



g+1

10

More ▾ Next Blog»

Create Blog Sign In

# INSANE CODING

'COZ GOOD THINKING REQUIRES GOING OUTSIDE THE BOX

TUESDAY, NOVEMBER 22, 2011

## How to read in a file in C++

Posted by insane coder at Tuesday, November 22, 2011

So here's a simple question, what is the correct way to read in a file completely in C++?

Various people have various solutions, those who use the C API, C++ API, or some variation of tricks with iterators and algorithms. Wondering which method is the fastest, I thought I might as well put the various options to the test, and the results were surprising.

First, let me propose an API that we'll be using for the function. We'll send a function a C string (char \*) of a filename, and we'll get back a C++ string (std::string) of the file contents. If the file cannot be opened, we'll throw an error why that is so. Of course you're welcome to change these functions to receive and return whatever format you prefer, but this is the prototype we'll be operating on:

```
std::string get_file_contents(const char *filename);
```

Our first technique to consider is using the C API to read directly into a string. We'll open a file using fopen(), calculate the file size by seeking to the end, and then size a string appropriately. We'll read the contents into the string, and then return it.

```
#include <string>
#include <stdio>
#include <cerrno>

std::string get_file_contents(const char *filename)
{
    std::FILE *fp = std::fopen(filename, "rb");
    if (fp)
    {
        std::string contents;
        std::fseek(fp, 0, SEEK_END);
        contents.resize(std::ftell(fp));
        std::rewind(fp);
        std::fread(&contents[0], 1, contents.size(), fp);
        std::fclose(fp);
        return(contents);
    }
    throw(errno);
}
```

I'm dubbing this technique "method C". This is more or less the technique of any proficient C++ programmer who prefers C style I/O would look like.

### BLOG ARCHIVE

- ▶ 2014 (13)
- ▶ 2013 (2)
- ▶ 2012 (2)
- ▼ 2011 (9)
  - ▶ December (1)
  - ▼ November (3)
    - Reading in an entire file at once in C++, part 2
    - How to read in a file in C++
    - Making modconf work with Linux 3
- ▶ October (2)
- ▶ June (2)
- ▶ April (1)
- ▶ 2010 (9)
- ▶ 2009 (17)
- ▶ 2008 (5)
- ▶ 2007 (17)

### SEARCH THIS BLOG



### SUBSCRIBE TO

The next technique we'll review is basically the same idea, but using C++ streams instead.

```
#include <fstream>
#include <string>
#include <cerrno>

std::string get_file_contents(const char *filename)
{
    std::ifstream in(filename, std::ios::in | std::ios::binary);
    if (in)
    {
        std::string contents;
        in.seekg(0, std::ios::end);
        contents.resize(in.tellg());
        in.seekg(0, std::ios::beg);
        in.read(&contents[0], contents.size());
        in.close();
        return(contents);
    }
    throw(errno);
}
```

PLEASE ADD THIS



I'm dubbing this technique "method C++". Again, more or less a straight forward C++ implementation based on the same principals as before.

The next technique people consider is using `istreambuf_iterator`. This iterator is designed for really fast iteration out of stream buffers (files) in C++.

```
#include <fstream>
#include <streambuf>
#include <string>
#include <cerrno>

std::string get_file_contents(const char *filename)
{
    std::ifstream in(filename, std::ios::in | std::ios::binary);
    if (in)
    {
        return(std::string((std::istreambuf_iterator<char>(in)), std::istreambuf_iterator<char>()));
    }
    throw(errno);
}
```

This method is liked by many because of how little code is needed to implement it, and you can read a file directly into all sorts of containers, not just strings. The method was also popularized by the [Effective STL](#) book. I'm dubbing the technique "method iterator".

