
Week 4

XMUT-NWEN 241 - 2025 T2

Systems Programming

String

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

String

Content

- Strings
- String Literal
- String Variable

What is String in C?

- C language **does not support strings** as a basic data type
- A C string is just an array that contains ASCII characters terminated by the **null character '\0'**
- A C string is stored in an array of chars

H	i		2	4	1		!	!	\0
---	---	--	---	---	---	--	---	---	----

"Hi 241 !!"

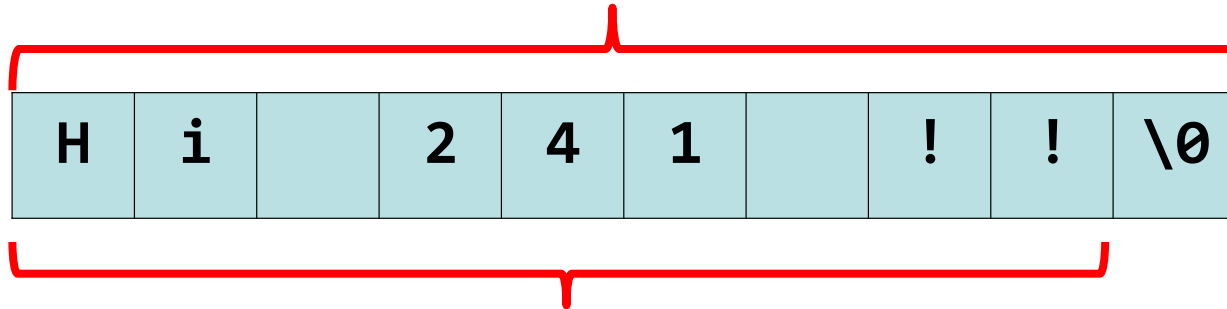
H	i		2	4	1		!		
---	---	--	---	---	---	--	---	--	--

Not a valid string

String Length

- Number of bytes/characters **excluding** the null character

Entire string occupies 10 bytes /characters



String length = 9 bytes /characters

- `strlen()` function in `<string.h>` returns the string length

String Literal vs String Variable

- In C, we distinguish between **string literals** and **string variables**
- A **string literal** refers to the string constant value which is stored in the read-only memory area of the program
- A **string variable** refers to a string that is stored in an array which can be modified

String Literal (1)

- Enclosed in double quotes (") an can contain character literals (plain and escape characters)
- Can broken up into multiple lines (each line ends with \) or separated by whitespaces

```
"Hello, world"
```

```
"Hello" ", " "world"
```

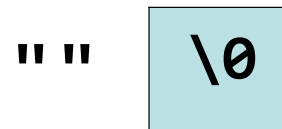
```
"Hello, \  
world"
```

String Literal (2)

- String literals may contain as few as **one** or **even zero** characters
- Do not confuse a single-character string literal, e.g. "A" with a character constant, 'A'
 - The former is actually two characters, because of the null-terminator stored at the end



- An **empty string**, "", consists of only the null-terminator, and is considered to have a string length of zero, because the null-terminator does not count when determining string lengths



String Literal (3)

- String literals are passed to functions as *pointers* to a stored string. For example, given the statement:

```
printf("Hello world!\n");
```

- The string literal "Hello world!\n" will be stored somewhere in memory, and the address will be passed to `printf()`
 - The first argument to `printf()` is actually defined as a `char *`
- We will revisit this when we talk about pointers

Operations on String Literals

- String literals may be subscripted

```
printf("%c\n", "Hello"[2]);    /* will print 'l' */
```

- Attempting to modify a string literal results in **undefined behaviour**, and may cause problems in different ways depending on the compiler, *e.g.*

```
"Hello"[2] = 'e';
```

Symbolic String Constants

- Similar to integer and float symbolic constants, symbolic string constants can be declared using **const** qualifier or **#define** pre-processor

```
const char *MSG = "Hello, world";  
const char *MSG_A = "Hello, \  
world";  
const char *MSG_B = "Hello" ", " "world";
```

```
#define MSG    "Hello, world"  
#define MSG_A "Hello, \  
world"  
#define MSG_B "Hello" ", " "world"
```

String Variables

- **String variables** are stored as arrays of chars, terminated by the **null character**
- A string variable can be initialized in 2 ways using the methods discussed in previous lecture:

```
char str[10];  
str[0] = 'H';  
str[1] = 'e';  
str[2] = 'l';  
str[3] = 'l';  
str[4] = 'o';  
str[5] = ' ';  
str[6] = '!';  
str[7] = '\\0';
```

```
char str[10] = {'H', 'e', 'l',  
               'l', 'o', ' ', '!', '\\0'};
```

