Subject: C++11
Posted by Mindtraveller on Wed, 12 Oct 2011 09:20:31 GMT
View Forum Message <> Reply to Message

http://en.wikipedia.org/wiki/C%2B%2B11
New version of C++ is finally out.
What do you think about it?
Will it change anything in how we use U++?

Subject: Re: C++11
Posted by mirek on Fri, 14 Oct 2011 10:18:09 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Wed, 12 October 2011 05:20http://en.wikipedia.org/wiki/C%2B%2B11
New version of C++ is finally out.
What do you think about it?
Will it change anything in how we use U++?

Unlikely. The closest thing that might have been useful for us is r-value references, which has the potential of replacing pick_. Unfortunately it does not compose, so directly replacing pick_ with && would break existing code. I do not know, MAYBE it would be worth it, after all there is only a couple of places where composition of pick is really used, OTOH fixing them would be pretty annoying.

What I mean by composition:

```
struct Foo {
  int a;
  Vector<int> foo;
};
```

in U++, this struct has well defined computer generated pick constructor and pick assignment.

Would pick_ be replaced by &&, it would have neither. Programmer would have to define special Foo(&&) constructor and operator=(&&). With some structs with dozen members, it would be tedious and error-prone.

Mirek

Subject: Re: C++11
Posted by lectus on Sat, 03 Dec 2011 11:03:08 GMT
View Forum Message <> Reply to Message

Interesting.
What compilers already support C++11?

Subject: Re: C++11
Posted by dolik.rce on Sun, 24 Jun 2012 07:51:48 GMT
View Forum Message <> Reply to Message

Hi everyone,

Allan McRae (one of the major devs of Arch Linux) wrote a series of nice and short articles about C++11 features. They're written for average joe programmer and I learned quite some new thing from them about the new standard. I really recommend them to anyone who haven't yet had the time yet to study C++11 changelist

From the features discussed in the first 5 articles, two would be IMHO useful for U++:

Initializer lists would be definitely a neat feature to have in U++ containers. We could than write things like "Vector v {10,3,12,8};" instead of "Vector v; v.Add(10); v.Add(3); v.Add(12); v.Add(8);". And if I understand correctly, we could also add some methods to do stuff like "v.Add({10,3,12,8});". The initializer_list works little bit similar to tuple, but it is syntactically much simpler and readable. It should be pretty easy to implement in U++, with backward compatibility assured by a flag.

Extern templates could speed up U++ compilation. The speedup should be significant for non-BLITZ case and probably noticeable even with BLITZ on. There might be however problems to implement a backward compatible usage of the extern templates...

As for the compiler support: GCC and Clang support is very good. I don't know about MSVC...

Best regards,
Honza

Subject: Re: C++11
Posted by mirek on Sun, 24 Jun 2012 10:50:54 GMT
View Forum Message <> Reply to Message

dolik.rce wrote on Sun, 24 June 2012 03:51We could than write things like "Vector v {10,3,12,8};" instead of "Vector v; v.Add(10); v.Add(3); v.Add(12); v.Add(8);". And if I understand correctly, we could

Well, you can also write, in "old" C++ and current U++

Vector v<int> = Vector<int>() << 10 << 3 << 12 << 8;

or

Vector v<int>;
v << 10 << 3 << 12 << 8;

which really is just a tiny bit more verbose (but right, it would run a bit slower than best C++11

implementation).

For me, the C++11 feature I like the most is 'auto'. Anyway, generally, I still have feeling that "polluting" U++ code with C++11 is not worth it for now. Perhaps in another 5 years...

Subject: Re: C++11
Posted by unodgs on Sun, 24 Jun 2012 11:02:36 GMT
View Forum Message <> Reply to Message

[quote title=mirek wrote on Sun, 24 June 2012 06:50]dolik.rce wrote on Sun, 24 June 2012 03:51
For me, the C++11 feature I like the most is 'auto'. Anyway, generally, I still have feeling that "polluting" U++ code with C++11 is not worth it for now. Perhaps in another 5 years...

I would also adjust u++ containers to work with C++11 foreach.

