# Strings

Dr Deepayan Bhowmik Dr Andrea Bracciali

# In this lecture

- Strings and they operators

    as "simple" as that

- The C String lib (to the rescue)
- Arrays of strings.

# Introduction

- It's important to separate between *string constants* (or *literals,* as they're called in the C standard) and string *variables*.
- Strings are arrays of characters in which a special character—the null character—marks the end (mandatory!).
- The C library provides a collection of functions for working with strings.

# 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:

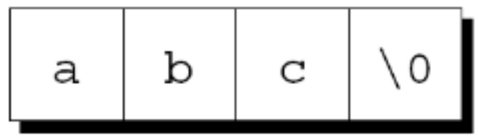
```
Candy
Is dandy
But liquor
Is quicker.
  --Ogden Nash
```

# 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:




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

# 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 – a char in this case:

  ```
  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 behaviour:

```
char *p = "abc";

*p = “xyz”;                    /*** 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 is not 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

- Any one-dimensional array of characters can be used to store a string.
- A string must be terminated by a null character.
- Difficulties with this approach:
  - It can be hard to tell whether an array of characters is being used as a string.
  - String-handling functions must be careful to deal properly with the null character.

# 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];
```

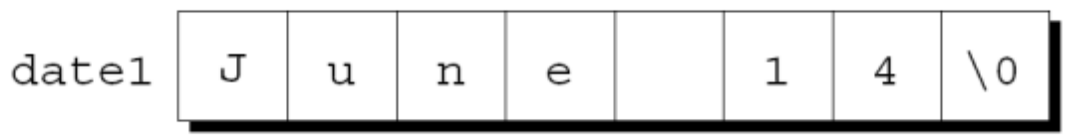
- Adding 1 to the desired length allows room for the null character at the end of the string.
- Defining a macro that represents 80 (the number of characters you are dealing with) and then adding 1 separately is a common practice.

# Initializing a String Variable

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

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

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




- `"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`:

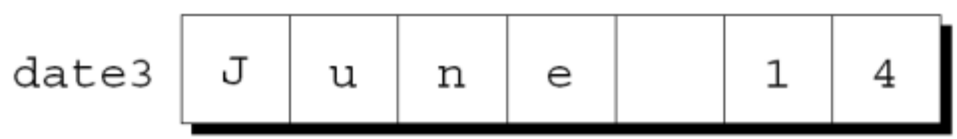
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| date2 | J | u | n | e | | 1 | 4 | \0 | \0 |

# Initializing a String Variable

- An initializer for a string variable can't be longer than the variable, but it can be the same length:

```
char date3[7] = "June 14";
```

- There is no room for the null character, so the compiler makes no attempt to store one:



# Initializing a String Variable

- The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

- The compiler sets aside eight characters for `date4`, enough to store the characters in `"June 14"` plus a null character.
- Omitting the length of a string variable is especially useful if the initializer is long, since computing the length by hand is error-prone.

# Character Arrays versus Character Pointers

- The declaration

  `char date[] = "June 14";`

  declares `date` to be an *array*,

- The similar-looking

  `char *date = "June 14";`

  declares `date` to be a *pointer.*

- Thanks to the close relationship between arrays and pointers, either version can be used as a string.

# Character Arrays versus Character Pointers

- However, there are significant differences between the two versions of `date`.

  - In the array version, the characters stored in date can be modified. In the pointer version, date points to a string literal that should not be modified.

  - In the array version, date is an array name. In the pointer version, date is a variable that can point to other strings.

# Character Arrays versus Character Pointers

- The declaration

  ```
  char *p;
  ```

  does not allocate space for a string.
- Before we can use `p` as a string, it must point to an array of characters.

- One possibility is to make `p` point to a string variable:

  ```
  char str[STR_LEN+1], *p;
  p = str;
  ```

- Another possibility is to make `p` point to a dynamically allocated string (using `malloc`, i.e. directly allocating/addressing memory from the program – future work!) .

# Character Arrays versus Character Pointers

- Using an uninitialized pointer variable as a string is a serious error.
- An attempt at building the string "abc":

