**CS302 Lecture notes**
**Sprintf, Sscanf, C strings**


- James S. Plank

- Directory: **/home/plank/cs302/Notes/Sprintf**
- August, 2015
- Latest revision: Wed Aug 23 15:23:35 EDT 2023
- Git Hash: 5940e159838836d9c2e971e46b3e3b36a28d04c5

---

## Topcoder Practice Problems

- EllysTimeMachine, the 250-point problem from TCO 2016, Round 1A. You can do this one extremely quickly if you use sscanf() and sprintf().

---

## Into the sewers

This lecture drags you through some pretty detailed information about C++ and C-style strings. I urge you to walk through these lecture notes pretty slowly, and run the code yourself on your own machine. I know you're going to glaze over pretty quickly, thinking something like the following:

*"I'm never going to use this stuff. I can use **cin** and **cout** and stringstreams, so why would I want to bother with pointers and arrays, ampersands and null characters? Plank's just old and doesn't know how to do modern programming."*

Now that we've acknowledged what you're thinking, I want you to resist the temptation to skip this stuff and instead work through it. Yes, I am old. Trust me -- **sprintf()** and **sscanf()** are really nice alernatives to stringstreams. Plus, understanding computer memory is one of the most important keys to being a good programmer. So buckle up, and spend an hour or two with these lecture notes.

---

## Introduction - C Style Strings

**Sprintf()** and **sscanf()** are string conversion procedures from the C Stdio Library. The functionality that you get with **sprintf()** and **sscanf()** is handled in C++ with stringstreams. However, I find **sprintf()** and **sscanf()** to be easier to use, which is why I want you to learn them.

Both of them work with "C style" strings. These are arrays of bytes (of type **char**). The convention with C-style strings is that the array contains printable characters, terminated with the null character (which you specify as '\0' -- Its actual value is zero). Note, I say that this is a "convention." That's because it is up to anyone using and manipulating C-style strings to make sure that the array of bytes is in the proper format – it is not automatically handled for you like strings are in C++.

*Slight digression: I assume that you've had this information before, but a little review never hurts -- printable characters in C and C++ are simply bytes, which are integers between -128 and 127. The data type is **char**. You can print them as integers by using **printf("%d", ...)**. When you print them as characters, you use **printf("%c", ...)**. There is a mapping of integers to characters called "ASCII." You don't need to care what this mapping is, except you should know that:*

- *Characters '0' through '9' are contiguous integers, and are smaller than 'A'-'Z' and 'a'-'z'.*
- *Characters 'A' through 'Z' are contiguous integers, and are smaller than 'a'-'z'.*
- *Characters 'a' through 'z' are contiguous integers.*

*When you print a string in C or C++, you are printing an array of bytes using the ASCII mapping. With **printf("%s", s)** in particular, you are passing a pointer to bytes, and **printf()** prints the character associated with each byte in succession, until it reaches the null character, specified as '\0', whose integer value is zero.*

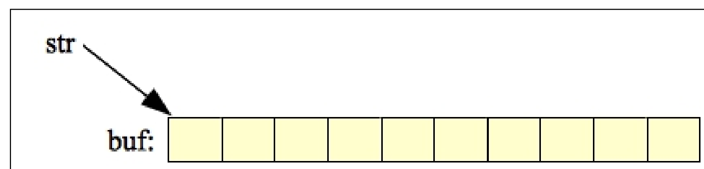


Copyright © 2023, James S. Plank. All rights reserved.

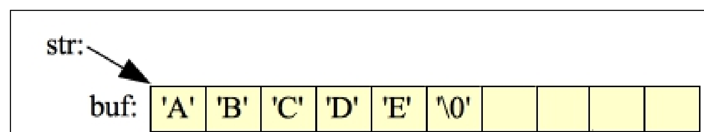
When you access a C-style string, you use a pointer to its first character. This pointer will be of type "**char ***." Alternatively, you can use an array of characters (an array, not a C++ vector) that you allocate yourself, such as, for example "**char buf[10]**," which is an array of 10 characters. Let's look at an example, in src/c_str.cpp:

```
/* Line  1 */    #include <string>
/* Line  2 */    #include <cstdio>
/* Line  3 */    #include <cstdlib>
/* Line  4 */    #include <iostream>
/* Line  5 */    using namespace std;
/* Line  6 */
/* Line  7 */    int main()
/* Line  8 */    {
/* Line  9 */      char buf[10];
/* Line 10 */      char *str;
/* Line 11 */      int i;
/* Line 12 */      string cpps;
/* Line 13 */
/* Line 14 */      str = buf;
/* Line 15 */
/* Line 16 */      for (i = 0; i < 6; i++) buf[i] = 'A'+i;
/* Line 17 */      buf[i] = '\0';
/* Line 18 */
/* Line 19 */      printf("When I print buf with percent s, I get: %s\n", buf);
/* Line 20 */      printf("When I print str with percent s, I get: %s\n", str);
/* Line 21 */
/* Line 22 */      cpps = buf;
/* Line 23 */      str[0] = 'X';
/* Line 24 */      cpps[1] = 'Y';
/* Line 25 */
/* Line 26 */      cout << "This is cpps: " << cpps << endl;
/* Line 27 */      cout << "This is str:  " << str << endl;
/* Line 28 */      cout << "This is buf:  " << buf << endl;
/* Line 29 */      return 0;
/* Line 30 */    }
```

