NWEN 241
Strings

School of Engineering and Computer Science
Victoria University of Wellington

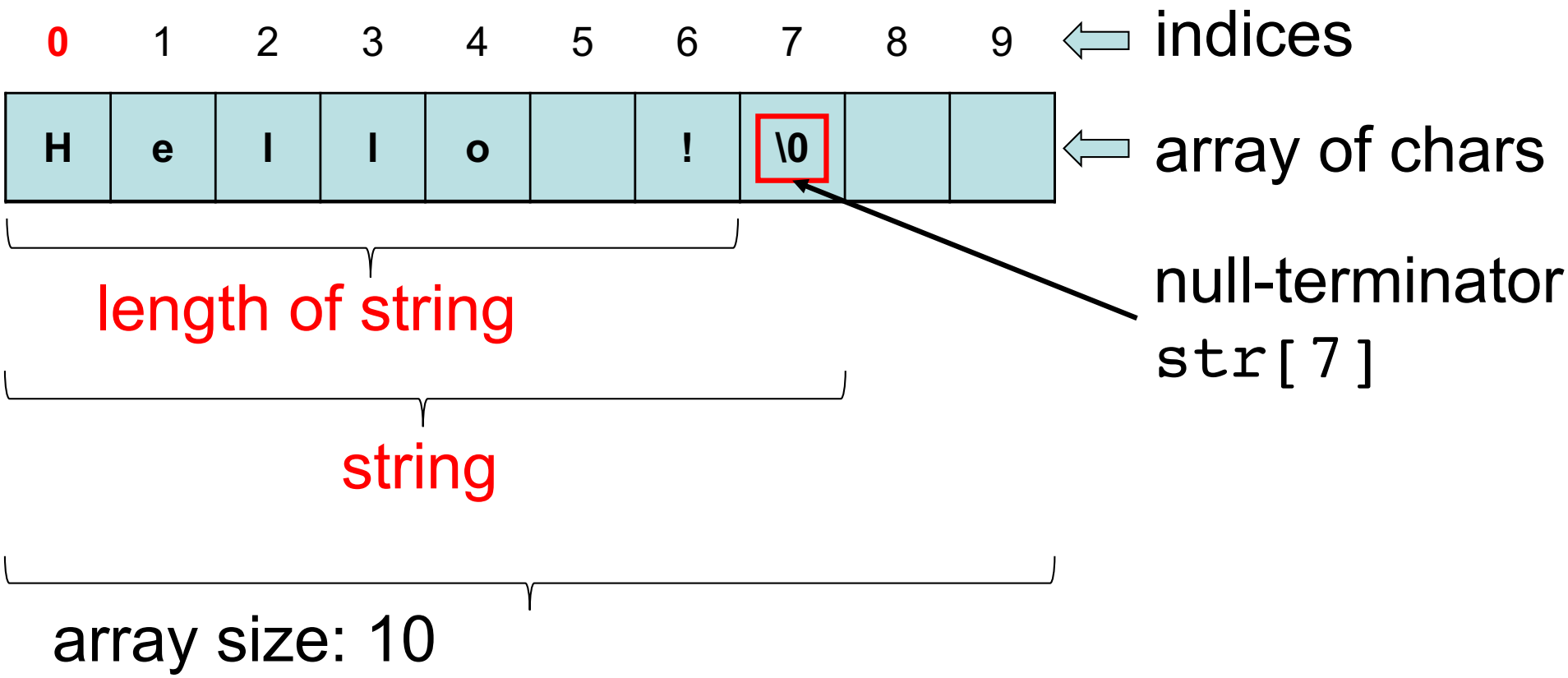
Strings in General

- **C** language **does not support strings** as a data type
- A **string** is a sequence of characters that is treated as a single data item and terminated by a **null character** also known as the **null-terminator**, **null byte** or just **'\0'**
  - In C language a string is actually a one-dimensional array of characters

---

- In C we distinguish between **String Literals** and **String Variables**
  - If we (declare and) initialize a string using a pointer, we speak about a **String Literal**
  - If we (declare and) initialize a string using an array of characters, we speak about a **String Variable**

Strings in General (cont.)



