**Lecture 22 - Chapter 8: C Characters and Strings – Mon Nov 6 or Tues Nov 7**

**Announcements**

Reading:

* Chapter 8

Assignments:

* Assignment #9 - due on **Nov 8** (MW class) or **Nov 9** (TR class) **(no late assignments accepted)**

**Today’s Goals**

1. String Manipulation Functions of the String-Handling Library
2. Comparison Functions of the String-handling Library
3. Search Functions of the String-Handling Library
4. Memory Functions of the String-Handling Library
5. Other Functions of the String-Handling Library
6. Secure C Programming

**Today’s Terminology**

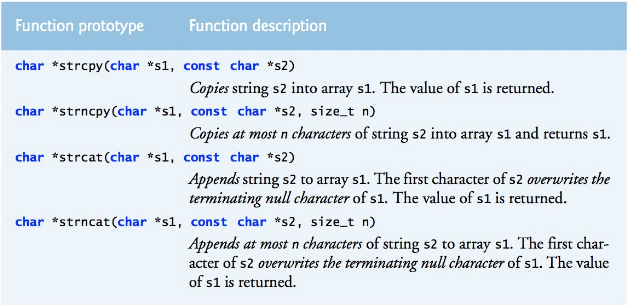
**Terminology**

* C Standard Library Functions
  + Contains functions we can use to process characters, strings, lines of text and blocks of memory.
* C11 Annex K Functions
  + *Optional* annex to C standard library
  + C11 specifies many security features
  + Reduce unexpected behavior and prevent common attacks
  + Contains more secure versions of many of the functions we will see in this chapter
  + Should use if available with your compiler.
  + More on this below
* Character Constants
  + **int** value represented as one character in single quotes
* String Constants (Literals)
  + Sequence of characters treated as one unit
  + Written in double quotes

**String Manipulation Functions of the String-Handling Library**

**<string.h> - String-handling library**

* Functions to perform string manipulation
* Every function, except strncpy, appends null character to its result. Strncpy may or may not copy ‘\0’
* Figure 8.14



* String copying functions
  + **strcpy**
    - Same issue as scanf – need to make sure the string is copied into a large enough array
    - Be aware of buffer overflow possibilities
    - **strcpy\_s** - More secure version in C11
  + **strncpy**
    - Includes a size – solves potential buffer overflow problem – if used properly!
    - Can still have buffer overflow!
    - If string if too big to fit into target array
      * Silently truncates string
      * Leaves target array without a terminating ‘\0’ – this is not good!
    - **strncpy\_s** - More secure version in C11
  + **Good strcpy()**
    - **strcpy (char \*s1, const char \*s2);**

**char** str[] = "123456789";

**char** s10[10];

// Copy string of length 9 into array that can hold 10 characters

**printf** ("s10 contains the string %s\n", **strcpy**(s10, str));

**puts**("Let's print the individual characters");

**for** (**unsigned** **int** i = 0; i < 10; i++) {

**printf** ("%s%d%s%c\n","s10[", i, "] = ", s10[i]);

}

// Check if the NULL terminating character was placed in array

// Note: strcpy appends null to the result

**if** (s10[9] == '\0') {

**puts** ("s10 contains the NULL terminating character");

}

**else** {

**puts** ("S10 does not contain the NULL terminating character");

}

**Displays**

s10 contains the string 123456789

Let's print the individual characters

s10[0] = 1

s10[1] = 2

s10[2] = 3

s10[3] = 4

s10[4] = 5

s10[5] = 6

s10[6] = 7

s10[7] = 8

s10[8] = 9

s10[9] =

s10 contains the NULL terminating character

* + **Bad strcpy()**

**char** str[] = "123456789";

**char** s7[7];

// Copy string of length 9 into array that can hold 7 characters

// This is buffer overflow and creates undefined behavior.

**printf** ("s7 contains the string %s\n", **strcpy**(s7, str));

**puts** ("Let's print the individual characters");

**for** (**unsigned** **int** i = 0; i < 7; i++) {

**printf** ("%s%d%s%c\n","s7[", i, "] = ", s7[i]);

}

**if** (s7[6] == '\0') {

**puts** ("s7 contains the NULL terminating character");

}

**else** {

**puts** ("S7 does not contain the NULL terminating character");

}

Produces undefined behavior!

