Strings in C

ITSC 2181: Introduction to Computer Systems
UNC Charlotte
College of Computing and Informatics

COLLEGE OF COMPUTING AND INFORMATICS

String Literals

- A ***string literal*** is a sequence of characters enclosed within double quotes:

```
"When you come to a fork in the road, take it."
```

- String literals may contain escape sequences.
- Character escapes often appear in **`printf`** and **`scanf`** format strings.
- For example, each `\n` character in the string

```
"Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"
```

causes the cursor to advance to the next line:

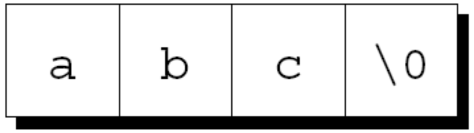
```
Candy
Is dandy
But liquor
Is quicker.
  --Ogden Nash
```

How String Literals Are Stored

- When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string.
- This memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string.
- The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

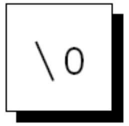
How String Literals Are Stored

- The string literal `"abc"` is stored as an array of four characters:



a	b	c	\0

- The string `""` is stored as a single null character:



\0

(see **`array_iteration.c`** in *Code samples and Demonstrations* in *Canvas*) for an example of code to access individual characters in an array.

How String Literals Are Stored

- Since a string literal is stored as an array, the compiler treats it as a pointer of type **`char *`**
- Both **`printf`** and **`scanf`** expect a value of type **`char *`** as their first argument.
- The following call of **`printf`** passes the address of **`"abc"`** (a pointer to where the letter `a` is stored in memory):

```
printf("abc");
```

COLLEGE OF COMPUTING AND INFORMATICS

Operations on String Literals

- We can use a string literal wherever C allows a **`char *`** pointer:

```
char *p;

p = "abc";
```

- This assignment makes **`p`** point to the first character of the string.

Operations on String Literals

- String literals can be subscripted (like arrays):

```
char ch;

ch = "abc"[1];
```

The new value of `ch` will be the letter `b`.

- A function that converts a number between `0` and `15` into the equivalent hex digit:

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

Operations on String Literals

- Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';    /*** WRONG ***/
```

- A program that tries to change a string literal may crash or behave erratically.

String Literals versus Character Constants

- A string literal containing a single character isn't the same as a character constant.

  `"a"` is represented by a *pointer*.

  `'a'` is represented by an *integer*.

- A legal call of **`printf`**:

```
printf("\n");
```

- An illegal call:

```
printf('\n');    /*** WRONG ***/
```

String Variables

- If a string variable needs to hold **80** characters, it must be declared to have length **81**:

```
#define STR_LEN 80
…
char str[STR_LEN+1];
```

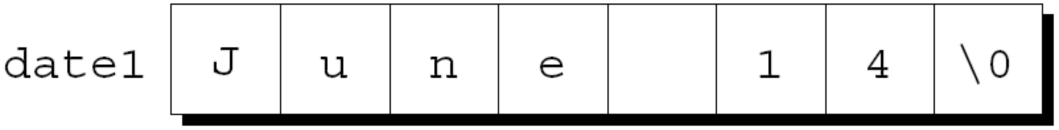
- Adding 1 to the desired length allows room for the *null* character at the end of the string.
- Defining a macro that represents the string’s length and adding 1 is a common practice.

Initializing a String Variable

- A string variable can be initialized at the same time it is declared:

```
char date1[8] = "June 14";
```

- The compiler will automatically add a *null* character so that `date1` can be used as a string:



date1	J	u	n	e		1	4	\0

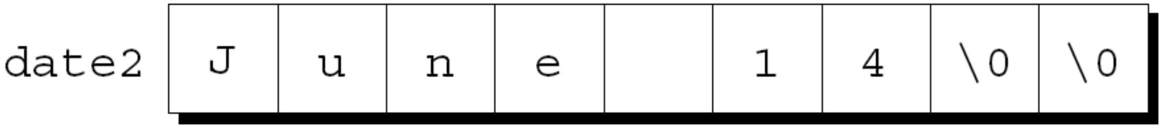
- `"June 14"` is not a string literal in this context. Instead, C views it as an abbreviation for an array initializer.

Initializing a String Variable

If the initializer is too short to fill the string variable, the compiler adds extra null characters:

```
char date2[9] = "June 14";
```

Appearance of **`date2`**:



date2	J	u	n	e		1	4	\0	\0

Reading and Writing Strings

- Writing a string is easy using either `printf` or `puts`.
- Reading a string is a bit harder, because the input may be longer than the string variable into which it is being stored.
- To read a string in a single step, we can use either `scanf` or `gets`.
- As an alternative, we can read strings one character at a time.

A Special Case of Array Declaration

- Declaring a pointer to a **string literal** also allocates the memory containing that string
- Example:

```
char *str = “This is a string”;
```

is equivalent to…

```
char str[] = “This is a string”;
```

Except! **first version is read only** (cannot modify string contents in your program)

(see `string_test.c` in *Code samples and Demonstrations* in *Canvas*).

Doesn’t work with other types or arrays, ex.:

