Lecture 12: Strings

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

Agenda

- String literals/constants
- String variables
- Reading and writing strings
- Accessing characters in a string
- Arrays of strings

Readings

- Textbook Chapter 13

Overview of Strings in C

- A string is a series/sequence of characters

- This is equivalent to a `char` array

- The differences between a “regular” array of characters and a string is that a string is always terminated by a “null character” (`'\0'`) which marks the end of the string

Overview of Strings in C

- A string is a series/sequence of characters

- This is equivalent to a `char` array

- The differences between a “regular” array of characters and a string is that a string is always terminated by a “null character” (`'\0'`) which marks the end of the string

String Literals

- A string literal is a sequence of characters enclosed by double quotes. E.g., “Today is Tuesday”

- A string literal is treated as a `char` pointer and is stored as a `char` array in memory

String Literals

- A string literal is a sequence of characters enclosed by double quotes. E.g., “Today is Tuesday”

- A string literal is treated as a `char` pointer and is stored as a `char` array in memory

A string literal of length *n* is stored in *n* + *1* bytes of memory

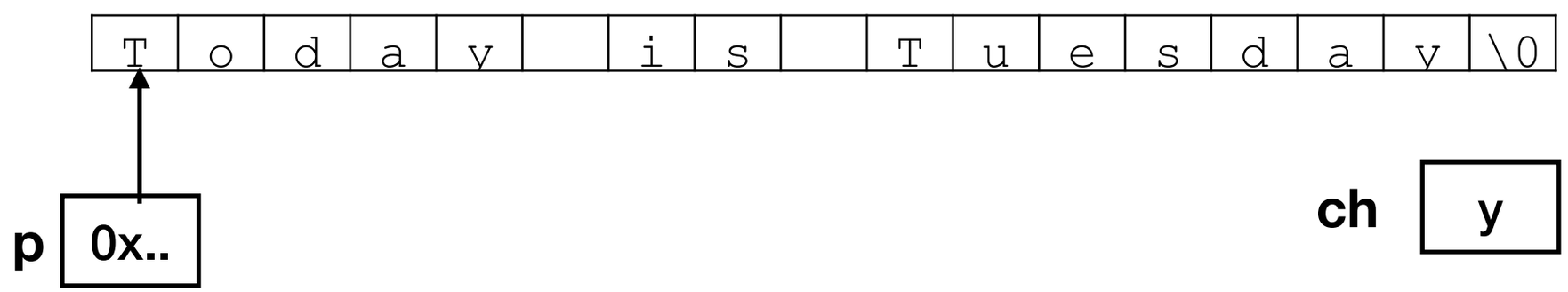
String Literals

- A string literal is a sequence of characters enclosed by double quotes. E.g., “Today is Tuesday”

- A string literal is treated as a `char` pointer and is stored as a `char` array in memory

```
char *p;
char ch;

p = "Today is Tuesday"; // p points to the 1st character in the string
ch = "Today is Tuesday"[4]; //ch = 'y'
```



A string literal of length *n* is stored in *n + 1* bytes of memory

String Literals *Cont'd*

String Literals *Cont'd*

- String literals CANNOT be modified

String Literals *Cont'd*

- String literals CANNOT be modified

```
char *p;

p = "Today is Tuesday";
*p = 'R'; // WRONG!!!!!!
```

String Literals *Cont'd*

- String literals CANNOT be modified

```
char *p;

p = "Today is Tuesday";
*p = 'R'; // WRONG!!!!!!
```

  - "a" is a string literal, stored as a an array of 2 characters, where a pointer points to the first character

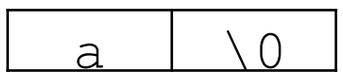
String Literals *Cont'd*

- String literals CANNOT be modified

```
char *p;

p = "Today is Tuesday";
*p = 'R'; // WRONG!!!!!!
```

- "a" is a string literal, stored as a an array of 2 characters, where a pointer points to the first character



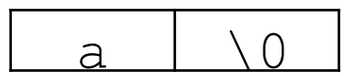
String Literals *Cont'd*

- String literals CANNOT be modified

```
char *p;

p = "Today is Tuesday";
*p = 'R'; // WRONG!!!!!!
```

- "a" is a string literal, stored as a an array of 2 characters, where a pointer points to the first character



- 'a' is a character constant that is represented as an integer (the ASCII code of the character)

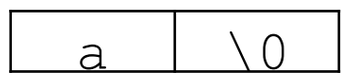
String Literals *Cont'd*

- String literals CANNOT be modified

```
char *p;

p = "Today is Tuesday";
*p = 'R'; // WRONG!!!!!!
```

- "a" is a string literal, stored as a an array of 2 characters, where a pointer points to the first character



- 'a' is a character constant that is represented as an integer (the ASCII code of the character)

- Thus, you must make sure to use a string when a string is expected and a char when a char is expected. e.g.