Subject: Re: C++11
Posted by mirek on Sun, 24 Jun 2012 11:13:37 GMT
View Forum Message <> Reply to Message

[quote title=unodgs wrote on Sun, 24 June 2012 07:02]mirek wrote on Sun, 24 June 2012 06:50dolik.rce wrote on Sun, 24 June 2012 03:51
For me, the C++11 feature I like the most is 'auto'. Anyway, generally, I still have feeling that "polluting" U++ code with C++11 is not worth it for now. Perhaps in another 5 years...

I would also adjust u++ containers to work with C++11 foreach.

Should not it work without adjusting? We do have begin/end defined already...

Mirek

Update: Confirmed

```
GUI_APP_MAIN
{
 Vector<int> v;
 v << 1 << 2 << 3;
 for(int& x: v)
  LOG(x);
}
```

Subject: Re: C++11
Posted by unodgs on Sun, 24 Jun 2012 11:21:45 GMT
View Forum Message <> Reply to Message

Excellent! (I thought some kind of new interface must be implemented)

Subject: Re: C++11
Posted by dolik.rce on Sun, 24 Jun 2012 14:24:43 GMT
View Forum Message <> Reply to Message

mirek wrote on Sun, 24 June 2012 13:13
```
GUI_APP_MAIN
{
 Vector<int> v;
 v << 1 << 2 << 3;
 for(int& x: v)
  LOG(x);
}
```
Or even with auto:
```
GUI_APP_MAIN
{
 Vector<int> v;
 v << 1 << 2 << 3;
 for(auto& x: v)
  LOG(x);
}
```

I agree that the intializer list syntax provides the same functionality as already existing code. The performance gain is not important because initialization shouldn't happen much in performance oriented code (where one should generaly reuse existing containers as much as possible). OTOH it is easy to read and I think the simplicity of it fits nice into U++. Also, the sooner people start encountering c++11 code in real world examples, the sooner it gets widely adopted, so maybe we could give a good example to the world It can't hurt, even if it stays semi-hidden under flag USECXX11 (or similar) for a first few years. This is just my opinion, and I don't force it to anyone... but I will probably starte experiment in this direction soon, and I will most probably try to show here what I can come up with

Honza

Subject: Re: C++11
Posted by lectus on Tue, 26 Jun 2012 23:29:07 GMT
View Forum Message <> Reply to Message

Interesting.
I was testing this stuff and I had to add -std=c++0x to the compiler options
Looks like a very clean way to iterate through a container.

Very handy indeed. I can say things like:

```
Vector<int> range(int x, int y) {
 Vector<int> v;
 for(int i=x; i<=y; i++)
  v.Add(i);
 return v;
}

...

for(auto x: range(0, 10))
  arrayCtrl1.Add(x);
```

Subject: Re: C++11
Posted by mirek on Thu, 28 Jun 2012 06:16:56 GMT
View Forum Message <> Reply to Message

BTW:

http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.a spx

---

Subject: Re: C++11
Posted by Lance on Sun, 02 Dec 2012 05:18:47 GMT
View Forum Message <> Reply to Message

For people who are interested in C++11, here is a pretty good book:

https://www.dropbox.com/s/pwpinrlme0hzhxp/%28Overview.of.the
.New.C.%EF%BC%9AC.0x%29.Scott.Meyers.pdf

Subject: Re: C++11
Posted by nlneilson on Sun, 02 Dec 2012 07:35:38 GMT
View Forum Message <> Reply to Message

lectus wrote on Sat, 03 December 2011 03:03Interesting.
What compilers already support C++11?

MSVC 2012 compilers support C++11/c++0x

Having the auto setup for those compilers would be good.

Subject: Re: C++11
Posted by Lance on Sun, 02 Dec 2012 14:08:26 GMT
View Forum Message <> Reply to Message

I think eventually U++ should get rid of pick_ and make use of rvalue reference instead. rvalue reference solves the same problem pick_ saught to solve and is standard compliant, and behaves more consistent across compilers: I believe pick_ is #define'd to different things on MSVC from on g++, and to avoid conflicts, U++ has to introduce a dummy parameter for deep copy semantics.

Last time when I mentioned this, Mirek said something like pick_ had more degree of automation; for the same purpose rvalue reference might involve more coding. But the benefit of switching might overwhelm the cost. C++ programmers turning to U++ will appreciate the effort saved for learning pick_ and will find the code easier to understand. U++ programmers don't have to speak a special dialect when there is no compelling reason for that.

