# Module 8: Strings

Readings: CP:AMA 13

The primary goal of this section is to be able to use strings.

Module 8. Section 1: Strings are null-terminated arrays

## **Strings**

There is no built-in C string type.

By **convention** a C string is an **array of characters**, terminated by a *null character*.

```
char my_string[4] = \{'c', 'a', 't', '\0'\};
```

The *null character*, also known as a null *terminator*, is the char with a value of zero. It is often written as '\0' instead of 0 to help indicate that a null character is intended.

'\0' is equivalent to 0. This is different from '0', which is equivalent to 48 (the ASCII character for the symbol zero).

#### String initialization

The following definitions create identical 4-character arrays:

```
char a[4] = {'c', 'a', 't', '\0'};
char b[4] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};
char d[4] = { 99, 97, 116, 0};
char e[4] = "cat";
char f[4] = "cat\0";
char g[] = "cat"; // This array also stores 4 bytes, including the '\0' terminator!
```

Because they all have a null terminator, they are also strings.

#### Null termination

With null terminated strings, we do not need to pass the length to functions.

The length of a string is the number of characters that come before the first '\0'.

```
char a[4] = {'c', 'a', 't', '\0'};
char t[] = "T0\0 fu\n";
```

The length of a is 3; the length of t is 2.

erci

Write a function  $int my_strlen(const char *s)$  that calculates the length of s.

In the  ${\tt string}$  library, there is a function  ${\tt strlen}$  that does the same thing.

It is usually better to use a library function that to "roll your own".

```
#include <string.h>
int main(void) {
  assert(strlen(a) == 3);
  assert(strlen(t) == 2);
}
```

Here is another example working with a string.

cerci

Rewrite it using only pointer notation: do not use any [ or ].

```
// e_count(s) counts the # of 'e' and 'E' in string s
int e_count(const char s[]) {
  int count = 0:
  int i = 0;
  while (s[i]) { // not the null terminator
    if ((s[i] == 'e') || (s[i] == 'E')) {
     ++count:
   ++i;
  return count;
```

#### Lexicographical order

The order of char values is well-defined:

a char is a number; two char values are in order if the corresponding numbers are in order.

'A' < 'Y' is just a way of writing 65 < 89.

'a' < 'Y' is just a way of writing 97 < 89.

So plainly 'A' < 'Y'  $\Rightarrow$  true.

So plainly 'a' < 'Y'  $\Rightarrow$  false.

We say 'A' comes before 'Y'.

We say 'a' comes after 'Y'.

Suppose we have:



Discuss: what is the value of a < b?

It is clear that  $(*a < *b) \Rightarrow ('f' < 'b') \Rightarrow (102 < 98) \Rightarrow false.$ 

But here we are looking at a and b; the "value" of an array is a pointer to the first element.

It is impossible to determine if a < b is true or false.

It depends on where in memory a and b are.

If we directly compare two char \* with <, <=, >, >=, ==, or !=, we are only looking at the values of the pointers. This is rarely useful.

To compare strings we are typically interested in using a **lexicographical order**. We compare character by character until either a string is empty, or the two strings differ. The character that comes first is in the string that comes first (including '\0'!)

```
For example, consider "Frobisher" and "Frontenac":
```

- 'F' is the same; 'r' is the same; 'o' is the same.
- 'b' comes before 'n', so "Frobisher" comes before "Frontenac".

Write a function int my\_strcmp(const char \*s1, const char \*s2). It shall return:

- a negative int if s1 comes before s2,
- a positive int if s1 comes after s2,
- o if they are equal.

Just like the string library function strcmp.

#### String equality

To compare if two strings are equal, use the strcmp function and check for **zero**.

```
#include <string.h>
char a[] = "the same?";
char b[] = "the same?";
char c[] = "different";

strcmp(a, b) ⇒ 0 // there is no difference
strcmp(a, c) ⇒ 1 // a comes after c
strcmp(c, a) ⇒ -1 // c comes before a
```

Never use relational operators (==, <, <=, etc.) to compare strings. They compare the *addresses* of the strings, not their contents.

## String I/O

The printf placeholder for strings is %s.

```
char a[] = "cat";
printf("the %s in the hat\n", a);
```

printf prints all the characters before the null character.

#### We see:

the cat in the hat

#### String I/O

scanf("%ns", ...) reads in one "word" at a time, stopping when it encounters whitespace, or after it has read n bytes. n is the **maximum field width**.