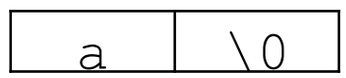
String Literals *Cont'd*

- String literals CANNOT be modified

```
char *p;

p = "Today is Tuesday";
*p = 'R'; // WRONG!!!!!!
```

- "a" is a string literal, stored as a an array of 2 characters, where a pointer points to the first character



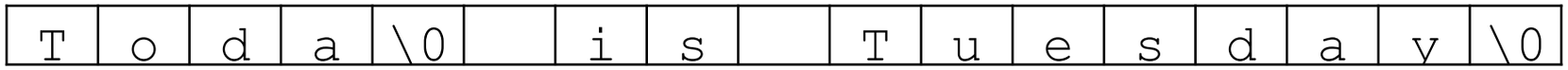
- 'a' is a character constant that is represented as an integer (the ASCII code of the character)

- Thus, you must make sure to use a string when a string is expected and a char when a char is expected. e.g.

```
atoi('5'); //WRONG!! it expects a char*
atoi("5"); //OK
```

String Variables

- There is no built-in string type in C
- Any 1-dimensional character array that has a null character represents a string
- The sequence of characters in the string are **from the first character in the array to the FIRST null character.** All characters after the first null character are ignored when processing strings.



represents “Toda”

String Variables *Cont'd*

- To store a string of length *n*, you need an array of size *n+1*

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

- A character array without a null character **cannot** be used as a string. This is very important when using functions from the string library as we will see in a bit.

String Variable Examples

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

String Variable Examples

treated as an array initializer, not a string literal

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

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

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

date2

J	u	n	e		1	4	\0	\0

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

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

date2

J	u	n	e		1	4	\0	\0

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

date3

J	u	n	e		1	4

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

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

date2

J	u	n	e		1	4	\0	\0

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

date3

J	u	n	e		1	4

no room for the null character so it is not stored.
This means that date3 cannot be used as a string

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

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

date2

J	u	n	e		1	4	\0	\0

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

date3

J	u	n	e		1	4

no room for the null character so it is not stored.
This means that date3 cannot be used as a string

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

date4

J	u	n	e		1	4	\0

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

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

date2

J	u	n	e		1	4	\0	\0

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

date3

J	u	n	e		1	4

no room for the null character so it is not stored.
This means that date3 cannot be used as a string

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

date4

J	u	n	e		1	4	\0

If no length provided, compiler will calculate the length, including enough room for the null character

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

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

date2

J	u	n	e		1	4	\0	\0

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

date3

J	u	n	e		1	4

no room for the null character so it is not stored.
This means that date3 cannot be used as a string

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

date4

J	u	n	e		1	4	\0

If no length provided, compiler will calculate the length, including enough room for the null character

```
char *date5 = "June 14";
```

J	u	n	e		1	4	\0

date5

String Variable Examples

treated as an array initializer, not a string literal

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

date1

J	u	n	e		1	4	\0

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

date2

J	u	n	e		1	4	\0	\0

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

date3

J	u	n	e		1	4

no room for the null character so it is not stored.
This means that date3 cannot be used as a string

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

date4

J	u	n	e		1	4	\0

If no length provided, compiler will calculate the length, including enough room for the null character

```
char *date5 = "June 14";
```

J	u	n	e		1	4	\0

date5

- **characters in date1, date2, date3, and date 4 can be modified.**
- **characters in date5 CANNOT be modified. It is a string literal**
- **date1, date2, date4, and date 5 can be used as strings**

Writing Strings

```
char str[] = "This is CMPUT 201!";
printf("%s", str); //prints: This is CMPUT 201!
printf("%.6s", str); //prints: This i
puts(str); //prints: This is CMPUT 201! followed by a newline
```

Reading Strings with `scanf`

```
char str[20];
scanf("%s", str);
```

Reading Strings with `scanf`

```
char str[20];
scanf("%s", str);
```

- Remember that the array name is treated as a pointer, so no need to put `&` in front of it

