Subject: Core chat... (reference counted pointers etc.)
Posted by mirek on Tue, 23 Oct 2007 21:48:56 GMT
View Forum Message <> Reply to Message

EDIT: by fudadmin - This topic is a pure gem and one day should go into documentation. I am making it sticky.

mdelfede wrote on Tue, 23 October 2007 15:47luzr wrote on Tue, 23 October 2007 19:45
Well, I think At is extremely effective way how to solve many problems. And similar issue existis even in the plain C

a[i++] = a[i];

it is simply something you have to care about...

Well, in your example, you can see that you're doing it wrong.
In theide bug, you can't.... or, at least, you can't if you don't know upp vector internals, what's normally the matter.

Wait a moment - this issue is quite rigirously documented. There is precisely defined which container methods invalidate references.

Quote:
Usually people see Vector<> as a blackbox, they don't know (at least, I didn't know) about invalid references caused by At().

http://www.ultimatepp.org/src$Core$Vector$en-us.html

Anyway, as long as you know what At does, I guess this is quite obvious - but I agree that it is a potential source of bugs. There are several gotachas in U++ like this; I guess we should have some document created...

Quote:
BTW, I still think that c++ standard is missing many useful constructs, one of the most useful ot them is the 'property' one, which borland introduced as a language extension in their compilers.
An object-oriented language without properties is an empty wine glass, I think

Just an example of it :

```
class C
{
  private:

  int iVal;

  void set_iVal(int _i) { i = _iVal; }
  int get_iVal(void) { return iVal; }

  public:

  __property i = { read = getIval, write = setIval };

};

C c ;

c.i = 5;  // calls set_Ival(5)
int a = c.i; // calls a = get_Ival()
```

That's what I call 'clean object-oriented language'.

I am still not quite sure why people insist on such verbose syntactic sugar (it is really nothing else).

IME/IMO, there is usually much more properties to set than get and setting properties is much more convenient U++/C++ way...

Mirek

Subject: Re: Core chat...
Posted by cbpporter on Wed, 24 Oct 2007 13:30:49 GMT
View Forum Message <> Reply to Message

luzr wrote on Tue, 23 October 2007 23:48
I am still not quite sure why people insist on such verbose syntactic sugar (it is really nothing else).

IME/IMO, there is usually much more properties to set than get and setting properties is much more convenient U++/C++ way...

Mirek

Lack of properties is one of my bigger complaints for C++ (lack of module support being on first place), and I consider them more than just syntactic sugar. I think they are better suited to express the abstract idea of a field/property than some function. And I don't see why you consider them verbose. But anyway, by the time C++ gets properties, I'm sure we'll all be 100 years old, so we can concentrate on the issue at hand (which wouldn't be solved by properties either).

Subject: Re: Core chat...
Posted by mirek on Wed, 24 Oct 2007 14:37:23 GMT
View Forum Message <> Reply to Message

---

cbpporter wrote on Wed, 24 October 2007 09:30And I don't see why you consider them verbose.

```
EditField e;
e.SetFont(Arial(20)).Password().InitCaps();
```

```
EditField e;
e.font = Arial(20);
e.password = true;
e.initcaps = true;
```

Mirek

Subject: Re: Core chat...
Posted by cbpporter on Wed, 24 Oct 2007 14:44:01 GMT
View Forum Message <> Reply to Message

I forgot about chaining. But it's not really used outside of U++ and properties would.

Subject: Re: Core chat...
Posted by mdelfede on Wed, 24 Oct 2007 15:46:28 GMT
View Forum Message <> Reply to Message

---

luzr wrote on Tue, 23 October 2007 23:48
I am still not quite sure why people insist on such verbose syntactic sugar (it is really nothing else).

IME/IMO, there is usually much more properties to set than get and setting properties is much more convenient U++/C++ way...

Well, if you think so, you can have object-oriented code in plain C also, look in GTK code to have just an example.... or even in assembler.
That doesn't mean 'clean object programming' in my way of thinking.
Please understand that I'm not criticizing UPP, I still think it's one of the best coded toolkits.
What I do criticise is the lack of some very useful constructs in C++ that could make code much cleaner to write and to mantain.

Constructs that don't cause loss of performance, as properties.

Quote:
Wait a moment - this issue is quite rigirously documented. There is precisely defined which container methods invalidate references.

Yes, all is documented somewhere, you did know that but... still the bug was there. In my thinking, for example, a function that has the main purpose of accessing array element should *not* as a side effect invalidate references. I know that write

```
a.SetSize(100);
a[99] = 10;
```

is one line longer than

```
a.At(99) = 10;
```

but in former case you clearly separate array dimensioning from accessing, in latter not. And more, you have a sure cause of nasty bugs on latter one. Such constructs are (IMHO) acceptable in dynamic languages such visual basic, not in C++.
You where complaining about shared ownership of objects (and you're right about it, if you use pointers), but in case of At() function you do have a case that is very similar, because you use [] operator that you think operates in an object that is invalidated by the At() function as a side effect.
Ok, it's documented, but then also realloc() is documented, but I've seen MANY times code like this :

```
int * a = malloc(100);
realloc(a, 200);
*a = 10;
```

and that's MUCH less subtle bug than yours with At().

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Wed, 24 Oct 2007 16:23:03 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Wed, 24 October 2007 11:46luzr wrote on Tue, 23 October 2007 23:48
I am still not quite sure why people insist on such verbose syntactic sugar (it is really nothing else).

IME/IMO, there is usually much more properties to set than get and setting properties is much more convenient U++/C++ way...

Well, if you think so, you can have object-oriented code in plain C also, look in GTK code to have just an example.... or even in assembler.
That doesn't mean 'clean object programming' in my way of thinking.
Please understand that I'm not criticizing UPP, I still think it's one of the best coded toolkits.
What I do criticise is the lack of some very useful constructs in C++ that could make code much cleaner to write and to mantain.
Constructs that don't cause loss of performance, as properties.

Well, in any case, GTK code is much more verbose than C++. U++ "modifiers" are less verbose than properties.