**Displays** This shouldn’t show 9 characters

s7 contains the string 123456789 No NULL terminating character so

Let's print the individual characters printf kept going until found ‘\0’

s7[0] = 1 and it ended up printing 123456789

s7[1] = 2 not 123456 has wanted

s7[2] = 3

s7[3] = 4

s7[4] = 5

s7[5] = 6

s7[6] = 7

S7 does not contain the NULL terminating character

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| s7[0] | s7[1] | s7[2] | s7[3] | s7[4] | s7[5] | | s7[6] | |
| ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | | ‘6’ | | ‘7’ |

S7 is 7 characters long and null should be here

but input was too big so it’s not and have overflow!

* + **Good strncpy()**
    - **strncpy (char \*s1, const char \*s2, size\_t n);**
    - **copies at most n characters of string s2 into array s1**
    - **the one function that does not necessarily copy null of 2nd argument to 1st argument!!!**
      * only copied if number of characters to be copied is MORE than length of string

**char** str[] = "123456789";

**char** s6[6];

// Copy first 3 characters from str into s6.

**strncpy** (s6, str, 3);

**for** (**unsigned** **int** i = 0; i < 6; i++) {

**printf** ("%s%d%s%c\n","s6[", i, "] = ", s6[i]);

}

// Check if the NULL terminating character was placed in array

// NULL is only copied if number of characters to be copied is MORE than

// the length of the string – in this case that is not true because the second

// argument is “123456789” which is 10 in length and the number of characters

// to be copied is 3 which is NOT more than 10

**if** (s6[3] == '\0') {

**puts** ("s6 contains the NULL terminating character");

}

**else** {

**puts** ("S6 does not contain the NULL terminating character");

}

**Displays**

s6[0] = 1

s6[1] = 2

s6[2] = 3

s6[3] =

s6[4] =

s6[5] =

s6 does not contain the NULL terminating character

* + **Bad strncpy() – cause buffer overflow**

**char** str[] = "123456789";

**char** s6[6];

**puts** ("Specified a size too big for s6 using strncpy – will cause overflow!");

**strncpy**(s6, str, **sizeof**(str));

**for** (**unsigned** **int** i = 0; i < 6; i++) {

**printf** ("%s%d%s%c\n","s6[", i, "] = ", s6[i]);

}

// Check if the NULL terminating character was placed in array

// Characters are in locations 0, 1, 2, 3, 4 so ‘\0’ should be 5th location

**if** (s6[5] == '\0') {

**puts** ("s6 contains the NULL terminating character");

}

**else** {

**puts** ("S6 does not contain the NULL terminating character");

}

**Displays**

Specified a size too big for s6 using strncpy – will cause overflow!

s6[0] = 1

s6[1] = 2

s6[2] = 3

s6[3] = 4

s6[4] = 5

s6[5] = 6

S6 does not contain the NULL terminating character

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| s6[0] | s6[1] | s6[2] | s6[3] | s6[4] | s6[5] |
| ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ |

S6 is only 6 characters long and null should be here

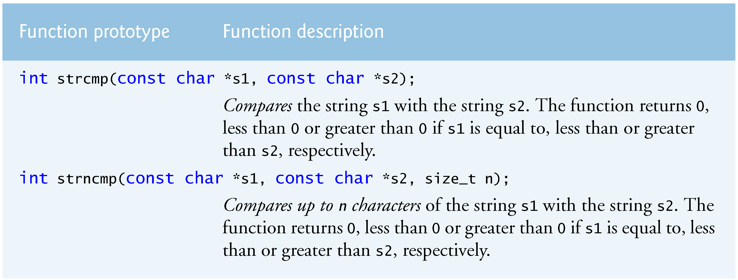
but input was too big so null is missing and have overflow!

* String concatenation functions
  + **strcat**
    - Target array must be big enough to hold original string, second string and the terminating character.
    - **strcat\_s** - More secure version in C11
  + **strncat**
    - Includes a size – solves potential buffer overflow problem if used properly!
    - Can still have buffer overflow!
    - **strncat\_s** - More secure version in C11