Reading Strings with `scanf`

```
char str[20];
scanf("%s", str);
```

- Remember that the array name is treated as a pointer, so no need to put `&` in front of it
- When `scanf` reads a string, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character. `scanf` always stores a null character at the end of the string

Reading Strings with `scanf`

```
char str[20];
scanf("%s", str);
```

- Remember that the array name is treated as a pointer, so no need to put `&` in front of it
- When `scanf` reads a string, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character. `scanf` always stores a null character at the end of the string
- This means that a string read using `scanf` will never contain white space, which means that `scanf` cannot read a full line of input (newline, tab, or space will cause scanf to stop reading)

Reading Strings with `scanf`

```
char str[20];
scanf("%s", str);
```

- Remember that the array name is treated as a pointer, so no need to put `&` in front of it
- When `scanf` reads a string, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character. `scanf` always stores a null character at the end of the string
- This means that a string read using `scanf` will never contain white space, which means that `scanf` cannot read a full line of input (newline, tab, or space will cause scanf to stop reading)
- Without specifying the number of characters to be read, `scanf` may store characters beyond the bounds of the array, causing undefined behavior

Reading Strings with `fgets`

```
char str[20];
fgets(str, 20, stdin);
```

Reading Strings with `fgets`

```
char str[20];
fgets(str, 20, stdin);
```

- fgets doesn't skip white space before starting to read a string

Reading Strings with `fgets`

```
char str[20];
fgets(str, 20, stdin);
```

- fgets doesn't skip white space before starting to read a string
- `fgets` keeps reading characters until either `n-1` characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. If a new line is read, it is stored in the array.

Reading Strings with `fgets`

```
char str[20];
fgets(str, 20, stdin);
```

- fgets doesn't skip white space before starting to read a string
- `fgets` keeps reading characters until either `n-1` characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. If a new line is read, it is stored in the array.
- `fgets` always includes the null character after the last read character

Reading Strings with `fgets`

```
char str[20];
fgets(str, 20, stdin);
```

- fgets doesn't skip white space before starting to read a string
- `fgets` keeps reading characters until either `n-1` characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. If a new line is read, it is stored in the array.
- `fgets` always includes the null character after the last read character
- On success, `fgets` returns the read string (`str` in this case). If there is an error or `EOF` is reached without reading any characters, `fgets` returns `NULL`.

Reading Strings with `fgets`

```
char str[20];
fgets(str, 20, stdin);
```

- fgets doesn't skip white space before starting to read a string
- `fgets` keeps reading characters until either `n-1` characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. If a new line is read, it is stored in the array.
- `fgets` always includes the null character after the last read character
- On success, `fgets` returns the read string (`str` in this case). If there is an error or `EOF` is reached without reading any characters, `fgets` returns `NULL`.

By using `fgets` and specifying a size that is <= size of array, you avoid writing characters beyond the length of the array. Note that there is a function called `gets` that is somewhat similar to fgets but doesn't protect against writing beyond the bounds of the array. You will get a warning if you use `gets`, which means it is not accepted in this course. In general, gets is unsafe to use.

Important Note

- To read in a string, you **must** declare a `char` array. You cannot just declare `char *str;` and read a string into it. When you just declare `char *str;` you are declaring a pointer variable. There is no memory unit it points to yet.

Accessing the Characters in a String

- Since a string is stored as a character array, we can use subscripting and pointers to access its characters.

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

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

  return count;
}
```

```
int count_spaces(const char *s){
  int count = 0;

  for (; *s ! = '\0'; s++)
    if (*s == ' ')
        count++;

  return count;
}
```

Accessing the Characters in a String

- Since a string is stored as a character array, we can use subscripting and pointers to access its characters.

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

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

  return count;
}
```

```
int count_spaces(const char *s){
  int count = 0;

  for (; *s ! = '\0'; s++)
    if (*s == ' ')
        count++;

  return count;
}
```

Note that `const` doesn't prevent the function from modifying `s`. The `const` prevents the function from modifying the contents that `s` points to. Additionally, since `s` is a copy of the pointer that is passed to the function, incrementing `s` doesn't affect the original pointer so this code is OK since in the end, it doesn't really change the value of the original array name in any way.

