c++ truths

A BLOG ON VARIOUS TOPICS IN C++ PROGRAMMING INCLUDING LANGUAGE FEATURES, STANDARDS, IDIOMS, DESIGN PATTERNS, FUNCTIONAL, AND OO PROGRAMMING.

tuesday, july 19, 2011

Want speed? Use constexpr meta-programming!

It's official: C++11 has two meta-programming languages embedded in it! One is based on templates and other one using constexpr. Templates have been extensively used for meta-programming in C++03. C++11 now gives you one more option of writing compile-time meta-programs using constexpr. The capabilities differ, however.

The meta-programming language that uses templates was discovered accidently and since then countless techniques have been developed. It is a pure functional language which allows you to manipulate compile-time integral literals and types but not floating point literals. Most people find the syntax of template meta-programming quite abominable because meta-functions must be implemented as structures and nested typedefs. Compile-time performance is also a pain point for this language feature.

The generalized constant expressions (constexpr for short) feature allows C++11 compiler to peek into the implementation of a function (even classes) and perform optimizations if the function uses constants (literals) only. Constants can be integral, floating point, as well as string literals. The signature of constexpr functions is just like regular C++ functions but the body has several restrictions, such as only one return statement is allowed. Nevertheless, the syntax of constexpr functions is significantly friendlier than that of template-based meta-functions. Contrary to the design of templates, designers of generalized constant expressions are well-aware of its meta-programming capabilities.

In my view, the most interesting aspect of constexpr is its speed. constexpr functions can perform compile-time computations at lightening speed. To compare the performance I implemented an is_prime algorithm in 3 different ways. Here is the algorithm in regular C++:

```
static bool IsPrime(size_t number)                                  ?
{
  if (number <= 1)
    return false;

  for (size_t i = 2; i*i <= number; ++i)
    if (number % i == 0)
      return false;

  return true;
}
```

Here is the a template version of the same algorithm. Obviously, it is implemented as a collection of meta-functions in terms of structures.

```
struct false_type                                                   ?
{
  typedef false_type type;
  enum { value = 0 };
};

struct true_type
{
  typedef true_type type;
  enum { value = 1 };
};

template<bool condition, class T, class U>
struct if_
{
  typedef U type;
};

template <class T, class U>
struct if_<true, T, U>
{
  typedef T type;
};

template<size_t N, size_t c>
struct is_prime_impl
{
  typedef typename if_<(c*c > N),
                       true_type,
                       typename if_<(N % c == 0),
                                    false_type,
                                    is_prime_impl<N, c+1> >::type >::type type;
  enum { value = type::value };
};

template<size_t N>
struct is_prime
```

about me

Sumant Tambe
G+ Follow
I am interested in C++, standards-based middleware for distributed systems, model-driven software development, and scalable fault-tolerant distributed systems.

View my complete profile

rss feeds

363 readers BY FEEDBURNER

ShareThis

G+1 +47 Recommend this on Google

Like 64 people like this.

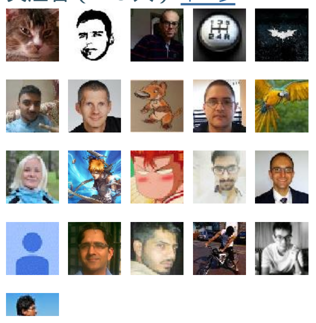
More C++ Idioms Wikibook
Watch this space for future items!
May 1, 2017

↑ Grab this Headline Animator

Writer SUPPORT WIKIPEDIA

followers

关注者（248 人）下一步



subscribe to

Posts

Comments

links

My homepage
The C++ Standards Committee
C++ soup!
LightSleeper
Thinking Asynchronously in C++
More C++ links

blog archive

► 2017 (1)
► 2016 (3)
► 2015 (3)
► 2014 (9)
► 2013 (6)
► 2012 (6)
▼ 2011 (4)
  ► October (1)
  ▼ July (2)

```
{
  enum { value = is_prime_impl<N, 2>::type::value };
};

template <>
struct is_prime<0>
{
  enum { value = 0 };
};

template <>
struct is_prime<1>
{
  enum { value = 0 };
};

```

The above meta-program is a recursive implementation because that's how pure functional languages work. The is_prime_impl meta-function does all the heavy lifting. To prevent infinite regression, lazy instantiation technique is used. All I can say at this point is you've to stare at the code to make sense out of it.