On lines 9 and 10 of this program, I declare an array of ten characters called **buf**, and a character pointer called **str**. The very first action that I perform (on line 14) is have **str** point to the first byte of **buf**. This looks as follows:




In other words, I have allocated ten bytes, and I may access them in two ways -- via the variable **buf** and via the variable **str**. Next, (lines 16 - 20), I set the first five characters to 'A', 'B', 'C', 'D', 'E' and 'F'. I set the next character to the null character, and I print out both **buf** and **str**. At this point, they are:



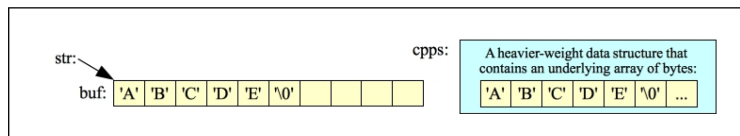


Copyright © 2023, James S. Plank. All rights reserved.

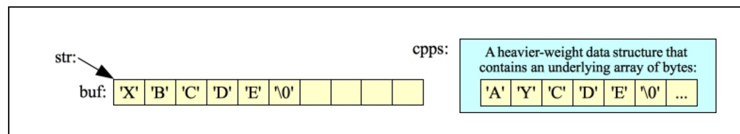
I can see this when I run the program by taking a look at the first two lines of output:

```
UNIX> make clean
rm -f a.out bin/*
UNIX> make bin/c_str
g++ -std=c++11 -Wall -Wextra -o bin/c_str src/c_str.cpp
UNIX> bin/c_str | head -n 2
When I print buf with percent s, I get: ABCDEF
When I print str with percent s, I get: ABCDEF
UNIX>
```

Next, (line 22), I assign the c++ string **cpps** to equal **buf**. This creates a new C++ string which is a heavier-weight data structure, because it contains more information than simply an array of bytes. It copies the bytes of **buf** into its own data structure. Here is a picture after line 22:




Finally, (lines 23) I change the first byte of **str** to 'X'. Because **str** is pointing to the same bytes as **buf**, that changes the first character of **buf**. I also change the second byte of **cpps** to 'Y'. Although this looks like the previous statement, it is a bit different, because the C++ string class overloads the bracket operators so that it finds the underlying bytes of the string, and changes the second one. Here's what they look like afterward:




This explains the last three lines of output, where I print out **cpps**, **str** and **buf**.

```
UNIX> bin/c_str | tail -n 3
This is cpps: AYCDEF
This is str:  XBCDEF
This is buf:  XBCDEF
UNIX>
```

Make sure you understand *every* line of this program, and in particular, why it is that **str** and **buf** utilize the same string buffer, and **cpps** utilizes a different string.

---

## C++ strings

Strings in C++ are very nicely handled – you may manipulate them, allocate them, read them and write them quite seamlessly. You may view a C++ string as a class that has, at its core, a C style string. It's something like the following (this is not exactly right, but for the purposes of this explanation, it's good enough):


Copyright © 2023, James S. Plank. All rights reserved.