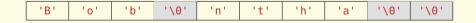
It always adds a null-terminator, so n must be 1 smaller than the length of the buffer.

```
char name[10] = {0};
while (1 == scanf("%9s", name)) {
   printf("Hello, %s!\n", name);
}
```

Give the input:
Samantha Bob [EOF]
It prints:
Hello, Samantha!
Hello, Bob!



Discuss: what is the contents of the name array?

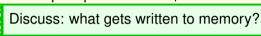


## String I/O

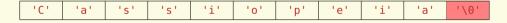
#### Suppose instead I do this:

```
int main(void) {
   char name[10] = {0};
   while (1 == scanf("%10s", name)) {
      printf("Hello, %s!\n", name);
   }
}
```

then as input I provide Cassiopeia.



It reads 10 characters, and **also adds a null-terminator**. So it attempts to write:



It overflows the buffer.

Always use a maximum field width 1 smaller than the length of the buffer.

#### Unsafe scanf

Sometimes you might find code that uses %s without a maximum width specifier. You might even find textbooks that suggest this. Like ours!

!

They are wrong!

Never use scanf("%s", buf) without a maximum width specifier!

The only reason to use scanf("%s", buf) without a maximum width specifier is if you are attempting to introduce bugs in your code. (Because you're a spy, or something.)

```
If we compile this code with AddressSanitizer turned off,
int main(void) {
  char a[6] = "ABCDE";
  char m[6] = "MNOPQ";
  while (1 == scanf("%s", a)) {
    printf("%10s - %10s\n", a, m);
  }
}
```

#### With this input:

```
auto
auto's
autopsy
automata
autograph
autocratic
```

#### We (might) get this output:

```
auto - MNOPQ
autos - MNOPQ
auto's -
autopsy - y
automata - ta
autograph - aph
autocratic - atic
```

#### This buffer overflow **corrupts** the data!

# Reading to the end of a line

If you need to read in a string that includes whitespace until a newline ( $\n$ ) is encountered, there is a function called gets (CP:AMA 13.3).

But it is deprecated! Never use it!

The documentation for gets says "Never use this function."

Never use gets.

Seriously, never use it.

If you need to read a line of text, use fgets instead.

```
strcpy(char *dest, const char *src) overwrites the contents of dest with the contents of src.
strcat(char *dest, const char *src) Copies (concatenates) src to the end of dest.
```

```
Write your own function my_strcpy(dest, src) that copies the contents of src to dest.
```

Write your own function my\_strcat(dest, src) that appends src to the end of dest.

Use pointer arithmetic; have only variables of type char \*.

```
char a[20] = "foo";
char b[20] = "bar";
char c[20] = "qux";

// Recall: strcmp \Rightarrow 0 when equal.
my_strcpy(c, a);
assert(0 == strcmp(c, "foo"));

my_strcat(a, b);
assert(0 == strcmp(a, "foobar"));
```

Any time we use these functions, we must ensure that the dest array is large enough, including space for the null terminator.

## Copying strings

Consider the following function call:

```
char s[9] = "spam";
my_strcpy(s + 4, s);
```



By hand, trace this function call until something fishy happens.

The null terminator of src is overwritten, so it will continue to fill up memory with spamspamspam....

This is undefined behaviour; anything might happen. Eventually, probably a crash.

7

The string.h version of streat **does not work** when dest and src overlap.



Rewrite my\_strcat so it (1) counts how many things to copy, then (2) copies that many. For example, above, s should contain "spamspam".

#### String literals

```
Draw a memory diagram:

int main(void) {

int a[5] = {2, 4, 6, 0, 1};
}
```

```
Draw a memory diagram:
int main(void) {
   char b[5] = "Jean";
}
```

These look virtually identical. In the stack frame:

```
b: 'J' 'e' 'a' 'n' '\0'
```

#### What about this?

Draw a memory diagram:
int main(void) {
 char \*c = "Valjean";
}

This picture is very different.

In the stack frame, we store only a char \*.

We draw a pointer as an arrow that points somewhere.

"Valjean" itself lives elsewhere. Where?

#### String literals

#### We use "Stuff in quotation marks" in two ways:

To initialize arrays of char:

```
char a[6] = "ABCDE";
```

This value is stored in the stack frame of the function where it occurs.

In expressions:

```
printf("%d + %d is %d\n", a, b, a + b);
strcpy(dst, "09 F9 11 02 9D 74 E3 5B D8 41 56 C5 63 56 88 C0");
int i = strlen("The only reason for making honey is so as I can eat it.");
scanf("%d", &i);
```

These are called **string literals**.

Where are they stored?

#### String literals

For each string literal, a null-terminated const char array is created in the **read-only data** section. This array has no name.