The only problem IMHO is the resources needed to implement the switch. It's bound to take a lot of time and break a lot of user codes, unless somebody can write a parser to automate the process.

Subject: Re: C++11
Posted by dolik.rce on Sun, 02 Dec 2012 15:04:20 GMT
View Forum Message <> Reply to Message

Lance wrote on Sun, 02 December 2012 15:08I think eventually U++ should get rid of pick_ and make use of rvalue reference instead. rvalue reference solves the same problem pick_ saught to solve and is standard compliant, and behaves more consistent across compilers: I believe pick_ is #define'd to different things on MSVC from on g++, and to avoid conflicts, U++ has to introduce a dummy parameter for deep copy semantics.
Hi Lance,

If I understand both picking and rvalue references correctly, there is still a very valid reason to keep using picking: Rvalue references work ONLY on temporary objects. In U++, you can use picking on any object and even reuse it, assuming you clean it up after it has been picked, typically by calling Clear() or by assigning content from another object. I think this can not be done simply with rvalues.

Also, I believe that introduction of move semantics to C++11 standard will actually make it easier to explain U++ pick to new programmers, because they will already be familiar with the concept. We should just adapt the introductory documentation to explain how pick constructor is similar to move constructor, what is different and how U++ further extends this concept.

Honza

EDIT: After bit more reading, I found you can actually make it work with any object, using std::move... Right now, that seems downright ugly and hackish to me I guess I need to read even more before I can make my mind whether the switch to move semantics is a good or bad idea.

Subject: Re: C++11
Posted by Lance on Sun, 02 Dec 2012 17:05:18 GMT
View Forum Message <> Reply to Message

Honza, you are right. A named object has to be std:move()'ed to communicate its moveable (temporary) status to the compiler, even when it's declared as temporary (using &&). std::move will appear a lot in new library code and user code as well, so people will get use to it soon.

You can do in move assignments or move constructors the same thing you used to do in pick constructor and pick assignment, eg, mark the right hand object as picked.

Now we can do

Vector<int> b=std:move(anotherVectorIntInstance); // move construction;

//or
Vector<int> b=anotherVectorIntInstance; //copy construction or deep copy, instead of
Vector<int> b(anotherVectorIntInstance, 0); // using dummy param to signal the intention of deep copy.

provide the underlying U++ facilities are modified using the rvalue reference language feature.

Subject: Re: C++11
Posted by Lance on Sun, 02 Dec 2012 17:47:32 GMT
View Forum Message <> Reply to Message

And any object can be std::move()'ed to become moveable, no matter it's a temporary or not.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <new>

using namespace std;

struct S
```

```
{
 S(){}
 S(const S& rhs) : text(rhs.text){}
 S(S&& rhs){
  struct T{ char _[sizeof(S)]; };
  std::swap(reinterpret_cast<T&>(*this),
   reinterpret_cast<T&>(rhs)
  );
 }

 S& operator=(const S& rhs)
 {
  this->~S();
  return *new(this)S(rhs);
 }

 S& operator=(S&& rhs)
 {
  this->~S();
  // std::move() is necessary even when rhs is declared as temoprary
  return *new(this)S(move(rhs));
 }

 string text;

};
ostream& operator<<(ostream& os, const S& s)
{
 return os<<s.text<<endl;
}



S global;



int main()
{
 global.text="This is the global text";
 cout<<global;

 S a;
 a.text="This is object a";

 //
 //
 S b(a);
```

```
 cout<<"after S b(a), now <b> is:"<<b<<"and <a> is:"<<a;

 S c(std::move(global)); // use move constructor
 cout<<"after S c=std::move(global), now <c> is:"<<c<<"and <global> is:"<<global;

