Subject: C++ FQA
Posted by unodgs on Tue, 06 Nov 2007 22:27:53 GMT
View Forum Message <> Reply to Message

May be interesting..
http://yosefk.com/c++fqa/

Subject: Re: C++ FQA
Posted by Mindtraveller on Tue, 06 Nov 2007 23:41:11 GMT
View Forum Message <> Reply to Message

Yes, this FQA led into massive discussion in Russian livejournal segment (more than 250 answers). I`d conclude the discussion mentioned this way: "Yes, C++ may sometimes be difficult as hell and it makes possible writing horrible code, yes there are many other problems with it. But as soon as we don`t have ANOTHER language accumulating all the might of OOP/FP along with C-like performance - C++ will live." (c)

P.S. Some time ago I`ve spent some hours learning new Pascal-languages branch: Oberon, Oberon-2, Zonnon. These are interesting things but, comparing to C++, they lack of some important abilities.

Also I`ve proposed graphical approach for creating programs, but was crucified with other massive livejournal discussion.

So, I would say for now we do not have any adequate alternatives for C++.

Subject: Re: C++ FQA
Posted by unodgs on Wed, 07 Nov 2007 09:40:14 GMT
View Forum Message <> Reply to Message

Quote:So, I would say for now we do not have any adequate alternatives for C++.
Some claim it is D (www.digitalmars.com/D). At least it has comparable set of features.

Subject: Re: C++ FQA
Posted by cbpporter on Wed, 07 Nov 2007 10:33:33 GMT
View Forum Message <> Reply to Message

I've read a good deal of the FQA (and the FAQ too) and while it is mostly right, it is also strongly biased. It still makes a point though, and I think a lot of people agree that C++ is too complicated for it's and everybody else's sake. I would love for an alternative, something that is more simple, straight-forward, but still powerful.

Right now D could be the only alternative (C++0x is even more complicated and redundant). I used D for 2500+ line project (which in C++ would have been about 4000-5000 lines) with about

20 files and I think I have a pretty good idea of what it can do. It has some strong points:

1. True module support.
No more including hundreds of kilobytes if not megabytes of header files in each compilation unit. No more writing every ddecalration twice. Modules act like Java packages, can be fully qualified to avoid name clashes and can be combined to create a library with different access levels. And because each module is compiled only once, D is lightning fast. My entire project compiled from scratch almost instantly, and if I only modified the content of a function or other miner detail, truly instantly.

2. A lot of built in features, like variable length array and hash maps, which combined with some extra functions, can be used to create types like Vector or Map, but even more easy to use and without templates. char[] and about 10 extra functions could make String and StringBuffer obsolete.

3. Templates + mixins + static ifs are stronger than templates in C++ + preprocessor. I never used them, because you can do in D a lot more without templates than in C++.

4. Optional (but defaulted to true) garbage collection.
As much as you could dislike the idea of garbage collection, memory management in C++ is a nightmare, and from this point of view, even U++ which has a lot less such issues, is not able to give such pain free management.

But is also has some disadvantages. Mostly because it has been branched to D 1.0, which is stable, and 2.0 which is a moving target, is not compatible with 1.0, and is a little more complex. It introduces const correctness as in C++, but D can live a lot easier without such extra access methods to have to deal with. Also, the standard library is widely disliked, and a third party library has been created, which is pretty good (even though a little to C++ style), but cross linking between the libraries is impossible. Also operator overloading is just a little bit less powerful.

Still, it is quite a good programing language, having a lot of pain free features and performance comparable to C/C++. I would love to see such a great GUI library as U++ for D, and I'm sure it will continue to evolve.

Subject: Re: C++ FQA
Posted by unodgs on Wed, 07 Nov 2007 12:15:07 GMT
View Forum Message <> Reply to Message

Quote:
1. True module support.
No more including hundreds of kilobytes if not megabytes of header files in each compilation unit. No more writing every ddecalration twice. Modules act like Java packages, can be fully qualified to avoid name clashes and can be combined to create a library with different access levels. And because each module is compiled only once, D is lightning fast. My entire project compiled from scratch almost instantly, and if I only modified the content of a function or other miner detail, truly instantly.

Yeah, I like it too but there is a paper about modules in new C++ too http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n207 3.pdf . I do not know how much similar (or not) they are to D modules but it seems to be a step in the right direction.
Quote:
2. A lot of built in features, like variable length array and hash maps, which combined with some extra functions, can be used to create types like Vector or Map, but even more easy to use and without templates. char[] and about 10 extra functions could make String and StringBuffer obsolete.
True. But I remember that D-people wanted to have String class. I prefere it too even if bulit-in char type is powerful and easy to use.
Quote:
3. Templates + mixins + static ifs are stronger than templates in C++ + preprocessor. I never used them, because you can do in D a lot more without templates than in C++.

Yes, some D coders (like Don Clugston for example) proved they are much more powerful than C++ ones.
Quote:
4. Optional (but defaulted to true) garbage collection.
As much as you could dislike the idea of garbage collection, memory management in C++ is a nightmare, and from this point of view, even U++ which has a lot less such issues, is not able to give such pain free management.

I prefer RAII approach and U++ is a very good example that this really works. Frankly if you use NTL or STL (and follow the RAII way) there is a rare situation when you have to worry about memory management. I don't know why this still is an issue.
D uses GC but fortunately it allows for deterministic destruction in "scope classes". At least in theory Must check it.

I think new C++ should break compatibility and be more like D. I don't understand why it cannot be since all current/old apps can be developed with old compilers. This way C++ will be fatter and fatter (and more complicated) with each standard revision.

Subject: Re: C++ FQA
Posted by Mindtraveller on Wed, 07 Nov 2007 13:03:58 GMT
View Forum Message <> Reply to Message

Talking about D, are there any comparisons in exe size / execution speed of the same code for C++ and D?

Again, I just don`t like an idea of uncontrolled garbage collection. Developing some time-cricical programms for industrial automation, I dislike the fact that my time-critical code can be interrupted or slowned down by some uncontrolled process of garbage collection.

Subject: Re: C++ FQA

Posted by cbpporter on Wed, 07 Nov 2007 13:57:59 GMT
View Forum Message <> Reply to Message

Quote:
Yeah, I like it too but there is a paper about modules in new C++ too http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n207 3.pdf . I do not know how much similar (or not) they are to D modules but it seems to be a step in the right direction.

Quote:
I think new C++ should break compatibility and be more like D. I don't understand why it cannot be since all current/old apps can be developed with old compilers. This way C++ will be fatter and fatter (and more complicated) with each standard revision.

That sound great and is pretty similar with the modules from D, with the only difference that they are already present in D, and by the time these features will be included in C, we'll all be retired programmers. I hate to be pessimistic, but if history has though us anything is that those behind C++ are a bunch of close minded people who have no intention to break compatibility and will gladly bloat the language with absolutely retarded and redundant features. Just have a look at C++0x, and see that most stuff just tries to fix old problems with new redundant features and by keeping the old ones. And even if they do introduce modules, by the time there will be a compliant compiler, I hope that C++ will be less popular for real-life application development.

Quote:
I prefer RAII approach and U++ is a very good example that this really works. Frankly if you use NTL or STL (and follow the RAII way) there is a rare situation when you have to worry about memory management. I don't know why this still is an issue.

It's probably because I'm not as skilled with high level C++ features, but the issues with memory management are quite present for me in U++.
While the language has deep copy, value references, copy constructors, etc., I think that it is impossible not to have such issues. And in U++ I spend a great deal of time just messing with const correctness and other low level details which I shouldn't be forced to worry about.

Quote:
Again, I just don`t like an idea of uncontrolled garbage collection. Developing some time-cricical programms for industrial automation, I dislike the fact that my time-critical code can be interrupted or slowned down by some uncontrolled process of garbage collection.

Well in D you can always chose not to use garbage collection for a class or even just a part of your code. You can use C style pointers, and even call malloc if there is a need to. But I consider this some kind of a myth. Has there been a documented case where garbage collection had serious impact on performance. It is true that such applications sometimes have the bad habit of hanging for 2-3 seconds when a garbage collection cycle starts, but this shouldn't be an issue in non GUI applications. And U++ and even STL does a lot of "garbage collection", only it is more deterministic, but not necessarily faster.

Subject: Re: C++ FQA

Posted by mirek on Wed, 07 Nov 2007 14:14:02 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Wed, 07 November 2007 08:03Talking about D, are there any comparisons in exe size / execution speed of the same code for C++ and D?

Dig in D website. There is an example where D beats C++/STL.

Of course, I could not resist to do the same thing in U++ (And yes, U++ is about twice as fast as D, with shorter code).

Mirek

Subject: Re: C++ FQA
Posted by mdelfede on Wed, 07 Nov 2007 14:39:55 GMT
View Forum Message <> Reply to Message

Well, speaking about 'different' languages, there's also Borland's Delphi (objectpascal) that makes stuffs much easier than C++.
You can have low level (and so *fast*) programs when you need, without loosing the comfort and type-safe pascal character.
Plus, the extensions Borland introduced over simple pascal are quite powerful... and makes true modularity possible.
Compilation is among the fastest, and resulting code is quite good, too... being only few percent slower (IMHO !) than c++ normal code.
It has also true RTTI built-in. It's only a pity that Kilix (Delphi ported to linux...) had no good luck in market.

Ciao

Max

Subject: Re: C++ FQA
Posted by mr_ped on Wed, 07 Nov 2007 16:05:48 GMT
View Forum Message <> Reply to Message

"have the bad habit of hanging for 2-3 seconds"

Oh.. There are applications which have to respond within 10 milliseconds otherwise something nasty happens to some device...

2-3sec is like ages for them.
(and yes, some of those applications are written in C)

Subject: Re: C++ FQA
Posted by Mindtraveller on Wed, 07 Nov 2007 17:02:05 GMT
View Forum Message <> Reply to Message

cbpportersuch applications sometimes have the bad habit of hanging for 2-3 seconds when a garbage collection cycle starts This is totally unacceptable for a great number of applications including industrial automation, net exchange, multimedia, device i/o, games, etc.

luzrU++ is about twice as fast as D, with shorter code Oops... with these calculations and 2-3 sec uncontrollable delays... just forgetting about D

Any more alternatives to discuss?

Subject: Re: C++ FQA
Posted by cbpporter on Wed, 07 Nov 2007 18:30:16 GMT
View Forum Message <> Reply to Message

Quote:Oops... with these calculations and 2-3 sec uncontrollable delays... just forgetting about D
I said that they can accur. They are by far not uncontrolable, and only happen in some apps. And I really wish they were so short in Java, but there they are a lot longer and totally uncontrolable, yet it is still used a lot (too much). With my application I was wery happy with it's performance. And very short allocation time plus almost zero deep copies can make up for it.