```
char *p;

p[0] = 'a';     /*** WRONG ***/
p[1] = 'b';     /*** WRONG ***/
p[2] = 'c';     /*** WRONG ***/
p[3] = '\0';    /*** WRONG ***/
```

- Since `p` hasn't been properly initialized, no allocation of memory (!), this causes undefined behaviour.

# Accessing the Characters in a String

- Since strings are stored as arrays, we can use subscripting to access the characters in a string.
- To process every character in a string `s`, we can set up a loop that increments a counter `i` and selects characters via the expression `s[i]`.

# Accessing the Characters in a String

- A function that counts the number of spaces in a string:

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ') {
      count++;
     }
  return count;
}
```

# Accessing the Characters in a String

- A version that uses pointer arithmetic instead of array subscripting :

```
int count_spaces(const char *s)
{
  int count = 0;
  for (; *s != '\0'; s++)
    if (*s == ' ')
      count++;
  return count;
}
```

# Using the C String Library

- Some programming languages provide operators that can **copy** strings, **compare** strings, **concatenate** strings, select substrings, and the like.

- C's operators, in contrast, are essentially useless for working with strings.

- Strings are treated as arrays in C, so they're restricted in the same ways as arrays.

- In particular, they can't be copied or compared using operators.

# Using the C String Library

- Direct attempts to copy or compare strings will fail.
- Copying a string into a character array using the = operator is not possible:

```
char str1[10], str2[10];
…
str1 = "abc";   /*** WRONG ***/
str2 = str1;    /*** WRONG ***/
```

  Using an array name as the left operand of = is illegal.

- *Initializing* a character array using = is legal, though:

```
char str1[10] = "abc";
```

TRY IT !

# Using the C String Library

- Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result:

```
if (str1 == str2) …    /*** WRONG ***/
```

- This statement compares `str1` and `str2` as *pointers*. Like in Java!
- Since `str1` and `str2` have different addresses, the expression `str1 == str2` will have value 0.

TRY IT !

# Using the C String Library (to the rescue!)

- 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).

# 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" */
```

# The `strncpy` (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 behaviour occurs.

# The **strncpy** (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** (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:

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

- `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 */
```

# The **strcat** (String Concatenation) Function

- Prototype for the `strcat` function:

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

- `strcat` appends the contents of the string `s2` to the end of the string `s1`.
- It returns `s1` (a pointer to the resulting string).
- `strcat` examples:

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);
  /* str1 now contains "abcdef" */

strcpy(str1, "abc");
strcat(str1, "def");
  /* str1 now contains "abcdef" */
```

# The `strcat` (String Concatenation) Function

- As with `strcpy`, the value returned by `strcat` is normally discarded.

- The following example shows how the return value might be used:

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
  /* str1 now contains "abcdefghi";
     str2 contains "defghi" */
```

# The `strcat` (String Concatenation) Function

- `strcat(str1, str2)` causes undefined behaviour if the `str1` array isn't long enough to accommodate the characters from `str2`.

- Example:

```
char str1[5] = "abc";
strcat(str1, "def");    /*** WRONG ***/
```

- `str1` is limited to five characters, causing `strcat` to write past the end of the array.

# The `strcmp` (String Comparison) Function

- Prototype for the `strcmp` function:

```
int strcmp(const char *s1, const char *s2);
```

- `strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2`.

You may have seen something similar in Java.

# The `strcmp` (String Comparison) Function

- `strcmp` considers `s1` to be less than `s2` if either one of the following conditions is satisfied:
  - The first *i* characters of `s1` and `s2` match, but the (*i*+1)st character of `s1` is less than the (*i*+1)st character of `s2`.
  - All characters of `s1` match `s2`, but `s1` is shorter than `s2`.

Essentially, a lexicographic order.