Efficient String Variable Initialization

- Another way to initialize a char array to hold a string variable: **assign a string literal to the array during declaration**

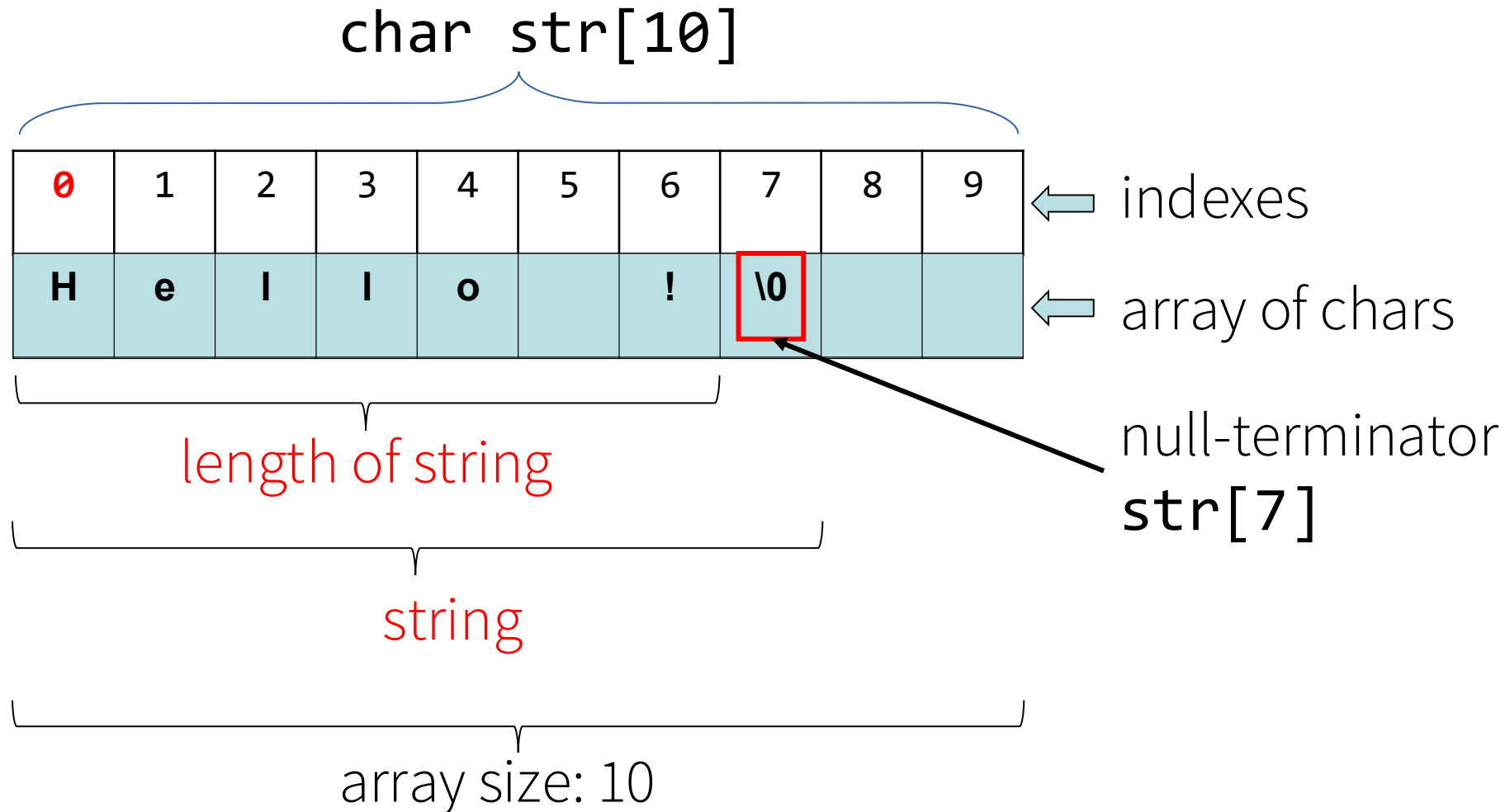
```
char str[10] = "Hello !";
```

```
char str[] = "Hello !";
```

What's the difference between the two?