And that's precisely the point: C++03 meta-programs are often unreadable and hard to comprehend. Not anymore! Here is the constexpr version of the same algorithm.

?

```
constexpr bool is_prime_recursive(size_t number, size_t c)
{
  return (c*c > number) ? true :
           (number % c == 0) ? false :
              is_prime_recursive(number, c+1);
}

constexpr bool is_prime_func(size_t number)
{
  return (number <= 1) ? false : is_prime_recursive(number, 2);
}

```

Well, I agree, although this version uses friendlier syntax, is not quite easy to read. Just like the previous one, it is recursive. I would say just get used to it! You can't live without recursion in C++ meta-programming world. Secondly, every function has a single return statement and the codifying logic for detecting primality requires abundant use of the ternary operator.

As long as parameter number is an integral constant, this constexpr version will compute the result at compile-time (C++11 compilers only). And when the number is a run-time integer, this same function is perfectly capable of computing the result at run-time. So you don't need two different versions of the same program: one for compile-time and another for run-time. One implementation does it all!

?

```
int main(void)
{
   static_assert(is_prime_func(7), "...");  // Computed at compile-time
   int i = 11;
   int j = is_prime_func(i)); // Computed at run-time
}

```

Now that we have the implementations, lets talk performance. Take 4256233. This is 30,000th prime number! How long does the template version take to check if it is prime? It cannot! Yes, g++ 4.7 fails to compile is_prime<4256233>::value because the computation exceeds the default (900) limit of the recursive template instantiations. So I used -ftemplate-depth-2100 allowing 2100 recursive instantiations! Does it work? Yes! How long does it take? About 1 second! That's not bad at all, you say. How fast is constexpr? About 0.154 seconds!

Seems like templates are not too far behind. Wrong! How about fifth million prime number! It is 86028121. I could not get it to work with g++ 4.7. Somewhere in the range of 9300 recursive instantiations g++ seg-faults. That's brutal! isn't it? 9000+ recursive instantiations? There has to be a better way.

So how long does the constexpr take for our fifth million prime number? 0.220 seconds! That's right! This large prime number makes almost no impact on constexpr compilation time. What could be the reason? There are no template instantiations. C++ compilers always did compile-time computations whenever it was possible. constexpr now allows user-defined abstractions to hide those computations behind functions and yet allow compilers to see through them.
By the way, I had to increase to depth of default allowed constexpr recursion using -fconstexpr-depth=9300. The compiler bails out after this limit and resorts to run-time computation. Remember, the function is perfectly good for run-time computation as well (unlike templates version).

I hope this little exercise will convince you that constexpr is the way to go for static meta-programming in C++ as long as you are dealing with literals (integral, floating point numbers, and strings). It is not clear whether constexpr functions are suitable for manipulating complex meta-programmable types, such as mpl::vector and related meta-functions, such as mpl::contains, mpl::push_back, and so on. If you are interested in playing with the above example more, here is the code: is_prime.cpp and prime-test.sh

posted by sumant tambe at 5:33 pm
labels: meta-programming constexpr performance

43 comments:

johan kotlinski said...

Compile-time regex matcher using constexpr
Want speed? Use constexpr meta-programming!
► June (1)
► 2010 (2)
► 2009 (2)
► 2008 (8)
► 2007 (18)
► 2006 (20)
► 2005 (27)

Sorry, I don't see exactly what benefit constexpr give over static in this case? The compiler should be able to apply the exact same optimizations no matter what!

Wed Jul 20, 05:23:00 PM PDT
johan kotlinski said...

This comment has been removed by the author.

