CMPSC 311 - Introduction to Systems Programming

IF YOU FORGET TO NULL TERMINATE YOUR STRINGS
INSTRUCTOR
YOU'RE GONNA HAVE A BAD TIME^C*&SC6VA*(&ER^#Q*R^BS&BV8 (D&V(BYUGF
memegenerator.net

Strings in C

Professors:
Suman Saha
(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)

ASCII

- American Standard Code for Information Interchange

```
  0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack    7 bel
  8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so    15 si
 16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn   23 etb
 24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs    31 us
 32 sp    33  !    34  "    35  #    36  $    37  %    38  &    39  '
 40  (    41  )    42  *    43  +    44  ,    45  -    46  .    47  /
 48  0    49  1    50  2    51  3    52  4    53  5    54  6    55  7
 56  8    57  9    58  :    59  ;    60  <    61  =    62  >    63  ?
 64  @    65  A    66  B    67  C    68  D    69  E    70  F    71  G
 72  H    73  I    74  J    75  K    76  L    77  M    78  N    79  O
 80  P    81  Q    82  R    83  S    84  T    85  U    86  V    87  W
 88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^    95  _
 96  `    97  a    98  b    99  c   100  d   101  e   102  f   103  g
104  h   105  i   106  j   107  k   108  l   109  m   110  n   111  o
112  p   113  q   114  r   115  s   116  t   117  u   118  v   119  w
120  x   121  y   122  z   123  {   124  |   125  }   126  ~   127 del
```

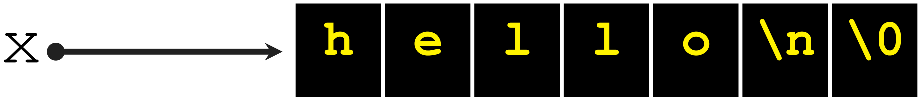
```
int a = 65;
printf( "a is %d or in ASCII \'%c\'\n", a, (char)a );
```

```
a is 65 or in ASCII 'A'
```

A string is just an array …

- C handles ASCII text through strings: null-terminated array characters

```
// Different ways to define strings
char *x = ”hello\n”;
char x1[] = ”hello\n”;
char x2[7] = ”hello\n”; // Why 7?
// All have the same output
printf(“%s\n”, x);
printf(“%s\n”, x1);
printf(“%s\n”, x2);
```



- There are a large number of interfaces for managing strings available in the C library, i.e., `string.h`.

How are strings stored in memory

- A subtle difference between the following definitions

```
// All of these are equivalent
char *x = ”hello\n”;
char x1[] = ”hello\n”;

x[4] = ‘j’;  // Segmentation fault
x1[4] = ‘j’; // OK
```

- String literals are in read-only region of memory
- Arrays (char x[]) are initialized by the compiler
  - Contents copied from string in read-only region
- Pointers (char *x) point directly to read-only region

sizeof vs strlen

- There are two ways of determining the “size” of the string, each with their own semantics
  - `sizeof(string)` returns the size of the declaration (sometimes, beware)
  - `strlen(string)` returns the length of the string, not including the null terminator

```
char *str = "text for example";
char str2[17] = "text for example";
printf( "str has size %lu\n", sizeof(str) );
printf( "str2 has size %lu\n", sizeof(str2) );
printf( "str has length %lu\n", strlen(str) );
printf( "str2 has length %lu\n", strlen(str2) );
```

```
str has size 8
str2 has size 17
str has length 16
str2 has length 16
```

Initializing strings …

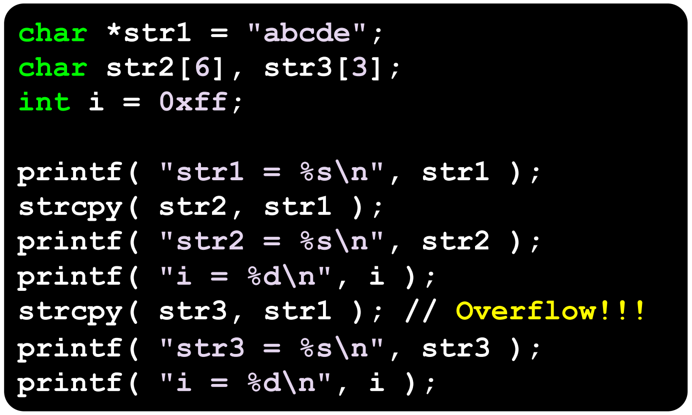
- Most legitimate (which are not?)
- The bad strings have no null terminator
  - This is called an unterminated string
  - Bad, scary things can happen when you work with unterminated strings (don’t do it).

```
char *str1 = "abc";
char str2[] = "abc";
char str3[4] = "abc";
char str4[3] = "abcd";
char str5[]  = {'a', 'b', 'c', '\0'};
char str6[3] = {'a', 'b', 'c'};
char str7[9] = {'a', 'b', 'c'};

printf( "str1 = %s\n", str1 );
printf( "str2 = %s\n", str2 );
printf( "str3 = %s\n", str3 );
printf( "str4 = %s\n", str4 );
printf( "str5 = %s\n", str5 );
printf( "str6 = %s\n", str6 );
printf( "str7 = %s\n", str7 );
```