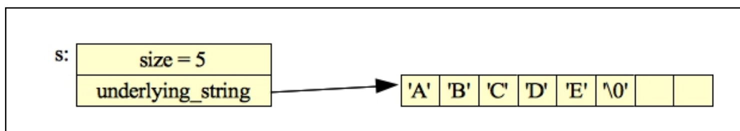
```
class string {
  public:
    unsigned long long size();
    char *c_str();
    ...
  private:
    char *underyling_string;
    unsigned long Size;
    ...
};
```

When you create a string, for example:

```
  int i;
  string s;

  for (i = 'A'; i < 'F'; i++) s.push_back(i);
```

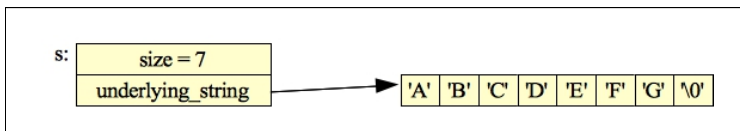
Then the string structure does some work to allocate memory for its string buffer, and when the loop is done, you'll have **Size** equaling 5, and **underlying_string** will point to a buffer of *at least* six bytes, the first 5 of which are 'A', 'B', 'C', 'D' and 'E', and the last of which is '\0'. We can view it as follows:




You'll note, in the drawing above, the amount of memory that **underlying_string** is pointing to is greater than the six characters required to store 'A' through 'E' and the null character. That can happen, and it will be different from machine to machine. However, let's go with this example. Let's suppose you do two more "push_back" commands:

```
    s.push_back('F');
    s.push_back('G');
```

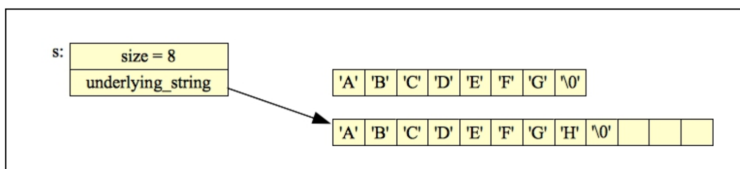
Our string now looks as follows:




Now, suppose you do one more "push_back":

```
    s.push_back('H');
```

The pointer to the buffer that holds the string has run out of memory. So, what the string class does is allocate a new buffer, and copy the string there. The state of our string will look something like this:




What I'm trying to convey here is that the old buffer is discarded and a new one is used. The old buffer will be released to the memory management system to be reused, and the string uses the new buffer until it fills up.


Copyright © 2023, James S. Plank. All rights reserved.

## The c_str() method of a C++ string

The **c_str()** method of a C++ string returns a **const char ***. This is a pointer to the first byte of the C-style string that is held in the class of the C++ string. The keyword "const" means that you should not try to modify this string, and if you do, the compiler will exit with an error. We can get around this, by the way, which we'll do later. As I intimate above, a C++ string will keep filling in its underlying C-style string until it runs out of room, at which point it allocates a new string. We can prove that with the following program, in src/buffer_changes.cpp:

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

/* This program demonstrates that as you call push_back(), the string class'
    underlying string buffer will change.  This is because the buffer "fills up",
    and then the string implementation allocates a bigger buffer and copies
    the string over to it.  If you try to maintain a pointer to this old
    buffer, the pointer will become "stale" when the buffer changes. */

int main()
{
    string s;
    const char *cs;
    int i;

    cs = s.c_str();                 // Store the pointer to the buffer in cs

    for (i = 1; i <= 10000; i++) {
    s.push_back('A');
    if (s.c_str() != cs) {       // Print when the pointer changes.
        printf("The underlying buffer changed at size: %d\n", i);
        cs = s.c_str();
    }
    }
    return 0;
}
```

This keeps adding characters to a C++ string, s, and it notes when storage for the underlying C-style string changes. Check it out as it runs (on my linux box):


Copyright © 2023, James S. Plank. All rights reserved.

```
UNIX> make bin/buffer_changes
g++ -std=c++11 -Wall -Wextra -o bin/buffer_changes src/buffer_changes.cpp
UNIX> bin/buffer_changes
The underlying buffer changed at size: 1
The underlying buffer changed at size: 2
The underlying buffer changed at size: 3
The underlying buffer changed at size: 5
The underlying buffer changed at size: 9
The underlying buffer changed at size: 17
The underlying buffer changed at size: 33
The underlying buffer changed at size: 65
The underlying buffer changed at size: 129
The underlying buffer changed at size: 257
The underlying buffer changed at size: 513
The underlying buffer changed at size: 1025
The underlying buffer changed at size: 2049
The underlying buffer changed at size: 8136
UNIX>
```

I think we can all figure out that the underlying buffers are allocated to be powers of two, although that last line is pretty odd. These things change from machine to machine – on my macbook in 2018, I got the following output from this program:

