

## Re: Copy to CString's internal buffer

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2007-06/msg01280.html>

---

- *From:* Joseph M. Newcomer <[newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)>
  - *Date:* Wed, 20 Jun 2007 23:18:05 -0400
- 

See below...

On Wed, 20 Jun 2007 22:43:40 -0400, "vvf" <[novvfspam@xxxxxxxxxxxxx](mailto:novvfspam@xxxxxxxxxxxxx)> wrote:

Hi Joe!

Please see embedded comments.

"Joseph M. Newcomer" <[newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)> wrote in message [news:lj3h731vf0ar8rdf56b2i9a737njt3vejn@xxxxxxxxxxxxx](mailto:news:lj3h731vf0ar8rdf56b2i9a737njt3vejn@xxxxxxxxxxxxx)

On Tue, 19 Jun 2007 21:46:14 -0400, "vvf" <[novvfspam@xxxxxxxxxxxxx](mailto:novvfspam@xxxxxxxxxxxxx)> wrote:

Hi All,

I have a function like this:

```
void func(char* buffer);
```

I know that this buffer is 20 characters long; Does anybody know why I

can't

do the following in the body of the function:

```
CString str;  
LPTSTR lptStr = str.GetBuffer(20);  
memcpy(lptStr, buffer, 20);  
lptStr.ReleaseBuffer();
```

\*\*\*\*

Seems an odd thing to do.

But note that char is obsolete and should not be used except in rare and

exotic

## Re: Copy to CString's internal buffer

circumstances, so you have to justify that this is one.

My only justification is that I know for a fact that this function will receive a non-unicode set of characters.

\*\*\*\*\*

Whenever I do this, first, I would not write 'char \*' but would write 'LPSTR', and I would always explicitly add the comment

```
// 8-bit characters on input!!!
```

so I would know it was not an accident that I was using 8-bit characters. Then under NO conditions would I use memcpy, because a CString is not necessarily 8-bit characters!

\*\*\*\*\*

Also, it seems a totally weird way to copy a string; you would be better writing

```
CString str;  
str = buffer;
```

since there is no justification for unwrapping the string and doing an

erroneous memcpy

(did you notice that you copied 20 of the 21 characters of the input

string? What

happened to that terminal NUL?

\*\*\*\*\*

As you observed, I didn't explain the problem properly. The char\* buffer passed to this function is not NUL terminated and I have to work with a subset of characters from it anyway. In other words, I can't just do CString str; str = buffer. For example, the buffer passed to the function may contain 512 characters and my function will have to "extract" fixed sized "fields"; In other words, I would have to extract characters 0 through 120 then characters 120 through 320 and so on. Indeed, I could 'extend' this buffer by one position and add the NUL terminator at the end so that I could do CString str = buffer. After that, I could do (pseudo-code):

\*\*\*\*\*

Then this is the wrong way to do it. Consider the CStringT::CopyChars operation instead! No need to break encapsulation and memcpy by hand! So you went to a lot of work to reimplement something that is already a part of the library!

Note that CStringT is derived from CStringT, and CString is actually CStringT. There

Re: Copy to CString's internal buffer

## Re: Copy to CString's internal buffer

are also CStringA and CStringW, and you might consider passing in a const CStringA& instead of a char \*.

\*\*\*\*\*

```
CString strBuffer = buffer;
CString strSubString;
for (int i = 0; i < 120; ++i)
    strSubString.SetAt(i, strBuffer.GetAt(i));
```

or maybe use the CString += operator.

Duh! OF COURSE this works (in non-Unicode apps) because you forgot to copy the NUL in the

above example. It would also have worked if you had done  
memcpy(buff, buffer, sizeof(buff)+1);  
but that is still a truly LOUSY way to store a string! Why are you

unwrapping the string

at all? Why not use the simple assignment operator, which always does the right thing?

It is my fault that I did not explain correctly what I wanted to do. I hope that now it is clear why I couldn't just assign the passed buffer to a CString.

\*\*\*\*\*

Yes, it is now clear, but it is also clear that you didn't need to use memcpy to do it, because once you explained what you were attempting to accomplish, it was also clear that the solution already existed (and furthermore, exists in a type-safe fashion!)

\*\*\*\*\*

You have certainly not justified why you enjoy writing erroneous code to solve simple

problems that already have clean and well-defined solutions.

Sorry! :)

\*\*\*\*\*

Re: Copy to CString's internal buffer

str = buff;

The reason why I am trying to use mempcy directly to the  
CString's buffer

is

to avoid the use of a for loop with a "SetAt" operation  
because in the

real

application, the 'buffer' is much longer and I think that  
mempcy does a  
better job in optimization as opposed to a for/SetAt.