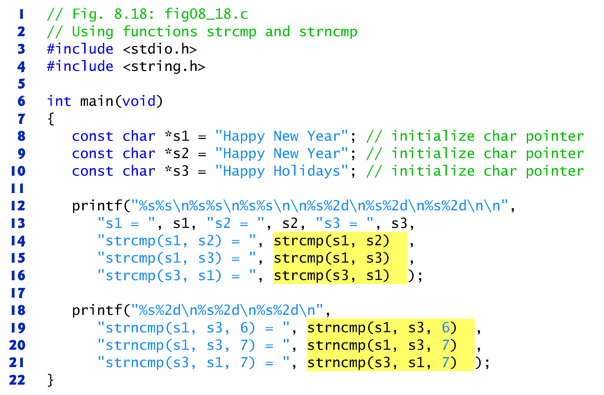
**Comparison Functions of the String-handling Library**

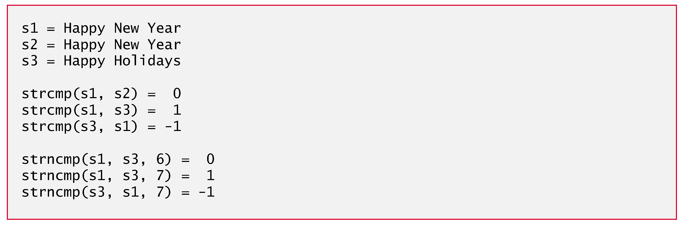
**<string.h> - String-handling library – comparison functions**

* Looked at this library last lecture
* String comparison functions are in Figure 8.17



* **strcmp**
  + Returns an integer
    - 0 means s1 = s2
    - < 0 means s1 < s2
    - > 0 means s1 > s2
  + Useful for testing user input against expected values
  + Example uses:
    - Checking if username/password match what is in database
* **strncmp**
  + Same as strcmp – except compares a certain number of characters
  + Does not compare characters following ‘\0’ character
* Go over code in Figure 8.18
  + Can you tell me why?
    - strcmp (s1, s3) = 1? (How does ASCII code fit in?)
    - strcmp (s3, s1) = -1?
  + Can you tell me why?
    - strncmp (s1, s3, 7) = 1?
    - strncmp (s3, s1, 7) = -1

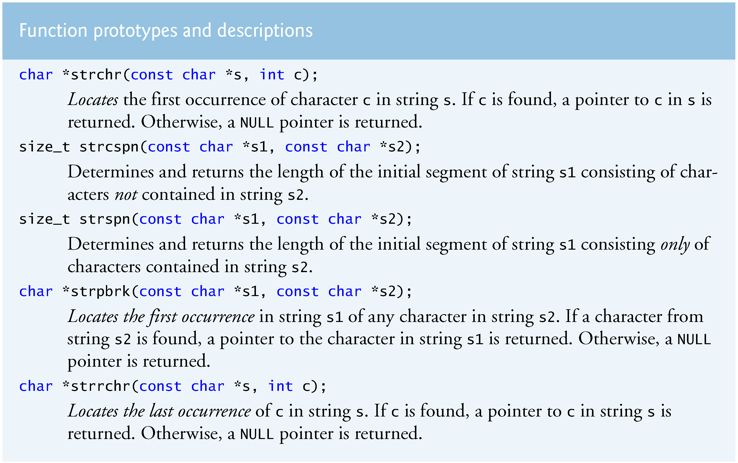


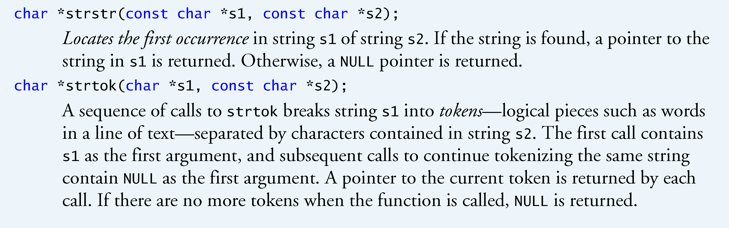


**Search Functions of the String-Handling Library**

**<string.h> - String-handling library – search functions**

* Functions to search strings are very helpful
* String search functions are in Figure 8.19





* **strchr**
  + Searches for ONE character – finds first occurrence
  + Useful when looking for specific characters in a string
  + Example uses:
    - Does password have at least one number
    - Counting occurrences of a character in a string
  + **Example #1** – does email address contain @ character

**char** email[SIZE] = "test@uccs.edu";

If this returns null

**if** (**strchr** (email, '@')) { go into “else” part

// Email contains @ now make sure there is a period and domain

**printf** ("So far, a valid email address");

**printf** ("%s\n", **strchr**(email, '@'));

}

**else** {

**printf** ("Invalid email address");

}

**Displays**

So far, a valid email address

@uccs.edu

* + **Example #2** – how many spaces are in a line of text.

**char** buffer[SIZE]; // Buffer to hold line of text

**int** numSpaces = 0; // Counter for number of spaces

**char** \*bufferPtr; // Pointer to move through string

// Ask user to enter text

// Use fgets stops reading when encounters newline, end-of-file indicator, or

// max # characters read

**puts** ("Enter a sentence");

**fgets** (buffer, SIZE, stdin);

// Set pointer to 1st space in buffer

// Searching through entire line of text and count the number of spaces

bufferPtr = **strchr** (buffer, ' ');

**while** (bufferPtr != NULL) {

// Count the space

// Move pointer to next character

numSpaces++;

bufferPtr++;

// Search for next space

bufferPtr = **strchr** (bufferPtr, ' ');

} // while

**printf** ("Text entered was - %s\nIt contains %d spaces\n", buffer, numSpaces);

**Displays**

Enter a sentence

It is too beautiful today to study so lets go for a hike.

Text entered was - It is too beautiful today to study so lets go for a hike.

It contains 12 spaces

* **strrchr**
  + Searches for ONE character – finds last occurrence (reverse order – that’s why the extra “r”)
* **strpbrk**
  + Stands for – String pointer break
  + Returns a pointer to the 1st occurrence in s1 of any character in s2
  + **strtok** uses **strpbrk** to find the delimiter characters
  + **Example:** strpbrk (str1, str2) returns “ll0 J”

str1 =

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| ‘H’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ | ‘’ | ‘J’ | ‘\0’ |

Str2 =

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| ‘l’ | ‘j’ | ‘\0’ |

* **strstr**
  + Example use:
    - Finding substring
    - Counting the occurrences of a substring
* **strtok**
  + Useful to extract **tokens** (a sequence of characters separated by delimiters) from a string
  + The original string is modified so make a copy if original string is needed later
  + Example uses:
    - Extracting the different pieces from a string:
      * birth data (mm/dd/yyyy)
      * SSN (xxx-xx-xxxx)
      * name (first name, last name)
      * words from a string
    - Microsoft word tells you how many words are in a document
  + More secure version in C11
  + **Example** – find the month, date and year in string representing a birthdate

**char** birthdate[];

**puts** ("Enter you birth date as mm/dd/yyyy");

**scanf** ("%s", birthdate);

1st call takes the string to

**char** \*tokenPtr = **strtok**(birthdate, "/"); tokenize and the delimiter.

**printf** ("%s%s\n", "Month = ", tokenPtr); ‘\0’ placed where delimiter

was in the INPUT string!

tokenPtr = **strtok**(NULL, "/");

**printf** ("%s%s\n", "Day = ", tokenPtr); Next calls take NULL and

delimiter. NULL tells it

tokenPtr = **strtok**(NULL, "/"); to continue tokenizing from

**printf** ("%s%s\n", "Year = ", tokenPtr); location saved in last call

**Displays**

Enter you birth date as mm/dd/yyyy

02/04/1945

Month = 02

Day = 04

Year = 1945

* + Notes on strtok
    - strtok modifies the original string so be careful!!!
    - If you said the following (instead an array)

**char** \*birthdate = "02/04/1945";

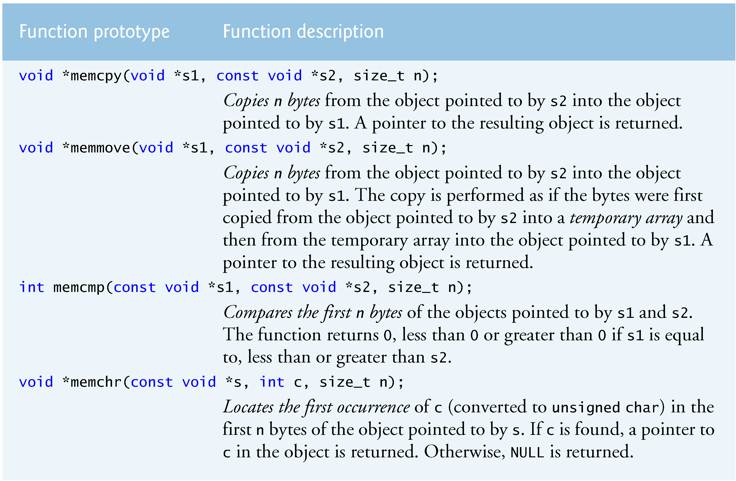
the first call to strtok will crash. Why? (because birthdate is pointing to read-only

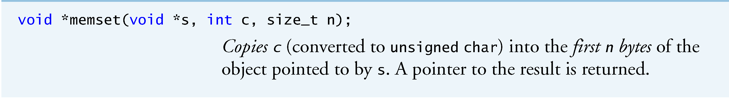
memory and strtok then tries to modify read-only memory and boom!)