```
The underlying buffer changed at size: 23
The underlying buffer changed at size: 48
The underlying buffer changed at size: 96
The underlying buffer changed at size: 192
The underlying buffer changed at size: 384
The underlying buffer changed at size: 768
The underlying buffer changed at size: 1536
The underlying buffer changed at size: 3072
The underlying buffer changed at size: 6144
```

One thing that you should get out of this program is that you should not store **c_str()** pointers if you change the C++ string, because the underlying buffer can change.

---

## Don't mess with the bytes that c_str() returns, #1.

The **const** keyword typically keeps you out of danger, but you can get around it. The following program should show you why you shouldn't do that. Here, I "typecast" the return value of **c_str()** to a **char *** that doesn't have the "const" keyword. Then I modify the C-style string so that it puts the null character after the first character. Then I print out the C++ string's size, plus I print out the string using both **printf()** and **cout**. The program is in src/bad_c_str.cpp:


Copyright © 2023, James S. Plank. All rights reserved.

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

/* This program shows what happens when you mess with
   the pointer returnd by the c_str() method of strings.
   You end up corrupting the string structure. */

int main()
{
  string s;
  char *cs;

  /* Here you set cs to point to the bytes of a string,
     and you set the character at index one to the
     NULL character. */

  s = "ABCDE";
  cs = (char *) s.c_str();
  cs[1] = '\0';

  /* You'll note that it still reports that the size
     is 5, even though when you print it, its size is one. */

  cout << "After setting index 1 to the NULL character.\n";
  cout << s.size() << endl;
  cout << s << endl;
  printf("%s\n", s.c_str());

  /* When you call push_back on s, it indeed pushes the character
     'F' on the end of the string -- you'll see that the string
     is still corrupted. */

  s.push_back('F');

  cout << endl << "After calling s.push_back('F'):\n";
  cout << s.size() << endl;
  cout << s << endl;
  printf("%s\n", s.c_str());

  return 0;
}
```

When I run it (again on my mac), you see some pretty odd behavior:



Copyright © 2023, James S. Plank. All rights reserved.

```
UNIX> make bin/bad_c_str
g++ -std=c++11 -Wall -Wextra -o bin/bad_c_str src/bad_c_str.cpp
UNIX> bin/bad_c_str
After setting index 1 to the NULL character.
5
ACDE
A

After calling s.push_back('F'):
6
ACDEF
A
UNIX>
```

You'll note that putting the null character into s turns the C style string into a one-character string, but the C++ string retains its size, and when you print it out with **cout**, it basically skips over the null character and keeps printing. When you print out the **c_str()** with **printf()**, it stops at the null character.

In other words, this is a program itching with bugs. Don't do what I've done here; however, it's good to see what's happening.

---

## sprintf()

**Sprintf()** does what **printf()** does, only it takes as its first argument a pointer to a buffer of bytes, and instead of printing to the screen, it puts its output into that buffer. Here's a very simple example of putting 5 numbers into a string (src/sprintf1.cpp):

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

/* This program uses sprintf() to put five numbers into a string. */

int main()
{
    char buf[100];
    string s;
    int i;

    cin >> i;

    sprintf(buf, "%d %d %d %d %d", i, i+1, i+2, i+3, i+4);
    s = buf;

    cout << s << endl;

    return 0;
}
```


Copyright © 2023, James S. Plank. All rights reserved.

When we run it, we see that the string s is set to "1 2 3 4 5":

```
UNIX> make bin/sprintf1
g++ -std=c++11 -Wall -Wextra -o bin/sprintf1 src/sprintf1.cpp
UNIX> echo 1 | bin/sprintf1
1 2 3 4 5
UNIX>
```

You want to make sure that you allocate a buffer that is big enough. If you don't, the **sprintf()** call will overrun memory, and when you do that, odd things may happen. Here's an example, in src/sprintf2.cpp

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

/* This program makes a big mistake, and does not allocate enough
    memory for the sprintf() call.  In particular, buf2 is only
    eight bytes, and the sprintf() call writes at least 10 bytes,
    and maybe more. */

int main()
{
    char buf1[8];
    char buf2[8];
    char buf3[8];
    int i;

    buf1[0] = '\0';
    buf2[0] = '\0';
    buf3[0] = '\0';

    cin >> i;

    printf("Before:\n");
    printf("buf1: %s\n", buf1);
    printf("buf2: %s\n", buf2);
    printf("buf3: %s\n", buf3);

    /* Here is where sprintf() overruns the bytes allocated for buf2. */

    sprintf(buf2, "%d %d %d %d %d", i, i+1, i+2, i+3, i+4);

    printf("After:\n");
    printf("buf1: %s\n", buf1);
    printf("buf2: %s\n", buf2);
    printf("buf3: %s\n", buf3);

    return 0;
}
```


Copyright © 2023, James S. Plank. All rights reserved.

