Week 11: Strings in C

CIT-593, Spring 2022

Sarah Santos and Le Pan

March 30, 2022

Penn Engineering

What are strings in C?

A string is an array of characters (`char`) with a null terminator.

What are strings in C?

A string is an array of characters (char) with a null terminator.


```
char not_string[5] = {'S','a','r','a','h'}
```

(no null terminator)

What are strings in C?

A string is an array of characters (char) with a null terminator.


```
char not_string[5] = {'S','a','r','a','h'}
```

(no null terminator)

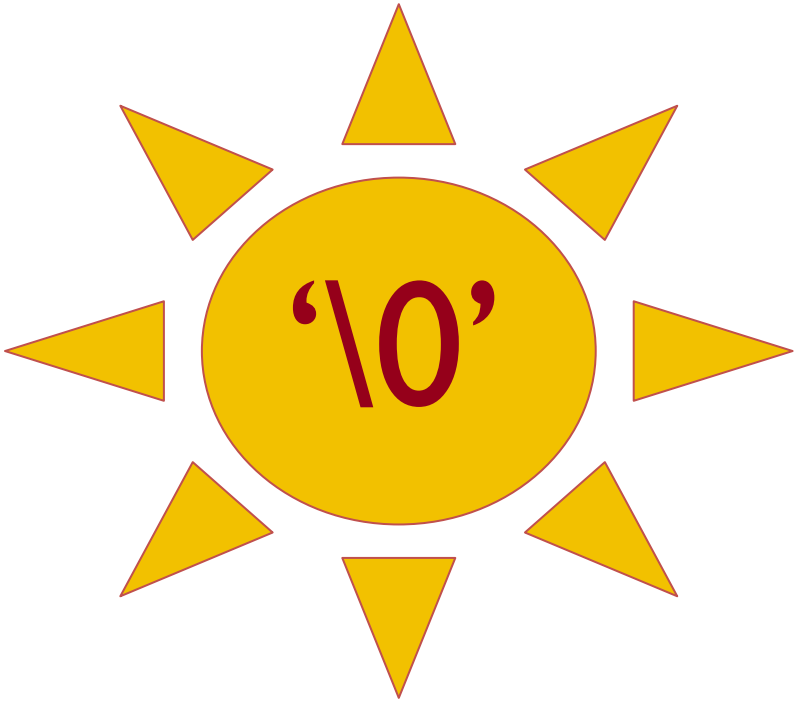
```
char not_string[6] = {'S', 'a','r','a','h','\0'}
```

null terminator - simply a 0 in memory that signifies end of a string

The null terminating char

The hero all strings deserve. Preventing segfaults since 1972.

Not all heroes wear capes. This one just wears a little backslash symbol.



If you have a string without a null terminating character, you are going to have a ~~bad time.~~ segfault or read/write error.

Adapted from slides by Katie Pizziketti/Ben Barba!

Different examples of strings

The following declarations are the same:

```
char my_string[6] = {'H','e','l','l','o','\0'};
```

Different examples of strings

The following declarations are the same:

```
char my_string[6] = {'H','e','l','l','o','\0'};
char my_string[6] = "Hello"; ← same (compiler adds null terminator for you)
```

Different examples of strings

The following declarations are the same:

```
char my_string[6] = {'H','e','l','l','o','\0'};
char my_string[6] = "Hello"; ← same (compiler adds null terminator for you)
```

We also don't have to include the size in brackets if we immediately initialize it during declaration:

```
char my_string[] = {'H','e','l','l','o','\0'};
```

Different examples of strings

The following declarations are the same:

```
char my_string[6] = {'H','e','l','l','o','\0'};
char my_string[6] = "Hello"; ← same (compiler adds null terminator for you)
```

We also don't have to include the size in brackets if we immediately initialize it during declaration:

```
char my_string[] = {'H','e','l','l','o','\0'};
char my_string[] = "Hello"; ← same (compiler adds null terminator for you)
```

What is a "string literal"?

- A string literal is a string enclosed in double-quotes
  - e.g., `"hey"`

What is a "string literal"?

- A string literal is a string enclosed in double-quotes
  - e.g., `"hey"`
- Character array initialized with string literal:
  - `char my_string[] = "Sarah";`

What is a "string literal"?

- A string literal is a string enclosed in double-quotes
  - e.g., `"hey"`
- Character array initialized with string literal:
  - `char my_string[] = "Sarah";`
- Pointer to a string literal:
  - `char* strLit = "I'm literally a string literal."`
  - Declared with `char*` (pointer to global memory will be on the stack)
  - The string literal is stored *in global/static memory (**not the stack**).*
  - *READ-ONLY*

You cannot edit pointers to string literals!