}
```

Subject: Re: C++11
Posted by Lance on Sun, 02 Dec 2012 18:30:49 GMT
View Forum Message <> Reply to Message

If U++ will adopt rvalue references, will it create the possibility of getting rid of Vector and many other containers/algorithms? Mirek will know better. My feeling is that in many a situation, the answer probably is 'yes'. If that is true, moving code into and out of U++ could be greatly simplified.

Subject: Re: C++11
Posted by dolik.rce on Sun, 02 Dec 2012 19:12:45 GMT
View Forum Message <> Reply to Message

Lance wrote on Sun, 02 December 2012 19:30If U++ will adopt rvalue references, will it create the possibility of getting rid of Vector and many other containers/algorithms? Mirek will know better. My feeling is that in many a situation, the answer probably is 'yes'. If that is true, moving code into and out of U++ could be greatly simplified.Actually many of the containers are already compatibly with STL (search for STL_*_COMPATIBILITY in Core). So things wouldn't get much easier than they are now. Another thing is that there are not only differences in move semantics, but also in algorithms used, interfaces and possibly other, so there are still reasons to keep the U++ containers. Not even to mention that you have to think about backwards compatibility

Honza

Subject: Re: C++11
Posted by Lance on Sun, 02 Dec 2012 20:00:08 GMT
View Forum Message <> Reply to Message

Hi Honza:

Thanks. I didn't know that.

It's probably better not to touch it at all, or add some additional interfaces to incorporate rvalue reference so that existing code doesn't break. But in that way copy construction/assignment still means pick_, and for deep copy, so that a class moving into U++ may have to revise its related constructors/= to behave correctly in U++.

A class not written with pick_ in mind will(or may?) not work correctly with Upp::Vector<>, even if it meets all the interface requirements superficially.

I agree benefit gain doesn't seem to justify the work involved and problems it may created.

Subject: Re: C++11
Posted by mirek on Mon, 03 Dec 2012 10:04:29 GMT
View Forum Message <> Reply to Message

Lance wrote on Sun, 02 December 2012 15:00
A class not written with pick_ in mind will(or may?) not work correctly with Upp::Vector<>, even if it meets all the interface requirements superficially.

Actually, that is not true. pick_ is in no way related with generic requirement of Vector elements (only 2-3 methods require it).

Please note that pick semantics and Moveable are two very different things.

pick indeed is a variant of rvalue and it might have sense to replace it with r-value. Unfortunately, rvalue lacks composition rules, which means that it has to be reimplemented for any composite type, while pick generates compiler generated composite pick operations without problems. How much more code it would mean in practice is something that I plan to test is some branch in future. But e.g. for something like RichTxt::Para, it will be nasty.

Anyway, pick is not used for performance reasons in Vector. Moveable is. And that is still a bit ahead than pick/&&.

Pick is rather interface issue, allows you to pass objects from place to place without copying them (which do not even has to be available).

Mirek

Subject: Re: C++11
Posted by mirek on Mon, 03 Dec 2012 10:10:25 GMT
View Forum Message <> Reply to Message

Lance wrote on Sun, 02 December 2012 13:30If U++ will adopt rvalue references, will it create the possibility of getting rid of Vector and many other containers/algorithms?

Note: You can use most STL algorithms with U++ containers now, as long as elements satisfy STL requirements, and vice versa.

However, sometimes U++ algorithms are better suited for U++ concrete types (and, again, vice versa). For example, U++ Sort is fater than than std::sort for Vector<String>, but slower for std::vector<std::string> (~30%), as there are subtle choices in algorithm that can favor one or another set of types.

Mirek

Subject: Re: C++11
Posted by Lance on Mon, 03 Dec 2012 23:55:48 GMT
View Forum Message <> Reply to Message

Thanks Mirek. I got it.

So U++'s Vector would still outperform std::vector significantly for classes have move constructor and move assignment defined?

Subject: Re: C++11
Posted by mirek on Tue, 04 Dec 2012 07:08:03 GMT
View Forum Message <> Reply to Message

Lance wrote on Mon, 03 December 2012 18:55Thanks Mirek. I got it.

So U++'s Vector would still outperform std::vector significantly for classes have move constructor and move assignment defined?

Well, it depends on many factors, but generally yes.

The difference is that when expanding std::vector, the code still has to iterate through all elements and move them (albeit using &&). Vector simply performs memcpy on raw data.

Now it is possible that the std::vector iteration could in the end be optimized by compiler to something like memcpy, as long as compiler sees the inlined move contructor, in that case the performace should be similar. But in the end, Vector always does memcpy

Mirek

Subject: Re: C++11
Posted by Lance on Tue, 19 Mar 2013 02:24:04 GMT
View Forum Message <> Reply to Message

C++11 compiler support shootout: Visual Studio, GCC, Clang, Intel

http://cpprocks.com/c11-compiler-support-shootout-visual-stu dio-gcc-clang-intel/