If you're lucky, you get a seg-fault. Here, you're not so lucky (again, this is on my mac – results of this program will differ from machine to machine). The weird behavior is **buf1**. Look at what it is before and after the **sprintf()** statement:

```
UNIX> make bin/sprintf2
g++ -std=c++11 -Wall -Wextra -o bin/sprintf2 src/sprintf2.cpp
src/sprintf2.cpp:32:3: warning: 'sprintf' will always overflow; destination buffer has size 8, but
    format string expands to at least 10 [-Wfortify-source]
    sprintf(buf2, "%d %d %d %d %d", i, i+1, i+2, i+3, i+4);
    ^
1 warning generated.
UNIX> echo 10 | bin/sprintf2
Before:
buf1:
buf2:
buf3:
After:
buf1:  13 14                        # You'll note that the end of "buf2" has spilled into "buf1".
buf2: 10 11 12 13 14                # This is a really nasty bug, which can be very hard to find.
buf3:
UNIX>
```

We will explore this phenomenon in great detail in CS360. For now, just remember to make sure that your **sprintf()** buffers are big enough to hold the final strings. Typically, if I use **sprintf()** in C++, I overallocate the buffer, and instantly copy the buffer to a C++ string right after the **sprintf()** call. That way, I'm not wasting memory, but I'm not exposing myself memory bugs like the one above.

---

## Don't mess with the bytes that c_str() returns, #2.

Students hate memory allocation, because it's a pain, and many languages let them get away without doing it. Therefore, I have seen students try to use **sprintf()** to create a string without allocating a buffer for the **sprintf()** call (this is in src/bad_c_str_2.cpp):

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

/* This is a program where you have sprintf() write bytes into
    a buffer that may or may not have been allocated.  The
    typecast statement is a good sign that you're doing something
    wrong here.  Without the typecast statement, the compiler won't
    compile this code, because you shouldn't be attempting to write
    into the bytes pointed to by c_str(). */

int main()
{
    string s;

    sprintf((char *) s.c_str(), "%d", 5);
    cout << s << endl;
    return 0;
}
```


Copyright © 2023, James S. Plank. All rights reserved.

This is a disaster waiting to happen, because the c++ string doesn't know what's going on: It may or may not have allocated enough memory in its underlying string. Moreover, when you do the **sprintf()** call, the rest of the c++ string (e.g. the size field) doesn't know what has happened. When I run this, it prints nothing. When you want to use **sprintf()**, you need to allocate the buffer yourself, and make sure that it's big enough. Below (in src/good_c_str_2.cpp), we only need two characters for the buffer (for the '5' and the null character), but we use a ten-character buffer to be safe:

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
    string s;
    char buf[10];

    sprintf(buf, "%d", 5);
    s = buf;
    cout << s << endl;
}
```

---

## sscanf()

**Sscanf()** does the opposite of **sprintf()**. It takes a C-style string as its first argument, and then a format string like **sprintf()**, and then it attempts to "read" from the first string, converting what it has read into the proper data types. The variables into which it "reads" must be specified as pointers. Let's look at an example (in src/sscanf1.cpp):

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

/* Read the integer 100 from the string "100" using sscanf(). */

int main()
{
    string s;
    int i;

    s = "100";
    sscanf(s.c_str(), "%d", &i);
    printf("i = %d\n", i);

    return 0;
}
```


Copyright © 2023, James S. Plank. All rights reserved.

This "reads" the string s, and converts it to an integer **i**, which it then prints:

```
UNIX> make bin/sscanf1
g++ -std=c++11 -Wall -Wextra -o bin/sscanf1 src/sscanf1.cpp
UNIX> bin/sscanf1
i = 100
UNIX>
```

You can specify multiple inputs to read, and **sscanf()** will return the number of items that it read successfully. The program below (src/sscanf2.cpp) reads a line of text, and then tries to interpret that line as a double, followed by a space, and an int. It then prints out how many "matches" it made, the double and the int.

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

/* This program reads a line of text with getline(), and then
    uses sscanf() which attempts to read the string as a double,
    followed by a space, and an integer.  The number of correct
    "matches" is returned from sscanf().  This program prints the
    number of matches, and then the double and integer.   If sscanf()
    was unsuccessful with a conversion, then the double and/or
    integer will remain as uninitialized variables. */

int main()
{
    string s;
    int i, n;
    double d;


    getline(cin, s);
    n = sscanf(s.c_str(), "%lf %d", &d, &i);
    printf("n = %d.  d = %lf.  i = %d\n\n", n, d, i);
    return 0;
}
```