Wed Jul 20, 05:26:00 PM PDT
ralph zhang said...

Wonderfull for meta programming!
But in terms of running this at runtime, would it be a hell lot of runtime recursion? Does C++0x have anything to say about tail recursion?

Thu Jul 21, 12:16:00 AM PDT
richard smith said...

Have you checked that the compiler actually performed that calculation at compile time? It's not required to do so: there is an implementation-defined limit on the constexpr recursion depth, and if that limit is hit, the resulting value is not a constant expression (the compile-time calculation bails out). The standard recommends that this limit is at least 512, which is a lot lower than the limits both of your examples need.

This would much more naturally explain why you get the exact same runtime for both constexpr evaluations: they're both performing the same number of steps before they give up.

Try:

constexpr i = is_prime_func(982451653);

I expect you'll get a compile-time error because the initializer is not a constant expression.

Thu Jul 21, 05:11:00 AM PDT
richard smith said...

D'oh, of course I meant

constexpr bool i = is_prime_func(982451653);

Thu Jul 21, 05:13:00 AM PDT
coder said...

D programming language has CTFE (Compile Time Function Evaluation) which evaluates most of the regular functions at compile time.

Could somebody explain how it differs from C++0x's constexpr?

Thu Jul 21, 06:58:00 AM PDT
coder said...

Excuse me, I meant Compile Time Function Execution

Thu Jul 21, 07:00:00 AM PDT
sumant said...

@Richard: Thank you very much for pointing out what I was doing wrong! I was not extremely sure why I got the same compilation-time for large as well as small prime numbers. I've updated the blog post now with new numbers. I'm now using static_assert and I had to use -fconstexpr-depth-N option to increase default recursion depth. Nevertheless, it compiles faster than the template version.

Thu Jul 21, 09:09:00 AM PDT
mmocny said...

This is an amazing experiment, thanks!

I am curious, however: may you post timings for comparing runtime execution of the static and constexpr versions?

If the performance of the constexpr is lower at runtime, then we may still need two versions of the same function..

-Michal Mocny

Thu Jul 21, 11:13:00 AM PDT
john haugeland said...

"The meta-programming language that uses templates was discovered accidently"

No, it wasn't. Please don't call your blog "c++ truths" if you're going to spread misinformation like this. There is nothing accidental of any form to what Erwin Unruh built.

Stop repeating fictions you learned on Wikipedia.

Fri Jul 22, 03:11:00 AM PDT
john haugeland said...

"compile-time computations at lightening speed"

Jesus, you really don't understand any of this, do you?

The call cost of a compile-time behavior is zero. "At lightning speed?" Is this all metaphor to you?

Fri Jul 22, 03:12:00 AM PDT
faisal vali said...

Not only that - but you can manipulate and process strings at compile time - which one could not do with templates!
Check out:

1) http://tinyurl.com/45xkdlq
(http://groups.google.com/group/comp.lang.c++.moderated/browse_thread/thread/d9bddd4105f1441e?hl=en#)

Fri Jul 22, 03:14:00 AM PDT
john haugeland said...
Oh man, it gets worse and worse.

"The above meta-program is a recursive implementation because that's how pure functional languages work."

Nonsense. That's how languages like Haskell, OcaML and Erlang work, but that has absolutely nothing to do with being a functional language. For example, CSS is a declarative functional language, and has no calling at all, let alone recursion; Prolog, F# and Postscript do not encourage recursion any more than does C, which is not functional; recursion is very common in C; et cetera.

"The is_prime_impl meta-function does all the heavy lifting. To prevent infinite regression, lazy instantiation technique is used. All I can say at this point is you've to stare at the code to make sense out of it."

The most obvious reaction here is "Have you even looked at that code? It doesn't instantiate anything. It's pretty obvious that the reason that all you've [sic] to say about it is read it: that's because you obviously have no idea how it works."

"And that's precisely the point: C++03 meta-programs are often unreadable and hard to comprehend"

For you, maybe. The thing you just botched explaining is actually clear as day, to someone who speaks C++."