**Memory Functions of the String-Handling Library**

**<string.h> - String-handling library – memory-processing functions**

* Functions to manipulate, compare and search blocks of memory
* Blocks of memory are treated as character arrays
* String memory functions are in Figure 8.27





* **memcpy**
  + Useful when you need to copy blocks of memory that don’t overlap
  + If the source and destination overlap in memory
    - Result is **undefined**
    - Possible when have two pointers into same array
    - If overlap is a possibility, use **memmove** instead
  + Memory overlap example

char letters[6] = “abcde”;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0028FD41 | 0028FD42 | 0028FD43 | 0028FD44 | 0028FD45 | 0028FD46 |
| ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘\0’ |

Memcpy takes 3 arguments

* + - Pointer to destination block of memory
    - Pointer to source block of memory
    - Number of bytes to be copied

Let’s say,

* Destination is 0028FD43 (‘c’)
* Source is 0028FD41 (‘a’)
* Number of bytes is 3?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0028FD41 | 0028FD42 | 0028FD43 | 0028FD44 | 0028FD45 | 0028FD46 |
| ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘\0’ |

Source Destination

**memcpy** (destination, source, 3);

‘a’ would be copied into 0028FD43

‘b’ would be copied into 0028FD44

‘a’ would be copied into 0028FD45 (might lose ‘c’ when we do the first copy!)

What we want:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0028FD41 | 0028FD42 | 0028FD43 | 0028FD44 | 0028FD45 | 0028FD46 |
| ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘c’ | ‘\0’ |

What we ***might*** get:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0028FD41 | 0028FD42 | 0028FD43 | 0028FD44 | 0028FD45 | 0028FD46 |
| ‘a’ | ‘b’ | ‘a’ | ‘b’ | **‘a’** | ‘\0’ |

* + memcpy vs strcpy
    - memcpy
      * Used to copy any memory location with any data type – (void \*)
      * Copies an exact number of bytes
    - strcpy
      * Used to copy null terminated strings
      * Copies source string including the ‘\0’ character into destination
      * Same issue with overlap as memcpy
      * Copies variable-length null terminated strings
  + More secure version in C11
* **memmove**
  + Same as **memcpy** but works when source and destination overlap (uses a temp array)
  + More secure version in C11
* **memset**
  + Useful for zeroing memory quickly
  + Example uses:
    - Set an array’s element to a specific value without using a loop
* **Example #1 – no difference between memcpy and strcpy for strings**

**char** src[] = "Difference between memcpy and strcpy";

**char** dest[SIZE];

// Now copy the string using memcpy

**memset** (dest, '\0', **sizeof** dest);

**printf** ("Before memcpy dest = %s\n", dest);

**printf** (" strlen(dest) = %d\n", **strlen**(dest));

**printf** (" sizeof dest = %d\n", **sizeof** dest);

**memcpy** (dest, src, SIZE);

**printf** ("After memcpy dest = %s\n", dest);

**printf** (" strlen(dest) = %d\n", **strlen**(dest));

**printf** (" sizeof dest = %d\n", **sizeof** dest);

**puts** ("");

**int** destlength = **strlen**(dest);

**if** (dest[destLength] == '\0') {

**printf** ("dest[%d] is NULL terminator\n", destLength);

}

**else** {

**printf** ("dest[%d] is not NULL terminator\n", destLength);

}

**puts** ("");

// Now copy the string using strcpy

**memset** (dest, '\0', **sizeof** dest);

**printf** ("Before strcpy dest = %s\n", dest);

**printf** (" strlen(dest) = %d\n", **strlen**(dest));

**printf** (" sizeof dest = %d\n", **sizeof** dest);

**strcpy** (dest, src);

**printf** ("After strcpy dest = %s\n", dest);

**printf** (" strlen(dest) = %d\n", **strlen**(dest));

**printf** (" sizeof dest = %d\n", **sizeof** dest);

**puts** ("");

**int** srcLength = **strlen**(src);

**if** (src[srcLength] == '\0') {

**printf** ("src[%d] is NULL terminator\n", srcLength);

}

**else** {

**printf** ("src[%d] is not NULL terminator\n", srcLength);

}

**puts** ("");

destlength = **strlen**(dest);