Here it is running on a variety of inputs. Make sure you understand all of these outputs. In particular, if it can't match the initial double, then it won't match the integer, ever:


Copyright © 2023, James S. Plank. All rights reserved.

```
UNIX> make bin/sscanf2
g++ -std=c++11 -Wall -Wextra -o bin/sscanf2 src/sscanf2.cpp
UNIX> echo 10.5 5 | bin/sscanf2
n = 2.  d = 10.500000.  i = 5

UNIX> echo 10.5 Fred | bin/sscanf2
n = 1.  d = 10.500000.  i = 0

UNIX> echo Fred 5 | bin/sscanf2
n = 0.  d = 0.000000.  i = 0    # Since it doesn't match the initial double,
                                  it won't match the integer either.

UNIX> echo 10.5xyz 55 | bin/sscanf2
n = 1.  d = 10.500000.  i = 0

UNIX> echo go vols | bin/sscanf2
n = 0.  d = 0.000000.  i = 0

UNIX>
```

Your input fields don't have to be separated by spaces. The following program reads lines of text, which are in the format "h:m:s". (in src/sscanf3.cpp):

```
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
    string l;
    int h, m, s, n;
    double d;

    getline(cin, l);
    n = sscanf(l.c_str(), "%d:%d:%d", &h, &m, &s);
    printf("n = %d.  h = %d.  m = %d.  s = %d.\n", n, h, m, s);
    return 0;
}
```

That may well be handy for lab 1.....

```
UNIX> g++ -std=c++11 sscanf3.cpp
UNIX> echo '55:33:22' | ./a.out
n = 3.  h = 55.  m = 33.  s = 22.
UNIX>
```

---

## But Dr. Plank, why is it ok to use c_str() in sscanf(), but not in sprintf()?

It's ok because **sscanf()** does not modify the bytes in its first argument. It merely reads them, so our code is fine. On the other hand, **sprintf()** does modify the bytes of its first argument, and that's why we shouldn't use it.



Copyright © 2023, James S. Plank. All rights reserved.

## What about snprintf() – doesn't that solve the problems with sprintf?

There is an alternative to **sprintf()** named **snprintf()**, and some compilers have started to emit a warning when they see **sprintf()**, advocating that you use **snprintf()** instead. With **snprintf()**, you pass the size of the buffer as an argument. For example, src/snprintf.cpp is identical to src/sprintf1.cpp, except it uses **snprintf()** instead of **sprintf()**:

```
// ...
snprintf(buf, 100, "%d %d %d %d %d", i, i+1, i+2, i+3, i+4);
// ...
```

It's a good habit to get into, because it will catch some bugs which can otherwise be disastrous and hard to find. It's not a panacea, though, because you can give it an incorrect size, and it will still compile and run in a buggy fashion. I won't demonstrate that here, so just take my word for it. You'll typically see me use **sprintf()** instead of **snprintf()**, partially because of habit, and partially because I feel like I feel it's unneccessary, so long as you are programming carefully.

## Q & A: What if there is a string before the time?

Please see pdf/2022-01-25-Piazza.pdf:
I have a question about sscanf() lecture.

On the lecture note, there is a code that reads the format of "h:m:s"

```
sscanf(l.c_str(), "%d:%d:%d", &h, &m, &s);
```

And I was wondering why it cannot store h, m, and s when the input is "abcdef 12:34:56" and how can I make it work?

Thank you!

----------------------------------------

**The Instructors' Answer** - *where instructors collectively construct a single answer*

If s = "abcdef 12:34:56", and you want to read the "12:34:56" part, then there are two avenues to go down.

1. Use a C++ stringstream, or perhaps the find() and substr() methods of C++ strings to isolate the "12:23:56" into it's own C++ string. Then use the sscanf() call above. This is what I'd recommend.

2. (Not recommended, but I'm putting it here in case someone recommends it). If you know that there is a single string in front of the time, and you know how big it is, then you can do:

```
sscanf(l.c_str(), "%s %d:%d:%d", buf, &h, &m, &s);
```

but will need to be a c-style array of bytes that is big enough to hold the initial string (plus the NULL character). The reason I don't recommend this is that if you mess up, and buf is too small, you open yourself up to memory corruption and buffer overflow attacks. You'll learn more about that in CS360. – JP

Copyright (c) 2023, James S. Plank


Copyright © 2023, James S. Plank. All rights reserved.