Get to Know the New C++11 Initialization Forms

By Danny Kalev

Mar 28, 2012

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.

Related Resources

Like this article? We recommend

C++ Standard Library, The: A Tutorial and Reference, 2nd Edition

Learn More Buy

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
struct S1
{
explicit S1(int n, int m) : x(n),
private:
 int x, y;
};
S1 s(0,1); //object initializers
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 a
int c1[2]={0,2};
char c2[]="message";
//or you can use the more verbose
```

```
char c3[]={'m','e','s','s','a','g
struct S
{
 int a,b;
};
S s={0,1};
```


You can use parentheses to in
well. The parentheses in this c
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
```

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", "g
map<string, string> stars
```

```
 { {"Superman", "+1 (212) 545-7890"},
  {"Batman", "+1 (212) 545-0987"}};
double *pd= new double [3] {0.5, 1.2,
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 i
int *p{}; //initialized to nullptr
double d{}; //initialized to 0.0
char s[12]{}; //all 12 chars are initi
string s{}; //same as: string s;
char *p=new char [5]{}; // all five ch
```

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{
};
```

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
C(int y) : x(y) {} //overrides the cla
};
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_li
{
//implementation (a pair of pointers o
public:
constexpr initializer_list(const E*, c
constexpr initializer_list(const E*, i
constexpr int size() const; // no. of
constexpr const T* begin() const; // f
constexpr const T* end() const; // one
};
```

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 `double`s.
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.

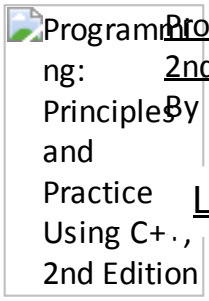
In Conclusion

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.

Danny Kalev *is a certified system analyst and software*

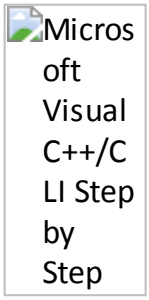
engineer specializing in C, C++ Objective-C and other programming languages. He was a member of the C++ standards committee until 2000 and has since been involved informally in the C++ standardization process. Danny is the author of ANSI/ISO Professional Programmer's Handbook *(1999) and* The Informit 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.*

You might also like:



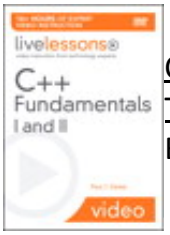
Programming: Principles and Practice Using C++, 2nd Edition
By Bjarne Stroustrup

Learn More



Microsoft Visual C++/CLI Step by Step
By Julian Templeman

Learn More



C++ Fundamentals I and II LiveLessons (Video Training)
By Paul Deitel

Learn More

Discussions

Comments for this thread are now closed.

Comments Community 1 Login

Recommend 3 Sort by Oldest

Itsthursdaythursday • 3 years ago

Great article, thanks; before the books come out it's nice to have such articles that let you learn incrementally. One nit, you put ';' after code braces in a couple of your class definitions.

2 • Share ›

nooneinparticular • 3 years ago

With respect to the initializer lists/sequence constructor, what about the following two cases:

a) An initializer list with different types which may or may not be convertible to a common type.

b) A class which has a sequence constructor *and* a constructor which takes two arguments and is initialized with {1, 2}. Which constructor is called? How do you force one over the other?

1 • Share ›

Danny Kalev ➦ nooneinparticular • 3 years ago

These are excellent questions.
With respect to an initializer list with heterogeneous types, say {1,0,2.5}, you'll get a compilation error because of a narrowing conversion from double to int (some compilers would issue a warning here, but that's also standard conforming because the standard merely requires a "diagnostic")

"diagnostic").

However, if the conversion is possible without narrowing, say {1,true, 0} then the conversion will take place, and the resulting initializer list will be {1,0,0}.
These are in essence the same rules that apply to template argument matching in general. There's nothing special about the class template initializer_list in that regard.
With respect to ambiguity, say a constructor that accepts {int, int} and an initializer list {1,1} the most likely result is a compilation error, similar to the one you'd get for using two overloaded functions with parameters that are too similar, say f(int) as opposed to f(const int).
In all these cases, the implementation is expected to issue clear diagnostics. To force one over the other you could use explicit casts, or better yet, redesign the class' constructors.

▲ | ▼ • Share ›

traveler • 3 years ago

@Danny Kalev :> However, if the conversion is possible without narrowing, say {1,true, 0} then the conversion will take place, and the resulting initializer list will be {1,0,0}.

Hmmm, I don't think that's right. E.g.:

```
auto x = { 1, true, 0 };
```

is an error, just like calling a template

```
template<class t=""> void f(T, T, T);
```

with f(1, true, 0) is an error.

▲ | ▼ • Share ›

Ivan Čukić ➔ traveler • 3 years ago

You're right, automatic type deduction can't work in that case. But when explicitly stating that something is

explicitly stating that something is std::vector < int > (as the author did), it will work.

Share ›

Danny Kalev • 3 years ago

Is this true that in MS Visual Sudio 2010 the

//C++11: default initialization using {}

is not implemented? The very elegant initialization :

char s[12]{}; //all 12 chars are initialized to '\0'
is not compiling in my MS environment, this is very sad. Is there anything perhaps that I am missing out like include files to be listed, name spaces etc?

Would I encounter more of such experiences in Mr Josutti's C++ Standard Library?
Thanks for answering a not so very ambitious question, regards, Harald

Share ›

Bengt Gustafsson • 3 years ago

As I understand the standard the compiler has to see if an initializer list can be interpreted as a initializer_list<t> for some T that the constructor needs. This means that the list {1, 2} is useful both for vector<int> and vector<double> as the