You cannot edit a pointer to a string literal

```
char* strLit = "You literally cannot change me.";
strLit[0] = 'y';   // NOT ALLOWED. Literals are read-only
```

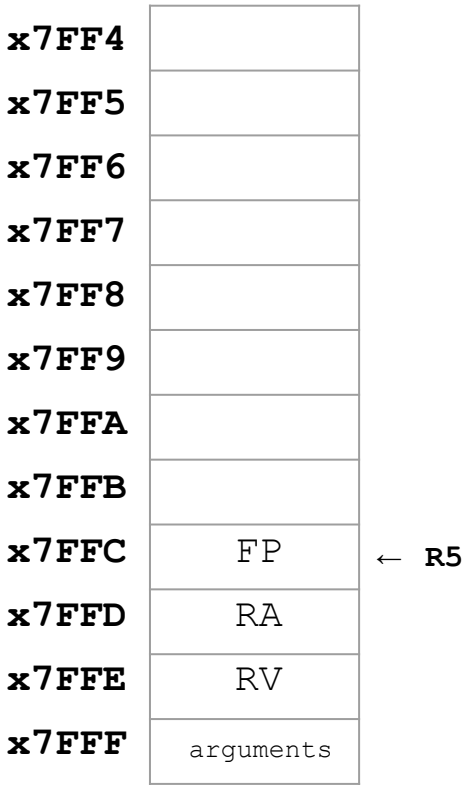
Editing a character array is completely fine

```
char[] myStr = "I'm not scared of change.";
myStr[0] = 'i';  // this is legal :)
```