String Literals

- **String Literal**, also known as a **string constant** or **constant string**, is a string of characters enclosed in **double quotes**, e.g.:

  "NWEN241-2017 is driving me crazy!"

- String Literals are stored in C as **an array of chars**, terminated by **'\0'**
  - '\0' is not the same as the '0' character, the integer 0, the double 0.0, or the pointer NULL
- 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 literals may contain any valid characters, including escape sequences such as \n, \t, etc.

Passing String Literals to Functions

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

```
printf("Hello World!");
```

- The string literal "Hello World!" 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 *

Continuing a String Literal

- If a string literal needs to be continued across multiple lines, there are three options:

```
printf("This will
        print over three
      lines, (and will include extra tabs or spaces)");
```

```
printf("This will \
        print over a single \
 line, (but will still include extra tabs or spaces)");
```

```
printf("This will "
       "print over a single "
       "line, (without extra tabs or spaces)");
```

Operations on String Literals

- Character pointers may hold the address of string literals
- String literals may be subscripted

```
char *str = "Hello";          // Character pointers

printf("%c\n", str[0]);       // will print 'H'

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

printf("%c\n", str[5]);       // will print "nothing".

          //Actually this will print the null-terminator
```

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

```
str[0] = 'S'; //NEVER do this when you declared and
              //initialized your string using pointers!
```

String Variables

- **String Variables** are typically stored as arrays of chars, terminated by a **null-terminator**
- String variables can be initialized **either**
  - with **individual characters (you have to supply the '\0' explicitly)**
  - or **more commonly** and **easily** with **string literals**

```
as: array size; ss: string size;
char str[ ] = {'H','e','l','l','o','\0'}; // as=6, ss=5
char str[6] = {'H','e','l','l','o','\0'}; // as=6, ss=5
```

```
char str[5] = "Hello"; //5 chars, sacrifices null-term.
char str[ ] = "Hello";  // as=6, ss=5
char str[20] = "Hello"; // ss=5, 15 null-term.
```

```
char str[ ]; str = "Hello";              // illegal method
char str[8]; str = "Hello World!";       // illegal method
char str[3] = "Hello";                   // illegal method
```

Character Arrays vs Character Pointers

```
char s6[ ] = "hello";
char *s7 =   "hello";
```

- s6 is a fixed constant address, determined by the compiler
- s6 allocates space for exactly 6 bytes
- the contents of s6, can be changed, e.g. s6[0] = 'J';

- s7 is a pointer variable, that can be changed to point elsewhere
- s7 allocates space for 10 (typically) - 6 for the characters plus another 4 for the pointer variable.
- the contents of s7 should not be changed

The null-terminator

- Any array of characters that ends with a '\0' is a string
- 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
  - The string "One\0Two”, and
  - another null-terminator, which was put at the end by the compiler

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

```
printf( "%.5s\n", "abcdefg" );

    // only ”abcde” will be displayed
```

- The width ('%N') parameter can be used to print a short string in a long space

```
printf( "%5s\n", "abc" );    // prints "  abc". Note

          // the leading two spaced 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"
printf("%s\n", str); //only the first 5 characters
                     //be read and a concluding '\0'
                     //will be put at the end
```

Reading in Strings – scanf() (cont.)

- 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);
```

- 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
- The gets() function will be terminated by a newline character
- The newline character won't be included as part of the string
- **The string may include white space characters**
- '\0' is always appended to the end of the string of stored characters

```
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**
  - **Here you must use the '&' symbol!!!**
- 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
  - Examples in course exercise

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
  - http://www.cplusplus.com/reference/cctype/ documents the library
- 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
  - http://www.cplusplus.com/reference/cstring/ documents the library
- Some of the more commonly used functions:
  - **strcpy()** – returns the length of the string, not counting the '\0'
  - **strcat()** – concatenates (appends) source to the end of destination
  - **strlen()** – returns length of the string, not counting the '\0'
  - **strcmp()** – compares strings str1 and str2, up until the first encountered null-term
    - Returns zero if the two strings are equal
    - Returns a positive value (1?) if the first encountered difference has a larger value in str1 than str2
    - Returns a negative value (-1?) if the first encountered difference has a smaller value in str1 than str2

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
  - http://www.cplusplus.com/reference/cstdlib/ 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