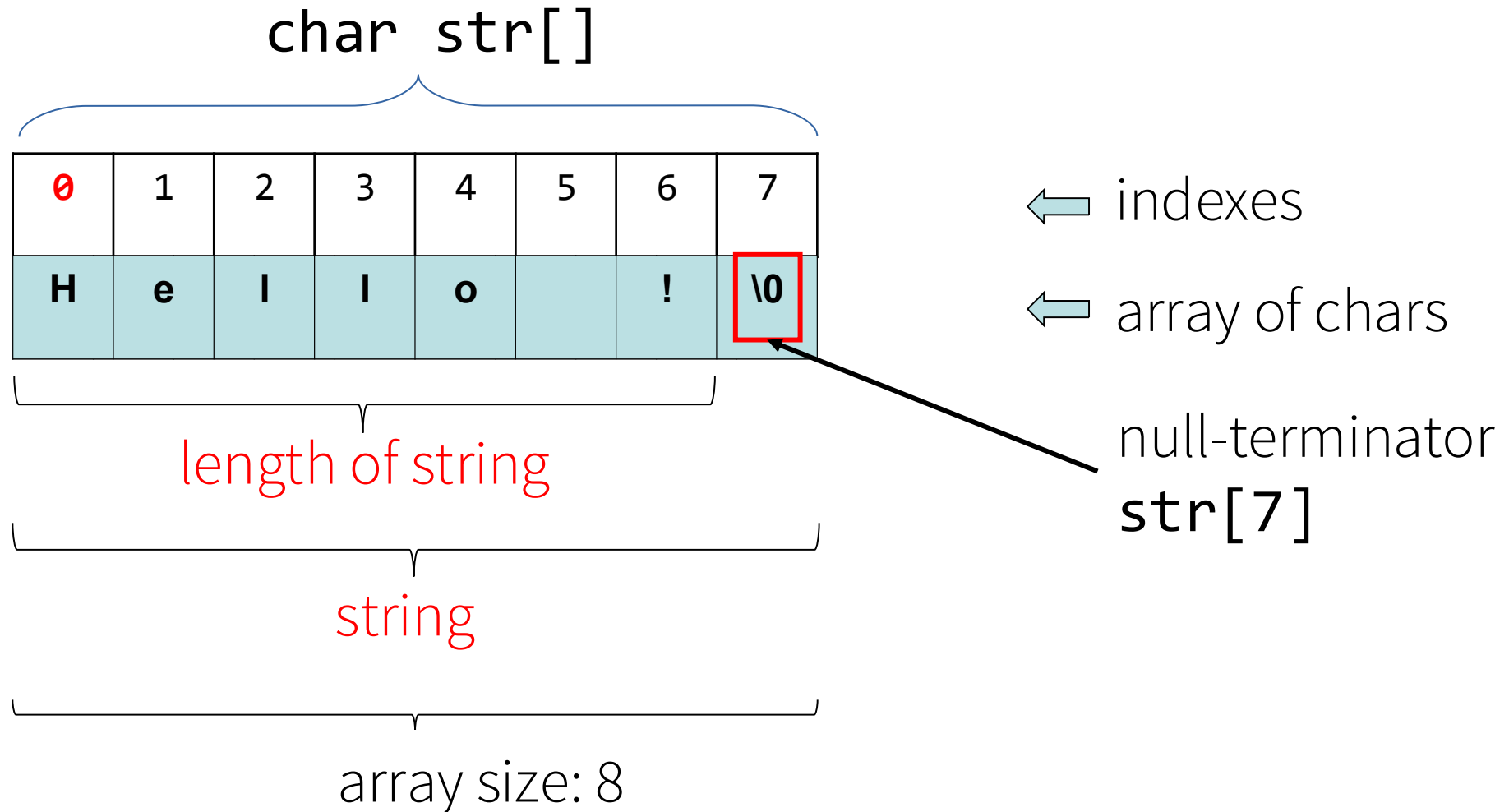
Arrays and C Strings

```
char str[10] = "Hello !";
```



Arrays and C Strings

```
char str[] = "Hello !";
```



Assigning a string after array declaration

```
char str[10];  
...  
str = "Hello !";
```

Illegal! Use strcpy() function

```
char str[10];  
...  
strcpy(str, "Hello !");
```

Strings

- `long int strlen(const char* source);`
 - Calculates the length of a given string, up to the first null character.
- `char* strcpy(char* destination, const char* source);`
 - Copies the source string to the destination character array.
- `int strcmp (const char* str1, const char* str2);`
 - Compares two strings and returns 0 if both strings are identical.
- `char *strcat(char *dest, const char *src);`
 - Concatenates two strings and stores the result in the first argument.