Representation in Memory

How is this string represented in memory?

```
char my_string[6] = "Hello";
```

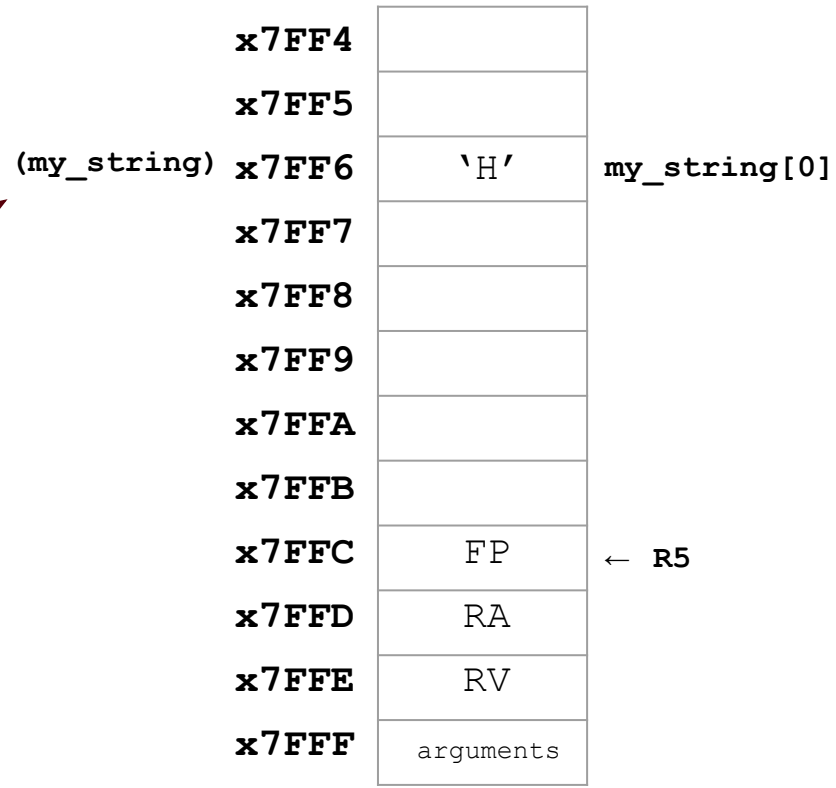


Representation in Memory

How is this string represented in memory?

```
char my_string[6] = "Hello";
```

- The name of the string becomes a label for the starting address

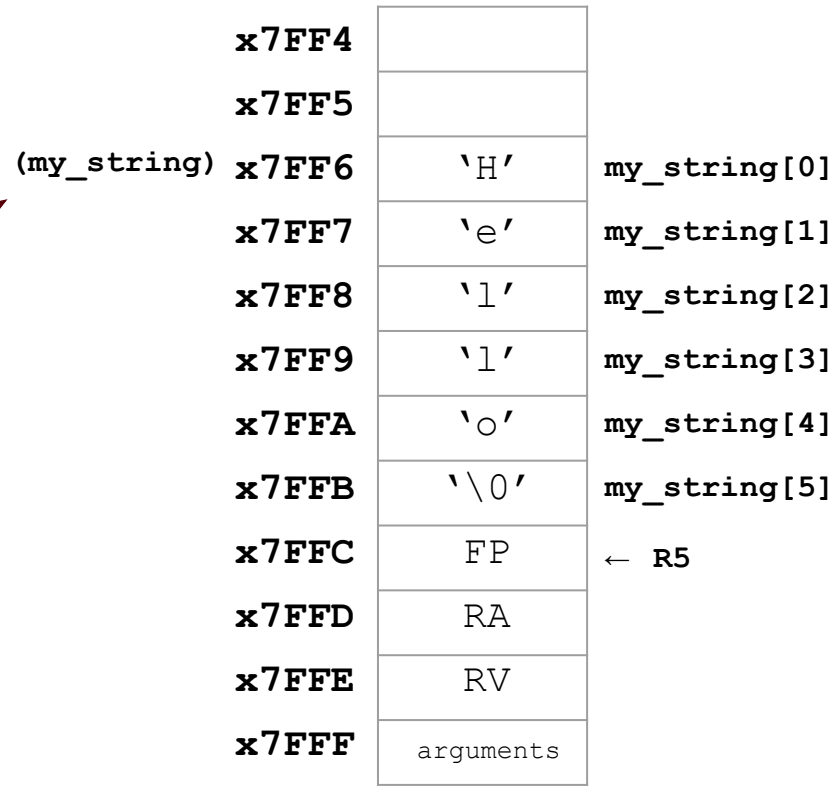


Representation in Memory

How is this string represented in memory?

```
char my_string[6] = "Hello";
```

- The name of the string becomes a label for the starting address



Pointer syntax

```
char myString[] = "Hello";
char* stringPntr = NULL;
```

Pointer syntax

```
char myString[] = “Hello”;
char* stringPntr = NULL; //these are the same!
char *stringPntr = NULL; //these are the same!
stringPntr = myString;
```

Pointer syntax

```
char myString[] = “Hello”;
char* stringPntr = NULL; //these are the same!
char *stringPntr = NULL; //these are the same!
stringPntr = myString;

char** doublePntr = &stringPntr;
//a double pointer is a pointer to a pointer.
```

Pointer syntax

```
char myString[] = “Hello”;
char* stringPntr = NULL; //these are the same!
char *stringPntr = NULL; //these are the same!
stringPntr = myString;

char** doublePntr = &stringPntr;
//a double pointer is a pointer to a pointer.


*doublePntr is equivalent to stringPntr which is equivalent to myString
**doublePntr is equivalent to *stringPntr which is equivalent to myString[0]
```

Pointer syntax

```
char myString[] = "Hello"
char* stringPntr = NULL; //these are the same!
char *stringPntr = NULL; //these are the same!
stringPntr = myString;

char** doublePntr = &stringPntr;
//a double pointer is a pointer to a pointer.
```

Pointers need to have a type.

- Otherwise, the computer will not know how to interpret the value returned when dereferencing the pointer.
- Also needed for pointer arithmetic

A void pointer (void*) needs to be cast to a type before you can dereference it.

```
int num = *(int*)aVoidPtr
```

```
*doublePntr is equivalent to stringPntr which is equivalent to myString
**doublePntr is equivalent to *stringPntr which is equivalent to myString[0]
```

Pointers vs Arrays

An array is a label for a memory address.

Pointer:

-

Array:

-

Pointer to array:

Array:

Pointers vs Arrays

An array is a label for a memory address.

Pointer:

- Can be dereferenced with *

Array:

- Can't be dereferenced, but elements can be directly accessed by their index

Pointer to array:	Array:
*pntr;	array; OR array[0];

Pointers vs Arrays

An array is a label for a memory address.

Pointer:

- Can be dereferenced with *
- Can use pointer arithmetic, such as incrementing the pointer, which will set the pointer contents to address + 1.

Array:

- Can't be dereferenced, but elements can be directly accessed by their index
- Can access elements in array using brackets: array[0]

Pointer to array:	Array:
*pntr;	array; OR array[0];
*(pntr +1);	array[1];
*(pntr + 2)	array[2];

Pointers vs Arrays

An array is a label for a memory address.

Pointer:

- Can be dereferenced with *
- Can use pointer arithmetic, such as incrementing the pointer, which will set the pointer contents to address + 1.
- Can change to point to different elements in an array

Array:

- Can't be dereferenced, but elements can be directly accessed by their index
- Can access elements in array using brackets: array[0]
- Can't be incremented to access a different element

Pointer to array:	Array:
*pntr;	array; OR array[0];
*(pntr +1);	array[1];
*(pntr + 2)	array[2];

Adapted from slides by Katie Pizziketti/Ben Barba!

Pointers vs Arrays

An array is a label for a memory address.

Pointer:

- Can be dereferenced with *
- Can use pointer arithmetic, such as incrementing the pointer, which will set the pointer contents to address + 1.
- Can change to point to different elements in an array
- **Can be returned from a function**

Array:

- Can't be dereferenced, but elements can be directly accessed by their index
- Can access elements in array using brackets: array[0]
- Can't be incremented to access a different element
- **Can't be returned from a function, but CAN be passed as an argument to a function**

Pointer to array:	Array:
*pntr;	array; OR array[0];
*(pntr +1);	array[1];
*(pntr + 2)	array[2];

Penn Engineering

Adapted from slides by Katie Pizziketti/Ben Barba!

Debugging practice

String Functions (time-permitting)

Walk through function signatures and expected functionality:

- https://www.tutorialspoint.com/c_standard_library/c_function_strlen.htm
- https://www.tutorialspoint.com/c_standard_library/c_function_strcpy.htm
- https://www.tutorialspoint.com/c_standard_library/c_function_strchr.htm
- https://www.tutorialspoint.com/c_standard_library/c_function_strcat.htm
- https://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm