C++ stringstream, string, and char* conversion confusion

My question can be boiled down to, where does the string returned from `stringstream.str().c_str()` live in memory, and why can't it be assigned to a `const char*`?

This code example will explain it better than I can

```
#include <string>
#include <sstream>
#include <iostream>

using namespace std;

int main()
{
    stringstream ss("this is a string\n");

    string str(ss.str());

    const char* cstr1 = str.c_str();

    const char* cstr2 = ss.str().c_str();

    cout << cstr1       // Prints correctly
        << cstr2;       // ERROR, prints out garbage

    system("PAUSE");

    return 0;
}
```

The assumption that `stringstream.str().c_str()` could be assigned to a `const char*` led to a bug that took me a while to track down.

For bonus points, can anyone explain why replacing the `cout` statement with

```
cout << cstr            // Prints correctly
    << ss.str().c_str() // Prints correctly
    << cstr2;           // Prints correctly (???)
```

prints the strings correctly?

I'm compiling in Visual Studio 2008.

c++ string memory stringstream

asked Sep 3 '09 at 16:22

Graphics Noob

4,485 ● 6 ● 29 ● 38

5 Answers

`stringstream.str()` returns a temporary string object that's destroyed at the end of the full expression. If you get a pointer to a C string from that (`stringstream.str().c_str()`), it will point to a string which is deleted where the statement ends. That's why your code prints garbage.

So you either have to copy that string to some other string and take the C string from that one:

```
const std::string tmp = stringstream.str();
const char* cstr = tmp.c_str();
```

Note that I made the temporary string `const`, because any changes to it might cause it to re-

allocate and thus render `cstr` invalid. It is therefor safer to not to store the result of at all and use `cstr` only until the end of the full expression:

```
use_c_str( stringstream.str().c_str() );
```

Of course, the latter might not be easy and copying might be to expensive. What you can do instead is to bind the temporary to a `const` reference. This will extend its lifetime to the lifetime of the reference:

```
{
  const std::string& tmp = stringstream.str();
  const char* cstr = tmp.c_str();
}
```

IMO that's the best solution. Unfortunately it's not very well known.

edited Oct 8 '10 at 6:40

answered Sep 3 '09 at 16:25

sbi
110k ● 27 ● 155 ● 306

10 It should be noted that doing a copy (as in your first example) will not necessarily introduce any overhead - if `str()` is implemented in such a way that RVO can kick in (which is very likely), the compiler is permitted to construct the result directly into `tmp` , eliding the temporary; and any modern C++ compiler will do so when optimizations are enabled. Of course, the bind-to-const-reference solution guarantees no-copy, so may well be preferable - but I thought it's still worth clarifying. – Pavel Minaev Sep 3 '09 at 16:34

1 "Of course, the bind-to-const-reference solution guarantees no-copy" <- it doesn't. In C++03, the copy constructor needs to be accessible and the implementation is allowed to copy the initializer and bind the reference to the copy. – Johannes Schaub - litb Sep 3 '09 at 16:38

Your first example is wrong. The value returned by c_str() is transient. It can not be relied on after the end of the current statement. Thus you can use it to pass a value to a function but you should NEVER assign the result of c_str() to a local variable. – Loki Astari Sep 3 '09 at 20:03

2 @litb: You are technically correct. The pointer is valid until the next non cost method call on the string. The problem is that the usage is inherently dangerous. Maybe not to the original developer (though in this case it was) but especially to subsequent maintenance fixes, this kind of code becomes extremely fragile. If you want to do this you should wrap the pointers scope so that it's usage is a short as possible (the best being the length of the expression). – Loki Astari Sep 4 '09 at 17:34

1 @sbi: Ok, thanks, that is more clear. Strictly speaking, though, since the 'string str' var is not modified in the code above, the str.c_str() remains perfectly valid, but I appreciate the potential danger in other cases. – William Knight Mar 12 '10 at 22:51

What you're doing is creating a temporary. That temporary exists in a scope determined by the compiler, such that it's long enough to satisfy the requirements of where it's going.