Now some have looked at the last technique, and felt it could be optimized further, since if the string has an idea in advance how big it needs to be, it will reallocate less. So the idea is to reserve the size of the string, then pull the data in.

```
#include <fstream>
#include <streambuf>
#include <string>
#include <cerrno>

std::string get_file_contents(const char *filename)
{
    std::ifstream in(filename, std::ios::in | std::ios::binary);
    if (in)
    {
```

```

    std::string contents;
    in.seekg(0, std::ios::end);
    contents.reserve(in.tellg());
    in.seekg(0, std::ios::beg);
    contents.assign((std::istreambuf_iterator<char>(in)), std::istreambuf_ite
rator<char>());
    in.close();
    return(contents);
}
throw(errno);
}

```

I will call this technique "method assign", since it uses the string's assign function.

Some have questioned the previous function, as assign() in some implementations may very well replace the internal buffer, and therefore not benefit from reserving. Better to call push\_back() instead, which will keep the existing buffer if no reallocation is needed.

```

#include <fstream>
#include <streambuf>
#include <string>
#include <algorithm>
#include <iterator>
#include <cerrno>

std::string get_file_contents(const char *filename)
{
    std::ifstream in(filename, std::ios::in | std::ios::binary);
    if (in)
    {
        std::string contents;
        in.seekg(0, std::ios::end);
        contents.reserve(in.tellg());
        in.seekg(0, std::ios::beg);
        std::copy((std::istreambuf_iterator<char>(in)), std::istreambuf_iterator<
char>(), std::back_inserter(contents));
        in.close();
        return(contents);
    }
    throw(errno);
}

```

Combining std::copy() and std::back\_inserter(), we can achieve our goal. I'm labeling this technique "method copy".

Lastly, some want to try another approach entirely. C++ streams have some very fast copying to another stream via operator<< on their internal buffers. Therefore, we can copy directly into a string stream, and then return the string that string stream uses.

```

#include <fstream>
#include <sstream>
#include <string>
#include <cerrno>

std::string get_file_contents(const char *filename)
{
    std::ifstream in(filename, std::ios::in | std::ios::binary);
    if (in)
    {
        std::ostringstream contents;
        contents << in.rdbuf();
        in.close();
        return(contents.str());
    }
}

```

```
}  
throw(errno);  
}
```

We'll call this technique "method rdbuf".

Now which is the fastest method to use if all you actually want to do is read the file into a string and return it? The exact speeds in relation to each other may vary from one implementation to another, but the overall margins between the various techniques should be similar.

I conducted my tests with libstdc++ and GCC 4.6, what you see may vary from this.

I tested with multiple megabyte files, reading in one after another, and repeated the tests a dozen times and averaged the results.

#### Method Duration

C	24.5
C++	24.5
Iterator	64.5
Assign	68
Copy	62.5
Rdbuf	32.5

Ordered by speed:

#### Method Duration

C/C++	24.5
Rdbuf	32.5
Copy	62.5
Iterator	64.5
Assign	68

These results are rather interesting. There was no speed difference at all whether using the C or C++ API for reading a file. This should be obvious to us all, but yet many people strangely think that the C API has less overhead. The straight forward vanilla methods were also faster than anything involving iterators.

C++ stream to stream copying is really fast. It probably only took a bit longer than the vanilla method due to some reallocations needed. If you're doing disk file to disk file though, you probably want to consider this option, and go directly from in stream to out stream.

Using the `istreambuf_iterator` methods while popular and concise are actually rather slow. Sure they're faster than `istream_iterator`s (with skipping turned off), but they can't compete with more direct methods.

A C++ string's internal `assign()` function, at least in libstdc++, seems to throw away the existing buffer (at the time of this writing), so reserving then assigning is rather useless. On the other hand, reading directly into a string,

or a different container for that matter, isn't necessarily your most optimal solution where iterators are concerned. Using the external `std::copy()` function, along with back inserting after reservation is faster than straight up initialization. You might want to consider this method for inserting into some other containers. In fact, I found that `std::copy()` of `istreambuf_iterator` with back inserter into an `std::deque` to be faster than straight up initialization (81 vs 88.5), despite a `Deque` not being able to reserve room in advance (nor does such make sense with a `Deque`).

