Read whole ASCII file into C++ std::string [duplicate]

Asked 14 years, 8 months ago   Modified 3 years, 9 months ago   Viewed 857k times

735

This question already has answers here:
How do I read an entire file into a std::string in C++? (24 answers)
Closed 8 years ago.

I need to read a whole file into memory and place it in a C++ `std::string`.

If I were to read it into a `char[]`, the answer would be very simple:

```
std::ifstream t;
int length;
t.open("file.txt");      // open input file
t.seekg(0, std::ios::end);    // go to the end
length = t.tellg();           // report location (this is the length)
t.seekg(0, std::ios::beg);    // go back to the beginning
buffer = new char[length];    // allocate memory for a buffer of appropriate
dimension
t.read(buffer, length);       // read the whole file into the buffer
t.close();                    // close file handle

// ... Do stuff with buffer here ...
```

Now, I want to do the exact same thing, but using a `std::string` instead of a `char[]`. I want to avoid loops, i.e. I **don't** want to:

```
std::ifstream t;
t.open("file.txt");
std::string buffer;
std::string line;
while(t){
std::getline(t, line);
// ... Append line to buffer and go on
}
t.close()
```

Any ideas?

c++ string caching file-io standard-library

Share
Improve this question

edited Jun 23, 2014 at 15:25
animuson ♦
54.7k ● 28 ● 141 ● 150

asked Apr 8, 2010 at 17:19
Escualo
42k ● 26 ● 91 ● 137

Follow

3 There will always be a loop involved, but it can be implicit as part of the standard library. Is that acceptable? Why are you trying to avoid loops? – Adrian McCarthy Apr 8, 2010 at 19:22

8 I believe that the poster knew that reading bytes involved looping. He just wanted an easy, perl-style *gulp* equivalent. That involved writing little code. – unixman83 Jan 5, 2012 at 10:57

This code is buggy, in the event that the std::string doesn't use a continuous buffer for its string data (which is allowed): stackoverflow.com/a/1043318/1602642 – Chris Desjardins Jul 30, 2013 at 18:34

3 @ChrisDesjardins: (1) Your link is outdated (C++11 made it contiguous) and (2) even if it wasn't, `std::getline(istream&, std::string&)` would still do the right thing. – MSalters Nov 23, 2015 at 13:50

7 Side note for anyone looking at this code: The code presented as an example for reading into char[] does not null-terminate the array (read does not do this automatically), which may not be what you expect. – Soren Bjornstad Apr 11, 2016 at 1:00

9 Answers

Sorted by: Highest score (default)

1128

There are a couple of possibilities. One I like uses a stringstream as a go-between:

```
std::ifstream t("file.txt");
std::stringstream buffer;
buffer << t.rdbuf();
```

Now the contents of "file.txt" are available in a string as `buffer.str()`.

Another possibility (though I certainly don't like it as well) is much more like your original:

```
std::ifstream t("file.txt");
t.seekg(0, std::ios::end);
size_t size = t.tellg();
std::string buffer(size, ' ');
t.seekg(0);
t.read(&buffer[0], size);
```

Officially, this isn't required to work under the C++98 or 03 standard (string isn't required to store data contiguously) but in fact it works with all known implementations, and C++11 and later do require contiguous storage, so it's guaranteed to work with them.

As to why I don't like the latter as well: first, because it's longer and harder to read. Second, because it requires that you initialize the contents of the string with data you don't care about, then immediately write over that data (yes, the time to initialize is usually trivial compared to the reading, so it probably doesn't matter, but to me it still feels kind of wrong). Third, in a text file, position X in the file doesn't necessarily mean you'll have read X characters to reach that point -- it's not required to take into account things like line-end translations. On real systems that do such translations (e.g., Windows) the translated form is shorter than what's in the file (i.e., "\r\n" in the file becomes "\n" in the translated string) so all you've done is reserved a little extra space you never use. Again, doesn't really cause a major problem but feels a little wrong anyway.

Share
Improve this answer
Follow

edited Sep 18, 2019 at 18:05

answered Apr 8, 2010 at 17:53
Jerry Coffin
489k ● 83 ● 647 ● 1.1k

50 The three-liner works like a charm! – Ryan H. Jul 20, 2011 at 13:33

110 This should've been marked as the answer. – unixman83 Aug 14, 2011 at 23:21

40 Important note for some, at least on my implementation, the three-liner works at least as good as the C fopen alternative for files under 50KB. Past that, it seems to lose performance fast. In which case, just use the second solution. – hiddensunset4 Nov 9, 2011 at 2:43

77 make sure to #include <sstream> – Pramod Jul 6, 2012 at 6:52

19 Most of the time, you're fine *not* testing whether the file has opened (the other operations will simply fail). As a rule, you should *avoid* printing out error messages on the spot, unless you're sure that fits with the rest of the program -- if you must do *something*, throwing an exception is usually preferable. You should almost never explicitly close a file either -- the destructor will do that automatically. – Jerry Coffin Aug 24, 2012 at 20:44

616

Update: Turns out that this method, while following STL idioms well, is actually surprisingly inefficient! Don't do this with large files. (See: http://insanecoding.blogspot.com/2011/11/how-to-read-in-file-in-c.html)

You can make a streambuf iterator out of the file and initialize the string with it:

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

std::ifstream t("file.txt");
std::string str((std::istreambuf_iterator<char>(t)),
                 std::istreambuf_iterator<char>());
```

Not sure where you're getting the `t.open("file.txt", "r")` syntax from. As far as I know that's not a method that `std::ifstream` has. It looks like you've confused it with C's `fopen`.

Edit: Also note the extra parentheses around the first argument to the string constructor. *These are essential*. They prevent the problem known as the "most vexing parse", which in this case won't actually give you a compile error like it usually does, but will give you interesting (read: wrong) results.

Following KeithB's point in the comments, here's a way to do it that allocates all the memory up front (rather than relying on the string class's automatic reallocation):

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

std::ifstream t("file.txt");
std::string str;

t.seekg(0, std::ios::end);
str.reserve(t.tellg());
t.seekg(0, std::ios::beg);

str.assign((std::istreambuf_iterator<char>(t)),
            std::istreambuf_iterator<char>());
```

Share

Improve this answer

Follow

edited Feb 23, 2017 at 19:31
resueman
10.6k ● 10 ● 33 ● 46

answered Apr 8, 2010 at 17:23
Tyler McHenry
76.5k ● 18 ● 125 ● 171

6 open is definitely a method of ifstream, however the 2nd parameter is wrong. cplusplus.com/reference/iostream/ifstream/open – Joe Apr 8, 2010 at 17:30

9 @KeithB If efficiency is important, you could find the file length the same was as in the `char*` example and call `std::string::reserve` to preallocate the necessary space. – Tyler McHenry Apr 8, 2010 at 17:36

57 No sure why people are voting this up, here is a quick question, say I have a 1MB file, how many times will the "end" passed to the std::string constructor or assign method be invoked? People think these kind of solutions are elegant when in fact they are excellent examples of HOW NOT TO DO IT. – Matthieu N. Mar 10, 2011 at 8:49

119 Benchmarked: both Tyler's solutions take about 21 seconds on a 267 MB file. Jerry's first takes 1.2 seconds and his second 0.5 (+/- 0.1), so clearly there's something inefficient about Tyler's code. – dhardy Oct 1, 2012 at 12:32

9 The insanecoding blog post is benchmarking solutions to a slightly different problem: it is reading the file as binary not text, so there's no translation of line endings. As a side effect, reading as binary makes ftell a reliable way to get the file length (assuming a long can represent the file length, which is not guaranteed). For determining the length, ftell is not reliable on a text stream. If you're reading a file from tape (e.g., a backup), the extra seeking may be a waste of time. Many of the blog post implementations don't use RAII and can therefore leak if there's an error. – Adrian McCarthy Oct 14, 2013 at 22:56

99

I think best way is to use string stream. simple and quick !!!

```
#include <fstream>
#include <iostream>
#include <sstream> //std::stringstream
int main() {
    std::ifstream inFile;
    inFile.open("inFileName"); //open the input file

    std::stringstream strStream;
    strStream << inFile.rdbuf(); //read the file
    std::string str = strStream.str(); //str holds the content of the file

    std::cout << str << "\n"; //you can do anything with the string!!!
}
```

Share
Improve this answer
Follow

edited Jun 5, 2019 at 12:57
L. F.
20.5k ● 9 ● 54 ● 83

answered Nov 12, 2013 at 6:08
mili
3,792 ● 1 ● 31 ● 30

8 Simple and quick, right! insanecoding.blogspot.com/2011/11/how-to-read-in-file-in-c.html – Narek Jun 28, 2014 at 6:21

4 Remember to close the stream afterwards... – Yngve Sneen Lindal Jan 4, 2016 at 10:35

35 @YngveSneenLindal Or let the destructor do it automatically - take advantage of C++! – Dan Nissenbaum Feb 12, 2016 at 5:26

1 @YngveSneenLindal are you sure you need to close the stream afterwards? Apparently the fstream's memory allocation should be deallocated once fstream is destroyed (out of the scope)? Although it can be good to use `.close()` for error checking? – darclander Aug 14, 2020 at 15:25

2 Why post this when it was already in the answer by Jerry Coffin from years before? – Arthur Tacca Apr 13, 2021 at 11:15

43

You may not find this in any book or site, but I found out that it works pretty well:

```
#include <fstream>
// ...
std::string file_content;
std::getline(std::ifstream("filename.txt"), file_content, '\0');
```

Share
Improve this answer
Follow

edited Mar 2, 2021 at 20:25
SRG
590 ● 1 ● 7 ● 19

answered Nov 10, 2015 at 16:48
Ankit Acharya
3,111 ● 3 ● 20 ● 29

15 Casting `eof` to `(char)` is a bit dodgy, suggesting some kind of relevance and universality which is illusory. For some possible values of `eof()` and signed `char`, it will give implementation-defined results. Directly using e.g. `char(0)` / `'\0'` would be more robust and honestly indicative of what's happening. – Tony Delroy Dec 28, 2015 at 4:44

2 @TonyD. Good point about converting eof() to a char. I suppose for old-school ascii character sets, passing any negative value (msb set to 1) would work. But passing \0 (or a negative value) won't work for wide or multi-byte input files. – riderBill Feb 22, 2016 at 20:54

7 This will only work, as long as there are no "eof" (e.g. 0x00, 0xff, ...) characters in your file. If there are, you will only read part of the file. – Olaf Dietsche Aug 12, 2017 at 10:25

1 @OlafDietsche There shouldn't be 0x00 in an ASCII file (or I wouldn't call it ASCII file). `0x00` appears to me like a good option to force the `getline()` to read the whole file. And, I must admit that this code is as short as easy to read although the higher voted solutions look much more impressive and sophisticated. – Scheff's Cat Nov 23, 2019 at 9:15

2 @Scheff After revisiting this answer, I don't know, how I reached to that conclusion and comment. Maybe I thought, that `(char) ifs.eof()` has some meaning. `eof()` returns `false` at this point, and the call is equivalent to `std::getline(ifs, s, 0);`. So it reads until the first 0 byte, or the end of file, if there's no 0 byte. – Olaf Dietsche Nov 23, 2019 at 21:51

7

Try one of these two methods:

```
string get_file_string(){
    std::ifstream ifs("path_to_file");
    return string((std::istreambuf_iterator<char>(ifs)),
                  (std::istreambuf_iterator<char>()));
}

string get_file_string2(){
    ifstream inFile;
    inFile.open("path_to_file");//open the input file

    stringstream strStream;
    strStream << inFile.rdbuf();//read the file
    return strStream.str();//str holds the content of the file
}
```

Share Improve this answer Follow

answered Feb 5, 2015 at 12:50

madx
7,153 ● 4 ● 58 ● 63

3

I figured out another way that works with most istreams, including std::cin!

```
std::string readFile()
{
    stringstream str;
    ifstream stream("Hello_World.txt");
    if(stream.is_open())
    {
        while(stream.peek() != EOF)
        {
            str << (char) stream.get();
        }
        stream.close();
        return str.str();
    }
}
```

Share
Improve this answer
Follow

edited Jun 5, 2019 at 12:58
L. F.
20.5k ● 9 ● 54 ● 83

answered Aug 20, 2014 at 5:23
yash101
671 ● 1 ● 8 ● 20

1

If you happen to use glibmm you can try Glib::file_get_contents.

```
#include <iostream>
#include <glibmm.h>

int main() {
    auto filename = "my-file.txt";
    try {
        std::string contents = Glib::file_get_contents(filename);
        std::cout << "File data:\n" << contents << std::endl;
    catch (const Glib::FileError& e) {
        std::cout << "Oops, an error occurred:\n" << e.what() << std::endl;
    }

    return 0;
}
```

Share Improve this answer Follow

answered Aug 7, 2016 at 14:55
Artem Vorotnikov
120 ● 1 ● 4

2 Imho: Although this works, providing a "glib" solution, which is the non-platform-independent equivalent of pandora's chest, might confuse enormously, even more, if there's a simple CPP-standard solution to it. – MABVT Feb 11, 2019 at 7:07

0

I could do it like this:

```
void readfile(const std::string &filepath,std::string &buffer){
    std::ifstream fin(filepath.c_str());
    getline(fin, buffer, char(-1));
    fin.close();
}
```

If this is something to be frowned upon, please let me know why

Share Improve this answer Follow

answered Oct 22, 2012 at 12:55
chunkyguy
3,679 ● 1 ● 30 ● 34

13 char(-1) is probably not a portable way to denote EOF. Also, getline() implementations are not required to support the "invalid" EOF pseudo-character as a delimiter character, I think. – reddish Jan 23, 2013 at 10:54

1 @reddish it indeed isn't, in modern C++ it is better to use `std::char_traits<char>::eof()`. If someone is still using an ancient compiler... `<cstdio>` contains `EOF` macro. – user11877195 Mar 7, 2022 at 9:27

-6

I don't think you can do this without an explicit or implicit loop, without reading into a char array (or some other container) first and ten constructing the string. If you don't need the other capabilities of a string, it could be done with `vector<char>` the same way you are currently using a `char *`.

Share Improve this answer Follow

answered Apr 8, 2010 at 17:30
KeithB
17k ● 3 ● 43 ● 45

20 -1 Not true... See above – unixman83 Aug 14, 2011 at 23:22

3 Well, to be fair, all of the answers above do include a loop in some way, be it as a boilerplate or *behind the scenes*... – ljleb Oct 17, 2020 at 10:15