CMPSC 311 - Introduction to Systems Programming
Module: Strings

Professor Patrick McDaniel
Fall 2013
A string is just an array ...

- C handles ASCII text through \textit{strings}
- A string is just an array of characters
  - Which is really just a pointer

```
// All of these are equivalent
char *x = "hello\n";
char x1[] = "hello\n";
char x2[7] = "hello\n"; // Why 7?
```

- There are a large number of interfaces for managing strings available in the C library, i.e., \texttt{string.h}.
### ASCII

- **American Standard Code for Information Interchange**

<table>
<thead>
<tr>
<th>ASCII Value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>Control</td>
</tr>
<tr>
<td>32-126</td>
<td>Alpha</td>
</tr>
<tr>
<td>127-255</td>
<td>Others</td>
</tr>
</tbody>
</table>

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

- `a is 65 or in ASCII 'A'`
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
  - `strlen(string)` returns the length of the string, not including the null terminator

```c
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) );
```
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
  ‣ `strlen(string)` returns the length of the string, not including the null terminator

```c
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 ...

```c
char *str1 = "abc";
char str2[] = "abc";
char str3[4] = "abc";
char str4[3] = "abcd"; // Wat?
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);
```
Initializing strings ...

```c
char *str1 = "abc";
char str2[] = "abc";
char str3[4] = "abc";
char str4[3] = "abcd"; // Wat?
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
```
Initializing strings ...

- All of these are legitimate except `str4`
- The initialization of `str4` only copies the first 3 characters and no NULL terminator
  - This is called an **unterminated string**
  - Big, scary things can happen when you work with unterminated strings (don’t do it).

```c
char *str1 = "abc";
char str2[] = "abc";
char str3[4] = "abc";
char str4[3] = "abcd"; // Wat?
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);
```
Copying strings

• `strcpy` allows you to copy one string to another
  ‣ It searches NULL terminator and copies everything up to that point, plus the terminator
  ‣ Copy from “source” string to “destination” string

\[
\text{strcpy} (\text{dest}, \text{src})
\]

is kinda like `dest = src`

```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 );
printf( "str3 = %s\n", str3 );
printf( "i = %d\n", i );
```
Copying strings

• `strcpy` allows you to copy one string to another
  ‣ It searches NULL terminator and copies everything up to that point, plus the terminator
  ‣ Copy from “source” string to “destination” string

\[
\text{strcpy}( \text{dest, src} )
\]

is kinda like \texttt{dest = src}

code

```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 );
printf("str3 = %s\n", str3);
printf("i = %d\n", i);
```

\[
\text{str1 = abcde}
\text{str2 = abcde}
\text{i = 255}
\text{str3 = abcde}
\text{i = 101}
\]
Copying strings

- `strcpy` allows you to copy one string to another
  - It searches NULL terminator and copies everything up to that point, plus the terminator
  - Copy from “source” string to “destination” string

\[ \text{strcpy}(\text{dest}, \text{src}) \]

is kinda like \( \text{dest} = \text{src} \)

```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 );
printf( "str3 = %s\n", str3 );
printf( "i = %d\n", i );
```

\begin{align*}
\text{str1} &= \text{abcde} \\
\text{str2} &= \text{abcde} \\
\text{i} &= 255 \\
\text{str3} &= \text{abcde} \\
\text{i} &= 101
\end{align*}
• A buffer overflow is when you intentionally overwrite some data on the stack to take over the process
  ‣ When adversary controls, they can take over the process.
  ‣ Specifically, the return pointer

```c
char buf[5];
printf( "Please enter some text:\n" );
scanf( "%s", buf );
```

Please enter some text:
thisissomelongtext
*** stack smashing detected ***: process terminated
Aborted (core dumped)
Buffer overflows ...

- A buffer overflow is when you intentionally overwrite some data on the stack to take over the process
  - When adversary controls, they can take over the process.
  - Specifically, the return pointer

```c
char buf[5];
printf( "Please enter some text:\n" );
scanf( "%s", buf );
```

Please enter some text: thisissomelongtext
*** stack smashing detected ***: process terminated
Aborted (core dumped)
Buffer overflows ...

- A buffer overflow is when you intentionally overwrite some data on the stack to take over the process
  - When adversary controls, they can take over the process.
  - Specifically, the return pointer

```c
char buf[5];
printf( "Please enter some text:\n" );
scanf( "%s", buf );
```

Please enter some text:
```text
thisissomelongtext
```
*** stack smashing detected ***: process terminated
Aborted (core dumped)
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

\[
\text{strncpy(dest, src, n)}
\]

```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);
strncpy(str3, str1, 2);
str3[2] = 0x0; // explicit terminator
printf("str3 = %s\n", str3);
printf("i = %d\n", i);
```
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

\[
\text{\texttt{strncpy(dest, src, n)}}
\]

```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 );
strncpy( str3, str1, 2 );
str3[2] = 0x0; // explicit terminator
printf( "str3 = %s\n", str3 );
printf( "i = %d\n", i );
```

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

\texttt{strncpy(dest, src, n)}
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

```
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 terminator
printf("str3 = %s\n", str3);
printf("i = %d\n", i);
```

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

No Stomp
Concatenating strings ...

- Often we want to “add” strings together to make one long string, e.g., as in C++ \((\text{str} = \text{str1} + \text{str2})\)
- In C, we use \texttt{strcat} (which appends src to dest)
  \[
  \text{strcat}(\text{dest}, \text{src});
  \]
- The \texttt{strncat} variant copies at most \texttt{n} bytes of src
  \[
  \text{strncat}(\text{dest}, \text{src}, \text{n});
  \]

```c
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 );
```
Concatenating strings ...

- Often we want to “add” strings together to make one long string, e.g., as in C++ (str = str1 + str2)
- In C, we use `strcat` (which appends src to dest)
  
  ```c
  strcat(dest, src);
  ```
- The `strncat` variant copies at most n bytes of src
  
  ```c
  strncat(dest, src, n);
  ```

```c
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 ...

• We often want to compare strings to see if they match or are lexicographically smaller or larger

• In C, we use `strcmp` (which compares `s1` to `s2`)
  ```c
  strcmp(s1, s2);
  ```

• `strncmp` compares first `n` bytes of strings
  ```c
  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`
How is a string greater than?

```c

for (i=0; i<6; i++) {
    printf( "Compare %2s to : n", str[i] );
    for (j=0; j<6; j++) {
        printf( "%2s=(%3d) ", str[j], strcmp(str[i], str[j]) );
    }
    printf( "\n" );
}
```

<table>
<thead>
<tr>
<th>Compare</th>
<th>a to</th>
<th>a=(</th>
<th>b=(</th>
<th>c=(</th>
<th>ac=</th>
<th>1=(</th>
<th>_(</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>b to</td>
<td>a=(</td>
<td>b=(</td>
<td>c=(</td>
<td>ac=</td>
<td>1=(</td>
<td>_(</td>
</tr>
<tr>
<td>Compare</td>
<td>c to</td>
<td>a=(</td>
<td>b=(</td>
<td>c=(</td>
<td>ac=</td>
<td>1=(</td>
<td>_(</td>
</tr>
<tr>
<td>Compare</td>
<td>ac to</td>
<td>a=(</td>
<td>b=(</td>
<td>c=(</td>
<td>ac=</td>
<td>1=(</td>
<td>_(</td>
</tr>
<tr>
<td>Compare</td>
<td>1 to</td>
<td>a=(</td>
<td>b=(</td>
<td>c=(</td>
<td>ac=</td>
<td>1=(</td>
<td>_(</td>
</tr>
<tr>
<td>Compare</td>
<td>_ to</td>
<td>a=(</td>
<td>b=(</td>
<td>c=(</td>
<td>ac=</td>
<td>1=(</td>
<td>_(</td>
</tr>
</tbody>
</table>
Searching strings

• Often we want to 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
    
    ```c
    strchr(str, ch_to_find);
    strrchr(str, ch_to_find);
    ```
  ‣ `strstr` searches front to back for a string
  ‣ `strcasestr` searches from front for a string (ignoring case)
    
    ```c
    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

```c
char *str = "xxxx0xxxFindmexxxx0xxxxFindme2xxxxx";
printf( "Looking for character %c, strchr : %s\n", 'c',
        strchr(str,'0') );
printf( "Looking for character %c, strrchr : %s\n", 'c',
        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 : 0xxxxFindmexxxx0xxxxFindme2xxxxx
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 forms (variables)
- In C, we use `sscanf` which extracts data by format
  
  ```
  scanf(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]
Tokenizing strings ...

• Input is often in a form ready for parsing, such as the `.csv` format (comma separated values)

  Patrick,McDaniel,CMPSC311,Professor
  Junpeng,Qiu,CMPSC311,TA
  Prashanth,Thinakaran,CMPSC311,TA

• We want to be able to pull that data apart so we can process it, where each field is a token
  ‣ Here we use the `strtok` function
    ```
    strtok(str, delim);
    ```
  ‣ First use pass the string to parse, thereafter NULL
Tokenizing example

```c
   "Patrick,McDaniel,CMPSC311,Professor",
   "Junpeng,Qiu,CMPSC311,TA",
   "Prashanth,Thinakaran,CMPSC311,TA"
};

for (i=0; i<3; i++) {
   // Duplicate the string (avoid modofying original)
   nptr = strdup(input[i]);

   // First time supply string to parse
   ptr = strtok( nptr, "\," );
   while (ptr != NULL) {
      // Subsequent times pass NULL
      printf( "Next token [%s]\n", ptr);
      ptr = strtok( NULL, "\," );
   }
   free( nptr );
}
```

- Next token [Patrick]
- Next token [McDaniel]
- Next token [CMPSC311]
- Next token [Professor]
- Next token [Junpeng]
- Next token [Qiu]
- Next token [CMPSC311]
- Next token [TA]
- Next token [Prashanth]
- Next token [Thinakaran]
- Next token [CMPSC311]
- Next token [TA]
System security/reliability

• Input received from outside the process must be validated to ensure it has the correct format/content.
  ‣ This is particularly true of strings because it is so easy to make a critical mistake and leave the system vulnerable
• Most of the attacks on the web happen because this was not done properly.
  ‣ Leads to things like cross-site scripting attacks, e.g., NASDAQ

“This means anyone could inject arbitrary HTML code into Nasdaq.com to display a fake web form demanding credit card numbers and other personal information or to inject malware to infect PC users. The only limit is the hacker’s imagination.”
- Ilia Kolochenko (2013)
Mid-term 1 results

Min 24.5, Max 92, Avg. 65, Median 70

Bar chart showing the distribution of grades:
- A: 4
- B: 35
- C: 25
- D: 15
- F: 20

Grade ranges:
- >=90: A
- >=80: B
- >=70: C
- >=60: D
- >=50: F
- <50: No grade