Subject: Re: Which is the biggest drawback of U++ "unpopuliarity"?
Posted by mirek on Sat, 26 Apr 2008 06:11:39 GMT
View Forum Message <> Reply to Message

tvanriper wrote on Fri, 25 April 2008 20:36
If I have it right, your primary concern with std:: involves its relatively terrible performance,

Well, not really. If am to put it in a very simple way, the main problem with std:: is that it makes you wish the C++ had garbage collector....

Quote:
If that's the concern, someone could potentially help you find a way to achieve the same performance you currently get with NTL, while using a more std::-like interface.

Well, what would be that? Something like these macros at the end of Core/topt.h?

```
// STL compatibility hacks

#define STL_INDEX_COMPATIBILITY(C) \
 typedef T            value_type; \
 typedef ConstIterator const_iterator; \
 typedef const T&     const_reference; \
 typedef int          size_type; \
 typedef int          difference_type; \
 const_iterator       begin() const        { return B::Begin(); } \
 const_iterator       end() const          { return B::End(); } \
 void                 clear()              { B::Clear(); } \
 size_type            size()               { return B::GetCount(); } \

#define STL_BI_COMPATIBILITY(C) \
 typedef T            value_type; \
 typedef ConstIterator const_iterator; \
 typedef const T&     const_reference; \
 typedef int          size_type; \
 typedef int          difference_type; \
 const_iterator       begin() const        { return Begin(); } \
 const_iterator       end() const          { return End(); } \
 void                 clear()              { Clear(); } \
 size_type            size()               { return GetCount(); } \
 typedef Iterator     iterator; \
 typedef T&           reference; \
 iterator             begin()              { return Begin(); } \
 iterator             end()                { return End(); } \

#define STL_MAP_COMPATIBILITY(C) \
```

```
 typedef T            value_type; \
 typedef ConstIterator const_iterator; \
 typedef const T&     const_reference; \
 typedef int          size_type; \
 typedef int          difference_type; \
 const_iterator       begin() const         { return B::Begin(); } \
 const_iterator       end() const           { return B::End(); } \
 void                 clear()               { B::Clear(); } \
 size_type            size()                { return B::GetCount(); } \
 typedef Iterator     iterator; \
 typedef T&           reference; \
 iterator             begin()               { return B::Begin(); } \
 iterator             end()                 { return B::End(); } \

#define STL_VECTOR_COMPATIBILITY(C) \
 typedef T            value_type; \
 typedef ConstIterator const_iterator; \
 typedef const T&     const_reference; \
 typedef int          size_type; \
 typedef int          difference_type; \
 const_iterator       begin() const         { return Begin(); } \
 const_iterator       end() const           { return End(); } \
 void                 clear()               { Clear(); } \
 size_type            size()                { return GetCount(); } \
 typedef Iterator     iterator; \
 typedef T&           reference; \
 iterator             begin()               { return Begin(); } \
 iterator             end()                 { return End(); } \
 reference            front()               { return (*this)[0]; } \
 const_reference      front() const         { return (*this)[0]; } \
 reference            back()                { return Top(); } \
 const_reference      back() const          { return Top(); } \
 void                 push_back(const T& x) { Add(x); } \
 void                 pop_back()            { Drop(); }
```

Quote:
I could, of course, be mistaken. I'm not completely clear on why you feel these are so incompatible... as perhaps I'm not 100% clear on your design goals, or I'm ignorant of the fundamental problem you see in std::.

The real trouble starts with the fact that you cannot use std::string as map keys. You cannot use any concrete class defined in std:: as element of any Vector flavor U++ container.

So far, the main "incompatibility complaint" was that "U++ guys seem to define their own containers and string". This is not easy to fix

I've read and re-read [url=http://www.ultimatepp.org/www$uppweb$vsstd$en-us.html]this page[/url], but I still can't quite see how U++ and std:: can be so incompatible that there's no hope of improving the std::-style system to the point of matching U++ performance.

Ah, but you could fix std::. But it is not likely to happen.

Moreover, adopting all U++ tricks into std:: would change its semantics and break existing code.

I only pose this idea because it feels to me like you and boost have similar goals. I could, of course, be wrong. I know, for example, that boost has less of an emphasis on performance and more of an emphasis on their idea of 'correctness', so you may differ significantly there. (This is certainly not to say you have no concern for 'correctness', but that you may have a slightly different idea of what is 'correct' from boost).

Oh, I have a very strong concern for 'correctness' - to the degree that I often rather break existing code by fixing some "incorrectness" in U++ Core.

Also, please, do not think I am not aware about boost or that I think these people are stupid. Of course not, boost is a very good effort and the code is pretty good.

I just feel U++ is not a good fit there. It is almost like suggesting boost to adopt Java

BTW: I mostly care about "optimality" with U++. If I would care about "popularity" more, I would certainly use another path and boost would be the part of it.

Quote:
Perhaps someone could submit an article to Dr. Dobb's Journal showcasing the use of Ultimate++; that's a fairly popular magazine, at least here in the United States (the CUJ folded to Dr. Dobb's a few years ago, sadly, or I would have recommended it instead).

I guess that would be much better idea:)

Mirek