In the code, the occurrence of the string literal is replaced with the address of the corresponding array.

```
const char glob[] = "Rabbit";
int main(void) {
  char arr0[5] = "Pooh";
  char *arr1 = "Piglet";
  printf("arr0: %p\n", arr0);
  printf("glob: %p\n", glob);
  printf("arr1: %p\n", arr1);
```

```
arr0: 0x7ffc71b229a3
glob: 0x5633d3ee5004
arr1: 0x5633d3ee500b
```

Notice that  $my_global$  and arr1 are very close to each other, and very far from the stack.

# Poll 3: Which program works? What goes wrong?

```
int main(void) {
    char *s = "hello";
    *s = 'j';
    printf("%s\n", s);
    // Should print "jello"
}
```

```
int main(void) {
    char s[] = "hello";
    *s = 'j';
    printf("%s\n", s);
    // Should print "jello"
}
```

- Both compile, but #1 doesn't work
- Both compile, but #2 doesn't work
- #1 doesn't compile
- #2 doesn't compile
- Neither works

## Poll 4: Which program works? What goes wrong?

An array is more similar to a **constant** pointer (that cannot change what it "points at").

```
int a[6] = \{4, 8, 15, 16, 23, 42\};
Tint * const p = a;
```

In most practical expressions a and p would be equivalent. The only significant differences between them are:

- a has the same value as &a, while p and &p have different values
- The size of a is 24 bytes, while sizeof(p) is 8

#### Arrays of Strings

#### Suppose we want to store some text:

'Twas brillig, and the slithy toves Did gyre and gimble in the wabe; All mimsy were the borogoves, And the mome raths outgrabe.

## A string in the stack?

```
char cha[134] = "'Twas brillig, and the
    slithy toves\n Did gyre and gimble in
    the wabe;\nAll mimsy were the borogoves
    ,\n And the mome raths outgrabe.";
```

How can we tell how many lines there are, or where they start?

# How about a 2-D array in the stack?

```
char chaa[4][36] = {
   "'Twas brillig, and the slithy toves",
   " Did gyre and gimble in the wabe;",
   "All mimsy were the borogoves,",
   " And the mome raths outgrabe."
};
```

Internally, this is a 1-D array, padded with '\0'. I don't like that magic 36.

#### Arrays of Strings

```
An array of char * pointers?
char *chap[4] = {
    "'Twas brillig, and the slithy toves",
    " Did gyre and gimble in the wabe;",
    "All mimsy were the borogoves,",
    " And the mome raths outgrabe."
};
```

Each string is a **string literal**, and chap is an array of pointers.

Internally, this is equivalent to

```
char *j0 =
   "'Twas brillig, and the slithy toves";
char *j1 =
   "   Did gyre and gimble in the wabe;";
char *j2 =
   "All mimsy were the borogoves,";
char *j3 =
   "   And the mome raths outgrabe.";
char *chapp[4] = {j0, j1, j2, j3};
```

Done this way, we cannot mutate the individual strings, since they are literals.

#### Arrays of Strings

Until we learn how to use dynamic memory, defining an array of *mutable* strings is a little more awkward.

We must define each mutable string separately.

Aside from mutability, this array of  ${\tt char}\ *$  is just like the previous.

```
char m0[] = "'Twas brillig, and the slithy toves";
char m1[] = " Did gyre and gimble in the wabe;";
char m2[] = "All mimsy were the borogoves,";
char m3[] = " And the mome raths outgrabe.";
char *champ[4] = {m0, m1, m2, m3};
```

There are advantages and disadvantages to each.

# Summary: we saw 3 ways to store a long piece of text:

```
a string:
                     char cha[134] = "'Twas brillig, and the slithy toves\n
                                                                               Did gyre and
                        gimble in the wabe;\nAll mimsy were the borogoves,\n
                                                                                And the
                       mome raths outgrabe.";
array of char *:
                     char * chap[4] = {
                       "'Twas brillig, and the slithy toves",
                           Did gyre and gimble in the wabe; ",
                       "All mimsy were the borogoves,",
                           And the mome raths outgrabe."
array of char[36]:
                     char chaa[4][36] = {
                       "'Twas brillig, and the slithy toves",
                           Did gyre and gimble in the wabe:".
```

For each of these types, write a function that takes such a value and determines the total number of characters.

Remember: any time you work with an array that is not a string, you must know its length!

#### Goals of this Section

At the end of this section, you should be able to:

- define and initialize strings
- explain and demonstrate the use of the null termination convention for strings
- explain string literals and the difference between defining a string array and a string pointer
- Understand lexicographical order
- use I/O with strings and explain the consequences of buffer overruns
- use string.h library functions (when provided with documentation)