Subject: Re: C++ FQA
Posted by Mindtraveller on Wed, 07 Nov 2007 23:13:34 GMT
View Forum Message <> Reply to Message

cbpporterThey are by far not uncontrolable, and only happen in some apps.OK, as a developer who likes D programming paradigm, and who wants writing efficient applications, I will consider changing language to D if
1) There`s a simple way to guarantee no hangups at all.
2) My .exe will be <10%-15% slower than corresponding C++ code in any case.

These are critical conditions. Can D, or any other alternative fit?

P.S. luzr, I just thought that D and U++ comparison is not quite honest. It would be better to compare internal language features of D and C++. According to the tests described in D site (I looked at them as you recommended), D is no slower than C++ (at least in some cases?). So porting the U++ classes and algorithms to D, adopting them for D specifics could make U+D (U++ for D) as fast as original U++. It`s just a theory, of course.

Subject: Re: C++ FQA
Posted by mirek on Thu, 08 Nov 2007 04:25:23 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 07 November 2007 08:57
Well in D you can always chose not to use garbage collection for a class or even just a part of your code. You can use C style pointers, and even call malloc if there is a need to.

Note that the price of this feature is the use of conservative GC.

Which makes D code behaviour dependent on data processed. E.g. it is dangerous to use D to process large cryptography application.

In the end, IMO, this makes D a toy language.

Mirek

Subject: Re: C++ FQA
Posted by mirek on Thu, 08 Nov 2007 04:34:24 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Wed, 07 November 2007 18:13cbpporterThey are by far not uncontrolable, and only happen in some apps.OK, as a developer who likes D programming paradigm, and who wants writing efficient applications, I will consider changing language to D if
1) There`s a simple way to guarantee no hangups at all.
2) My .exe will be <10%-15% slower than corresponding C++ code in any case.

3) Add to mix memory requirements. In that "Alice" benchmark, D used about 2-3 times more memory than U++ (if I remember well).

BTW, it is some time when I last tried it. I would really be very glad if somebody reproduced my results; here is U++ code:

```
#include <Core/Core.h>

using namespace Upp;

#define BENCHMARK // for benchmark purposes, output is omitted
#ifdef  BENCHMARK
#define BENCHBEG for(int i = 0; i < 1000; i++) {
#define BENCHEND }
#else
#define BENCHBEG
#define BENCHEND
#endif


void main(int argc, const char *argv[])
{
```

```
 VectorMap<String, int> map;
 BENCHBEG
 for(int i = 1; i < argc; i++) {
  String f = LoadFile(argv[i]);
  int line = 1;
  const char *q = f;
  for(;;) {
   int c = *q;
   if(IsAlpha(c)) {
    const char *b = q++;
    while(IsAlNum(*q)) q++;
    map.GetAdd(String(b, q), 0)++;
   }
   else {
    if(!c) break;
    if(c == '\n')
     ++line;
    q++;
   }
  }
 }
 BENCHEND
 Vector<int> order = GetSortOrder(map.GetKeys());
#ifndef BENCHMARK
 for(int i = 0; i < order.GetCount(); i++)
  Cout() << map.GetKey(order[i]) << ": " << map[order[i]] << '\n';
#endif
 printf("%d\n", map.GetCount());
}
```

(I had to loop over it more times, otherwise the execution for Alice.txt was too fast to be measurable).

Quote:
P.S. luzr, I just thought that D and U++ comparison is not quite honest. It would be better to compare internal language features of D and C++. According to the tests described in D site (I looked at them as you recommended), D is no slower than C++ (at least in some cases?). So porting the U++ classes and algorithms to D, adopting them for D specifics could make U+D (U++ for D) as fast as original U++. It`s just a theory, of course.

I think this is not quite possible, as language features are way too different.

OTOH, in this particular benchmark, D is using internal language features, while C++/U++ is using library. Still, we are faster and the code is shorter. That IMO says a lot about language flexibility.

Mirek

Subject: Re: C++ FQA
Posted by Zardos on Thu, 08 Nov 2007 12:33:30 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Thu, 08 November 2007 00:13OK, as a developer who likes D programming paradigm, and who wants writing efficient applications, I will consider changing language to D if
1) There`s a simple way to guarantee no hangups at all.
2) My .exe will be <10%-15% slower than corresponding C++ code in any case.

These are critical conditions. Can D, or any other alternative fit?

I switched from D to UPP because:

D uses an old object format which makes it impossible to link you d programs with ordinary c libraries. You have to compile the c libraries with digital mars c. Or you have to use some convertes which converts the libs in the old format

Lack of D libraries: While I was in the language - D was an constant moving target. Every weekly new version broke your code (and the code of the libraries)

No professional GUI library available. Well you could build one of you own.. But see the point above.

Performance: Some D code compiles to very efficient code. Sometimes even better than MSVC, but some parts are unacceptable slow. For example the built in associative arrays are horrible slow. ...And you can not fix this by fixing the library, because they are really built into the language

Garbage collection: Even if the last version of the grabage collection is less conservative it is still a conservative garbage collection. -> See Mireks comment.

No useful IDE with integrated dubugger avilable. Productivity is not only related to the language...

And I completely lost interest of the language at the time when Walter decided to at CONST to the language (currently only in the development branch). Well he claims "you don't have to use it..." but I doubt this. I expect a similar disaster as in C++: Add one const and the source is infected...

IMHO C++ sucks. But I don't see a serious choice for my requirements. UPP showed me again that you can ship (most times) around the flaws of C++...
But I'm still dreaming of a language like D without the shortcomings. May be compiled to C++ in the first incarnation to make all C++ libraries available automatically?

So don't get me wrong. I think D is really great (except of the added const). It has fantastic template and meta programming capabilities and its design is clean and ellegant. It has all the state of the art constructs you would expect from a modern language...

Subject: Re: C++ FQA
Posted by mirek on Thu, 08 Nov 2007 18:25:44 GMT
View Forum Message <> Reply to Message

Zardos wrote on Thu, 08 November 2007 07:33
Well he claims "you don't have to use it..." but I doubt this. I expect a similar disaster as in C++: Add one const and the source is infected...

Well, as of const, I generally tend to thing that while initially it feels like plague, after a while you can find it quite useful, at least to describe the interface better.

Been through it ages ago....

Subject: Re: C++ FQA
Posted by mdelfede on Thu, 08 Nov 2007 22:39:51 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 08 November 2007 19:25
Well, as of const, I generally tend to thing that while initially it feels like plague, after a while you can find it quite useful, at least to describe the interface better.

Been through it ages ago....

me too.... at the beginning I found it boring and ennoying stuff, now I find it very useful. The bad thing is that c++ allow you to recast const to non-const... so a badly written library can do what he wants.

Max

Subject: Re: C++ FQA
Posted by mirek on Fri, 09 Nov 2007 07:51:40 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Thu, 08 November 2007 17:39luzr wrote on Thu, 08 November 2007 19:25
Well, as of const, I generally tend to thing that while initially it feels like plague, after a while you can find it quite useful, at least to describe the interface better.

Been through it ages ago....

me too.... at the beginning I found it boring and ennoying stuff, now I find it very useful. The bad thing is that c++ allow you to recast const to non-const... so a badly written library can do what he

wants.

Max

Well, but that is useful feature and this is one of things I like with C++ - the "default" mode is "safe", but you can always do dirty things when you need them.

In other words, you can also say that a well written library can do what it needs

Actually, interestingly it seems like I am the only one here who in fact likes C++ as it is (except some quite small issues and the standard library, which IMO only looks like a good design).

Mirek

Subject: Re: C++ FQA
Posted by tvanriper on Fri, 09 Nov 2007 11:56:15 GMT
View Forum Message <> Reply to Message

Actually, I rather like C++ as it is, too.

But then, after having worked with it for so many years, I'm maybe rather comfortable with it, enough to make it do pretty much whatever I want.

I don't want a language that coddles me (by trying to prevent me from doing something dumb), but I also don't want to deal with machine-code. I want something that can scale, but let me deal with the bit-twiddling details if necessary. C++ provides all of this for me.

And, most of the time, if I've found my application is starting to look a little wordy, it's probably because of a bad design.

This isn't to say that I'm not intrigued by some of the other developments I've heard about in C++0x and others, but I'm happy with C++ as it is.

C++0x, if I recall, is really an attempt to make the language easier to handle for folks new to the language. As such, at this point in my life, I have no need for it.

I haven't really looked deeply into D, so I can't really comment on it, except to say that I haven't met a lot of people who know how to program in D, so commercial projects written in D would require a very special person to maintain it, which could drive up costs. I might consider writing fun stuff with D, but until the language becomes more mainstream, I'd hesitate to use it for work that provides me with a paycheck.

Subject: Re: C++ FQA

luzr wrote on Fri, 09 November 2007 08:51
Well, but that is useful feature and this is one of things I like with C++ - the "default" mode is "safe", but you can always do dirty things when you need them.

no, here I don't agree.... such things make virtually impossible write 'safe' libraries. A 'const' should be a 'const', not a 'maybe const'.... as 'private' should be so, not a maybe one.
I've seen constructs like that :

#define private public
#include "alib.h"

just to overcome a private class declaration and access the low-level stuffs inside it.... Then, when library changes, people (maybe also people that hasn't nothing to do with such a hack) starts wondering why his program that up to the day before worked like a charm just crash. IMHO that has nothing to do with commercial-grade applications.

Quote:
In other words, you can also say that a well written library can do what it needs

well, a well written lib should do what the coder will, *not* what the user is missing. Before using C++ hacks to overcome libs limitations, you have 3 solutions :
1) Patch the sources, if you have them
2) Ask the original programmer to enhance the lib
3) Just find another lib that suit your needs

Quote:
Actually, interestingly it seems like I am the only one here who in fact likes C++ as it is (except some quite small issues and the standard library, which IMO only looks like a good design).

Well, I agree that C++ *is* useful and *is* the only widespread system-wide programming language. But I really can't say that is a good language. Besides static memory management, which I prefere against a gc approach (I like do code what I want, not what the compiler want...), it contains really too many caveats dues mostly (but not all) because of compatibility issues.
It is :
- slow compiling
- not modular at all
- object model is missing too many useful stuffs (properties, delegates, a true rtti system, just among all)
- operator overloading is just awful, as is missing rvalue-lvalue different behaviour
- missing high-level types (strings, arrays.....)
- cumbersome templates
- no binary objects specifications... in particular with respect to to name mangling
- this damn'd preprocessor that does what he wants
Just an example about this... on a really poorly written code :

#define a_type mytype
#define an_include </my/include/dir/a_type/mytype.hxx>
#define another_include "/my/include/dir/a_type/mytype.hxx"
#include an_include
#include another_include

That has the wonderful (sigh) result of :

#include </my/include/dir/mytype/mytype.hxx>
#include "/my/include/dir/a_type/mytype.hxx"

I stumbled about such a problem and it tooks half a day to understand that inside <> you have macro substitution, but inside "" not.... and I'm still not sure that it isn't a compiler behaviour.

IMHO, what we needs is a new system wide language, that maybe resembles to C++, but gets rid of all caveats and introduces the missing things. C++ is a language that, in order to be able to compile 1980's code is just becoming a monster and still missing what a modern oo language should have.

Ciao

Max

Subject: Re: C++ FQA
Posted by mirek on Fri, 09 Nov 2007 13:48:52 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 09 November 2007 07:41luzr wrote on Fri, 09 November 2007 08:51
Well, but that is useful feature and this is one of things I like with C++ - the "default" mode is "safe", but you can always do dirty things when you need them.

no, here I don't agree.... such things make virtually impossible write 'safe' libraries. A 'const' should be a 'const', not a 'maybe const'.... as 'private' should be so, not a maybe one.
I've seen constructs like that :

#define private public
#include "alib.h"

just to overcome a private class declaration and access the low-level stuffs inside it.... Then, when library changes, people (maybe also people that hasn't nothing to do with such a hack) starts wondering why his program that up to the day before worked like a charm just crash. IMHO that has nothing to do with commercial-grade applications.

Sure, this is awful, but very often, the alternative is that it is not possible to finish your job.

Note that all these "high-level" language, whose propagators despise C/C++, have interfaces for these languages so that the dirty stuff can be done.

Quote:
well, a well written lib should do what the coder will, *not* what the user is missing. Before using C++ hacks to overcome libs limitations, you have 3 solutions :
1) Patch the sources, if you have them
2) Ask the original programmer to enhance the lib
3) Just find another lib that suit your needs

And if neither is possible? You quit the job?

Mirek

Subject: Re: C++ FQA
Posted by mdelfede on Fri, 09 Nov 2007 14:52:26 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 09 November 2007 14:48

Sure, this is awful, but very often, the alternative is that it is not possible to finish your job.

Note that all these "high-level" language, whose propagators despise C/C++, have interfaces for these languages so that the dirty stuff can be done.

I'm not telling about 'make a new ""high level"" language, just speaking about making a system-level language more consistent and comfortable. Missing properties, delegates, right handling for rvalues-lvalues, awful macro language has *nothing* to do with the ability of doing system-level stuffs. GC is another matter but there I agree, i see GC as a way to do things without thinking, and has nothing to do with a language that has to be fast and real-time. Modularity also brings only advantages... so why not ? As I told in another thread, Borland did a great job with delphi, adding many useful extensions to pascal language, whithout loosing anything... better said, adding also the ability of low level machine access somewhere.
As an example, bring pick_ and reference counting inside the language would not break anything, if they're put as an option. The same for properties, delegates and some better handling of overloaded operators. Modularity should be not difficult too, it's just a matter of define a new object format that contains precompiled declarations too, as borland did with their packages for delphi. All that could stay side-by-side with actual c++ implementation.
Adding also a good string and array base types should not be a big problem too, and could also be much faster than actual template solutions.... so why not ?

Quote:
Quote:
well, a well written lib should do what the coder will, *not* what the user is missing. Before using C++ hacks to overcome libs limitations, you have 3 solutions :

1) Patch the sources, if you have them
2) Ask the original programmer to enhance the lib
3) Just find another lib that suit your needs

And if neither is possible? You quit the job?

Well, you must agree that the cases on which neither of the 3 solutions is possible are very rare... and yes, in such a case, I'd quit the job

Ciao

Max

Subject: Re: C++ FQA
Posted by waxblood on Fri, 09 Nov 2007 16:07:43 GMT
View Forum Message <> Reply to Message

mdelfede wrote:

Quote:#define private public
#include "alib.h"

Nice trick... I think I'll start using it.....

As for C++ language, I'd like much a 100% ansi-compliant C++ interpreter... Nowadays many people write their C++ programs and link them to some so-called scripting langauge (one for all: Python) to have more flexibility, but why can't I use C++ to perform the same task?

I've looked into cint
: while covers most (if not all) C, it is still at 85% of ansi c++, and the same authors say it will never reach the 100% goal.

David

Subject: Re: C++ FQA
Posted by mdelfede on Fri, 09 Nov 2007 18:51:10 GMT
View Forum Message <> Reply to Message

waxblood wrote on Fri, 09 November 2007 17:07mdelfede wrote:

Quote:#define private public
#include "alib.h"

Nice trick... I think I'll start using it.....

eheheheheh

Quote:
As for C++ language, I'd like much a 100% ansi-compliant C++ interpreter... Nowadays many people write their C++ programs and link them to some so-called scripting langauge (one for all: Python) to have more flexibility, but why can't I use C++ to perform the same task?

I've looked into cint
: while covers most (if not all) C, it is still at 85% of ansi c++, and the same authors say it will never reach the 100% goal.

If you need a nice scripting c-like language, there is also squirrel (www.squirrel-lang.org) that has some nice features.
Of course, is *not* ansi c++, but it has dynamic built-in structures that make it useful as a scripting lang.
Among others, it's used also on codeblocks scripting engine.

Ciao

Max

Subject: Re: C++ FQA
Posted by mirek on Sat, 10 Nov 2007 14:14:53 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 09 November 2007 09:52
Quote:
Quote:
well, a well written lib should do what the coder will, *not* what the user is missing. Before using C++ hacks to overcome libs limitations, you have 3 solutions :
1) Patch the sources, if you have them
2) Ask the original programmer to enhance the lib
3) Just find another lib that suit your needs

And if neither is possible? You quit the job?

Well, you must agree that the cases on which neither of the 3 solutions is possible are very rare... and yes, in such a case, I'd quit the job

Ciao

Max

Interesting, I find them quite common. E.g. U++ has to use similar trick with X11, because X11 polutes the global namespace with too many simple names as "Font".

There it no chance Xlib.h being changed and in order to use X11, I cannot use different library. I cannot realistically patch the sources too. Or, BTW, you can consider that hackery as "patching sources from outside" It is dirty, but at least it is possible.

Mirek

Subject: Re: C++ FQA
Posted by mirek on Sat, 10 Nov 2007 14:22:40 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 09 November 2007 09:52
As an example, bring pick_ and reference counting inside the language would not break anything, if they're put as an option. The same for properties, delegates and some better handling of overloaded operators.

Well, I guess they are reluctant to make already very complex language even more complex. And many of these issues do not bring anything really new to the table. (but I would certailny liked better syntax sugar for "pick_", this is the only thing in C++ I seriously miss).

Quote:
Modularity should be not difficult too, it's just a matter of define a new object format that contains precompiled declarations too, as borland did with their packages for delphi. All that could stay side-by-side with actual c++ implementation.

Well, but keep in mind that C++ *standard* is intended as multiplatform solution. It e.g. must not have anything in it preventing the use of language on platform that is only capable of working with 36 bit words...

What you demand is possible even now - there is nothing in C++ standard that would make it impossible for specific implementation.

Quote:
Adding also a good string and array base types should not be a big problem too, and could also be much faster than actual template solutions.... so why not ?

Or you would be stuck with slow implementation and no way how to improve it...

Mirek

Subject: Re: C++ FQA
Posted by mdelfede on Sat, 10 Nov 2007 15:58:28 GMT
View Forum Message <> Reply to Message

luzr wrote on Sat, 10 November 2007 15:22

Well, I guess they are reluctant to make already very complex language even more complex. And many of these issues do not bring anything really new to the table. (but I would certailny liked better syntax sugar for "pick_", this is the only thing in C++ I seriously miss).

Uhmmm.. they already made huge changes with templates, at least from the beginning of C++ standards... and other stuffs. pick_, refcounts and properties would *not* break existing code, so I really don't understand why they're not inside... in particular, properties do belong to a good OO language, and give absolutely no problem to existing code.

Quote:
Well, but keep in mind that C++ *standard* is intended as multiplatform solution. It e.g. must not have anything in it preventing the use of language on platform that is only capable of working with 36 bit words...

What you demand is possible even now - there is nothing in C++ standard that would make it impossible for specific implementation.

I know, but it wouldn't be standard. __property construct was added by borland to his C++ Builder, and effectively simplified much code writing. Also __published properties were a very good addition, as they brought good RTTI inside classes and made easy to write RAD tools.

Take in mind that C++ object files are, IMHO, *not* standardized, in particular with respect to code mangling... M$ has his one, borland another one and I guess GCC again a different one. So, it's impossible to link objs made with different C++ compilers, which was indeed possible with plain C. I think that was the real big problem of Borland tools... M$ came later, made worse compilers and made object files totally incompatible with borland ones.... so people that wanted to write code linking with M$ libs *had* to go to M$ tools.
That's again a big miss of C++ standart. So, what prevents put in C++ standard a documented obj format WITH module support ? That I don't really understand. It could also be done keeping the compatibility with old obj format (at least, on one compiler....).

Quote:
Or you would be stuck with slow implementation and no way how to improve it...

Well, here you must agree that bringing refcounts and/or pick_ INSIDE the language would make it possible compile-level optimizations that now are not possible. Of course, if you have a bad compiler that's slow, but that belongs to many other stuffs too.

Ciao

Max

Subject: Re: C++ FQA
Posted by mdelfede on Sat, 10 Nov 2007 16:06:00 GMT
View Forum Message <> Reply to Message

luzr wrote on Sat, 10 November 2007 15:14
Interesting, I find them quite common. E.g. U++ has to use similar trick with X11, because X11 polutes the global namespace with too many simple names as "Font".

There it no chance Xlib.h being changed and in order to use X11, I cannot use different library. I cannot realistically patch the sources too. Or, BTW, you can consider that hackery as "patching sources from outside" It is dirty, but at least it is possible.

Well, I don't know what have you done (and why did you need to..) on respect to Xlib. I know that Xlib comes from old times and has old stuffs inside. But I think you could do it without hacks, of course that would have been more difficult to avoid name clashes and so. But IMHO, if you use undocumented features to ease your job, you're not guaranteed your app will work on next Xlib release. You'll loose control of your app, and so will do your customers. Your app will depend on 3dy part changes in code.
That's the same with the '#define private public' hack. You can access all level of foreign code, but you loose control on it.

Ciao

Max

Subject: Re: C++ FQA
Posted by cbpporter on Sat, 10 Nov 2007 16:31:12 GMT
View Forum Message <> Reply to Message

Well Delphi is really great. I used it for many years and I never understood why it wasn't used

more extensively (especially now, that there is a free version available). And it is great that you can use it comfortably without templates and other stuff.

For those looking into Delphi and not afraid to experiment a little, here is a nice third party lib: http://kolmck.net/. It is as easy to use as the original, but it uses some very clever compiler and preprocessor tricks to obtain very small exe sizes. I saw an mp3 player and a zip extractor with about 100kb exe size. It's quite stable and bug free, but very hard to debug the sources of the lib.

And I'm quite surprised to see people who don't like gc, but have nothing against reference counting, which is slower and almost impossible to use efficiently in a multi-threading application.

Subject: Re: C++ FQA
Posted by mirek on Sat, 10 Nov 2007 16:54:15 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sat, 10 November 2007 11:31
And I'm quite surprised to see people who don't like gc, but have nothing against reference counting, which is slower and almost impossible to use efficiently in a multi-threading application.

I mostly agree with this...

Mirek

Subject: Re: C++ FQA
Posted by mr_ped on Sat, 10 Nov 2007 21:26:41 GMT
View Forum Message <> Reply to Message

Excuse me for my blatant ignorance, but the last time I was checking GC inner working they were pretty much all based upon ref counting.

They did either evolve great deal since I take that fast glance on them (quite possible), or by using GC you get everything ref counted by default.

In the latter case I don't understand this thing at all:
Quote:And I'm quite surprised to see people who don't like gc, but have nothing against reference counting, which is slower

Please, don't send me links on the modern GC designs, just tell me I'm wrong and miss some centuries of evolution.
(I will maybe check the modern GC design later, right now I have more important things to do.)

Subject: Re: C++ FQA
Posted by mirek on Sat, 10 Nov 2007 21:35:01 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Sat, 10 November 2007 16:26Excuse me for my blatant ignorance, but the last time I was checking GC inner working they were pretty much all based upon ref counting.

Nope, real GC cannot be based on ref counting - that does not solve the circular references.

Quote:
They did either evolve great deal since I take that fast glance on them (quite possible), or by using GC you get everything ref counted by default.

Real GC is based on mark&sweep algorithm. Google it

Mirek

Subject: Re: C++ FQA
Posted by mdelfede on Sat, 10 Nov 2007 22:57:52 GMT
View Forum Message <> Reply to Message

luzr wrote on Sat, 10 November 2007 17:54cbpporter wrote on Sat, 10 November 2007 11:31
And I'm quite surprised to see people who don't like gc, but have nothing against reference counting, which is slower and almost impossible to use efficiently in a multi-threading application.

I mostly agree with this...

As with GC, refcount can be made thread safe, IMHO.
And, as GC, it can be made more or less efficient code.
Even pick_ can be not thread safe, if it's bad coded, and must have some sort of synchronizing code to be thread safe. Of course, the 'blocking' code in pick_ should be a bit less than in refcount, and (I guess) much less than in GC.
Even copying a whole array can be a big problem in multithreading stuff, and can become slow.
At least, refcount (and GC) can be made thread safe by design, without requiring additional sync code by user program; new(), delete() and malloc() and manual memory management requires thread safe stuffs in user code. Of course they CAN be faster, but more error-prone too.
I *don't* like GC, but only because I want to know when memory is being freed; I *do* like refcount because gives you some coding comfort without too much performance penalty; pick_ gives you less code comfort with best performance, and has great advantage over more manual stuff that gives you an error if you access picked objects.

That said, GC is nothing modern, lisp has it since before 1980, and also appleII pc with its basic

had it, awfully slow, too, for string management. If I remember well, some GC have a mix of refcount AND other stuffs to solve circular references, but I'm not sure. It could be made working with refcounts, some linked list stuff and the GC traversing them to solve circular refs, at a cost of performance.

I would not accept a language based mostly on GC, but I've got nothing against an optional gc among other management stuffs.

BTW, I can't see how refcount can be slower than GC... maybe I'm wrong, but I'd like to have it explained !

ciao

Max

Subject: Re: C++ FQA
Posted by mirek on Sun, 11 Nov 2007 08:54:11 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Sat, 10 November 2007 17:57luzr wrote on Sat, 10 November 2007 17:54cbpporter wrote on Sat, 10 November 2007 11:31
And I'm quite surprised to see people who don't like gc, but have nothing against reference counting, which is slower and almost impossible to use efficiently in a multi-threading application.

I mostly agree with this...

As with GC, refcount can be made thread safe, IMHO.

The things is that even thread unsafe reference counting is about as fast or slower than mark&sweep GC.

And, w.r.t. thread safety, the another trouble is that you cannot safely use atomic operations only when the object is really shared between threads (when it is needed).

Quote:
Even pick_ can be not thread safe, if it's bad coded, and must have some sort of synchronizing code to be thread safe.

Everything can be thread unsafe if you really try. Anyway, the simplest pick_ implementation is naturaly thread safe.

Quote:
I would not accept a language based mostly on GC, but I've got nothing against an optional gc among other management stuffs.

The problem is that this is not quite possible.

Quote:
BTW, I can't see how refcount can be slower than GC... maybe I'm wrong, but I'd like to have it explained !

OK, only think about the amount of operations that have to be performed in simple "return the value" scenario

```
RefCounted<Foo> Fn() {
  RefCounted<Foo> x = new Foo;
  .....
  return x;
}

void Fn2() {
  RefCounted<Foo> y = new Foo;
  ...
  y = Fn();
}
```

Mirek

Subject: Re: C++ FQA
Posted by mdelfede on Sun, 11 Nov 2007 10:25:19 GMT
View Forum Message <> Reply to Message

luzr wrote on Sun, 11 November 2007 09:54mdelfede wrote on Sat, 10 November 2007 17:57luzr wrote on Sat, 10 November 2007 17:54cbpporter wrote on Sat, 10 November 2007 11:31
And I'm quite surprised to see people who don't like gc, but have nothing against reference counting, which is slower and almost impossible to use efficiently in a multi-threading application.

I mostly agree with this...

As with GC, refcount can be made thread safe, IMHO.

The things is that even thread unsafe reference counting is about as fast or slower than mark&sweep GC.

Well, here that depends on *what* do you mean with 'faster'.
Let's speak for example about linux kernels, the normal one and the low-latency one. Which is faster ? That depends on what do you mean! Low latency kernel responds faster than normal one, but in overall time it is slower. The normal kernel can have troubles with real-time apps, but is overall faster.
So, refcount DOES add overhead on allocation operations, so for the single op IS slower than GC. But, when GC comes in place, you do have a long latency. In overall application time, GC is of course faster than many refcount ops.... with the counterpart of some 'program stops' during GC.
In conclusion :
GC is faster in overall app time, and is faster on single memory operation, BUT it has the big problem of GC stop time
Refcount is slower on single memory ops, is slower globally BUT has no stop times and it's execution is smoother.
If you don't need real-time response, GC is better, otherwise can be very bad.
The big problem, as you say here later, is that is very difficult to use other memory allocations in conjunction with GC.

Quote:
And, w.r.t. thread safety, the another trouble is that you cannot safely use atomic operations only when the object is really shared between threads (when it is needed).

I didn't understand that one...

Quote:
Quote:
Even pick_ can be not thread safe, if it's bad coded, and must have some sort of synchronizing code to be thread safe.

Everything can be thread unsafe if you really try. Anyway, the simplest pick_ implementation is naturaly thread safe.

Let's say that pick_ is very easy to make thread safe stuff

Quote:
Quote:
I would not accept a language based mostly on GC, but I've got nothing against an optional gc among other management stuffs.

The problem is that this is not quite possible.

I agree with this one... mostly. That's why I don't like GC.

And, in conclusion, I agree with you that pick_ is the best when you don't need a true object copy, or even when you do need it but you're ready to think much on what your code will do.
Refcount lets you to 'turn brain off', you must not bother about what your code will do with your object, mostly. So, less error prone. GC is the best for 'lazy people', it does all the dirty job but it leads often to slow apps... because people start allocating hundreds of objects on the fly without thinking about optimization.

Ciao

Max

Subject: Re: C++ FQA
Posted by mdelfede on Sun, 11 Nov 2007 10:36:25 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sat, 10 November 2007 17:31Well Delphi is really great. I used it for many years and I never understood why it wasn't used more extensively (especially now, that there is a free version available). And it is great that you can use it comfortably without templates and other stuff.

I think that real problem on Delphi and even CBuilder where the binary incompatibility with M$ ones. You could *not* link a delphi or BC++ app with M$ libs, or even (and that's the real bad) with C++ DLL without huge efforts. I made many apps in BC++ builder to extend Autocad, and I could not use ARX internals because of code mangling incompatibility. I resorted about a COM app embedded with the use of some hacks calling 2-3 funcs of ARX internals, just translating the mangling schemas between compilers... a cumbersome work, and those were only 2-3 funcs to call, not the 1000+ of the whole stuff.
Borland should have taken the decision to make its binaries compatible with the (whorse) M$ one in order to stay on the market... but they didn't.

Subject: Re: C++ FQA
Posted by Mindtraveller on Sun, 11 Nov 2007 17:22:06 GMT
View Forum Message <> Reply to Message

For me, main BC++B problems were:
1) Compiler errors and IDE issues (buggy IDE and debugger)
2) Ineffective and poor framework (even that was much better than MFC) which wasn`t really upgraded since smth like 1997-1999 (leading VCL developer was bought by M$ for .NET platform).
3) Wrong development direction: since first versions BC++B was strongly modified for database applications against the interest of all others.

Subject: Re: C++ FQA
Posted by mdelfede on Sun, 11 Nov 2007 17:52:34 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Sun, 11 November 2007 18:22For me, main BC++B problems were:
1) Compiler errors and IDE issues (buggy IDE and debugger)

well, not too many bugs... often less than M$ counterparts.
It was quite usable.

Quote:
2) Ineffective and poor framework (even that was much better than MFC) which wasn`t really upgraded since smth like 1997-1999 (leading VCL developer was bought by M$ for .NET platform).

well, IMHO not so ineffective nor poor... at least, an order of magnitude better than M$ one.

Quote:
3) Wrong development direction: since first versions BC++B was strongly modified for database applications against the interest of all others.

That depends on what you code... I don't like database apps, too, but if they did so, maybe it was for some reason.

Subject: Re: C++ FQA
Posted by Mindtraveller on Sun, 11 Nov 2007 23:58:14 GMT
View Forum Message <> Reply to Message

mdelfedenot too many bugs... often less than M$ counterparts.
It was quite usable.
I`ve used BCB until this autumn, when I found U++. And I had problems with it`s bug. The worst compiler bug is that it sometimes ignores source code line next to the "}". Also, debugging multithreaded applications frequently lead BCB IDE to hang. There are more some critical bugs and issues. VCL also may show different behaviour on Win98 and WinXP systems with the same code (ini files, etc).

mdelfedenot so ineffective nor poor... at least, an order of magnitude better than M$ one. Yes, as I mentioned, it`s much better than MFC and much more lightweight than .NET.
My common BCB application has something like 30% of code with Win32 API calls (threads and core objects, serial i/o, grid cells drawing, advanced registry work, ...). This shows for me that VCL is rather poor when we discuss what we can do with it. Yes, it`s much-much more than nothing but it is still insufficient.
Again VCL is good, it was wonderful back in 90-s, but now after so much time passed, Borland could do much, but they didn`t.
So VCL is still a pascal-derived library, ignoring most of C++ features.
So, adding notes about efficiency, do you know how VCL handles it`s forms? Forms and

components are converted to textual representation. The borland IDE gives text to linker, which adds these text resources to the end of .exe files. When you start any BCB application, it runs special parser (which is by default in dll!). Text resources are parsed. How? Application gives this text to internal engine, then to newly-created components which serialize their properties from text parts. That is how BCB application starts.
Also, BCB has it`s own memory manager which gives you something like 1Mb-size pool for variables. And there is no way to switch to any other memory manager, as soon as you use VCL components or even VCL strings.
The greatest issue for me personally is that IDE "insists" on the only one programming style. More your application uses VCL forms and components, more time you must spend making your code the way you need, not BCB IDE. For example, BCB makes you having all your form and component event handler in one file. Even when they are logically belong to very different parts of program logics (different classes).
And, yes, it`s .lib files are incompatible with Microsoft ones. The utility for coversion (implib) isn`t that good also.
Isn`t that enough? I can count more critical issues if it`s not clearly fow now. All I say is that VCL was good in 90-s, but it was abandoned by Borland. VCL had to be improved, upgraded, optimized for general-purpose applications. Then - for specific needs. It had a chance to become much more popular than M$ thumb "framework" like MFC.

mdelfedeI don't like database apps, too, but if they did so, maybe it was for some reason. They thought that their reason was serious enough and they lost. BCB is nearly dead for last years and according to the information I have, Borland`s made a decision to abandon VCL completely, switching all development to QT or .NET.

Subject: Re: C++ FQA
Posted by mirek on Mon, 12 Nov 2007 08:46:10 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Sun, 11 November 2007 18:58
So, adding notes about efficiency, do you know how VCL handles it`s forms? Forms and components are converted to textual representation. The borland IDE gives text to linker, which adds these text resources to the end of .exe files. When you start any BCB application, it runs special parser (which is by default in dll!). Text resources are parsed. How? Application gives this text to internal engine, then to newly-created components which serialize their properties from text parts. That is how BCB application starts.

Actually, this might seem horrible, but in reality, it does not need to cause any real problems... Those text descriptions are usually not too big and easy to parse. I do not know too much about VCL, but this could be "OK solution" IMO...

Quote:
The greatest issue for me personally is that IDE "insists" on the only one programming style.

Similar experience with MFC or some other environments. This is what made me sceptical about

"visual tools".

(Ironically, it seems like U++ starts to be quite ide supported too... I guess if you have that power - to support your library in your ide - it is just hard to resist..)

Mirek

Subject: Re: C++ FQA
Posted by mdelfede on Mon, 12 Nov 2007 09:19:07 GMT
View Forum Message <> Reply to Message

luzr wrote on Mon, 12 November 2007 09:46Mindtraveller wrote on Sun, 11 November 2007 18:58
So, adding notes about efficiency, do you know how VCL handles it`s forms? Forms and components are converted to textual representation. The borland IDE gives text to linker, which adds these text resources to the end of .exe files. When you start any BCB application, it runs special parser (which is by default in dll!). Text resources are parsed. How? Application gives this text to internal engine, then to newly-created components which serialize their properties from text parts. That is how BCB application starts.

Actually, this might seem horrible, but in reality, it does not need to cause any real problems... Those text descriptions are usually not too big and easy to parse. I do not know too much about VCL, but this could be "OK solution" IMO...

I agree, BCB apps were fast starting, I don't see anything bad on loading forms from a resource file and/or from exe file. It show only that they had good serialization stuffs.

Quote:
Quote:
The greatest issue for me personally is that IDE "insists" on the only one programming style.

Similar experience with MFC or some other environments. This is what made me sceptical about "visual tools".

(Ironically, it seems like U++ starts to be quite ide supported too... I guess if you have that power - to support your library in your ide - it is just hard to resist..)

I think that it's unavoidable.... every Ide tends to support best their own class library, as people have more skills on it.
That's the same for Codeblocks.... it can work with many widget libraries, but is focused on wxwidgets, as it's written using that library. They have wizards for a few more frameworks, but then no support for layout editing and more stuffs. It's a great tool, but not for quick development, IMHO. And wxwidgets is an old and heavy framework.
Theide is focused on Upp, has a good layout editor, is still young in particular in respect to help

system and (IMO) debugger, but is lightweight and fast to use. You could use it with wxwidgets or other tools but.... why ?

Ciao

Max

Subject: Re: C++ FQA
Posted by cbpporter on Mon, 12 Nov 2007 09:28:23 GMT
View Forum Message <> Reply to Message

Just a note on VCL. It stands for Visual Components Library (or something similar) and it was developed a visual components library (), and as such other features where introduced to support the visual stuff. You can't expect it to be a general purpose library. But as the version number increased, the library got a good deal of non-visual features. But you still needed a good systems interaction library if you didn't want to use windows API. And under Delphi, not BCB, using it felt more integrated. I couldn't shake the feeling when working in BCB that VCL for C was more of an afterthought than a real alternative. An BCB 1.0 was extremely buggy.

Subject: Re: C++ FQA
Posted by mdelfede on Mon, 12 Nov 2007 10:43:34 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 12 November 2007 10:28Just a note on VCL. It stands for Visual Components Library (or something similar) and it was developed a visual components library (), and as such other features where introduced to support the visual stuff. You can't expect it to be a general purpose library. But as the version number increased, the library got a good deal of non-visual features. But you still needed a good systems interaction library if you didn't want to use windows API. And under Delphi, not BCB, using it felt more integrated. I couldn't shake the feeling when working in BCB that VCL for C was more of an afterthought than a real alternative. An BCB 1.0 was extremely buggy.

Yes, BCB 1.0 was buggy, but following versions were more than ok.
BTW, the buggiest Borland product I've ever seen was C#builder 1.0 ... That one was quite unusable.

But VCL is a good *visual* library, with some more stuffs. And, besides, you could get many more components for BCB/Delphi on open source/shareware market.

Subject: Re: C++ FQA
Posted by exolon on Mon, 12 Nov 2007 12:09:19 GMT

unodgs wrote on Wed, 07 November 2007 09:40Quote:So, I would say for now we do not have any adequate alternatives for C++.
Some claim it is D (www.digitalmars.com/D). At least it has comparable set of features.
D does seem good, and its built-in string handling and GC are apparently fast.
Check out the computer language benchmarks game for some tests of various languages.

Personally, I've been looking into programming in Eiffel (with the free SmartEiffel compiler) lately, but there seems to be no useful GUI library... the built-in one supplied with the compiler libraries, "Vision", is so ugly it hurts.

p.s. Funny quote from the FQA"There's a basic assumption behind C++ that extra features can't be a problem - only missing features can. That's why there are so many features in C++, and in particular so many duplicate ones. Real world analogies ("imagine a dog with twelve legs") are pale compared to this reality."

Subject: Re: C++ FQA
Posted by Mindtraveller on Mon, 12 Nov 2007 14:54:07 GMT

It is funny, I`m criticizing the tool, I`ve been using for years. Just to make clear - programms written under BCB were my main job throughout these years. Yes, while I used it heavily, I just wanted something better - and this is purpose of my being here. Event this critics aimed not to blame Borland with VCL, but to think of something better and more advanced than good old VCL.

It was also mentioned above, that VCL (it really stands for Visual Components Library) - is not framework, it is simple bunch of approaches and code for rapid GUI writing. I`ve forgotten this, considering VCL a complete framework - it`s wrong of course.

Quote:I don't see anything bad on loading forms from a resource file and/or from exe file
The main issue is that you need to have all components in your code - because application doesn`t know which components are used at the compile time. This way we have very-big-applicaction-code. Borland solved this by dividing components by packages, which you can plug-in (or not) in your app.
And, yes, they managed to make serialization rather quick. I wonder if we`d have even more effective and fast-starting applications if VCL used more simple ways to start.

The second issue about working with these resources is that heavy graphics usage led to fast-growing exe files - because picture resources where also included into text resources in the form of textual binary data representation:

```
Glyph.Data = {
  76010000424D760100000000000076000000280000002000000010000000100
  0400000000000001000012B0000120B0000100000000000000000000000000000
  8000008000000080800080000000800080008080000007F7F7F00BFBFBF000000
```

FF0000FF000000FFFF00FF000000FF00FF00FFFF0000FFFFFF00555555555555
5000555555555555577755555555555550B0555555555555F7F7555555555550
00B055555555555777575555555555550B3B05555555555F7F557555555555000
3B055555555557775575555555555500B3B055555555557755575555555555550B3B
055555FFFF5F7F5575555700050003B055555777757775575555570BBB00B3B05
555577555775557555550BBBBBB3B05555557F555555575555550BBBBBBB0555
55557F55FF557F5555550BB003BB075555557F577F5575F5555577B003BBB055
555575F7755557F5555550BB33BBB0555555575F555557F555555507BBBB0755
55555575FFFF77555555555700007555555555577777755555555}

The third issue about text resources is that one may "hack" application. Imagine, you have access rights in your banking operations program. Operator access rights don`t allow him to move money from one account to another - where app simply disables unavailable buttons (this is very common approach).
Ok. You just run special resource editor (there are visual ones also) and make button enabled by default. Or add button with the same handlers, or 1000 more ways to change how your program works - without actual debugging, back engineering, etc. Of course, there are 3rd party utilities encrypting .exe, but think of how many programs are written by people who actually can`t imagine that their programms can be hacked in any moment, without even installing a debugger and living all the application code alone.
Once I was needed to try some application, but it had password protection. I applied some resource hacks and entered it (more just for fun). But I just didn`t like an idea that anyone else may do the same with apps I`ve written.

Quote:Ironically, it seems like U++ starts to be quite ide supported too Yes, but U++ restrictions are way too wider than common IDE`s. Also, I like the way of programming U++ proposes to me. After many problems with standard GUI approaches, to have components in class, where they are needed - is like breath of fresh air for me. That is why I don`t see any problems with U++ restrictions on my code (maybe I`m wrong with this, but alternative restrictions seem to me far worse).
Yesterday I`ve finished rewriting some of my helper classes for U++ (previuosly created for BCB) - the ones I commonly used in my apps. The first version of one of them (ConveyorThread) - being posted into forum. It turned that they were four times smaller, and all my tool application, rewritten under U++, fit into 10Kb of source code (without helper classes). This is more than three times smaller than BCB version - even to mention I`m no professional in U++ for now.

Quote:D does seem good, and its built-in string handling and GC are apparently fast.It was mentioned above that D`s GC may sometimes hang your application for 2-3 seconds. It soesn`t sound like good for serious programms. Also, I don`t like an idea that something else would control freeing of blocks I allocated. I don`t think that good-structured code have much problems with loosing allocated memory. More problematic for me personally is keeping things under control when you heavily use pointers with address arithmetics on many types. It`s very effective and quick (and sometimes necessary) but very dangerous.

Subject: Re: C++ FQA

Posted by mdelfede on Mon, 12 Nov 2007 15:47:37 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Mon, 12 November 2007 15:54
.....
The third issue about text resources is that one may "hack" application. Imagine, you have access rights in your banking operations program. Operator access rights don`t allow him to move money from one account to another - where app simply disables unavailable buttons (this is very common approach).
....

brrrrrrrrr ! Thinkin' that an app is 'secure' just hiding some buttons makes me think to take my (few) money from bank and put it under my bed....
With modern debuggers, it's a child game to hack a program, with or without resources embedded on it.
With data sensible apps, the only way is to rely on passwords, cryptography and such techniques, not on 'hidden data' inside executables !
Quote:
Quote:Ironically, it seems like U++ starts to be quite ide supported too Yes, but U++ restrictions are way too wider than common IDE`s. Also, I like the way of programming U++ proposes to me. After many problems with standard GUI approaches, to have components in class, where they are needed - is like breath of fresh air for me. That is why I don`t see any problems with U++ restrictions on my code (maybe I`m wrong with this, but alternative restrictions seem to me far worse).
.....

BTW, I don't see nothing bad on a gui approach to Upp.
It's a good framework and a good layout editor/Rad tool will help people to jump in.
In modern programs more than 60-70% of work is spent on layouts, appearance, ecc ecc... So having a good framework that is also a good 'visual' framework is surely a plus.

Subject: Re: C++ FQA
Posted by cbpporter on Mon, 12 Nov 2007 15:51:38 GMT
View Forum Message <> Reply to Message

Quote:It was mentioned above that D`s GC may sometimes hang your application for 2-3 seconds. It soesn`t sound like good for serious programms. Also, I don`t like an idea that something else would control freeing of blocks I allocated. I don`t think that good-structured code have much problems with loosing allocated memory. More problematic for me personally is keeping things under control when you heavily use pointers with address arithmetics on many types. It`s very effective and quick (and sometimes necessary) but very dangerous.
Well these execution freezes are not worse than in JVN or .NET platforms. Actually, they can even be shorter. I would like to see some real-life samples of GC performance, not just speculation or my personal experience. Have you ever used a bigger .NET or JVM application. For example Eclipse (I don't know of any big applications in .NET). D applications would seem

somewhat livelier than these, not because of GC, but because of generally better performance and less laggy GUI.

And in D you never need pointer arithmetics. You don't even need to bother with pointers and allocations at all, except when working with C libs and once in a while when you have a shared object and need to clone in first (nasty bug).

Subject: Re: C++ FQA
Posted by mirek on Mon, 12 Nov 2007 19:15:25 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 12 November 2007 10:51
I would like to see some real-life samples of GC performance, not just speculation or my personal experience. Have you ever used a bigger .NET or JVM application.

BTW, I did only that single benchmark D vs U++ - U++ was about 2x faster, but what was really shocking is that D consumed 5 times as much memory....

Quote:
And in D you never need pointer arithmetics. You don't even need to bother with pointers and allocations at all, except when working with C libs and once in a while when you have a shared object and need to clone in first (nasty bug).

I guess it is the same for U++. OTOH sometimes pointer arithemtic is handy....

Mirek

Subject: Re: C++ FQA
Posted by mirek on Mon, 12 Nov 2007 19:21:20 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Mon, 12 November 2007 09:54
The second issue about working with these resources is that heavy graphics usage led to fast-growing exe files

BTW, I am proud to say that U++ is now highly optimized here; not only are .iml compressed using zlib, but even more importantly, several images are always compressed in a single block; means you compress usually about 4KB of often related data together.

Mirek

Subject: Re: C++ FQA
Posted by cbpporter on Mon, 12 Nov 2007 20:39:18 GMT
View Forum Message <> Reply to Message

Quote:
BTW, I did only that single benchmark D vs U++ - U++ was about 2x faster, but what was really shocking is that D consumed 5 times as much memory....

As said before, D has a known performance problem with built in hash maps. I guess the developers are more concerned with the shape of the language and haven't had time to optimize such details yet.

As for the memory consumption, I am not surprised at all. Mark and sweep abandons traditional memory paradigms as allocation when you need and freeing again when you are done with the object. As much as you can optimize these algorithms, allocation and memory freeing are very time consuming operations (relatively speaking of course). In mark and sweep, allocation consists of an if to see if there is enough space. If not, a huge block is allocated. If there is enough space, a simple pointer incrementation is performed. Very fast. You continue to allocate and only do time costly deallocations when you are out of physical memory or have reached a certain threshold. Another advantage is zero memory fragmentation is the mark-and-sweeper is also a moving-compactor. And if you make it generational too, you can really optimize it. So in theory, GC programs should run a lot faster. Memory is cheap, and if with 1-2 GB of memory extra you can gain sufficient extra speed, I think GC will continue to get more and more popular. But this is only theory, and that's why I'm interested in some scientifically sane benchmarks.

Subject: Re: C++ FQA
Posted by mirek on Mon, 12 Nov 2007 21:43:31 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 12 November 2007 15:39Quote:
BTW, I did only that single benchmark D vs U++ - U++ was about 2x faster, but what was really shocking is that D consumed 5 times as much memory....

As said before, D has a known performance problem with built in hash maps. I guess the developers are more concerned with the shape of the language and haven't had time to optimize such details yet.

As for the memory consumption, I am not surprised at all. Mark and sweep abandons traditional memory paradigms as allocation when you need and freeing again when you are done with the object. As much as you can optimize these algorithms, allocation and memory freeing are very time consuming operations (relatively speaking of course). In mark and sweep, allocation consists of an if to see if there is enough space. If not, a huge block is allocated. If there is enough space, a simple pointer incrementation is performed. Very fast.

Sorry for being rude, but your understanding of actual GC (and especially conservative GC) is sort of lacking

Quote:
Another advantage is zero memory fragmentation is the mark-and-sweeper is also a moving-compactor.

Not in all cases and AFAIK, not in D. I might be wrong, but I am afraid that D, using conservative GC, is still prone to memory fragmentation.

Quote:
And if you make it generational too, you can really optimize it.

I am no expect in GC and perhaps I am wrong about this, but IMO generational GC and moving GC are mutually exclusive.

Quote:
So in theory, GC programs should run a lot faster. Memory is cheap, and if with 1-2 GB of memory extra you can gain sufficient extra speed, I think GC will continue to get more and more popular. But this is only theory, and that's why I'm interested in some scientifically sane benchmarks.

...or you can do twice as much work with existing hardware....

Anyway, I think the main argument w.r.t. GC is management of other resources than memory. With GC, it is virtually impossible. And no, you cannot have destructors and GC working together.

I believe that with a couple of tricks, I am getting (with U++) more than I could get with GC - all resources are managed by program structure.

Mirek

Subject: Re: C++ FQA
Posted by cbpporter on Mon, 12 Nov 2007 22:22:28 GMT
View Forum Message <> Reply to Message

Quote:
Sorry for being rude, but your understanding of actual GC (and especially conservative GC) is sort of lacking.

That could very well be the case, but I believe I have some if not enough teoretical and practical experience. What exactly did I say that was inaccurate? I was referring to generic garbage collection (no ref-counting though), but biased toward mark and sweep style algorithms, not exactly to D's implementation (about which I have only superficial knowledge). And what do you mean by conservative GC?

Quote:
I am no expect in GC and perhaps I am wrong about this, but IMO generational GC and moving GC are mutually exclusive.

Well a moving GC (which makes little sense without a compacting GC) moves memory chunks around when neaded and modifies pointers to point to the new locations. A generational GC simply optimizes the allocated objects list so that objects that were created recently are faster to deallocate than old objects (plainly put), because newly created objects have a larger change to get destroyed. So I don't see any reason for them to be mutually exclusive. I could be wrong though.

Quote:
And no, you cannot have destructors and GC working together.

Well you can. With a little extra care, you can have fully functional destructors (just be sure never to physically deallocate memory, just do cleanups). But with GC you rarelly need non-trivial destructors. And if the programing language has a "scope" clause like D, things get a lot simpler.

Quote:
I believe that with a couple of tricks, I am getting (with U++) more than I could get with GC - all resources are managed by program structure.

No arguing here. This is one of the main reasons I like U++. But I would say "almost all", because once in a while you do need to manually manage memory outside of program structure. But those cases are insignificant to overall code size and scope.

Subject: Re: C++ FQA
Posted by mdelfede on Mon, 12 Nov 2007 22:33:40 GMT
View Forum Message <> Reply to Message

Well, also being absolutely not an expert of GC algorithms, I always think that to allocate a lot of memory just because there is enough of it it's not a great practice.
First thing, you must look to what does OS in respect of free memory.... If your OS tells you that you've got 1 GB free ram, even when it's already swapping out another GB on disk, what this behaviour does is slowing down your system.

IMHO a truly efficient GC should be hardware implemented, or at least have a strong hardware support. Doing it software you'll face everytimes with some sort of problem, as latency, memory inefficiency or both.
And it should also collect freed memory asap.
Cbpporter spoke about some sort of 'threshold' for collecting garbage.... but what should be such a threshold ? On oldest OS that was simple, no swapping mechanics, so you could decide, 30% of physical ram, ok. On modern OS, you can't. That depends on too many factors, on how many processes are running, and so on. You could implement a maybe efficient GC only at OS level,

not at the application level, I guess.

No doubt that manual memory allocation is more efficient than GC, even if it can be a bot slower on the short time.
And a good framework can help to keep things simple.

Id rather extend C++ (or make some more modern language, without GC) to include some helpful features, than switch to less efficient languages.

Ciao

Max

Subject: Re: C++ FQA
Posted by mirek on Mon, 12 Nov 2007 22:34:32 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 12 November 2007 17:22Quote:
Sorry for being rude, but your understanding of actual GC (and especially conservative GC) is sort of lacking.

That could very well be the case, but I believe I have some if not enough teoretical and practical experience. What exactly did I say that was inaccurate? I was referring to generic garbage collection (no ref-counting though), but biased toward mark and sweep style algorithms, not exactly to D's implementation (about which I have only superficial knowledge). And what do you mean by conservative GC?

Well, your description of allocation as only "moving the pointer" is accurate (IMO, again) for moving collector. Anyway, D, by principle, has conservative GC, which cannot be moving (because you simply do not know where pointers are, therefore you cannot adjust them).

Obviously, for non-moving collectors, allocations are more complicated and fragmentation exists as well.

Quote:
Quote:
And no, you cannot have destructors and GC working together.

Well you can. With a little extra care, you can have fully functional destructors (just be sure never to physically deallocate memory, just do cleanups). But with GC you rarelly need non-trivial destructors. And if the programing language has a "scope" clause like D, things get a lot simpler.

No, you cannot scope is nice, but IMO limited. And finalizers are not destructors, if anything else, they are asynchronous.

Alternatively, you can suggest that GC only cares about memory management and destructors are on demant, but in that case you simply reduce GC to sort of leak checker... (because you have to carefully watch that all destructions do not leave GC collected destructable orphans).

Mirek

Subject: Re: C++ FQA
Posted by Zardos on Tue, 13 Nov 2007 00:56:25 GMT
View Forum Message <> Reply to Message

luzr wrote on Sat, 10 November 2007 17:54cbpporter wrote on Sat, 10 November 2007 11:31
And I'm quite surprised to see people who don't like gc, but have nothing against reference counting, which is slower and almost impossible to use efficiently in a multi-threading application.

I mostly agree with this...

Mirek

If we talk about a naive reference counting implementation I agree without hestiation.

About the prejudice: "GC is fast" / "Reference counting is slow":

I have read 3 papers approximately one year ago which showed reference counting can be as fast as garbage collection. In addition there are some benefits. Unfortunately I don't have the links anymore. But you may try goolgle with something like "cycles reference counting" and read the /newer/ papers!

But some basic ideas I still remember:

1.) Aggressive removal of AddRef/RemRef by a static flow analysis of the programm: The idea is to only inc. or dec. reference counters if neccessary. Usually only if a (smart)pointer manipulation is done inside a struct/class/... It is not neccessary to do inc./dec. operation for (smart)pointers on the stack or as function arguments. Probably special care has to be taken for threads. Even for (smart)pointers inside structs many inc./dec. operation can be elimanated by a (not so complicated) static analysis of the programm flow.

2.) Avoiding "Atmoic operations": The simple but effective idea is this. Instead of doing AtmoicInc/AtomicDec operations directly on the refence counters all Increments and Decrements are logged in a buffer with a fixed length. Each thread has it's own buffers! So, no locks are required for the ordinary Inc/Dec. If the buffer is full the buffer is given to a "memory management" thread. This requires a Lock operation, but if we assume a buffer has place for 10000 "Inc/Dec" ops. the synchronization cost boils down to 1/10000 which is negligible.
The memory management thraed finally can perform the inc/dec operations without the expensive "Atmoic" vesions just ordinary ++/--
Benefit: The memory management thread can be tuned to only do a fixed number of delete

operations per second. This avoids "cascading deletes" where one destructer call triggers a chain of delete operations. => Even with manual memory management there can be stalls!

=> So we have a concurent - nearly log free - reference counting garbage collection.

3.) Cycles: I have forgotten how they get removed. But it was not to complicated to understand. Yes it costs time to detect the cycles... The cost is reduced by point 1.). In addition the cylcle detection can be performed by the memory management - concurrently - without any additional locks. And again this memory management can be tuned to do only N. operations per second. To really spread the memory mangament cost over time (Of course, this means for some time periods more memory is allocated than necessary).

I think it's a really interesting topic. I always had the feeling Refernce Counting can be supperior over garbage collection... May be the furture brings a revival of the counters....

EDIT: I think 2.) was implemented differently: Instead of having buffer with a fixed length: The memory management thraed performed a "stop the world" (pausing all threads) periodically and fetched the buffers from all running threads. While stop the world sounds like stalling - you have to remember only some pointers to buffers have to be transfered to the menory management thread... After the transfer the world starts rotating again.

Subject: Re: C++ FQA
Posted by Mindtraveller on Tue, 13 Nov 2007 01:41:38 GMT
View Forum Message <> Reply to Message

luzrnot only are .iml compressed using zlib, but even more importantly, several images are always compressed in a single block More I know, more I like it

cbpporterWell these execution freezes are not worse than in JVN or .NET platforms. Actually, they can even be shorter. I would like to see some real-life samples of GC performance, not just speculation or my personal experience. Have you ever used a bigger .NET or JVM application.Just don`t get me wrong: I develop nearly real-time applications (not RTA actually at all when we discuss Windows issues). I do work with actual hardware devices with a number of protocols. It all runs under highly truncated version of Windows, which doesn`t know much of system-hanging device drivers like CD-ROM ones. So, generally we have no big problems with OS latensy on protocol timeouts like 50-500 msecs.
This way neither Java or .NET, nor similar "heavy" platforms can be used (usually those industrial computers are not that quick as Pentium3/4 to support virtual machines, and memory installed may be below even 64/128 MB).
But I need to use GUI as much as time-critical code working with hardware devices. It all of course is devided into different threads etc. The thing I need most - is efficiency and predictability, afterwards - ease of use.
This all situation makes GC or other hanging solutions totally inacceptable for my tasks. Huge memory consumption is inacceptable too.
Besides, I don`t like complex solutions in a situation when simple solution can be applied without high cost. So that U++ satisfies me by a number of criteria, becoming succesful replacement for Borland C++ Builder.

You can throw rocks at me, but I don`t see necessarity of having general purpose GC when you have experience with constructing/destructing of objects. It all may be good at learning-style languages like basic, but well-organized code generally knows where and when to delete objects - more of that, you may choose a moment to do costly memory operations - when they are the mostly invisible for user (knowing program logic). I just don`t believe that GC can determine these (most effective) moments automatically.

Also.
Speaking about reference counting and GC in such a general way makes hard to think of what is really better. Maybe it would be more efficient to discuss some real-life class to be more spicific and sure in reasons. Just a thought.

Subject: Re: C++ FQA
Posted by mirek on Tue, 13 Nov 2007 08:25:20 GMT
View Forum Message <> Reply to Message

Zardos wrote on Mon, 12 November 2007 19:56
2.) Avoiding "Atmoic operations": The simple but effective idea is this. Instead of doing AtmoicInc/AtomicDec operations directly on the refence counters all Increments and Decrements are logged in a buffer with a fixed length. Each thread has it's own buffers!

Yes, considered this one too. I guess this is promissing idea, but there are many caveats.

Quote:
EDIT: I think 2.) was implemented differently: Instead of having buffer with a fixed length: The memory management thraed performed a "stop the world" (pausing all threads) periodically and fetched the buffers from all running threads. While stop the world sounds like stalling - you have to remember only some pointers to buffers have to be transfered to the menory management thread... After the transfer the world starts rotating again.

Yep, you need to process all buffers unfortunately, otherwise you would get false deletes.

Mirek

Subject: Re: C++ FQA
Posted by cbpporter on Tue, 13 Nov 2007 09:14:46 GMT
View Forum Message <> Reply to Message

Quote:Just don`t get me wrong: I develop nearly real-time applications (not RTA actually at all when we discuss Windows issues). I do work with actual hardware devices with a number of protocols. It all runs under highly truncated version of Windows, which doesn`t know much of system-hanging device drivers like CD-ROM ones. So, generally we have no big problems with OS latensy on protocol timeouts like 50-500 msecs.

This way neither Java or .NET, nor similar "heavy" platforms can be used (usually those industrial computers are not that quick as Pentium3/4 to support virtual machines, and memory installed may be below even 64/128 MB).
I understand that and I was not trying to convince you to use GC on those machines. I was just giving arguments that a generic GC on a modern PC or similar hardware can save a lot of pain. You can live without it, doing all the memory management manually like in other frameworks, you can give smart pointers from BOOST a shot, you can use something like U++ or you can use GC (even with C++; AFAIK D GC is available/inspired by the C++ GC). For people with a lot of experience and good technical know-how, it's just a matter of compromise between choice and the needs of the project. Yet really knowledgeable C++ programmers are not that common, and managers often have teams formed of a little less experienced programmers. These are the people who introduce memory leaks and who misuse language features, not the guys who can perfectly understand were complex template code with multiple inheritance, virtual inheritance and some hidden macro-magic at the first glance. In such situations, GC can be a good alternative. Also, training time and cost for these programmers is reduced and there are people who refuse to go all the way and acquire full C++ feature knowledge, remaining in a "C with classes + some design patterns(+ use of the STL without really understanding it; still so many hand written vectors, lists, etc. in commercial code)"

Subject: Re: C++ FQA
Posted by mirek on Tue, 13 Nov 2007 09:52:41 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 13 November 2007 04:14Quote:Just don`t get me wrong: I develop nearly real-time applications (not RTA actually at all when we discuss Windows issues). I do work with actual hardware devices with a number of protocols. It all runs under highly truncated version of Windows, which doesn`t know much of system-hanging device drivers like CD-ROM ones. So, generally we have no big problems with OS latensy on protocol timeouts like 50-500 msecs.
This way neither Java or .NET, nor similar "heavy" platforms can be used (usually those industrial computers are not that quick as Pentium3/4 to support virtual machines, and memory installed may be below even 64/128 MB).
I understand that and I was not trying to convince you to use GC on those machines. I was just giving arguments that a generic GC on a modern PC or similar hardware can save a lot of pain. You can live without it, doing all the memory management manually like in other frameworks, you can give smart pointers from BOOST a shot, you can use something like U++ or you can use GC (even with C++; AFAIK D GC is available/inspired by the C++ GC). For people with a lot of experience and good technical know-how, it's just a matter of compromise between choice and the needs of the project. Yet really knowledgeable C++ programmers are not that common, and managers often have teams formed of a little less experienced programmers. These are the people who introduce memory leaks and who misuse language features, not the guys who can perfectly understand were complex template code with multiple inheritance, virtual inheritance and some hidden macro-magic at the first glance. In such situations, GC can be a good alternative. Also, training time and cost for these programmers is reduced and there are people who refuse to go all the way and acquire full C++ feature knowledge, remaining in a "C with classes + some design patterns(+ use of the STL without really understanding it; still so many hand written vectors, lists, etc. in commercial code)"

Well, of course. Nobody disputes that finding average Java programmer is much more simple than hiring good C++ programmer - and that hiring average C++ programmer is a threat to the project...