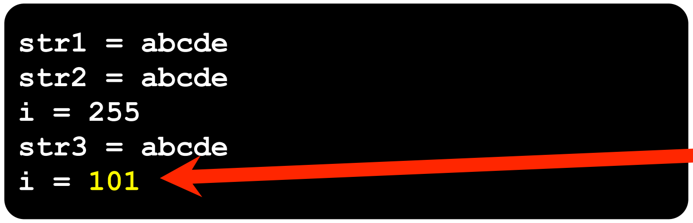
```
str1 = abc
str2 = abc
str3 = abc
str4 = abc*@
str5 = abc
str6 = abc
str7 = abc
```

Copying strings

`strcpy(dest, src)` copies src to dest up to and including null-terminator



```c
char *str1 = "abcde";
char str2[6], str3[3];
int i = 0xff;

printf( "str1 = %s\n", str1 );
strcpy( str2, str1 );
printf( "str2 = %s\n", str2 );
printf( "i = %d\n", i );
strcpy( str3, str1 ); // Overflow!!!
printf( "str3 = %s\n", str3 );
printf( "i = %d\n", i );
```



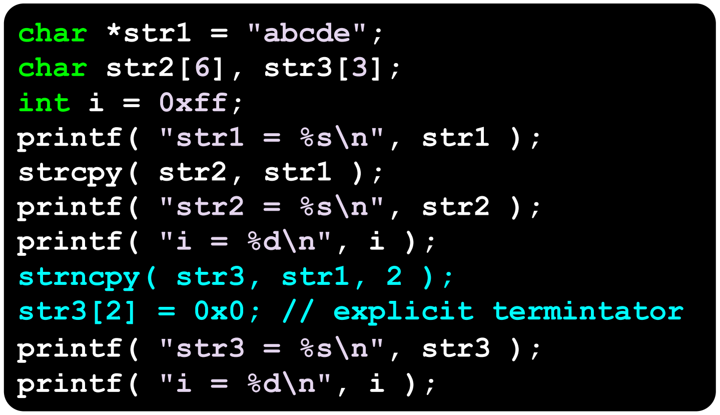
```
str1 = abcde
str2 = abcde
i = 255
str3 = abcde
i = 101
```

Stomp

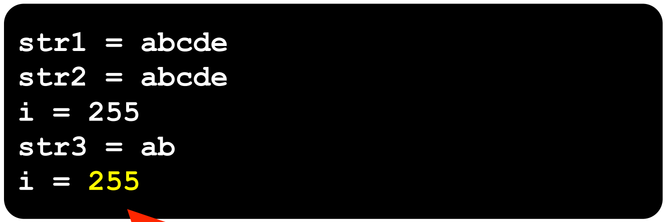
n-variants of string functions

- The best way to thwart buffer overflows (and generally make more safe code) is to use the “n” variants of the string functions
  - For example, you can copy a string to make it safe

```
strncpy(dest, src, n)
```



```
char *str1 = "abcde";
char str2[6], str3[3];
int i = 0xff;
printf( "str1 = %s\n", str1 );
strcpy( str2, str1 );
printf( "str2 = %s\n", str2 );
printf( "i = %d\n", i );
strncpy( str3, str1, 2 );
str3[2] = 0x0; // explicit termintator
printf( "str3 = %s\n", str3 );
printf( "i = %d\n", i );
```



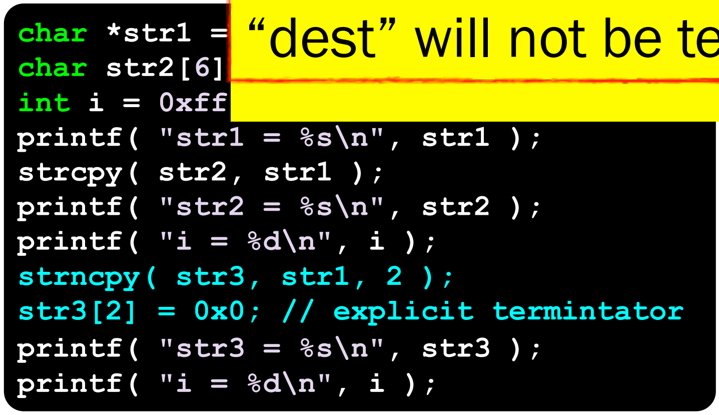
```
str1 = abcde
str2 = abcde
i = 255
str3 = ab
i = 255
```

