Lecture 10 - Strings

Meng-Hsun Tsai
CSIE, NCKU

```
char s1[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
char s2[6] = "Hello";
char s3[6] = "He" "llo";
```

H	e	l	l	o	\0

Introduction

- This lecture covers both 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.
- The C library provides a collection of functions for working with strings.

IMSLab Since 2010

10.1 String Literals

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
```

Continuing a String Literal

- There are two ways to continue a string literal.
- The backslash character (\) can be used to continue a string literal from one line to the next:

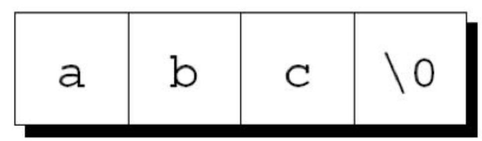
```
printf("When you come to a fork in the road, take it.  \
--Yogi Berra");
```

- When two or more string literals are adjacent, the compiler will join them into a single string.
  This rule allows us to split a string literal over two or more lines:

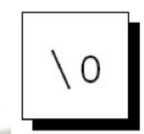
```
printf("When you come to a fork in the road, take it.  "
       "--Yogi Berra");
```

How String Literals Are Stored

- When a C compiler encounters a string literal of length *n*, 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 escape sequence whose bits are all zero, so it's represented by the `\0`.
- The string literal `"abc"` is stored as an array of four characters:



- Similarly, the string `""` is stored as a single null character:



How String Literals Are Stored (cont.)

- 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");
```

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 (cont.)

- String literals can be subscripted:

```
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 (cont.)

- Attempting to modify a string literal causes undefined behavior:

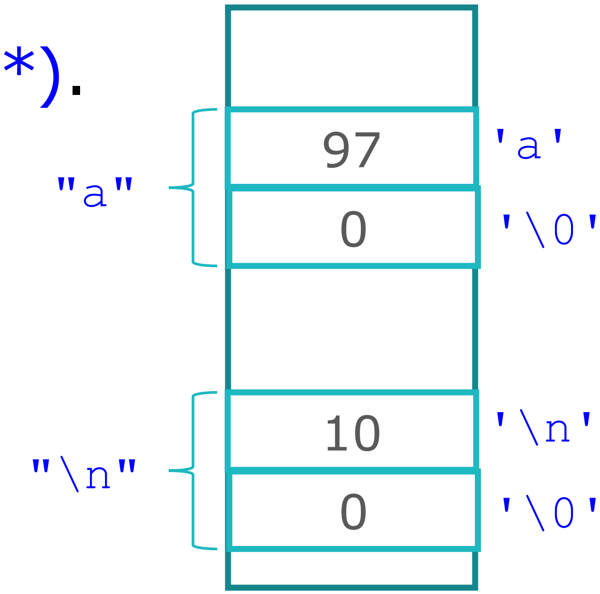
```
char *p = "abc";

*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 (char *)*.
  - `'a'` is represented by an *integer*.



- A legal call of `printf`:

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

- An illegal call:

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

10.2 String Variables

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.
  - Finding the length of a string requires searching for the null character.

String Variables (cont.)

- 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 and then adding 1 separately is a common practice.

String Variables (cont.)

- Be sure to leave room for the null character when declaring a string variable.
- Failing to do so may cause unpredictable results when the program is executed.
- The actual length of a string depends on the position of the terminating null character.
- An array of `STR_LEN` + 1 characters can hold strings with lengths between 0 and `STR_LEN`.

IMSLab Since 2010

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:

date1	J	u	n	e		1	4	\0

Initializing a String Variable (cont.)

