By Danny Kalev

The Biggest Changes in C++11 (and Why You Should Care)

Develop | Posted June 20, 2011

By Danny Kalev

It's been 13 years since the first iteration of the C++ language. Danny Kalev, a former member of the C++ standards committee, expl 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." Ind C++11 has changed significantly. It now supports lambda expressions, automatic type deduction of objects, uniform initialization s delegating constructors, deleted and defaulted function declarations, `nullptr`, and most importantly, rvalue references — a featur 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" built-in garbage collector (GC), and that it probably wouldn't support multithreading because of the technical complexities involved defining a portable threading model. Thirteen years later, the new C++ standard, C++11, is almost complete. Guess what? It lacks a 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 n only change. The new standard builds on the decades of expertise and makes C++ even more relevant. As Rogers Cadenhead poin "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 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 followin 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 l body gets a reference to *Uppercase* so it can modify it. Without the ampersand, *Uppercase* would be passed by value. C++11 lambd 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 initializ 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 template

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 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 confusion.

C++11 offers a similar mechanism for capturing the type of an object or an expression. The new operator `decltype` takes an expre

"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 meml

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. D

functions have two advantages: They are more efficient than manual implementations, and they rid the programmer from the cho

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 construc

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 tha

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*. Rva
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 *p*
resources of the source object, leaving the source in an "empty" state. In certain cases where making a copy of an object is both ex
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. 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 thi

```
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, function object wrapper and more. With the approval of C++11, TR1 is officially incorporated into standard C++ standard, along wi 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 represents an execution thread, promises and futures, which are objects that are used for synchronization in a concurrent environ the async() function template for launching concurrent tasks, and the thread_local storage type for declaring thread-unique data. F

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

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 the 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 collecto

However, it does offer plenty of new features that will make C++ more secure, efficient (yes, even more efficient than it has been th

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 you will probably agree with Stroustrup: C++11 *does* feel like a new language — a much better one.

See also:

- C11: A New C Standard Aiming at Safer Programming
- C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator
- C++11 Tutorial: Lambda Expressions — The Nuts and Bolts of Functional Programming