\*\*\*\*

So why did you ask a question unrelated to your problem?

I wrote the question in a hurry and didn't take the time to explain it  
thoroughly. Sorry! :)

\*\*\*\*\*

Had you asked the correct question, it would have been easy to give the correct answer.

\*\*\*\*\*

What you MEANT to ask was "how  
can I efficiently concatenate characters to a CString?" and you failed to

give anything

other than some vague handwave about the actual lengths. "much longer"

than 20 could mean

100, 10000, or 1000000. What order of problem are you working on?

Again, I'm sorry for not explaining the problem correctly. The buffer passed  
to this function will not be bigger than 512 characters. This buffer  
contains a bunch of 'fields' of various fixed lengths. For example, "Field  
1" may be from 0 to 120, "Field 2" may be from 120 to 320 and so on. I don't  
have the specifications in front of me right now to give you all the  
details. So, as I mentioned before, I am attempting to 'extract' these  
fields efficiently. That is why I thought about "mempcy". My reasoning was

## Re: Copy to CString's internal buffer

that this function has been around for a long time and has been used a lot so I was hoping that its implementation would be a very efficient copy (like the equivalent of a x86 stosd :) ) I thought that instead of me looping, calling GetAt and SetAt, I would call memcpy which would just do it more efficiently. Also, I didn't want to do this:

```
void func(char* buffer)
{
    char localBuff[SIZE_FIELD1+1];
    memcpy(localBuff, buff, SIZE_FIELD1);
    localBuff[SIZE_FIELD1] = '\0';
    CString str = localBuff;
}
```

because I felt it was not necessary to copy data to a local buffer and THEN assign it to a CString; I wanted to copy it directly to the CString's internal buffer.

\*\*\*\*\*

I agree there is no need to make a local copy, but you could just as easily have written

```
CString str(&buffer[computed_offset], field_length);
```

and it would have worked as well, but been typesafe and not required breaking into the implementation!

Again, asking the right question tends to get a useful answer; creating bogus examples and pretending they are the question will get shrieks of agony as to why you would ever consider doing something so bizarre.

So by asking the wrong question, we were not able to give you the very simple and straightforward solutions that are already defined.

\*\*\*\*\*

Instead, you asked a completely different question, gave a bad example, an incorrect

example, and an example using obsolete data types.

Yes, I agree. Sorry for misleading you and the other readers. Thanks for answering though.

If you have problems with SetAt, you might consider storing the array one character at a

Re: Copy to CString's internal buffer

time, but NOT by using memcpy. But since you have not actually explained your problem, it

is hard to give a correct answer.

OK, thanks.

Note that if you are simply storing a string, then memcpy is merely erroneous. If you are

storing one character at a time (which is NOT the example you gave) then you need to

create a well-formed string. A well-formed string has a NUL at the end.

Alternatively,

you can do a ReleaseBuffer with an explicit length.

Right, I should have read the documentation. That's one of the mistakes I've done initially. I thought that "ReleaseBuffer" would automatically add the NUL terminator but now I know how it works after reading the documentation.

NEVER use 'char' unless you are working in really weird situations.

Except for the rare

situations such as dealing with 8-bit files and embedded communications, you should never

see this data type in a program.

Yes, I agree. This project, though, uses embedded communications.

How about showing the ACTUAL problem you are trying to solve instead of an

artificial

example that is poorly written?

Re: Copy to CString's internal buffer

I will try to summarize it: My function receives an ANSI set of characters in the form of a buffer (char\*) and I know its length, 512. My function has to extract "sub-sets" of characters from this buffer and store them in their own 'data containers' (such as a CString.) What I am trying to do is to extract these sub-sets efficiently. Lastly, I would like to mention that I know each of the sub-set's lengths (no. of characters.)

\*\*\*\*\*

See the above. Trivial. No need to use memcpy to do it efficiently. How many millions of these 512-byte records do you need to process (if the answer is not in the millions, then issues of "efficiency" aren't even worth talking about; the above built-in methods will be as efficient as you could possibly need)

joe

\*\*\*\*\*

You might also consider using the PreAllocate method for performance, which doesn't require unwrapping the representation and opening the possibility of writing erroneous code.

\*\*\*\*\*

Thank you for taking the time to answer my question.

Joseph M. Newcomer [MVP]  
email: [newcomer@xxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxx)  
Web: <http://www.flounder.com>  
MVP Tips: [http://www.flounder.com/mvp\\_tips.htm](http://www.flounder.com/mvp_tips.htm)

.