- 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`:

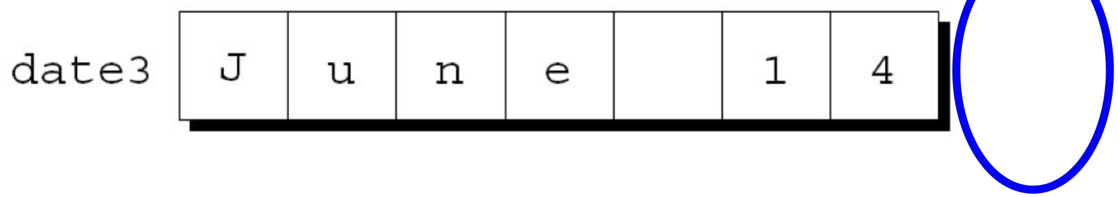
date2	J	u	n	e		1	4	\0	\0

Initializing a String Variable (cont.)

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



Initializing a String Variable (cont.)

- 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

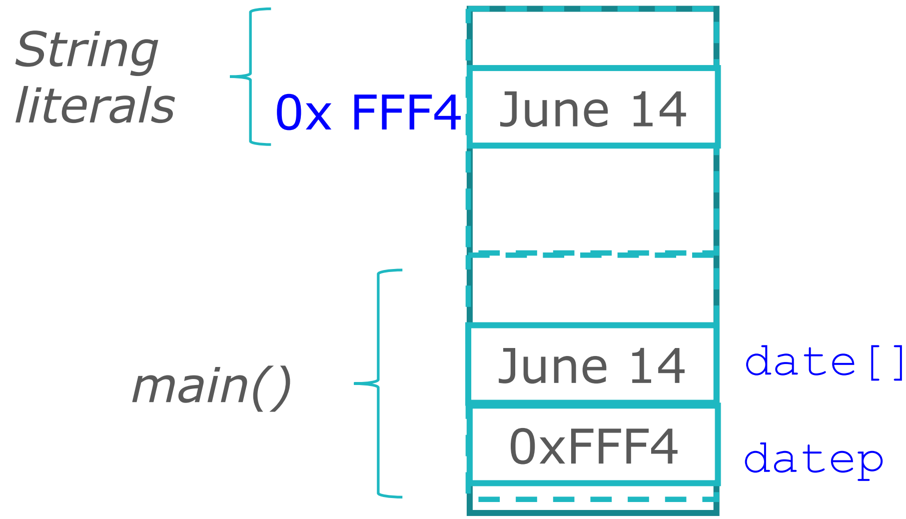
  `char date[] = "June 14";`

  declares `date` to be an *array*,

- The similar-looking

  `char *datep = "June 14";`

  declares `datep` 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 (cont.)

- 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, `datep` points to a string literal that shouldn't be modified.
  - In the array version, `date` is an array name.
    In the pointer version, `datep` is a variable that can point to other strings.

Character Arrays versus Character Pointers (cont.)

- 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.

Character Arrays versus Character Pointers (cont.)

- 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 initialized, this causes undefined behavior.

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]`.
- 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 (cont.)

- 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;
}
```

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

IMSLab Since 2010

Accessing the Characters in a String (cont.)

- Questions raised by the `count_spaces` example:
  - *Is it better to use array operations or pointer operations to access the characters in a string?*
    -> We can use either or both. Traditionally, C programmers lean toward using pointer operations.
  - *Should a string parameter be declared as an array or as a pointer?*
    ->There's no difference between the two.
  - *Does the form of the parameter (`s[]` or `*s`) affect what can be supplied as an argument?*
    -> No.

IMSLab Since 2010

10.3 Reading and Writing Strings

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's 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.

Writing Strings Using `printf` and `puts`

- The `%s` conversion specification allows `printf` to write a string:

```
char str[] = "Are we having fun yet?";

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

The output will be

```
Are we having fun yet?
```

- `printf` writes the characters in a string one by one until it encounters a null character.

Writing Strings Using `printf` and `puts` (cont.)

- To print part of a string, use the conversion specification `%.`*p*`s`.
- *p* is the number of characters to be displayed.
- The statement

```
printf("%.6s\n", str);
```

will print

```
Are we
```

Writing Strings Using **`printf`** and **`puts`** (cont.)