```
int *nums = {0, 1, 2, 3, 4}; /* won’t work!   */
char *str = {‘T,‘h’,‘i’,‘s’}; /* no NULL char */
```

The C Standard Library

Manipulating Strings and Characters

Standard Library: `<ctype.h>`

- The C Standard Library has many functions for **checking** whether a character is a digit, is upper case, …
  - `isalnum(c)`, `isalpha(c)`, `isspace(c)`,…
- Also, functions for **converting** to upper case and converting to lower case
  - `toupper(c)`, `tolower(c)`, …
- The input argument is an `int` and the return value is an `int`
  - Works fine with unsigned chars or 7-bit character types
  - Need to cast to `unsigned char` for safety

`<ctype.h>` (cont’d)

Checking:

`isalnum (c)`	c is a letter or a digit
`isalpha(c)`	c is a letter
`isdigit (c)`	c is a decimal digit
`islower (c)`	c is a lower-case letter
`isspace (c)`	c is white space `(\f \n \r \t \v`)
`isupper (c)`	c is an upper-case letter

Converting:

`tolower (c)`	convert c to lower case
`toupper (c)`	convert c to upper case

Only a partial list. For full list see library or https://en.wikibooks.org/wiki/C_Programming/ctype.h/Function_reference.

COLLEGE OF COMPUTING AND INFORMATICS

`scanf()` and `printf()` for Strings

`sscanf(s, "…", …)` scans a **string** (instead of stdin) for expected input

`sprintf(s, "…", …)` outputs to a **string** (instead of stdout) the specified output

(see `sscanf_example.c` in *Code samples and Demonstrations* in *Canvas*)

sscanf and **sprintf** Example

```
char input[80] = "55 cars";
char output[80] = "";
int total_cars = 0;

sscanf(input, "%d", &total_cars);

sprintf(output, "Total Cars: %d\n", total_cars);
printf(output);
```

(see **sscanf_example.c** in *Code samples and Demonstrations* in *Canvas*)

COLLEGE OF COMPUTING AND INFORMATICS

Using the C String Library

- The C library provides a rich set of functions for performing operations on strings.
- Programs that need string operations should contain the following line:

```
#include <string.h>
```

- In subsequent examples, assume that `str1` and `str2` are character arrays used as strings.

The `strcpy` (String Copy) Function

- Prototype for the `strcpy` function:

```
char *strcpy(char *s1, const char *s2);
```

- `strcpy` copies the string `s2` into the string `s1`.
  - To be precise, we should say "`strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`."
- `strcpy` returns `s1` (a pointer to the destination string).

(see `remind.c` in *Code samples and Demonstrations* in *Canvas*)

The **strcpy** (String Copy) Function

- A call of **strcpy** that stores the string "**abcd**" in **str2**:

```
strcpy(str2, "abcd");
/* str2 now contains "abcd" */
```

- A call that copies the contents of str2 into **str1**:

```
strcpy(str1, str2);
/* str1 now contains "abcd" */
```

(see **remind.c** in *Code samples and Demonstrations* in *Canvas*)

COLLEGE OF COMPUTING AND INFORMATICS

The `strcpy` (String Copy) Function

- In the call `strcpy(str1, str2)`, `strcpy` has no way to check that the `str2` string will fit in the array pointed to by `str1`.
- If it doesn't, undefined behavior occurs.

(see `remind.c` in *Code samples and Demonstrations* in *Canvas*)

The `strncpy` (**Safe** String Copy) Function

- Calling the `strncpy` function is a safer, albeit slower, way to copy a string.
- `strncpy` has a third argument that limits the number of characters that will be copied.
- A call of `strncpy` that copies `str2` into `str1`:

```
strncpy(str1, str2, sizeof(str1));
```

The strncpy (**Safe** String Copy) Function

- strncpy will leave str1 without a terminating null character if the length of str2 is greater than or equal to the size of the str1 array.
- A safer way to use strncpy:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

- The second statement guarantees that str1 is always null-terminated.

The **strlen** (String Length) Function

- Prototype for the **strlen** function:

```
size_t strlen(const char *s);
```

- **size_t** is a **typedef** name that represents one of C's unsigned integer types.

The **strlen** (String Length) Function

- **strlen** returns the length of a string **s**, not including the null character.
- Examples:

```
int len;

len = strlen("abc");  /* len is now 3 */
len = strlen("");     /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);   /* len is now 3 */
```

Standard Library: `<string.h>`

- (`<strings.h>` on some machines)
- Lots of string processing functions for
  - copying one string to another
  - comparing two strings
  - determining the length of a string
  - concatenating two strings
  - finding a substring in another string
  - …
- Function headers at end of slides
- A good reference site is http://www.cplusplus.com/

(see `string_comparison_example.c` in *Code samples and Demonstrations* in *Canvas*)

`<stdlib.h>` String Functions

- **`double atof( char s[] )`** converts a string to a **`double`**, ignoring leading white space
- **`int atoi( char s[] )`** converts a string to an **`int`**, ignoring leading white space
  - These don’t return information about errors
- (instead of…)

```
int num = 0;
while (isspace(c = getchar()))
    ;
while (isdigit(c)) {
    num = num * 10 + c - '0';
    c = getchar();
}
```

- Could also use
  - **`strtol`**
  - **`strtod/f`**

References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at: https://diveintosystems.org/book/

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.

- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.