I also found this to be a cute way to get a file into a container backwards, despite a `Deque` being rather useless for working with file contents.

```
std::deque<char> contents;
std::copy((std::istreambuf_iterator<char>(in)), std::istreambuf_iterator<char>(), std::front_inserter(contents));
```

Now go out there and speed up your applications!

If there's any demand, I'll see about performing these tests with other C++ implementations.

LABELS: API, COMPETITION, FILES, GCC, GOOD CODE, INSANE IDEAS, OPTIMIZATION



+10 Recommend this on Google

## 22 COMMENTS:

 insane coder said...

I tested with clang LLVM 3.0 and got the following:

C/C++: 7.5

rdbuf: 31.5

copy: 97


assign: 102

iter: 110

The iterator tests also varied in their running times between 90 and 120, whereas the others, or with GCC, the variation between tests were a difference of 1-2.

As with GCC, LLVM is smart enough to see that the `reserve()` followed by `assign()` makes the former a null op and optimizes it out.

NOVEMBER 24, 2011 AT 12:02 PM

 insane coder said...

Okay, I tested with Visual C++ 2005.

C: 18.3

C++: 21

rdbuf: 199

iter: 209.3  
assign: 221  
copy: 483.5

Something tells me that either their STL isn't designed too well, or the compiler really isn't smart enough to optimize all that template code properly.

Results here were also very erratic. Times between tests varied anywhere from 2 to 40.

I'll try to test 2010 later.

NOVEMBER 24, 2011 AT 12:50 PM


 Freddie Witherden said...

Interesting; I am surprised by the poor performance of the 'heavier' solutions although the naff VS performance /may/ be due to checked iterators. (It has been a long time since I've used VS but IIRC it used to enable checked iterators by default.)

Might be worth comparing the performance to a mmap + memcpy type solution (I think boost provides a memory mapped file wrapper somewhere). Or to see how EKOPath performs which uses the Apache STL.

Regards, Freddie.

NOVEMBER 24, 2011 AT 12:54 PM

 insane coder said...

Hi Freddie,

If I really wanted the fastest possible way to read in a file, I would be using POSIX 2008.

First I'd open the file, get its size, then tell the OS to optimize for a sequential read. Then I'd create a buffer which is in sync with the blocking factor of the partition.

Doing so, I'd easily cut the best time here in half. But I'm focusing here on standard C++.

NOVEMBER 24, 2011 AT 3:39 PM

 insane coder said...

Visual C++ 2010:

C: 16.5

C++: 20.4

rdbuf: 176.2

assign: 222.8

iter: 224.4

copy: 320

Results with this were also less erratic. Variance between passes were now 2-10.


NOVEMBER 25, 2011 AT 1:10 AM

 Eitan Adler said...

Were these timings done with a warm cache or cold cache? Was cron disabled? Was it done with the network card plugged in? What about the filesystems mounted read only?

There are still too many variables to take these numbers as usable.

NOVEMBER 28, 2011 AT 2:50 PM

 insane coder said...

Hi Eitan,

Tests were done from a RAM drive on an isolated machine with no cron. The tests were also run in random orders. But that's besides the point. The numbers are not to indicate exact speeds of anything. The numbers are to indicate which method averages better performance relative to others.

NOVEMBER 29, 2011 AT 12:24 PM

 Eitan Adler said...

Averages better is meaningless statistically. The question to ask is "Are these numbers significantly different at 95% confidence?". In order to get that answer you need more data than just the average (ie the std deviation)

NOVEMBER 29, 2011 AT 12:31 PM

 insane coder said...


I already mentioned the standard deviation and variance.

NOVEMBER 29, 2011 AT 2:59 PM

 Eitan Adler said...