Null Terminator '\0'

- A string is an array of characters that ends with the first occurrence of ' \0 '
- What comes **after** the end of the string doesn't matter, since the string has ended

```
char str[] = "One\0Two";  
printf("%s\n", str);
```

- The program will print only the string "One"
 - The ' \0 ' character terminates the string
 - What comes after, does not matter
- The array will contain 8 elements

Displaying Strings: `printf()`

- Strings can be displayed on the screen using `printf()`

```
printf("%s\n", str);
```

- The **precision** (`'%.N'`) parameter limits the length of longer strings to at most N

```
printf("%.5s\n", "ABCDEFGH");  
// only "ABCDE" will be displayed
```

- The **width** (`'%N'`) parameter can be used to print a short string in a long space, at least N characters

```
printf("%5s\n", "abc" );  
// prints "  abc". Note the leading  
// two spaces at the beginning.
```

Displaying Strings: puts()

- The puts() function writes the string out to standard output and automatically appends a newline character at the end

```
char str[] = "This is an ";  
printf("%s", str);  
puts("example string.");  
printf("See??\n");
```

- The output will be:

```
This is an example string.  
See??
```

Reading in strings – scanf()

- The standard format specifier for reading strings with `scanf()` is `%s` that the `'&'` is **not required** in the case of strings, since the string is a memory address itself
- `scanf()` appends a `'\0'` to the end of the character string stored
- `scanf()` does skip over any leading whitespace characters in order to find the first non-whitespace character

Reading in strings – scanf()

- The width field can be used to limit the maximum number of characters to read from the input
- You should use one character less as input than the size of the array used for holding the result

```
char str[6];  
printf("Hi\n");  
scanf("%5s", str);  
    // If you enter "HelloBello123xyz", only the  
    // first 5 characters will be read and a  
    // concluding '\0' will be put at the end  
  
printf("%s\n", str);
```

Reading in strings – scanf()

- `scanf()` reads in a string of characters, only up to the first non-whitespace character
 - it stops reading when it encounters a space, tab, or newline character
- C supports a format specification known as the edit set conversion code `%[ . . . ]`
 - it can be used to read a line containing a variety of characters, including white spaces

```
char str[20];  
printf("Enter a string:\n");  
scanf("%[^\n]", str);  
printf("%s\n", str);
```

Reading in strings – scanf()

- Always use the width field to limit the maximum number of characters to read with "%s" and "[%...]" in all production quality code!
 - No exceptions!

Reading in strings – gets()

- `gets()` is used to scan a line of text from a standard input device, until a newline character input
- The string may include white space characters
- The newline character won't be included as part of the string
- `'\0'` is always appended to the end of the string of stored characters

Reading in strings – gets()

```
char str[15];  
printf("Enter your name: \n");  
gets(str);  
printf("%s\n", str);
```

- gets() has no provision for limiting the number of characters to read
 - This can lead to overflow problems!

Reading strings character by character

- Read in character by character is useful when
 - you don't know how long the string might be,
 - or if you want to consider other stopping conditions besides spaces and newlines
 - e.g. stop on periods, or when two successive slashes, //, are encountered.
- The `scanf()` format specifier for reading individual characters is `%c`
- If a width greater than 1 is given (`%2c`), then multiple characters are read, and stored in successive positions in a char array

sscanf() and sprintf() functions

- `scanf()` and `printf()` functions are used to read from and write to the standard input/output
- `sscanf()` and `sprintf()` are used for the same goal but instead of the standard input/output, they use strings
- One of their main advantage is when you need to prepare a string for later use

The `<ctype.h>` header

- `<ctype.h>` declares a set of functions to classify and transform individual chars
 - `#include <ctype.h>` is required to use any of these functions
 - https://www.tutorialspoint.com/c_standard_library/ctype_h.htm documents the library

The `<ctype.h>` header

- Some of the more commonly used functions:
 - **isupper()** – checks if a character is an uppercase letter
 - A value different from zero is returned if the character is an uppercase alphabetic letter, zero otherwise
 - **islower()** – checks if a character is a lowercase letter
 - A value different from zero is returned if the character is a lowercase alphabetic letter, zero otherwise
 - **toupper()** – converts a character to its uppercase equivalent if the character is an lowercase letter and has an uppercase equivalent
 - If no such conversion is possible, the returned value is unchanged
 - **tolower()** – converts a character to its lowercase equivalent if the character is an uppercase letter and has a lowercase equivalent
 - If no such conversion is possible, the returned value is unchanged

The `<string.h>` header

- `<string.h>` defines several functions to manipulate null-byte terminated arrays of chars
 - `#include <string.h>` is required to use any of these functions
 - https://www.tutorialspoint.com/c_standard_library/string_h.htm documents the library

The `<stdlib.h>` header

- `stdlib.h` defines several functions, including searching, sorting and converting
 - `#include <stdlib.h>` is required to use any of these functions
 - https://www.tutorialspoint.com/c_standard_library/stdlib_h.htm documents the library
- Some of the more commonly used functions:
 - `atoi()`, `atof()`, `atol()`, `atoll()` – parses a string of numeric characters into a number of type `int`, `double`, `long int`, or `long long int`, respectively

Next Lecture

- Pointers