Quote:
Wait a moment - this issue is quite rigirously documented. There is precisely defined which container methods invalidate references.

Yes, all is documented somewhere, you did know that but... still the bug was there. In my thinking, for example, a function that has the main purpose of accessing array element should *not* as a side effect invalidate references.
[/quote]

I agree. However, the main purpose of At is not accessing array element, but to create one if it does not exist yet.

Quote:
 I know that write

```
a.SetSize(100);
a[99] = 10;
```

is one line longer than

```
a.At(99) = 10;
```

But that is not the correct equivalent.

Quote:
```
if(a.GetCount() < 100)
   a.SetCount(100)
a[99] = 10;
```

is the equivalent. OK, that would be something to live with, but consider composition:

```
Vector< Vector<int> > a;
a.At(10).At(20) = 10;
```

- that is the real reason to provide such utility method.

Quote:
You where complaining about shared ownership of objects (and you're right about it, if you use pointers), but in case of At() function you do have a case that is very similar, because you use [] operator that you think operates in an object that is invalidated by the At() function as a side effect.
Ok, it's documented, but then also realloc() is documented, but I've seen MANY times code like this :

```
int * a = malloc(100);
realloc(a, 200);
*a = 10;
```

and that's MUCH less subtle bug than yours with At().

Well, I guess there is always instrinsict conflict between performance, convenience and safety. "At", and in fact, the whole concept of "inplace creation" (instances get created in container, reference is returned) is nothing new, it is part of U++ design since the very beginning and proved to be very useful.

In any case, I believe I would make more bugs in "unrolling"

```
a.At(x).At(y) = ...
```

code than using such utility method.

Mirek

Subject: Re: Core chat...
Posted by sergei on Wed, 24 Oct 2007 17:49:56 GMT
View Forum Message <> Reply to Message

luzr wrote on Tue, 23 October 2007 23:48mdelfede wrote on Tue, 23 October 2007 15:47luzr wrote on Tue, 23 October 2007 19:45
Well, I think At is extremely effective way how to solve many problems. And similar issue existis

even in the plain C

a[i++] = a[i];

it is simply something you have to care about...

Well, in your example, you can see that you're doing it wrong.
In theide bug, you can't.... or, at least, you can't if you don't know upp vector internals, what's normally the matter.

Could someone please explain the workings of a[i++] = a[i]? I guess something here goes wrong because i++ probably gets evaluated earlier than rvalue (it it?), but I'm not sure why/what. And does this happen with Vector too (reference on left evaluated before rvalue)?

P.S. on-topic - I consider At() behaviour just fine, since if you don't want to allocate anything, just use [].

Subject: Re: Core chat...
Posted by mdelfede on Wed, 24 Oct 2007 18:06:45 GMT
View Forum Message <> Reply to Message

sergei wrote on Wed, 24 October 2007 19:49
Could someone please explain the workings of a[i++] = a[i]? I guess something here goes wrong because i++ probably gets evaluated earlier than rvalue (it it?), but I'm not sure why/what. And does this happen with Vector too (reference on left evaluated before rvalue)?

Well, I think that

a[i++] = a[i];

""should"" be equivalent to

a[i] = a[i+1];
i++;

but then, I was thinking also that in

a[i] = a.At(j)

the reference out of [] operator should come AFTER the right side At() function.... But I was wrong... at least for GCC.
As the stuff was ok with MSC, now I'm thinking that it's an undefined, compiler-dependent behaviour.

Ciao

Max

Subject: Re: Core chat...
Posted by mdelfede on Wed, 24 Oct 2007 18:24:03 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 24 October 2007 18:23
Well, I guess there is always instrinsict conflict between performance, convenience and safety.

Speaking about performance, I agree... it shouldn't traded with convenience. With safety.... hmmm... there I have some reserve.
But, in modern compilers, doing a 3 line construct or an equivalent 1 liner At() should be the same in terms of performance.
BTW, I really see few practical usage of At(); normally (about 100% of cases) you know in advance your needed array size, and, if you don't, I don't really see the point of increasing (and such copying the whole stuff) the array on a 1 element basis; se that example :

```
Array<int> a(1);
for(int i = 2, i < 1000; i++)
  a.At(i) = i;
```

of course that works, but that means, in the worst case of a badly written Array code, 998 realloc() calls, with 998 buffer copy, memory releases and allocations, ecc ecc.
In a better and more realistic case, if the array is grown in chunks of 10 elements, you'd still have to do it 98 times. That's no efficiency neither good code.
Of course there's a better solution (here I added 1 member functions to Array class) :

```
Array<int> a(1);
for(int i = 2, i < 1000; i++)
{
  a.CheckSize(i, 1000);
  a[i] = i;
}
```

Where the CheckSize(minSize, increment) is a member function that checks the actual/required size of array and, if too small, make it to grow of a specified size (here 1000).
That'd be an huge preformance gain with a small line of code added.

Quote:
 "At", and in fact, the whole concept of "inplace creation" (instances get created in container, reference is returned) is nothing new, it is part of U++ design since the very beginning and proved to be very useful.

In any case, I believe I would make more bugs in "unrolling"

a.At(x).At(y) = ...

code than using such utility method.

Here I'd let the At() the sole purpose of element accessor and put somewhere else the array resize. I don't see nothing bad on a.At().At().At(), as they don't change array sizes.

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Wed, 24 Oct 2007 19:46:23 GMT
View Forum Message <> Reply to Message

sergei wrote on Wed, 24 October 2007 13:49luzr wrote on Tue, 23 October 2007 23:48mdelfede wrote on Tue, 23 October 2007 15:47luzr wrote on Tue, 23 October 2007 19:45
Well, I think At is extremely effective way how to solve many problems. And similar issue existis even in the plain C

a[i++] = a[i];

it is simply something you have to care about...

Well, in your example, you can see that you're doing it wrong.
In theide bug, you can't.... or, at least, you can't if you don't know upp vector internals, what's normally the matter.

Could someone please explain the workings of a[i++] = a[i]? I guess something here goes wrong because i++ probably gets evaluated earlier than rvalue (it it?), but I'm not sure why/what. And does this happen with Vector too (reference on left evaluated before rvalue)?