As soon as the statement `const char* cstr2 = ss.str().c_str();` is complete, the compiler sees no reason to keep the temporary string around, and it's destroyed, and thus your `const char *` is pointing to free'd memory.

Your statement `string str(ss.str());` means that the temporary is used in the constructor for the `string` variable `str` that you've put on the local stack, and that stays around as long as you'd expect: until the end of the block, or function you've written. Therefore the `const char *` within is still good memory when you try the `cout` .

answered Sep 3 '09 at 16:25

Jared Oberhaus
9,452 ● 3 ● 27 ● 48

The `ss.str()` temporary is destroyed after initialization of `cstr2` is complete. So when you print it with `cout` , the c-string that was associated with that `std::string` temporary has long been destoryed, and thus you will be lucky if it crashes and asserts, and not lucky if it prints garbage or does appear to work.

```
const char* cstr2 = ss.str().c_str();
```

The C-string where `cstr1` points to, however, is associated with a string that still exists at the

time you do the `cout` - so it correctly prints the result.

In the following code, the first `cstr` is correct (i assume it is `cstr1` in the real code?). The second prints the c-string associated with the temporary string object `ss.str()`. The object is destroyed at the end of evaluating the full-expression in which it appears. The full-expression is the entire `cout << ...` expression - so while the c-string is output, the associated string object still exists. For `cstr2` - it is pure badness that it succeeds. It most possibly internally chooses the same storage location for the new temporary which it already chose for the temporary used to initialize `cstr2`. It could aswell crash.

```
cout << cstr           // Prints correctly
    << ss.str().c_str() // Prints correctly
    << cstr2;          // Prints correctly (???)
```

The return of `c_str()` will usually just point to the internal string buffer - but that's not a requirement. The string could make up a buffer if its internal implementation is not contiguous for example (that's well possible - but in the next C++ Standard, strings need to be contiguously stored).

In GCC, strings use reference counting and copy-on-write. Thus, you will find that the following holds true (it does, at least on my GCC version)

```
string a = "hello";
string b(a);
assert(a.c_str() == b.c_str());
```

The two strings share the same buffer here. At the time you change one of them, the buffer will be copied and each will hold its separate copy. Other string implementations do things different, though.

answered Sep 3 '09 at 16:33

Johannes Schaub - litb

284k ● 56 ● 586 ● 954

In this line:

```
const char* cstr2 = ss.str().c_str();
```

`ss.str()` will make a *copy* of the contents of the stringstream. When you call `c_str()` on the same line, you'll be referencing legitimate data, but after that line the string will be destroyed, leaving your `char*` to point to unowned memory.

answered Sep 3 '09 at 16:27

fbrereto

23.3k ● 7 ● 79 ● 146

The std::string object returned by ss.str() is a temporary object that will have a life time limited to the expression. So you cannot assign a pointer to a temporary object without getting trash.

Now, there is one exception: if you use a const reference to get the temporary object, it is legal to use it for a wider life time. For example you should do:

```
#include <string>
#include <sstream>
#include <iostream>

using namespace std;

int main()
{
    stringstream ss("this is a string\n");

    string str(ss.str());

    const char* cstr1 = str.c_str();

    const std::string& resultstr = ss.str();
    const char* cstr2 = resultstr.c_str();

    cout << cstr1       // Prints correctly
        << cstr2;       // No more error : cstr2 points to resultstr memory that is still
alive as we used the const reference to keep it for a time.

    system("PAUSE");
```

```
    return 0;
}
```

That way you get the string for a longer time.

Now, you have to know that there is a kind of optimisation called RVO that say that if the compiler see an initialization via a function call and that function return a temporary, it will not do the copy but just make the assigned value be the temporary. That way you don't need to actually use a reference, it's only if you want to be sure that it will not copy that it's necessary. So doing:

```
std::string resultstr = ss.str();
const char* cstr2 = resultstr.c_str();
```

would be better and simpler.

answered Sep 3 '09 at 16:30

Klaim

26.9k ● 14 ● 74 ● 146