Understood. I was just commenting on "The numbers are to indicate which method averages better performance relative to others."  
In general your methodology is better than most people's that I've seen :)

NOVEMBER 29, 2011 AT 5:32 PM

 insane coder said...

A follow up has been posted.

NOVEMBER 30, 2011 AT 10:52 AM

 Tygawr said...

Thanks for this wonderful blog! :)  
Can you also provide us an example on how to quickly write a huge file? Oh and how to quickly parse through a huge file.

APRIL 2, 2012 AT 4:46 PM

 insane coder said...

Parsing is outside the scope of the article, and when writing, size matters not.

When you want to write an array or a continuous container, you just write its contents, no trickery will improve that situation (except for informing the OS for a sequential write in POSIX 2008).

If your data is non continuous, then you'll need some kind of loop to write it all.

JULY 19, 2012 AT 1:45 PM

 Odne Hellebø said...


*This comment has been removed by the author.*

APRIL 21, 2013 AT 8:20 PM

 Odne Hellebø said...

Hello, thank you for this article. I was leaning toward the assign method, going with the c++ method instead now. Have you tested your results with gcc 4.8? Would love to see a followup where you do the same tests with todays implementations.

APRIL 21, 2013 AT 8:25 PM

 Unknown said...



I'm interested in your thoughts on

<http://stackoverflow.com/questions/1042940/writing-directly-to-stdstring-internal-buffers/1043318#1043318>.

In summary: This article discusses the incorrect assumption by some programmers that you can write to the pointer returned by `c_str()` AND it also discusses how one should not treat the address of a character reference (`string[]`) as a location that can be written to. It provides the reasons why - those being that the internal format of the string may not be contiguous memory, even though `c_str()` returns a pointer to contiguous memory.

However it doesn't discuss using `&str[0]` as a writable buffer pointer when you've resized an empty string. In other words, it would be difficult to conceive of an implementation that would resize a null string to a set of non-contiguous buffers. I presume this is why you feel it's safe to use `str.resize()` followed by a write to `&str[0]`.

OCTOBER 3, 2013 AT 5:13 PM

 insane coder said...

My thoughts are that more people need to read Effective C++.

`&str[0]` is safe, and mentioned in C++ 2003+.

Writing to a `c_str()` is a very bad idea. It returns a pointer to a const for a reason.

I did not base my decision on feelings of safety, but rather real documentation. I highly advise reading the standard material, and The C++ Programming Language, and Effective C++.

NOVEMBER 18, 2013 AT 4:20 AM



deepesh kumar said...

Interesting. In a similar problem, I am trying to read the content of a text file continuously in VS C++ and the content of that text file is changing by python platform at random time interval. While I am doing so VS C++ is not reading the text file except for the first time. Can any one please tell me how to do it.

JUNE 19, 2014 AT 10:39 PM

 James Hawk III said...

Wouldn't it have been more efficient to pass the target string via the call to avoid making a copy of the string during the return? At the time

you wrote this there weren't any move semantics available in C++, and even now I'm not sure move semantics would kick in.

Or is there something about `std::string` assignment operators that makes copy operations not produce intermediate duplicates?


NOVEMBER 6, 2014 AT 9:15 AM

 insane coder said...

Modern C++ compilers optimize out duplicate creation from return.

[http://en.wikipedia.org/wiki/Return\\_value\\_optimization](http://en.wikipedia.org/wiki/Return_value_optimization)

NOVEMBER 9, 2014 AT 3:03 AM

 mobeen ashraf said...

My problem solver.

i was trying to read a full file neglecting eof. and this blog solved my this issue in no time. Hey insane blogger can you explain me or give reference of exact operations you are performing in in your last method c++.

NOVEMBER 17, 2014 AT 10:13 AM

 insane coder said...

Pretty much everything here is explained.

Is there a point in particular that you feel needs more clarification?

NOVEMBER 20, 2014 AT 8:19 AM

Post a Comment

Newer Post

Home

Older Post

Subscribe to: Post Comments (Atom)