The problem is that the order of evaluation is left to the compiler, it is undefined (by C/C++ standard). Therefore, such statement has undefined semantics.

Mirek

Subject: Re: Core chat...
Posted by mirek on Wed, 24 Oct 2007 20:05:56 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Wed, 24 October 2007 14:24luzr wrote on Wed, 24 October 2007 18:23
Well, I guess there is always instrinsict conflict between performance, convenience and safety.

But, in modern compilers, doing a 3 line construct or an equivalent 1 liner At() should be the same in terms of performance.

Nobody argues that...

Quote:
BTW, I really see few practical usage of At();

Well, in that case, just do not use it:)

Anyway, it is used about 100 times in the uppsrc. There are many cases when it is the right thing to do - if you do not want to duplicate the code.

Quote:
normally (about 100% of cases) you know in advance your needed array size

Wrong, in 99% you do not know the GetCount of resulting Vector or Array. Or you do not WANT to know it - I mean, sure, it is e.g. possible to count all elements in the file first, then create the array of the right size, the reopen the file and load them in. But load them in into dynamic array is much more simple...

Quote:
, and, if you don't, I don't really see the point of increasing (and such copying the whole stuff) the array on a 1 element basis; se that example :

```
Array<int> a(1);
for(int i = 2, i < 1000; i++)
  a.At(i) = i;
```

of course that works, but that means, in the worst case of a badly written Array code, 998 realloc() calls, with 998 buffer copy, memory releases and allocations, ecc ecc.

Well but that is completely different issue altogether... But be sure that Vector code is not badly written

Quote:
In a better and more realistic case, if the array is grown in chunks of 10 elements, you'd still have to do it 98 times. That's no efficiency neither good code.

Growing by static chunks is very stupid method. You always need exponential growth - this is the same for NTL and STL. In that case, the total number of copying stuff is amortized constant - both for STL and NTL (but for NTL, unlike STL, the copy of Vector element is performed by raw binary move, which can be much faster).

Quote:
Where the CheckSize(minSize, increment) is a member function that checks the actual/required size of array and, if too small, make it to grow of a specified size (here 1000).

Works well speed wise only for small array sizes - but for small ones, it wastes memory. What you really need is exponential growth.

Quote:
That'd be an huge preformance gain with a small line of code added.

Current At method is more optimal.

Quote:
Here I'd let the At() the sole purpose of element accessor and put somewhere else the array resize. I don't see nothing bad on a.At().At().At(), as they don't change array sizes.

Why should you duplicate operator[] with a method?

BTW, if you want to study easy to undestand practical examples of using At, look at ArrayCtrl::SetDisplay or Switch::Set.

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Wed, 24 Oct 2007 21:36:44 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 24 October 2007 22:05

Well, in that case, just do not use it:)

Impossible, if some library that I use do use it

Quote:

Well but that is completely different issue altogether... But be sure that Vector code is not badly written

I never thought that   That was only an example of how it can be made different. Your Vector::Checksize() can also be made working exponentially, just drop the second argument and change code inside it... even more easy to use.

a.CheckSize(i);
a[i] = i;

in previous example. The matter doesn't change. All you spare with At() is a line of code at a cost of the danger of hidden bugs.....

Quote:
Growing by static chunks is very stupid method. You always need exponential growth - this is the same for NTL and STL. In that case, the total number of copying stuff is amortized constant - both for STL and NTL (but for NTL, unlike STL, the copy of Vector element is performed by raw binary move, which can be much faster).

I told you that I'm not the boggest fan of STL

Quote:
Current At method is more optimal.

More than my example with linear/constant growth, ok.
But with exponential growth, all you spare is a line of code.

Quote:
Why should you duplicate operator[] with a method?

You shouldn't. If the only purpose of At() is allow creating elements on the fly just before accessing them, I see on it no true benefit, besides some 20 keystrokes less typing.

Quote:
BTW, if you want to study easy to undestand practical examples of using At, look at ArrayCtrl::SetDisplay or Switch::Set.

I'll look for it next days, thanx !

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Thu, 25 Oct 2007 03:33:15 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Wed, 24 October 2007 17:36
I never thought that   That was only an example of how it can be made different. Your Vector::Checksize() can also be made working exponentially, just drop the second argument and change code inside it... even more easy to use.

```
a.CheckSize(i);
a[i] = i;
```

in previous example. The matter doesn't change. All you spare with At() is a line of code at a cost of the danger of hidden bugs.....

Consider composition a.At(x).At(y)... It can become tedious.

Also, this is not the only place you need be aware of the problem:

```
a.Add() = a[0];
```

```
VectorMap<int, int> x;
x.GetAdd(10) = x[0];
```

are very similar cases.

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Thu, 25 Oct 2007 12:06:05 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 25 October 2007 05:33
Consider composition a.At(x).At(y)... It can become tedious.

yes, but finding At() kind of bugs can be even more tedious...
There you could write :

```
a.CheckSize(x).CheckSize(y);
a[x][y];
```

if [][] operators can be joined, if not

```
a.CheckSize(x).CheckSize(y);
a.At(x).At(y)
```

You need only CheckSize to return a reference to array a as usual in upp.

Quote:
Also, this is not the only place you need be aware of the problem:

a.Add() = a[0];

That could be solved with a construct like

a.Dup(0);

where Dup() should have an obvious function.or

a.Grow(1);
a.Last() = a[0]

for example, or something similar. The point is to avoid potentially dangerous cases.
Of course, all that cost something in term of code lines, (besides the Dup() example...) but nothing in terms of code speed, but avoids many possible caveats.
I *do not* criticize retourning references, that's needed for speed sakes, but I think a generic class like an array should make potentially dangerous constructs impossible.

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Thu, 25 Oct 2007 12:34:15 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Thu, 25 October 2007 08:06luzr wrote on Thu, 25 October 2007 05:33
Consider composition a.At(x).At(y)... It can become tedious.

yes, but finding At() kind of bugs can be even more tedious...

Well, sometimes finding bugs IS tedious. But this one was the first At related I had to spend more than 10 minutes.

Quote:
There you could write :

a.CheckSize(x).CheckSize(y);
a[x][y];

if [][] operators can be joined, if not

a.CheckSize(x).CheckSize(y);
a.At(x).At(y)

You need only CheckSize to return a reference to array a as usual in upp.

Hey, think about it a little bit more. To make what you suggest work, CheckSize has to return a reference to contained element. So it behaves exactly the same as At.

Quote:
Of course, all that cost something in term of code lines, (besides the Dup() example...) but nothing in terms of code speed, but avoids many possible caveats.

Actually, would be a bit slower, as you get the variable address evaluated in the At once. Your version evaluates it twice (and in composition example, access the outer container twice too).

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Thu, 25 Oct 2007 13:19:01 GMT

View Forum Message <> Reply to Message

luzr wrote on Thu, 25 October 2007 14:34
Well, sometimes finding bugs IS tedious. But this one was the first At related I had to spend more than 10 minutes.

That's because you're a guru of your UPP code, in my case that would have taken days, I guess.... I haven't seen it even when you pointed at the source....

Quote:
Hey, think about it a little bit more. To make what you suggest work, CheckSize has to return a reference to contained element. So it behaves exactly the same as At.

yes, of course, but in my way you can't do very few bad thins with that reference.... you could even write

a.CheckSize(100)[100] = 5;

whith no danger. Of course, you could even force it do do bad things such

a.CheckSize(100)[100] = a[10];

but then, you're forcing things to be buggy
BTW, I still think that a CheckSize() function should return nothing, to avoid such caveats..... I'd

rather

```
a.CheckSize(100);
a[100] = 5;
```

Quote:
Actually, would be a bit slower, as you get the variable address evaluated in the At once. Your version evaluates it twice (and in composition example, access the outer container twice too).

well, that depends of compiler code... usually "modern" compilers take care of avoiding double access when unneeded.

BTW, all that chat becomes question of personal taste... like commenting code. I usually prefere to write some more lines and have less "hidden" bugs possibilities.... as I usually comment about each code line in order to be able to know what I did even 1 year later. Other people like more to write a single-1000-chars line of code with no comment at all, and they can understand it even after 10 years.

I'd find more interesting a chat about reference counted objects.... I think I'll get back my old (and poorly written) array class, just to see the performance differences
It was done mainly to manage DispInterfaces arrays in a COM app managing Autocad, so no great speed requirements, but I guess it could be polished to be of some interest.

Just a last word, I *don't* think to propose you to change from pick_ to refcounted arrays (usually I hate when someone writes to me of changing core parts of my code !), I'm just curious of performance differences and possible caveats of this kind of solution.

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Thu, 25 Oct 2007 16:02:34 GMT

View Forum Message <> Reply to Message

mdelfede wrote on Thu, 25 October 2007 09:19
Quote:
Actually, would be a bit slower, as you get the variable address evaluated in the At once. Your version evaluates it twice (and in composition example, access the outer container twice too).

well, that depends of compiler code... usually "modern" compilers take care of avoiding double access when unneeded.

Only if everything is inlined

Quote:
Just a last word, I *don't* think to propose you to change from pick_ to refcounted arrays (usually I hate when someone writes to me of changing core parts of my code !), I'm just curious of performance differences and possible caveats of this kind of solution.

OK. I like diggin' in this stuff.

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Thu, 25 Oct 2007 17:52:12 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 25 October 2007 18:02

OK. I like diggin' in this stuff.

Uhmmm... what about if I try to reimplement your Vector<> class as a refcounted one ?
The comparaison would be quite fair, then. I looked at my old Array<> class and it was really unoptimized stuff.

Subject: Re: Core chat...
Posted by mirek on Thu, 25 Oct 2007 19:33:26 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Thu, 25 October 2007 13:52luzr wrote on Thu, 25 October 2007 18:02

OK. I like diggin' in this stuff.

Uhmmm... what about if I try to reimplement your Vector<> class as a refcounted one ?
The comparaison would be quite fair, then. I looked at my old Array<> class and it was really unoptimized stuff.

You can try. However, long time ago, such class template was part of U++. But there was no use for it. "pick" is confusing at first, but quite powerful concept.

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Thu, 25 Oct 2007 21:26:17 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 25 October 2007 21:33

You can try. However, long time ago, such class template was part of U++. But there was no use for it. "pick" is confusing at first, but quite powerful concept.

I do find "pick" quite clear. You get the top performance at at the expense of some loss of easy to use stuff.
With deepcopy behaviour, you have the opposite.... large performance loss with the gain of ease to use.
I think using refcount objects is a compromise between both, you have the same easy to use as deepcopy behaviour at the expense of a *small* performance loss in respect to your pick mechanics, that I guess is mostly due to a double indirection accessing elements, when you don't need a deep copy.
In your class :

```
array<int> a, b;
a.At(1000) = 1;
b = a;
a[10] = 2;  <==ERROR
b[10] = 2;  <==OK
```

in my class :

```
array<int> a, b;
a.At(1000) = 1;
b = a;      <== b and a share same memory area
a[10] = 2;  <== here an automatic deep copy, ok
b[10] = 2;  <== here no deep copy, just array access, ok
```

in usual deep copy behaviour :

```
array<int> a, b;
a.At(1000) = 1;
b = a;      <== deep copy
a[10] = 2;  <==OK
b[10] = 2;  <==OK
```

in 95% of cases, my array behaves exactly as yours, but... in the rest 5%, you DO have to specify

WithDeepCopyOption, in mine that's done automatically, with a 'small' performance penalty.
What I'm really curious about is how 'small' that is....

BTW, returning an array from a function, my class don't need a deep copy, as yours, as the origin array is released when function ends, leaving only the result reference to array.

```
array<int> a;
a = MyFunc(x);
```

array a gets 2 references to it for a while, just when MyFunc returns a value. Then, temporary from MyFunc get destroyed, leaving a with a single reference. Any subsequent access to a[] don't need a deep copy.

Max

Subject: Re: Core chat...
Posted by mirek on Thu, 25 Oct 2007 21:38:04 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Thu, 25 October 2007 17:26luzr wrote on Thu, 25 October 2007 21:33

You can try. However, long time ago, such class template was part of U++. But there was no use for it. "pick" is confusing at first, but quite powerful concept.

I do find "pick" quite clear. You get the top performance at at the expense of some loss of easy to use stuff.

Well, performance is nice, but really not that important.

What IS imporant is that there is only ONE PLACE where instance of (possibly) non-copyable object can exist.

Quote:

```
array<int> a, b;
a.At(1000) = 1;
b = a;      <== b and a share same memory area
a[10] = 2;  <== here an automatic deep copy, ok
b[10] = 2;  <== here no deep copy, just array access, ok
```

Note that above is impossible to implement reliably in C++ (as long as you want read operator[]

access to perform no copy at all).

Quote:
array a gets 2 references to it for a while, just when MyFunc returns a value. Then, temporary from MyFunc get destroyed, leaving a with a single reference. Any subsequent access to a[] don't need a deep copy.

Sure. Anyway, the real point of pick is here:

```
Array<Ctrl> CreateWidgets()
{
   Array<Ctrl> x;
   ...
   return x;
}
```

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Thu, 25 Oct 2007 21:47:46 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 25 October 2007 23:38

Well, performance is nice, but really not that important.

What IS imporant is that there is only ONE PLACE where instance of (possibly) non-copyable object can exist.

I don't get the point...
Quote:
Quote:

```
array<int> a, b;
a.At(1000) = 1;
b = a;      <== b and a share same memory area
a[10] = 2;  <== here an automatic deep copy, ok
b[10] = 2;  <== here no deep copy, just array access, ok
```

Note that above is impossible to implement reliably in C++ (as long as you want read operator[] access to perform no copy at all).

Yes, you got the true caveat of my way.... now maybe you understand *why* I do miss __property construct in c++....

Quote:

Sure. Anyway, the real point of pick is here:

```
Array<Ctrl> CreateWidgets()
{
   Array<Ctrl> x;
   ...
   return x;
}
```

uh ? My Array class behaves exactly as yours, here...

```
Array<Ctrl> CreateWidgets()
{
   Array<Ctrl> x;  <== here, a single reference to memory object
   ...
   return x; <== here, for a while, 2 references to THE SAME memory object
}
```

ctrls = CreateWidgets() <== here, the first reference is destroyed, leaving a single reference in ctrls

In your pick_ behaviour, you have a single reference ever to a single memory object. In my case, I have just for a while 2 references to a single memory object, then the first one is released leaving the same result as yours.

Max

Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 07:36:07 GMT

View Forum Message <> Reply to Message

mdelfede wrote on Thu, 25 October 2007 17:47
Quote:

Sure. Anyway, the real point of pick is here:

```
Array<Ctrl> CreateWidgets()
{
   Array<Ctrl> x;
   ...
   return x;
}
```

uh ? My Array class behaves exactly as yours, here...

```
Array<Ctrl> CreateWidgets()
{
   Array<Ctrl> x;  <== here, a single reference to memory object
   ...
   return x; <== here, for a while, 2 references to THE SAME memory object
}
```

ctrls = CreateWidgets() <== here, the first reference is destroyed, leaving a single reference in ctrls

In your pick_ behaviour, you have a single reference ever to a single memory object. In my case, I have just for a while 2 references to a single memory object, then the first one is released leaving the same result as yours.

Max

Sure. But then, what now:

ctrls[10].Create<Button>()

(To make me more clear, "COPY" of the "COPY ON WRITE" is impossible...)

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Fri, 26 Oct 2007 10:13:44 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 26 October 2007 09:36
Sure. But then, what now:

ctrls[10].Create<Button>()

(To make me more clear, "COPY" of the "COPY ON WRITE" is impossible...)

uhmmmm... no, you were not clear enough !
Looking on Array::Create<>() is defined as

```
 template<class TT> TT& Create()     { TT *q = new TT; Add(q); return *q; }
```

and no Create<>() is defined for Ctrl class. So I suppose that your ctrls is an array of array of controls... maybe. Your examples becomes difficult to understand...
So, you're taking the 11-th element of your array, wich is an array of controls, and add to it a new button, ok.
Let's start from the beginnin (I'm thinking as I write, sorry!):
with ctrls[10] you get a reference to 11-th array (element must exist, but I guess it's not important).
You add then a new button to it, ok. That should be equivalent to :

```
Array<Array<Ctrl>> a;
Array<Ctrl> b;
b.Create<Button>();
a.At(10) = b;  // just to be sure that 11-th element exists...
```

which in your pick behaviours leaves a owning b contents and b in picked state.
I still don't see caveats in my Copy-On-Write behaviour, besides of the impossibility of using [] operator because missing lvalue-rvalue signature difference.
If I do the same with my class, I have :

```
Array<Array<Ctrl>> a; // empty array of arrays of Ctrl
Array<Ctrl> b; // empty array of Ctrl
b.Create<Button>(); // no problem, a button is added to b
a.At(10) = b; // a[10] adds a reference to b object... no problem
```

then, I do have 2 possibilities :

```
b[0] = myCtrl; // don't look for a while at [] problem...
```

b becomes a DIFFERENT array as a[10], and a[10] points to an object that has a single reference
Or... b exits from its scope, leaving object at a[10] again with a single reference.
You can join all that on a single line, just replacing this damn' [] operator with some member function OR don't care about havng copy-on-write even on reads if array has more references.
Let's look at the second case (keeping []) :

```
Array<Array<Ctrl>> ctrls;
ctrls[10].Create<Button>()
```

the 11-th element of ctrls gets a button added, I don't see caveats. Where am I wrong ????

BTW, the *real* caveat of copy-on-write behaviour is the missing difference of signature between lvalues anr rvalues, which gives you 2 choices :

1- Make(if possible) operator[] returning a read-only element, and use some other function to set element value; that is the most efficient, which leads to true copy-on-write when array as more than 1 reference on it.
2- Let [] do the jobs, so you can have also copy-on-read if array has more references, which is a waste of time.
As I said before, C++ standard as *many* caveats and missing stuffs...

Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 10:42:48 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 26 October 2007 06:13luzr wrote on Fri, 26 October 2007 09:36
Sure. But then, what now:

ctrls[10].Create<Button>()

(To make me more clear, "COPY" of the "COPY ON WRITE" is impossible...)

uhmmmm... no, you were not clear enough !

Sorry. My mistake. Forget about [10]. Only

ctrls.Create<Button>();

or (to be more clear):

```
Array<Ctrl> a, b;
a.Create<Label>();
a.Create<EditField>();

b = a;

b.Create<Label>(); // Now what?!
```

Mirek

Subject: Re: Core chat...

Posted by mdelfede on Fri, 26 Oct 2007 11:03:54 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 26 October 2007 12:42
Sorry. My mistake. Forget about [10]. Only

```
ctrls.Create<Button>();
```

or (to be more clear):

```
Array<Ctrl> a, b;
a.Create<Label>();
a.Create<EditField>();

b = a;

b.Create<Label>(); // Now what?!
```

```
Array<Ctrl> a, b;  // two empty arrays, reference to nothing
a.Create<Label>(); // adds a Label to a, a becomes an array with a single reference at underlying data
a.Create<EditField>(); // adds an EditField to a, a is grown by 1 element, as it had a single reference to data, nothing strange happens.
b = a; // now a AND b have both reference to the same underlying array
b.Create<Label>(); // underlying data is copied ....
```

I see your point, here you *did* want a single array of controls, and you *did* want to destroy a when copying to b. You could have easily write :

```
Array<Ctrl> a;
a.Create<Label>();
a.Create<EditField>();

Array<Ctrl>&b = a;

b.Create<Label>();
```

That leaves both a and b pointing to SAME array. You then could say that when a is destroyed, b points to nothing, that's true, but I can hardly imagine such a case. For example :

```
Array<Ctrl> MyFunc()
{
  Array<Ctrl> a;
  a.Create<Label>();
  a.Create<EditField>();

  Array<Ctrl>&b = a;

  b.Create<Label>();

  return b;
}

Array<Ctrl> c = MyFunc();
```

still works... just before a is destroyed (and thus b points to nothing...), c adds a reference to its contents, then it remains the sole owner of it when function ends.
Let me say that in my case it's clear what you pretends to do, in yours you must know pick behaviour, and still you have the possibility to write an erroneus a.Create<SomeControl>() AFTER a is picked. In my case, you can AND you get what you want.
You may still say that if you do a.Create<SomeControl>() you get an error, ok... and I can tell you that if you forget the & you have double controls on screen!

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 12:01:33 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 26 October 2007 07:03luzr wrote on Fri, 26 October 2007 12:42
Sorry. My mistake. Forget about [10]. Only

```
ctrls.Create<Button>();
```

or (to be more clear):

```
Array<Ctrl> a, b;
a.Create<Label>();
a.Create<EditField>();

b = a;

b.Create<Label>(); // Now what?!
```

Array<Ctrl> a, b; // two empty arrays, reference to nothing
a.Create<Label>(); // adds a Label to a, a becomes an array with a single reference at underlying data
a.Create<EditField>(); // adds an EditField to a, a is grown by 1 element, as it had a single reference to data, nothing strange happens.
b = a; // now a AND b have both reference to the same underlying array
b.Create<Label>(); // underlying data is copied ....

How is it copied? There is no copy operation for widgets. And it does not even have sense.

Quote:
[/code]
I see your point, here you *did* want a single array of controls, and you *did* want to destroy a when copying to b. You could have easily write :

Array<Ctrl> a;
a.Create<Label>();
a.Create<EditField>();

Array<Ctrl>&b = a;

b.Create<Label>();

That leaves both a and b pointing to SAME array.

Of course, but that is completely different thing.

In fact, yes, one of reasons to give away with reference counting for containers (or other "smart" approaches) is that you only seldem need to transfer the value of container at all.

OTOH, if you DO need such operation, then in most cases you either need exactly "pick" behaviour or you just do not care (function return value). Very seldom you need "deep copy" - in that case, you still have optional deep operations available.

And yes, sometimes you get "broken pick semantics" runtime assert. But I get 100 times more "invalid index" asserts than this one.

Quote:
 You then could say that when a is destroyed, b points to nothing, that's true, but I can hardly imagine such a case. For example :

Array<Ctrl> MyFunc()
{

```
  Array<Ctrl> a;
  a.Create<Label>();
  a.Create<EditField>();

  Array<Ctrl>&b = a;

  b.Create<Label>();

  return b;
}

Array<Ctrl> c = MyFunc();
```

still works...

But does not compile Your template needs deep public copy constructor for T, even if it is not actually used (because reference count is always 1 for COW ops).

Quote:
and I can tell you that if you forget the & you have double controls on screen!

Sounds like a possible solution, but in that case you need to solve the quite complex problem of polymorphic copy - and all that only to make flawed code working somehow.

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Fri, 26 Oct 2007 12:18:05 GMT

View Forum Message <> Reply to Message

Well, well... I give up !
I still think that refcounted array are very useful somewhere, but in case of widget arrays, they aren't.
And I still think that is better to have code that corrects your mistakes (refcounted...) OR displays your errors (pick_).
Going back to At() behaviour, that is one thing I really don't like!
BTW, I still don't see the point of

a.At(10) = aString;

on an empty array... What does a[2] becomes, for example ?
Empty string ? null object ? a default one ? Does At(n) initialize the n previous elements if the array was empty ?

Ciao

Max

p.s. Did you have time to give a look at OpenGL bug, or should I look at it ?

Subject: Re: Core chat...
Posted by mdelfede on Fri, 26 Oct 2007 16:47:27 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 26 October 2007 14:01
But does not compile Your template needs deep public copy constructor for T, even if it is not actually used (because reference count is always 1 for COW ops).

Quote:
and I can tell you that if you forget the & you have double controls on screen!