Then you say you also can't read the constexpr version, which is hilarious, because it's a one-liner loop on modulus, suggesting that you really shouldn't be writing a software blog.

"You can't live without recursion in C++ meta-programming world. "

I use C++ TMP every day, and I very rarely do so recursively. You don't know what you're talking about.

"How fast is constexpr? About 0.154 seconds!"

If you're measuring how long the compiler takes and comparing it to how long the app takes in runtime, then you're making a fantastically inappropriate comparison.

That the two pieces of code aren't actually doing the same thing, hilariously, appears to be beyond you.

"9000+ recursive instantiations? There has to be a better way."

Yes, there is, and the typical college freshman can find it.

"That's right! This large prime number makes almost no impact on constexpr compilation time. What could be the reason?"

Well for one, you seem to have confused compile time with how long it takes to compile the source inside, and you appear to have no idea how benchmarking works.

"C++ compilers always did compile-time computations whenever it was possible. "

Incorrect. C++ truths my ass. You're just guessing and presenting your wrong guesses as facts.

"I hope this little exercise will convince you that constexpr is the way to go for static meta-programming in C++ as long as you are dealing with literals"

Translation: "I don't understand the difference between the two, so I'm going to say a bunch of wrong stuff then tell you which one to use."

"It is not clear whether constexpr functions are suitable for manipulating complex meta-programmable types"

It is entirely clear.

What a dishonest embarrassment.

Fri Jul 22, 03:23:00 AM PDT
ralph zhang said...

The gentleman above seems to be really angry about other people talking about c++ and meta-programming.
It's a blog post, people say something that they just found out, it can be OK to you or not. But I don't see any point of posting such an aggressive comment.
It's not shameful at all to get wrong, but it's shameful to be as rude as that.

Fri Jul 22, 05:15:00 AM PDT
john haugeland said...

It is shameful to make guesses about what's happening in a well defined situation and present them as facts.

The primary reason this is a problem is that someone might make the mistake of beleiving him.

I'm sorry you imagine that it's worse to point out a lie than to tell the lie. However, it is not, and that's an aversion which causes very serious unnecessary problems.

Fri Jul 22, 06:37:00 AM PDT
sumant said...

@Johan: Return value of the functions labelled constexpr may be evaluated at compile-time provided all their parameters are known at compile-time.

@Ralph: IMHO, flattening tail recursion is implementation specific. Compilers may or may not be able to do that and their capabilities may vary. The is_prime_recursive is tail recursion 101 so I guess g++ 4.7 flattened it and that's why the performance of static and the recursive is identical.

@mmocny: Run-time were the same (0.004s) for regular version ("static" in the code examples) and the constexpr for really really large values. Note that for really really large values the answers were not computed at compilet-time due to limit of constexpr depth.

Fri Jul 22, 04:23:00 PM PDT
sumant said...

@John: So you blog bullof.bs and now the comments too! LoL!!

Check Chapter #1, section 4 of the “C++ Template Meta-programming” book by David Abrahams and Aleksey Gurtovoy. They talk about accidental discovery of C++ meta-programming.

Spot on with the examples of functional programming languages!

I've seen my code because I wrote it.

I’m glad someone thinks a C++ metaprogram is clear as day! When was it you last saw your code? You know there is more to programming than holding down the spacebar. Oh right! that’s infinite recursion in your part of the world.

Finally, I’m talking compilation time only. No run-time. Please go back and reread.

Now, read it again! Seriously!

Sat Jul 23, 02:32:00 AM PDT
laurelbostan said...

Obtain copy of Official Florida bankruptcy records, Bankruptcy discharge papers $9.99, bankruptcy creditors listing $19.99 and complete file for $29.99 at lowest cost on web.

Mon Jul 25, 11:19:00 PM PDT
johan kotlinski said...

> @Johan: Return value of the functions labelled constexpr may be evaluated at compile-time provided all their parameters are known at compile-time.