# The `strcmp` (String Comparison) Function

- Testing whether `str1` is less than `str2`:

```
if (strcmp(str1, str2) < 0)    /* is str1 < str2? */
   …
```

- Testing whether `str1` is less than or equal to `str2`:

```
if (strcmp(str1, str2) <= 0) /* is str1 <= str2? */
   …
```

- By choosing the proper operator (<, <=, >, >=, ==, !=), we can test any possible relationship between `str1` and `str2`.

# The `strcmp` (String Comparison) Function

- As it compares two strings, `strcmp` looks at the numerical codes for the characters in the strings.
- Some knowledge of the underlying character set is helpful to predict what `strcmp` will do.

- Important properties of ASCII:
  - 0–9, A–Z, and a–z have consecutive codes.
  - Spaces are less than all printing characters.
  - Digits are less than letters.
  - All upper-case letters are less than all lower-case letters.

# Arrays of Strings

- There is more than one way to store an array of strings.
- One option is to use a two-dimensional array of characters, with one string per row:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- The number of rows in the array can be omitted, but we must specify the number of columns.

# Arrays of Strings

- Unfortunately, the `planets` array contains a fair bit of wasted space (extra null characters):

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | M | e | r | c | u | r | y | \0 |
| 1 | V | e | n | u | s | \0 | \0 | \0 |
| 2 | E | a | r | t | h | \0 | \0 | \0 |
| 3 | M | a | r | s | \0 | \0 | \0 | \0 |
| 4 | J | u | p | i | t | e | r | \0 |
| 5 | S | a | t | u | r | n | \0 | \0 |
| 6 | U | r | a | n | u | s | \0 | \0 |
| 7 | N | e | p | t | u | n | e | \0 |
| 8 | P | l | u | t | o | \0 | \0 | \0 |

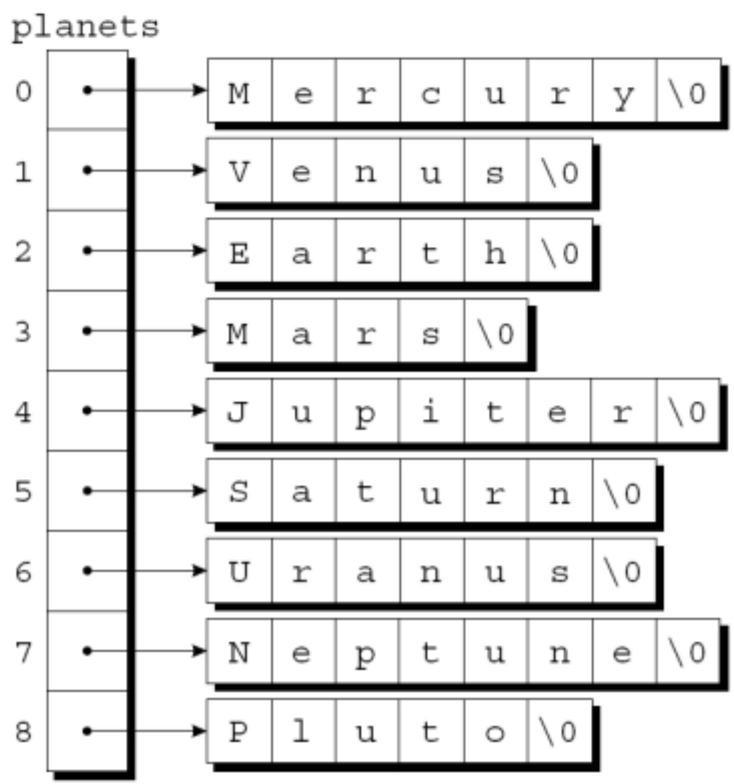
# Arrays of Strings

- Most collections of strings will have a mixture of long strings and short strings.
- What we need is a **ragged array**, whose rows can have different lengths.
- We can simulate a ragged array in C by creating an array whose elements are *pointers* to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

# Arrays of Strings

- This small change has a dramatic effect on how `planets` is stored:



# Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- Accessing a character in a planet name is done in the same way as accessing an element of a two-dimensional array.
- A loop that searches the `planets` array for strings beginning with the letter M:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```