**if** (dest[destLength] == '\0') {

**printf** ("dest[%d] is NULL terminator\n", destLength);

}

**else** {

**printf** ("dest[%d] is not NULL terminator\n", destLength);

}

**puts** ("");

**Displays**

Before memcpy dest =

strlen(dest) = 0

sizeof dest = 80

After memcpy dest = Difference between memcpy and strcpy

strlen(dest) = 36

sizeof dest = 80

dest[36] is NULL terminator

Before strcpy dest =

strlen(dest) = 0

sizeof dest = 80

After strcpy dest = Difference between memcpy and strcpy

strlen(dest) = 36 Copied src into dest

sizeof dest = 80 including NULL

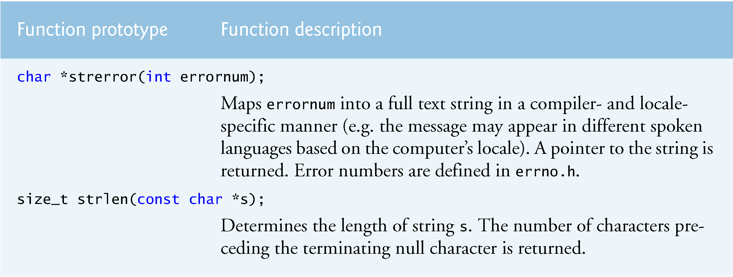
dest[36] is NULL terminator

* Notes
  + These functions all return a void pointer and accept void pointers
    - This allows the functions to manipulate any data type
  + Void pointers cannot be dereferenced so each function needs a size passed to it
    - Function needs to know how many bytes to manipulate
  + These functions DO NOT check for terminating ‘\0’
    - The blocks of memory might not be string

**Other Functions of the String-Handling Library**

**<string.h> - String-handling library – other functions**

* String search functions are in Figure 8.33



* **strerror**
  + Useful to turn error codes into human readable strings.
* **strlen**
  + Useful to find number of characters in a string
  + Example: determining of a password is 8 to 20 characters long
  + strlen and sizeof are quite different so don’t confuse
  + strlen vs sizeof
    - strlen
      * Returns number of characters in a string, not amount of memory
      * Does not count ‘\0’!
    - sizeof
      * Returns how many bytes of memory are allocated for a variable
  + Example

**char** buffer[80] = "String showing difference between strlen and sizeof";

**printf** ("strlen(buffer) = %d\n", **strlen**(buffer));

**printf** ("sizeof(buffer) = %d\n", **sizeof**(buffer));

**Displays**

strlen(buffer) = 51

sizeof(buffer) = 80

**Secure C Programming**

**Secure Programming**

* To write code that uses techniques that can stand up to attacks
* This topic is an entire class so we won’t be focusing on this topic
* We will discuss some of the techniques

**CERT C Secure Coding Standard**

* CERT – Computer Emergency Response Team - [www.cert.org](http://www.cert.org)
* Publishes and promotes secure coding standards
* Standard for C
  + <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
* Standard for other languages:
  + <https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards>

**Secure String-Processing Functions**

* More secure versions of string-processing functions are described the C11 Standard’s optional Annex K.
* Not all compilers support Annex K!
* If available should use.
* These secure functions are available only if
  + \_\_STDC\_LIB\_EXT1\_\_ is defined by the implementation
  + User defines \_\_STDC\_WANT\_LIB\_EXT1\_\_ to integer constant 1 before including <stdio.h>

**#define** \_\_STDC\_WANT\_LIB\_EXT1\_\_ 1

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <string.h>

**#ifdef** \_\_STDC\_LIB\_EXT1\_\_

printf\_s("My complier supports C11\n");

**char** aString[20];

puts ("Enter some text to test scanf\_s");

scanf\_s("%s", aString, 20);

printf\_s("My compiler works with scanf\_s - aString = %s\n\n", aString);

**#endif**

**Reading Numeric Input Validation**

* Always good to verify data inputted by user
* That is where the functions in this chapter are useful!