But this holds for any function that does not modify external state. So I don't see why it has to be labeled constexpr in this case.

Tue Jul 26, 02:38:00 AM PDT
vivek ragunathan said...

How did you find the time taken for the evaluating a template instantiation and constexpr?

Thu Aug 04, 07:22:00 AM PDT
sumant said...

@Vivek: I just measured the time taken to compile two different programs. I used the "time" command on Linux. This is quite course grain comparison but that't what is visible to programmers.

Thu Aug 04, 09:20:00 AM PDT
mukul said...

The standard recommends that this limit is at least 512, which is a lot lower than the limits both of your examples need.yeast infection treatment ! yeast infection prevention

Tue Aug 16, 01:43:00 AM PDT
anonymous said...

I see that you have run into the pathetic shit known as John Haugeland. Don't worry about him! He's a troll all over the internet. Just check out this post about him on reddit. He's a pathetic loser without a life who gets off on being a douche to others.

Sat Oct 08, 12:48:00 PM PDT
anonymous said...

I see that you have run into the pathetic shit known as John Haugeland. Don't worry about him! He's a troll all over the internet. Just check out this post about him on reddit. He's a pathetic loser without a life who gets off on being a douche to others.

Sat Oct 08, 12:49:00 PM PDT
id said...

@John Haugeland:

The aggressive tone of your comment is totally unnecessary. If you notice, Richard Smith up above you was perfectly capable of correcting a factual error without an aggressive tone.

And, in fact, template metaprogramming was discovered, not designed. The fact you could do metaprogramming with templates was accidental. It is true that Erwin Unruh was the first to discover this fact. But he didn't design the feature into C++ in the first place. He noticed it was there after it had already been accidentally added.

Sat Dec 24, 11:36:00 AM PST
omnifarious (aka eric hopper) said...

Also, I just wrote a constexpr that computes Fibonacci numbers. Interestingly, the compile-time version compiles significantly faster than the run-time version runs. This is because the compile-time version can take advantage of optimizations like memoization.

http://pastebin.com/P3v3eTxY

```
$ time g++ -std=c++0x -pedantic -Wall -Wextra -Ofast cefib.cpp

real 0m0.602s
user 0m0.042s
sys 0m0.027s
$ time ./a.out
fib(46) == 2971215073
fib(46) == 2971215073

real 0m9.163s
user 0m0.994s
sys 0m0.142s
```

Sat Dec 24, 01:24:00 PM PST
sumant said...

@omnifarious: The Fibonacci algorithm you are using, the classic one, is very very slow. Using it to compare compile-time and run-time does not make sense because as you point out compiler uses memoization. Consider using the "fib_fast" algorithm here (http://code.google.com/p/cpptruths/source/browse/trunk/c/fibo.c). I'm sure regular run-time will be several orders of magnitude faster than compile-time.

Sat Dec 24, 10:49:00 PM PST
xander345 said...

if you like c++ you can compile it online here: http://codecompiler.info/

32, 64 - windows & Linux - and more programming languages

Mon Jan 30, 10:57:00 AM PST
eugene yakubovich said...

This article inspired me to think about recursion limits of constexpr functions. I've written two blog posts about the subject here:
http://dev-perspective.blogspot.com/2012/02/recursing-constexpr.html
http://dev-perspective.blogspot.com/2012/03/recursing-constexpr-part-ii.html

Sun Mar 04, 11:51:00 AM PST
titanik 3d said...

thank you

Fri Mar 23, 12:40:00 PM PDT
anonymous said...

I VOTE FOR removing incompetent author from all pages of internet.
john haugeland - IS RIGHT !

Thu Mar 29, 09:17:00 PM PDT
arthur clarke said...

Hi.

Sorry, it might be an off topic, but i notice that you are interested in compilation-time containers.
boost::mpl containers have one critical disadvantage - container must change its name, while adding any element, so, you need to know last name of the container (which makes it almost useless).
Based on googling and some research, i found solution, which doesn't require to change the name of the container. You can find it here:
http://ideone.com/DUCeX
You need gcc-4.7 to compile it.
I've used some macros, but it needed only for readability, you can write the same without any macros.
This code have quite a bad style and almost certainly can be optimized.