No Stomp

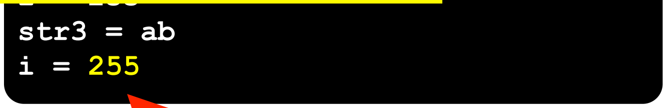
n-variants of string functions

- The best way to thwart buffer overflows (and generally make more safe code) is to use the “n” variants of the string functions
  - For example, you can copy a string to make it safe

Warning: if the source does not have a NULL terminator in first `n` bytes, “dest” will not be terminated.



```
char *str1 =
char str2[6]
int i = 0xff
printf( "str1 = %s\n", str1 );
strcpy( str2, str1 );
printf( "str2 = %s\n", str2 );
printf( "i = %d\n", i );
strncpy( str3, str1, 2 );
str3[2] = 0x0; // explicit termintator
printf( "str3 = %s\n", str3 );
printf( "i = %d\n", i );
```



```
i = 255
str3 = ab
i = 255
```

No Stomp

Concatenating strings …

- Often we want to “add” strings together (concatenate) to make one long string, e.g., as in C++ `(str = str1 + str2)`
- In C, we use strcat (which appends src to dest)

  `strcat(dest, src);`
- The strncat variant copies at most n bytes of src

  `strncat(dest, src, n);`

```
char str1[20] = "abcde",
     *str2 = "efghi",
     str3[20] = "abcde";
    strcat( str1, str2 );
    printf( "str1 is [%s]\n", str1 );
    strncat( str3, str2, 20 );
    printf( "str3 is [%s]\n", str3 );
```

```
str1 is [abcdeefghi]
str3 is [abcdeefghi]
```

String comparisons …

- Compare strings to see if they match or are lexicographically smaller or larger
- In C, we use `strcmp` (which compares s1 to s2)

```
strcmp(s1, s2);
```

- `strncmp` compares first n bytes of strings

```
strncmp(s1, s2, n);
```

- The comparison functions return
  - negative integer if s1 is less than s2
  - 0 if s1 is equal to s2
  - positive integer is s1 greater than s2

Searching strings

- Search through strings to find something we are looking for:
  - strchr searches front to back for a character
  - strrchr searches back to front for a character

```
strchr(str, char_to_find);
strrchr(str, char_to_find);
```

  - strstr searches front to back for a string
  - strcasestr searches from front for a string (ignoring case)

```
strstr(str, str_to_find);
strcasestr(str, str_to_find);
```

- All of these functions return a pointer within the string to the found value or NULL if not found

Example searches

```
char *str = "xxxx0xxxFindmexxxx0xxxxFindme2xxxxx";
printf( "Looking for character %c, strchr  : %s\n", '0',
        strchr(str,'0') );
printf( "Looking for character %c, strrchr : %s\n", '0',
        strrchr(str,'0') );
printf( "Looking for string %5s, strstr     : %s\n", "Findme",
        strstr(str,"Findme") );
printf( "Looking for string %5s, strstr     : %s\n", "FINDME",
        strstr(str,"FINDME") );
printf( "Looking for string %5s, strcasestr : %s\n", "FINDME",
        strcasestr(str,"FINDME") );
```

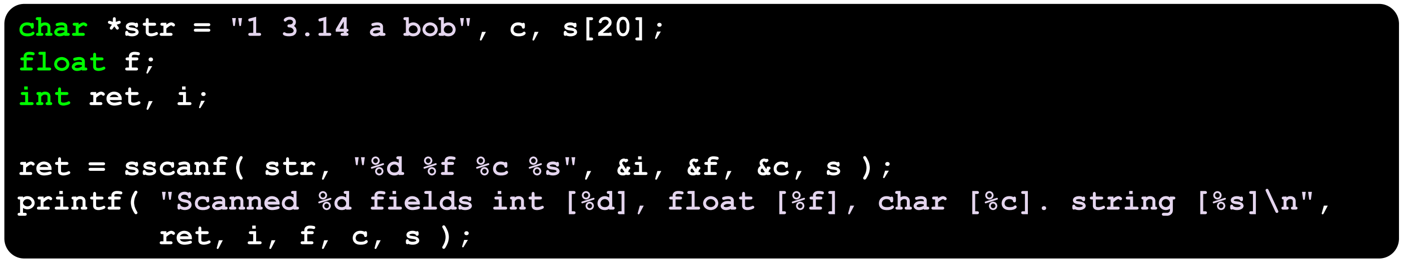
```
Looking for character 0, strchr  : 0xxxFindmexxxx0xxxxFindme2xxxxx
Looking for character 0, strrchr : 0xxxxFindme2xxxxx
Looking for string Findme, strstr    : Findmexxxx0xxxxFindme2xxxxx
Looking for string FINDME, strstr    : (null)
Looking for string FINDME, strcasestr: Findmexxxx0xxxxFindme2xxxxx
```