Sounds like a possible solution, but in that case you need to solve the quite complex problem of polymorphic copy - and all that only to make flawed code working somehow.

Well, thinking a bit more about it.... I have the solution : put dummy copy constructors in Ctrl class, that throw an exception or fail an assert... So :
1- You're sure that you don't use the copy-on-write for classes you don't want to.
2- You must not write some cumbersome polymorphic class copy constructor.
Copy constructors *are* there, so my previous example does work

So, the solution (with refcounted classes) is to displace the error from picked array to copy constructor

---

Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 20:09:55 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 26 October 2007 08:18Well, well... I give up !
I still think that refcounted array are very useful somewhere, but in case of widget arrays, they aren't.
And I still think that is better to have code that corrects your mistakes (refcounted...) OR displays your errors (pick_).
Going back to At() behaviour, that is one thing I really don't like!

BTW, I still don't see the point of

a.At(10) = aString;

on an empty array... What does a[2] becomes, for example ?
Empty string ? null object ? a default one ? Does At(n) initialize the n previous elements if the array was empty ?

Yes, it initializes non-existing elements to default value (String()). Also, there is another 2 parameter version that provides the intialization value.

Quote:
p.s. Did you have time to give a look at OpenGL bug, or should I look at it ?

I am sorry, I am not in Linux yet. Right now I am working hard to make U++ look even more native in Vista (and you need to be tough to use Vista for so long . In fact, this effort will provide more native Linux look too (the problem to solve is the visual appearence of DropList and DropChoice), so that will be the natural follow-up project.

Mirek

Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 20:12:55 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 26 October 2007 12:47luzr wrote on Fri, 26 October 2007 14:01
But does not compile Your template needs deep public copy constructor for T, even if it is not actually used (because reference count is always 1 for COW ops).

Quote:
and I can tell you that if you forget the & you have double controls on screen!

Sounds like a possible solution, but in that case you need to solve the quite complex problem of polymorphic copy - and all that only to make flawed code working somehow.

Well, thinking a bit more about it.... I have the solution : put dummy copy constructors in Ctrl class, that throw an exception or fail an assert... So :
1- You're sure that you don't use the copy-on-write for classes you don't want to.
2- You must not write some cumbersome polymorphic class copy constructor.
Copy constructors *are* there, so my previous example does work

So, the solution (with refcounted classes) is to displace the error from picked array to copy constructor

Now I am not quite sure if this is meant as joke or not

(Note that classes like Ctrl derive from NoCopy, which makes it copy constructor private to catch its misuse at compile time).

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Fri, 26 Oct 2007 21:09:57 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 26 October 2007 22:12
Now I am not quite sure if this is meant as joke or not

(Note that classes like Ctrl derive from NoCopy, which makes it copy constructor private to catch its misuse at compile time).

Well, it was a joke but a working one

Seriously speaking, those are 2 opposite ways of doing the job... pick_ and refcount way, each with its own goods and bads.
It would be obviously impossible to implement your Array as a refcounted one without rewriting 50% of upp code, I guess.

Comparing things, you're catching as runtime errors the assignements to picked arrays and as compile errors the copy of Ctrl objects. With refcounted, I'd catch as runtime errors the copy of Ctrls objects and I'd have no picked problems... at a small expense of code speed. Personally I'd prefere the refcounted way *if* c++ had a way to distinguish between lvalue and rvalue on [] operator. Being (for now) the opposite, your way may be better.

Ciao

Max

Subject: Re: Core chat...
Posted by mdelfede on Fri, 26 Oct 2007 21:13:37 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 26 October 2007 22:09
I am sorry, I am not in Linux yet. Right now I am working hard to make U++ look even more native in Vista (and you need to be tough to use Vista for so long . In fact, this effort will provide more native Linux look too (the problem to solve is the visual appearence of DropList and DropChoice), so that will be the natural follow-up project.

Well, I'll give it a try tomorrow... maybe it isn't an hard bug to find.

I've never tried Vista, but from what I hear, I think I'll stay with Ubuntu (and Win Xp just for Cad applications) !

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 21:51:29 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 26 October 2007 17:13
I've never tried Vista, but from what I hear, I think I'll stay with Ubuntu (and Win Xp just for Cad applications) !

I would like to too, but if you are going to develop GUI toolkit, you have to be tough...

Mirek

Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 21:55:23 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 26 October 2007 17:09With refcounted, I'd catch as runtime errors the copy of Ctrls objects and I'd have no picked problems... at a small expense of code speed.

Well, the expense is a little bit bigger than "small" too...

Reference counting means atomic increments and decrements with unstable conditions. These are very expensive to be called for each mutating operation...

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Sat, 27 Oct 2007 09:08:05 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 26 October 2007 23:55

I would like to too, but if you are going to develop GUI toolkit, you have to be tough... Smile

ehehehehe... I'd don't like to be in your place, now

Quote:

Well, the expense is a little bit bigger than "small" too...

Reference counting means atomic increments and decrements with unstable conditions. These are very expensive to be called for each mutating operation...

mhhhhhhh.... I'm not so sure about it. When I have a bit more time, I'll try to rewrite Array class using refcounts, so we can make some test about it. I find the matter very interesting.

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Sat, 27 Oct 2007 09:16:53 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Sat, 27 October 2007 05:08luzr wrote on Fri, 26 October 2007 23:55

I would like to too, but if you are going to develop GUI toolkit, you have to be tough... Smile

ehehehehe... I'd don't like to be in your place, now

Quote:

Well, the expense is a little bit bigger than "small" too...

Reference counting means atomic increments and decrements with unstable conditions. These are very expensive to be called for each mutating operation...

mhhhhhhh.... I'm not so sure about it. When I have a bit more time, I'll try to rewrite Array class using refcounts, so we can make some test about it. I find the matter very interesting.

Ciao

Max

So do I, but I have already measured it

And, BTW, better benchmark against Vector...

(BTW2, Array does not have that At or Add reference problem - it never invalidates references).