Mirek

Subject: Re: C++ FQA
Posted by exolon on Tue, 13 Nov 2007 16:30:03 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Mon, 12 November 2007 22:33Well, also being absolutely not an expert of GC algorithms, I always think that to allocate a lot of memory just because there is enough of it it's not a great practice.
First thing, you must look to what does OS in respect of free memory.... If your OS tells you that you've got 1 GB free ram, even when it's already swapping out another GB on disk, what this behaviour does is slowing down your system.

IMHO a truly efficient GC should be hardware implemented, or at least have a strong hardware support. Doing it software you'll face everytimes with some sort of problem, as latency, memory inefficiency or both.
And it should also collect freed memory asap.
Well, these last two statements don't seem logical together. If you don't want latency, why collect freed memory ASAP, unless you're on a very constrained (hard real-time embedded board...?) platform always operating close to the limit of available memory or starting to thrash the swap? This implies doing GC runs _all_ the time.

mdelfede wrote on Mon, 12 November 2007 22:33No doubt that manual memory allocation is more efficient than GC, even if it can be a bot slower on the short time.
And a good framework can help to keep things simple.

Id rather extend C++ (or make some more modern language, without GC) to include some helpful features, than switch to less efficient languages.
I think you should really look at some proper comparisons of real efficiency impacts of using GC, rather than automatically assuming that it kills your program's performance.

Also, you shouldn't just assume for sure that manual memory allocation must be more efficient than GC.
This FAQ is worth a read, with an open mind, rather than being a hardened C++ "oldskool all the (hard and error-prone) way" purist.

http://www.iecc.com/gclist/GC-faq.htmlFolk myths
* GC is necessarily slower than manual memory management.
* GC will necessarily make my program pause.
* Manual memory management won't cause pauses.
* GC is incompatible with C and C++.
Folk truths
* Most allocated objects are dynamically referenced by a very small number of pointers. The most

important small number is ONE.
* Most allocated objects have short lifetimes.
* Allocation patterns (size distributions, lifetime distributions) are bursty, not uniform.
* VM behavior matters.
* Cache behavior matters.
* "Optimal" strategies can fail miserably.

Subject: Re: C++ FQA
Posted by mdelfede on Tue, 13 Nov 2007 17:24:18 GMT
View Forum Message <> Reply to Message

exolon wrote on Tue, 13 November 2007 17:30
Well, these last two statements don't seem logical together. If you don't want latency, why collect freed memory ASAP, unless you're on a very constrained (hard real-time embedded board...?) platform always operating close to the limit of available memory or starting to thrash the swap? This implies doing GC runs _all_ the time.

When I say "I don't want latency", I mean "I don't want a big, unpredictable latency".
As I said, GC *can* globally be more performant, and *can* be in the very short time very performant too. What i can't is assure the performance *all* the time.
If I do a benchmark with a lot of allocations/deallocations, GC may behave wonderful. But then, if you stress benchmark beyond a threshold, you'll have your app stopping for maybe 2-3 seconds collecting stuffs. If you go forth, global app time *can* be shorter than with manual allocation, but in the middle you had the infamous stop.

Besides real-time tasks (on wich that's not acceptable), there are also many apps on which you couldn't accept such a behaviour, like multimedia, but even in a GUI a 3 second stop makes a bad feeling.

Quote:
I think you should really look at some proper comparisons of real efficiency impacts of using GC, rather than automatically assuming that it kills your program's performance.

As I said before, one must know what "killing program performance" means. if you accept that your app can lag for seconds, GC is the best for you.

Quote:
Also, you shouldn't just assume for sure that manual memory allocation must be more efficient than GC.
This FAQ is worth a read, with an open mind, rather than being a hardened C++ "oldskool all the (hard and error-prone) way" purist.

I'm surely not a purist, as I tried so many languages in the past. I'd prefere a more modern C++, but with no GC, but that's of course my opinion. I like very much C# syntax, for example, but you can't call him a fast or multipurpose language. When Java came out, it seemed to all people that it'd be the language of future.... but now I don't see many people thet like it. Nor it seems to me a

quick lang, or a very error-free one. D language seems interesting, but I didn't test it.
Of one thing I'm quite sure : I don't want a C++ with memory management mixed between malloc() and GC.

BTW, in the faq you showed, the focus seems to be on "you program is surely buggy, it shourely have leaks, so GC is wonderful for you". I receive about 100 spam mails in this idiom each day
Quote:
.....
* Most allocated objects are dynamically referenced by a very small number of pointers. The most important small number is ONE.

for that, the best is pick_ behaviour, of course

Quote:
* Most allocated objects have short lifetimes.

And ? does it mean that GC is better or not ?

Max

Subject: Re: C++ FQA
Posted by exolon on Tue, 13 Nov 2007 17:58:55 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Tue, 13 November 2007 17:24As I said, GC *can* globally be more performant, and *can* be in the very short time very performant too. What i can't is assure the performance *all* the time.
If I do a benchmark with a lot of allocations/deallocations, GC may behave wonderful. But then, if you stress benchmark beyond a threshold, you'll have your app stopping for maybe 2-3 seconds collecting stuffs. If you go forth, global app time *can* be shorter than with manual allocation, but in the middle you had the infamous stop.

Besides real-time tasks (on wich that's not acceptable), there are also many apps on which you couldn't accept such a behaviour, like multimedia, but even in a GUI a 3 second stop makes a bad feeling.
I find it hard to believe that modern GC implementations would cause a lag of 3 seconds in a normal GUI or even multimedia app.
You talk about a stress test, but surely this isn't really the general case?

So it looks to me like you're talking about special, extreme cases; the worst-case scenarios for a stopping GC (there are incremental GC schemes too, but I don't see that they exist in the real world).

Do you really prefer handling all the memory (de)allocation, when objects are being passed around and you don't know how many times a function will necessarily be called... in every case? In complex programs?

I respect your choice to favour whatever mechanism, manual or GC (and I certainly don't think you're stupid if your choice is different to mine - I make stupid choices all the time) but I really don't think the overhead is as bad as people make out.
If it turns out that GC is as good or nearly as good as manual management in 90% of cases, then I would definitely be going that route, unless we're dealing with cases where absolutely no unexpected latency can be introduced whatsoever... i.e. real-time with hard deadlines, in which case we'd probably be using something like C in uCOS or even assembly, anyway.

That said, I do understand that there's a non-performance-related problem with calling destructors during GC - namely that of two objects about to be collected, say FileReader and FileHandle, FileReader has a reference to FileHandle, and in its destructor calls handle->Close() or something, which will die if the FileHandle happens to be destroyed first (since the only reference to it is another 'dead' object), and I guess this might be what Luzr alluded to.
So I'm not sure what the best solution there is, without explicitly calling some "finalise" type method which would be just as annoying as deleting it anyway.

[edit]
Afterthought... then again, why not just have Close() in FileHandle's destructor, rather than having the client object do the cleanup?
Anyway, I'm sure there are cases out there that are more valid than my crappy example.

Subject: Re: C++ FQA
Posted by Mindtraveller on Thu, 15 Nov 2007 23:43:08 GMT
View Forum Message <> Reply to Message

It looks like a temporal armistice.
Just to have fun, some words about C++ from The-God-Of-Free-Whatever-Anything-You-Know - Linus Torvalds

---

Subject: Re: C++ FQA
Posted by mirek on Fri, 16 Nov 2007 10:59:33 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Thu, 15 November 2007 18:43It looks like a temporal armistice.
Just to have fun, some words about C++ from The-God-Of-Free-Whatever-Anything-You-Know - Linus Torvalds

Yep, I know that one.

Perhaps he could try to compete with idmap benchmark in C

Mirek

Subject: Re: C++ FQA
Posted by waxblood on Fri, 16 Nov 2007 15:24:57 GMT
View Forum Message <> Reply to Message

Git looked interesting to me, but after reading 'that one', I *need* Uvs3. Really.

David

Subject: Re: C++ FQA
Posted by mdelfede on Fri, 16 Nov 2007 15:37:03 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Fri, 16 November 2007 00:43It looks like a temporal armistice.
Just to have fun, some words about C++ from The-God-Of-Free-Whatever-Anything-You-Know - Linus Torvalds

Well, I did read (a part of) that thread.... interesting stuff.

extremistic. I can agree that one likes a language more than others, but saying that others are bullshit IMO doesn't solve problems nor apports something constructive to the discussion.
Second, I think I'll give D language a shot
Third, I think more and more that the usual argument of C supporters that "C++ is BS, because

get much spare time to check my code for bugs or to enhance it. Such arguments make me think that if one does all the job in assembler, it will be surely bug free .

ciao

Max

Subject: Re: C++ FQA
Posted by mirek on Sat, 17 Nov 2007 09:04:02 GMT
View Forum Message <> Reply to Message

exolon wrote on Tue, 13 November 2007 11:30
http://www.iecc.com/gclist/GC-faq.htmlFolk myths
* GC is necessarily slower than manual memory management.
* GC will necessarily make my program pause.
* Manual memory management won't cause pauses.
* GC is incompatible with C and C++.
Folk truths
* Most allocated objects are dynamically referenced by a very small number of pointers. The most important small number is ONE.
* Most allocated objects have short lifetimes.
* Allocation patterns (size distributions, lifetime distributions) are bursty, not uniform.

* VM behavior matters.
* Cache behavior matters.
* "Optimal" strategies can fail miserably.

BTW, if you want to see some real data, U++ has quite good heap statistics feature, up to the point of simple GUI in CtrlLib: See MemoryProfileInfo.

Or, for example, in TheIde About dialog, press Alt+M.

Mirek

Subject: Re: C++ FQA
Posted by mirek on Thu, 22 Nov 2007 04:43:16 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 13 November 2007 04:14
Yet really knowledgeable C++ programmers are not that common, and managers often have teams formed of a little less experienced programmers. These are the people who introduce memory leaks and who misuse language features, not the guys who can perfectly understand were complex template code with multiple inheritance, virtual inheritance and some hidden macro-magic at the first glance. In such situations, GC can be a good alternative.

BTW, interesting anecdotal evidence from the current project I am involved in:

It is now medium sized U++ application being developed "under the pressure" by a team of now 5.

2 of us were hired two months ago, with C# background. They literally had *DAYS* to (re)learn C++ and to learn U++.

Despite that, they are productive (well, certainly not as me , but they do deliver). And they have introduced the first "delete" statement yesterday, which will be removed, with explanation today (and thus will be the last "delete" in the project ever introduced).

So I guess, it is not as bad w.r.t. C++/U++ combo and less experienced programmers... and GC really not as important for productivity here.

Mirek

Subject: Re: C++ FQA
Posted by Mindtraveller on Thu, 27 Dec 2007 13:47:33 GMT
View Forum Message <> Reply to Message

http://shootout.alioth.debian.org/sandbox/benchmark.php?test =all&lang=gpp&lang2=dlang