CMPSC 311 - Introduction to Systems Programming
Module: Strings

Professor Patrick McDaniel
Fall 2015
A string is just an array ...

- C handles ASCII text through *strings*
- A string is just an array of characters
  ‣ Which is really just a pointer

```c
// 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., *string.h*. 
• American Standard Code for Information Interchange

```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 (sometimes, beware)
  ‣ `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 ...

- All legitimate except \texttt{str4} \texttt{str6} \texttt{str7}

- The bad strings have \underline{no} NULL terminator
  - This is called an \emph{unterminated string}
  - Bad, scary things can happen when you work with unterminated strings (don’t do it).
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

  ```
  strcpy(dest, 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 );
```

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

Stomp
Buffer overflows ...

- A buffer overflow is when you 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)
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)}

```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 );
```

\texttt{str1 = abcde}  \hspace{1cm} \texttt{str2 = abcde}  \hspace{1cm} \texttt{i = 255}  \hspace{1cm} \texttt{str3 = ab}  \hspace{1cm} \texttt{i = 255}

\textbf{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.

```c
#include <stdio.h>

int main() {
    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);

    return 0;
}
```
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`)
  ```
  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`
How is a string greater than?

```c
char *str[6] = { "a", "b", "c", "ac", "1", "_"};
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" );
}
```

```
Compare  a to : n a=(  0)  b=( -1)  c=( -2)  ac=(-99)  1=( 48)  _=(  2)
Compare  b to : n a=(  1)  b=(  0)  c=( -1)  ac=(  1)  1=( 49)  _=(  3)
Compare  c to : n a=(  2)  b=(  1)  c=(  0)  ac=(  2)  1=( 50)  _=(  4)
Compare  ac to : n a=( 99)  b=( -1)  c=( -2)  ac=(  0)  1=( 48)  _=(  2)
Compare   1 to : n a=(-48)  b=(-49)  c=(-50)  ac=(-48)  1=(  0)  _=(-46)
Compare   _ to : n a=( -2)  b=( -3)  c=( -4)  ac=( -2)  1=( 46)  _=(  0)
```
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, 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)
    
    ```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 : 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 forms (variables).
- In C, we use `sscanf` which extracts data by format:
  ```c
  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]
Tokenizing strings ...

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

  Patrick,McDaniel,CMPSC311,Professor
  Devin,Pohly,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
    
    ```c
    strtok(str, delim);
    ```
  ‣ First use pass the string to parse, thereafter NULL
Tokenizing example

```c
    "Patrick,McDaniel,CMPSC311,Professor",
    "Devin,Pohly,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 [Devin]
Next token [Pohly]
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)