Reading Strings Character by Character

```
/* read a string of max length n, return the length */
int read_line(char str[], int n) {
    int ch, i = 0;
    while ((ch = getchar()) != '\n') {
        if (i < n)
            str[i++] = ch;
    }
    if (i != n)
      str[i] = '\0'; //add the null character if you have the space

    return i;
}
```

Reading Strings Character by Character

```
/* read a string of max length n, return the length */
int read_line(char str[], int n) {
    int ch, i = 0;
    while ((ch = getchar()) != '\n') {
        if (i < n)
            str[i++] = ch;
    }
    if (i != n)
      str[i] = '\0'; //add the null character if you have the space

    return i;
}
```

In this case, it is not guaranteed that you will always have a string. What can you do to guarantee that you always have a string?

The String Library

Using the C String Library

```
char str1[10], str2[10];
…
str1 = "abc"; //THIS DOES NOT WORK IN C!
str2 = str1; /*THIS ALSO DOESN'T WORK. It will NOT copy the contents of one array
               into the other*/

if (str1 == str2) /*this does NOT compare the contents of the arrays. Instead, it
   compares their addresses and since the addresses are always different,
   the result of this comparison will always be 0 */
…
```

Using the C String Library

```
char str1[10], str2[10];
…
str1 = "abc"; //THIS DOES NOT WORK IN C!
str2 = str1; /*THIS ALSO DOESN'T WORK. It will NOT copy the contents of one array
               into the other*/

if (str1 == str2) /*this does NOT compare the contents of the arrays. Instead, it
   compares their addresses and since the addresses are always different,
   the result of this comparison will always be 0 */
…
```

Since C's built-in operators do not work for strings, there is a library that provides a rich set of functions for the commonly used operations on strings.

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

strcpy & strncpy

```
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
```

- `strcpy`:
  - Copies `s2` into `s1` and returns `s1`
  - Copies up to the first '\0' (see why '\0' is important?)
    - If the string `s2` points to is longer than the one `s1` points to, undefined behavior will occur since the function will keep going till it find the null character
- `strncpy`:
  - limits the number of characters copied to `n` to make it more safe
  - However, if the length of the string that `s2` points to is greater than or equal to `n`, then the null character will not be added. The programmer needs to then ensure that the s1 is null-terminated.

strlen

`size_t strlen(const char *s);`

- returns the number of characters in s up to, but NOT including, the first null character

```
int len;
char str1[10];

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

strcat & strncat

```
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
```

- strcat:
  - Provides string concatenation functionality. It appends the contents of string s2 to the end of string s1. It returns s1 (a pointer to the resulting string)
    - If the length of the array pointed to by s1 is not long enough to hold the result of the concatenation, undefined behavior may occur.
- strncat
  - n controls the number of characters to be copied, not including the null character
  - strncat always terminates s1 with the null character

strcmp & strncmp

```
char *strcmp(const char *s1, const char *s2);
char *strncmp(const char *s1, const char *s2, size_t n);
```

- Compares s1 and s2, character by character.
  - If equal: returns 0
  - If s1 is lexicographically less than s2: returns a value < 0
  - if s1 is lexicographically greater than s2: returns a value > 0
- Internally, the numeric value of the characters is compared:
  - Characters in each of the sequences A-Z, a-z, and 0-9 have consecutive ASCII codes
  - All upper-case letters are less than lower-case letters, because of the ASCII codes
  - Digits are less than letters
  - Spaces are less than all printing characters

demo: string.c

Printing a One-Month Reminder List

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0

Day Reminder
  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused“
 24 Susan's birthday
 26 Movie - "Chinatown"
```

demo: remind.c

Array of Strings

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

	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

Array of Strings

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

	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

This wastes lots of characters, because it assumes that all strings have the same length… We need a *ragged* array: a two dimensional array whose rows can have different lengths.