Parsing strings …

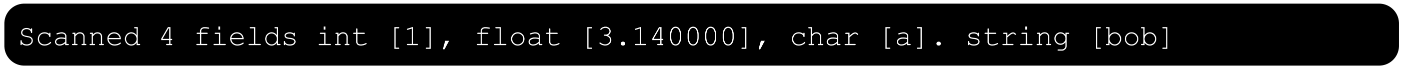
- Strings carry information we want to translate (parse) into other (variables)
- In C, we use `sscanf` which extracts data by format

  `sscanf(str, “format”, ...);`

- The syntax is very similar to that of `printf`, but your arguments must be passed by reference.
  - Returns the number of arguments successfully parsed



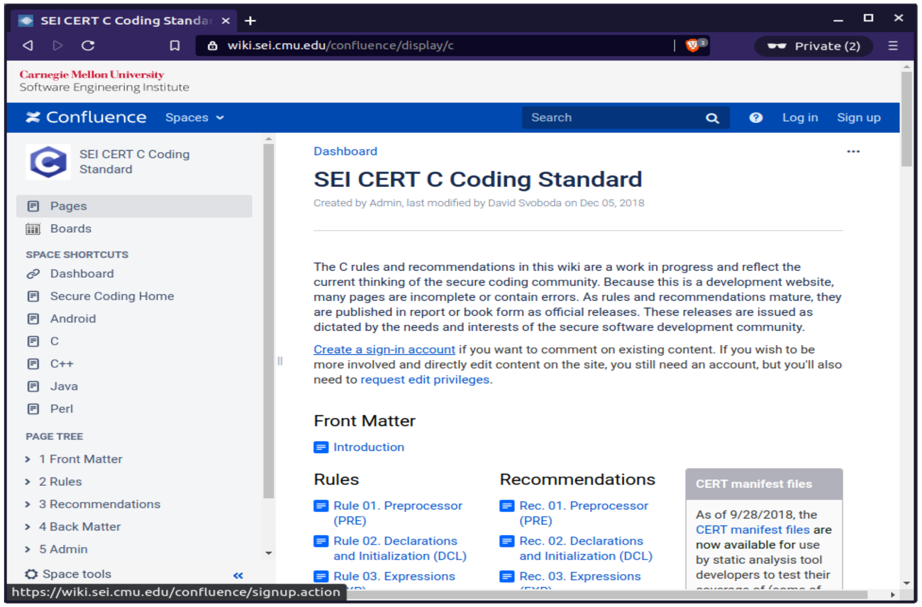
```c
char *str = "1 3.14 a bob", c, s[20];
float f;
int ret, i;

ret = sscanf( str, "%d %f %c %s", &i, &f, &c, s );
printf( "Scanned %d fields int [%d], float [%f], char [%c]. string [%s]\n",
        ret, i, f, c, s );
```



```
Scanned 4 fields int [1], float [3.140000], char [a]. string [bob]
```

C string API is extremely error-prone

- Study secure C coding guidelines
- Or learn a safe systems language: Rust



Dashboard

SEI CERT C Coding Standard

Created by Admin, last modified by David Svoboda on Dec 05, 2018

The C rules and recommendations in this wiki are a work in progress and reflect the current thinking of the secure coding community. Because this is a development website, many pages are incomplete or contain errors. As rules and recommendations mature, they are published in report or book form as official releases. These releases are issued as dictated by the needs and interests of the secure software development community.

Create a sign-in account if you want to comment on existing content. If you wish to be more involved and directly edit content on the site, you still need an account, but you'll also need to request edit privileges.

Front Matter

Introduction

Rules

- Rule 01. Preprocessor (PRE)
- Rule 02. Declarations and Initialization (DCL)
- Rule 03. Expressions (EXP)

Recommendations

- Rec. 01. Preprocessor (PRE)
- Rec. 02. Declarations and Initialization (DCL)
- Rec. 03. Expressions (EXP)

CERT manifest files

As of 9/28/2018, the CERT manifest files are now available for use by static analysis tool developers to test their coverage of (some of