Sorry for my English.

Sat Sep 22, 12:02:00 AM PDT
anonymous said...

@Sumant Nice article but why don't you delete those spam comments?
Definitively spam:
- laurelbostan
- Nikola Andelic
- sanjay
Maybe spam:
- Anonymous
- xander345
- Anonymous
Duplicate:
- Anonymous
Also don't hesitate to delete this comment too when done :)

Wed Aug 28, 07:29:00 AM PDT
anonymous said...

I'm surprized to see that the real message behind John Haugeland comments is not discussed here and I had decided to give it a try explaining my understanding of his comments, just to make sure that other perspectives are not lost.
Let me start by saying that I can understand that people are willing to share their discoveries via a blog but also we need to be aware of the perspective John Haugeland brings in(at least for me), about the inexactities generated by our lack of knowledge or limited knowledge of the subject.
This is where I have to agree to John Haugeland: as a user coming to this blog to increase my knowledge about the constexpr topic I might get misleaded by the information inside the post, especially when it does not clearly make the distinctions between personal opinion and facts.
The "acid" comments of John Haugeland are probably his style but the message it brings should not be blurred by that.

Fri Nov 29, 04:36:00 AM PST
sumant tambe said...

@anonymous: If you can extract signal from noise in John's comments, please let me know.

Sat Nov 30, 12:11:00 AM PST
larry evans said...

Apparently there's some differing
opinions about whether constexpr
metaprogramming is faster than
template metaprogramming, at least
according to Zach Laine in this post:

http://lists.boost.org/Archives/boost/2014/05/213389.php

Would you care to comment?

-regards,
Larry

Wed May 21, 01:18:00 PM PDT
sumant tambe said...

@Lrry Evans:The discussions in the mailing list correctly point out that the speedup is gained mostly because of avoiding recursive template instantiasions of integral parameters.

Thu May 22, 10:57:00 AM PDT
larry evans said...

@Sumant Tambe said... on
Thu May 22, 10:57:00 AM PDT

The discussions in the mailing list correctly point out that the speedup is gained mostly because of avoiding recursive template instantiasions of integral parameters.

Thanks for the response Sumant; however,
since the constexpr is_prime_func calls
a recursive function, is_prime_recursive,
and, the number of recursive function
calls is the same as the number of
recursive is_prime_impl class template
instantiations, I still don't understand why there's a difference
in times.

Could you explain a little more?

TIA.

-regards,
Larry

Sat May 31, 11:52:00 AM PDT
jony mehar said...

algolint is best online website dealing with all kind of online complier and online ide.. code snipets and compile and execute program online can be access and an be deal with no issue and erros free

Mon Jun 16, 08:05:00 AM PDT

vabna islam said...

Waw, Great site. I have read your article. really your have shared your natural Idea. I like your article very much. I think it is the best Webdesign site. Everyone can understand everything easily.
App Programmierung

Thu Jul 30, 07:41:00 PM PDT

akmal niazi khan said...

Programming is very interesting and creative thing if you do it with love. Your blog code helps a lot to beginners to learn programming from basic to advance level. I really love this blog because I learn a lot from here and this process is still continuing.
Love from Pro Programmer

Fri May 13, 07:24:00 AM PDT

nikolay mihaylov said...

Today I played with both gcc and clang and seems at -O2, both of them precompute the functions if parameters are known at compile time. you do not really need constexpr, inline, static or anonymous namespaces. However I forgot and I did not tried any static asserts.

I also want to point out that C++14 constexpr is more relaxed, and you no longer need recursion.

Wed Sep 28, 02:04:00 PM PDT

silver jckpot tips said...

I was looking for these kind of posts

Crude Oil HNI Free Tips

Fri Nov 11, 03:14:00 AM PST

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)