus far avoided two side issues pertaining to `override` and `final`. The first one is their unique location. Unlike virtual, inline, explicit extern, and similar function specifiers, these two keywords appear after the closing parenthesis of a function's parameter list, or (in the case of non-subclassable classes) after the class name in a class declaration.

The peculiar location of these keywords is a consequence of another unusual property: `override` and `final` aren't ordinary keywords. In fact officially, they aren't keywords at all. C++11 considers them as identifiers that gain special meaning only when used in the specific contexts and locations as I have shown. In any other location or context, they are treated as identifiers. Consequently, the following listing makes perfectly valid C++11 code:

```
//valid C++11 code
int final=0;
bool override=false;
if (override==true)
   cout<<"override is: "<<override<<endl;
struct D{} final;
struct A
{
   virtual bool func();
};
struct B:A
{
   bool func() override final;
};
```

It may seem surprising that `final` and `override` behave exactly like PL/1's context sensitive keywords (CSK). Since 1972, C and later C++ always avoided CSK, adhering instead to the reserved keywords approach.

So why did the committee make `final` and `override` an exception? The CSK choice was a compromise. Adding `override` and `final` as reserved keywords might have caused existing C++ code to break. If the committee had introduced new reserved keywords, they probably would have chosen funky strings such as`final_decl` or `_Override`, tokens that were less likely to clash with user-declared identifiers in legacy C++ code. However, no one likes such ugly keywords (ask C users what they think of C99's `_Bool` for example). That is why the CSK approach won eventually.

`override` and `final` become keywords in C++11, but only when used in specific contexts. Otherwise, they are treated as plain identifiers. The committee was reluctant to call `override` and `final` "context sensitive keywords" (which is what they truly are) though. Instead, they are formally referred to as "identifiers with special meaning." Special indeed!

In Conclusion

The two new context-sensitive keywords override and final give you tighter control over hierarchies of classes, ridding you of some irritating inheritance-related bugs and design gaffes. `override` guarantees that an overriding virtual function matches its base class counterpart. `final` blocks further derivation of a class or further overriding of a virtual function. With respect to compiler support, GCC 4.7, Intel's C++ 12, MSVC 11, and Clang 2.9 support these new keywords.

Danny Kalev is a certified system analyst and software engineer specializing in C, C++ Objective-C and other programming languages. He was a member of the C++ standards committee until 2000 and has since been involved informally in the C++ standardization process. Danny is the author of ANSI/ISO Professional Programmer's Handbook (1999) and The Informit C++ Reference Guide: Techniques, Insight, and Practical Advice on C++ (2005). He was also the Informit C++ Reference Guide. Danny earned an M.A. in linguistics, graduating summa cum laude, and is now pursuing his Ph.D. in applied linguistics.