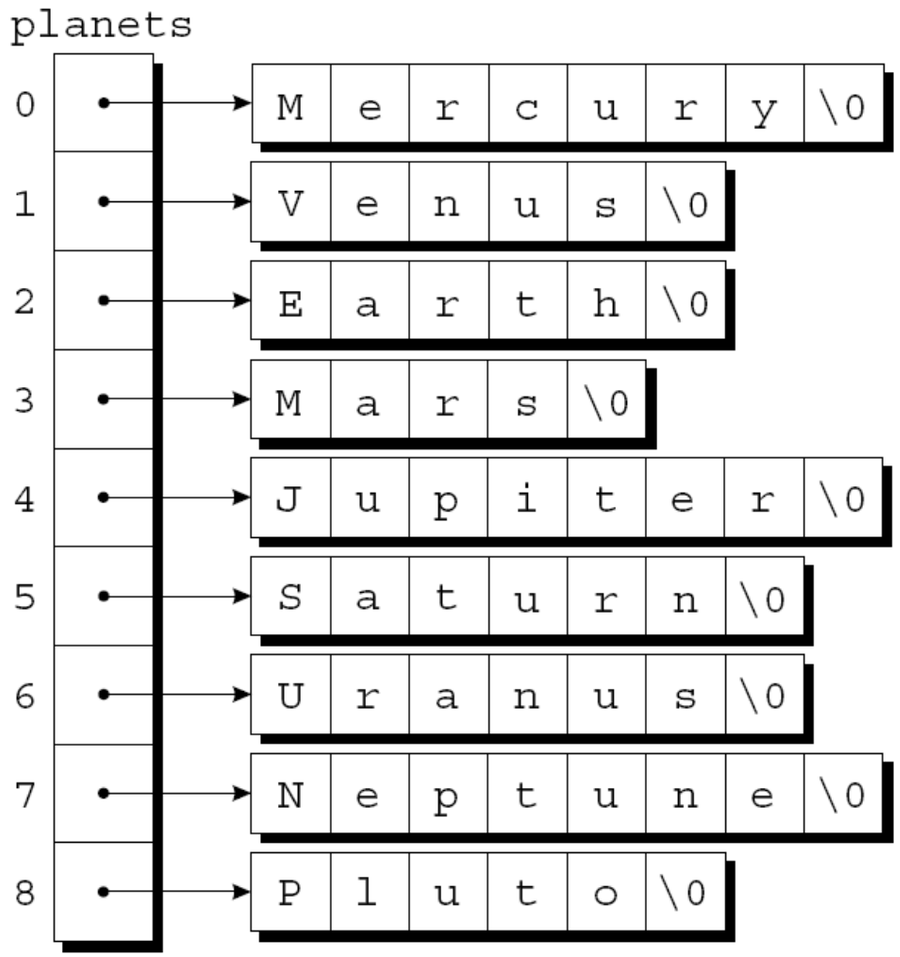
Ragged Array of Strings

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

demo: array-string.c

Ragged Array of Strings

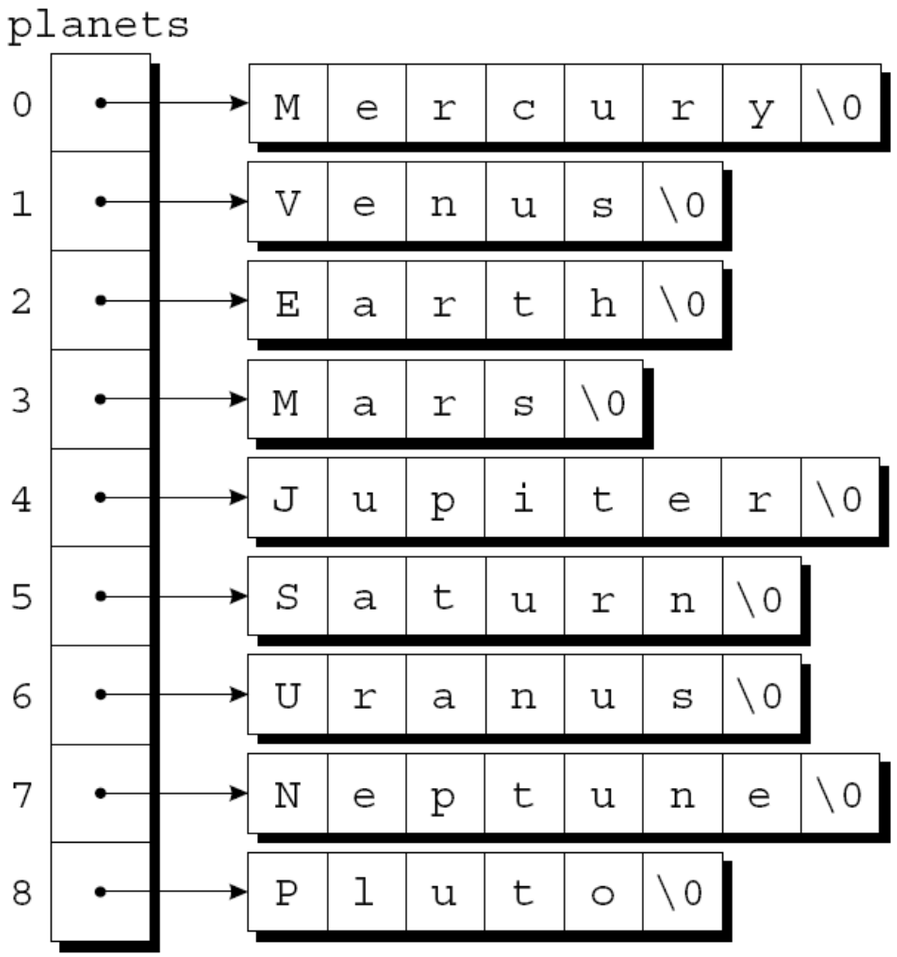
```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```



demo: array-string.c

Ragged Array of Strings

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

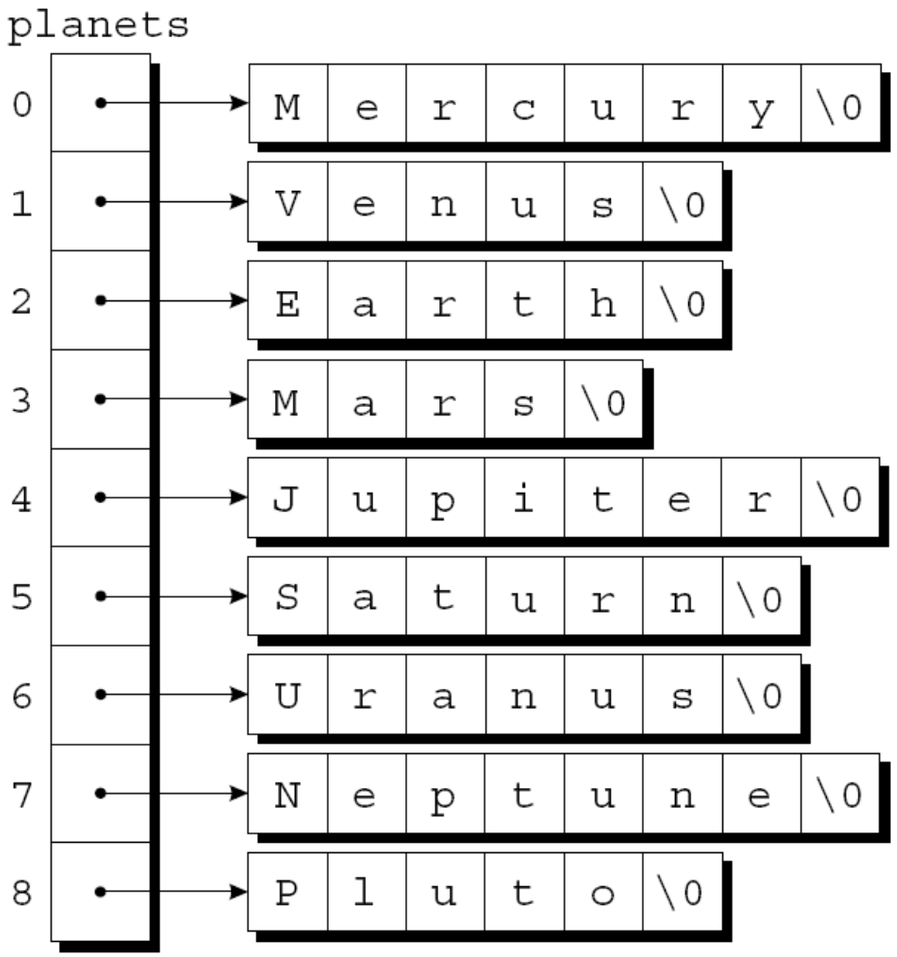


This is actually a 1D array, where each element of the array is a pointer to a null-terminated string

demo: array-string.c

Ragged Array of Strings

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



This is actually a 1D array, where each element of the array is a pointer to a null-terminated string

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

demo: array-string.c

Understanding `argv`

`ls -l remind.c`

```
int  main(int argc, char *argv[]){

  char **p;

  for (p = &argv[1]; *p != NULL; p++)
    printf("%s\n", *p);
}
```