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Sat, 27 Oct 2007 11:06:35 GMT

View Forum Message <> Reply to Message

luzr wrote on Sat, 27 October 2007 11:16

So do I, but I have already measured it

And, BTW, better benchmark against Vector...

(BTW2, Array does not have that At or Add reference problem - it never invalidates references).

So I suppose your Array class is built on top of a sort of linked list... not a contiguous area, right ?

Ciao

Max

p.s.: just a small question about object ownership...
When you write

a.Add(aControl);

Who has the ownership of aControl ? Array a[] or who created aControl ?
In former case this :

OpenGLExample aControl;
a.Add(aControl);

Should be wrong; in the latter case this :

a.Create<OpenGLExample>();

should be wrong. As you did show the latter example on this thread, I suppose a[] has the ownership...

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Sat, 27 Oct 2007 12:50:33 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Sat, 27 October 2007 07:06luzr wrote on Sat, 27 October 2007 11:16

So do I, but I have already measured it

And, BTW, better benchmark against Vector...

(BTW2, Array does not have that At or Add reference problem - it never invalidates references).

So I suppose your Array class is built on top of a sort of linked list... not a contiguous area, right ?

Basically *implemented* as Vector<T*>...

Quote:
p.s.: just a small question about object ownership...
When you write

a.Add(aControl);

Who has the ownership of aControl ? Array a[] or who created aControl ?

Very well, getting to the real issues

If aControl is an instance of widget, such statement is simply impossible (because it requires some form of copy, which Ctrl lacks).

Quote:
In former case this :

OpenGLExample aControl;

a.Add(aControl);

Should be wrong; in the latter case this :

Yes. You cannot copy widgets.

Quote:

a.Create<OpenGLExample>();

should be wrong. As you did show the latter example on this thread, I suppose a[] has the ownership...

Yes, although the whole idea of "ownership" is a little bit moot here. The widgets is simply an element of the container, the "ownership" issue is simple and obvious...

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Sat, 27 Oct 2007 13:21:15 GMT
View Forum Message <> Reply to Message

I did ask the former question because I was lookin' inside MainWindow code.... Looking for the OpenGL bug.
But then I realized that Ctrl::Add() is quite different from Array::Add(), building an array of references instead of objects.

BTW, I still didn't find the bug there... The only thing I found (up to now) is shown in my code here :

```
 int zzz;
 MyAppWindow *win, *win2;
 win = new MyAppWindow;
 win2 = new MyAppWindow;
 OpenGLExample gl, gl2;
 gl.SetFrame(InsetFrame());
 gl2.SetFrame(InsetFrame());
 win->Add(gl.HSizePos(10, 10).VSizePos(10, 10));
 win2->Add(gl2.HSizePos(10, 10).VSizePos(10, 10));
 win->Sizeable().Zoomable();
 win2->Sizeable().Zoomable();

 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 0
 win->OpenMain();
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 1
 win2->OpenMain();
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 2
 delete win;
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 2 !!!
 delete win2;
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 2 !!!
    Ctrl::EventLoop();
```

If I suppress the lines :

```
 win->Add(gl.HSizePos(10, 10).VSizePos(10, 10));
 win2->Add(gl2.HSizePos(10, 10).VSizePos(10, 10));
```

The code works ok :

```
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 0
 win->OpenMain();
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 1
 win2->OpenMain();
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 2
 delete win;
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 1
 delete win2;
 zzz = Ctrl::GetTopCtrls().GetCount(); // zzz = 0
```

That works even if I leave both lines BUT OpenGLExample is *not* derived from GLControl.
I'd like to know if the bug is Linux-dependent or not... But I haven't an Ide setup on my win xp machine. Don't you have a bit time to test on windows ?

Back to refcounted objects. What about if Ctrl would be an object built with PIMPL idiom and refcounted ? You then could write :

```
aControl a;  // control is created
aControl b = a; // just reference to inner pimpl object is copied
```

or, what sound even better:

```
Vector<Ctrl>*a, *b;
Ctrl c; // control is created, RefCount == 1
a = new Vector<Ctrl>;
b = new Vector<Ctrl>;
a->Add(c); // a gets a *copy* of c, but in reality it adds just to refcount of c, that becomes 2
b->Add(c); // b gets a *copy* of c, but in reality it adds just to refcount of c, that becomes 3
delete a; // a gets destroyed, RefCount in c becomes 2
```

The advantage of this instead of references of an object ? Well... you must not care of ownership.... and you can be sure object is freed on last reference lost.

As usual, that brings some performance lost.

Ciao

Max

Subject: Re: Core chat...
Posted by mirek on Sat, 27 Oct 2007 14:02:50 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Sat, 27 October 2007 09:21
Back to refcounted objects. What about if Ctrl would be an object built with PIMPL idiom and refcounted ? You then could write :
[code]

Sure, that is one possibility. But that would result in completely differnt library:)

IMO/IME, pimpl is nice for closed library (in terms of extensibility), but it is a pain in the ass if you want to be flexible.

E.g. consider something as simple as inheriting things

All in all, you can circle around the issue as you want. But I believe that U++ paradigm is the most optimal Minimum code to write, maximum flexibility, near to optimal performance.

That is not a bad tradeoff for pick_ IMO

Mirek

Subject: Re: Core chat...
Posted by mdelfede on Sat, 27 Oct 2007 14:28:55 GMT
View Forum Message <> Reply to Message

luzr wrote on Sat, 27 October 2007 16:02
Sure, that is one possibility. But that would result in completely differnt library:)

of course... that's only a chat, we're not planning to rewrite UPP

Quote:
IMO/IME, pimpl is nice for closed library (in terms of extensibility), but it is a pain in the ass if you want to be flexible.

E.g. consider something as simple as inheriting things

Of course, inheritance is the biggest problem of pimpl... well, not difficult but boring, as you have to derive 2 classes in parallel.

Quote:
All in all, you can circle around the issue as you want. But I believe that U++ paradigm is the most optimal Minimum code to write, maximum flexibility, near to optimal performance.

That is not a bad tradeoff for pick_ IMO

ok, ok... I give up!!!

Ciao

Max