- The `%`*m*`s` conversion will display a string in a field of size *m*.
- If the string has fewer than *m* characters, it will be right-justified within the field.
- To force left justification instead, we can put a minus sign in front of *m*.
- The *m* and *p* values can be used in combination.
- A conversion specification of the form `%`*m*`.`*p*`s` causes the first *p* characters of a string to be displayed in a field of size *m*.

Writing Strings Using **`printf`** and **`puts`** (cont.)

- `printf` isn't the only function that can write strings.
- The C library also provides `puts`:

```
puts(str);
```

- After writing a string, `puts` always writes an additional new-line character.

Reading Strings Using `scanf` and `gets`

- The `%s` conversion specification allows `scanf` to read a string into a character array:

  ```
  scanf("%s", str);
  ```

- `str` is treated as a pointer, so there's no need to put the `&` operator in front of `str`.
- When `scanf` is called, 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 Using **`scanf`** and **`gets`** (cont.)

- `scanf` won't usually read a full line of input.
- A new-line character will cause `scanf` to stop reading, but so will a space or tab character.
- To read an entire line of input, we can use `gets`.
- Properties of `gets`:
  - Doesn't skip white space before starting to read input.
  - Reads until it finds a new-line character.
  - Discards the new-line character instead of storing it; the null character takes its place.

IMSLab Since 2010

Reading Strings Using **scanf** and **gets** (cont.)

- Consider the following program fragment:

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

- Suppose that after the prompt

```
Enter a sentence:
```

  the user enters the line

```
  To C, or not to C: that is the question.
```

- `scanf` will store the string `"To"` in `sentence`.

Reading Strings Using **scanf** and **gets** (cont.)

- Suppose that we replace `scanf` by `gets`:

```
gets(sentence);
```

- When the user enters the same input as before, `gets` will store the string

```
"  To C, or not to C: that is the question."
```

  in `sentence`.

Reading Strings Using `scanf` and `gets` (cont.)

- As they read characters into an array, `scanf` and `gets` have no way to detect when it's full.
- Consequently, they may store characters past the end of the array, causing undefined behavior.
- `scanf` can be made safer by using the conversion specification `%`*n*`s` instead of `%s`.
- *n* is an integer indicating the maximum number of characters to be stored.
- `gets` is inherently unsafe; `fgets` is a much better alternative.

Reading Strings Character by Character (cont.)

- Programmers often write their own input functions.
- Issues to consider:
  - Should the function skip white space before beginning to store the string?
  - What character causes the function to stop reading: a new-line character, any white-space character, or some other character? Is this character stored in the string or discarded?
  - What should the function do if the input string is too long to store: discard the extra characters or leave them for the next input operation?

Reading Strings Character by Character (cont.)

- Suppose we need a function that (1) doesn't skip white-space characters, (2) stops reading at the first new-line character (which isn't stored in the string), and (3) discards extra characters.
- A prototype for the function:

```
int read_line(char str[], int n);
```

- If the input line contains more than `n` characters, `read_line` will discard the additional characters.
- `read_line` will return the number of characters it stores in `str`.

Reading Strings Character by Character (cont.)

- `read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there's room left:

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';    /* terminates string */
  return i;         /* number of characters stored */
}
```

- `ch` has `int` type rather than `char` type because `getchar` returns an `int` value.

Reading Strings Character by Character (cont.)

- Before returning, `read_line` puts a null character at the end of the string.
- Standard functions such as `scanf` and `gets` automatically put a null character at the end of an input string.
- If we're writing our own input function, we must take on that responsibility.

10.4 Using the C String Library

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 (cont.)

- 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";
```

  In this context, `=` is not the assignment operator.

Using the C String Library (cont.)

- 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*.
- Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0.

Using the C String Library (cont.)

- 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 (cont.)

- 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" */
```

- 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.

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

IMSLab Since 2010

The **strcat** (String Concatenation) Function (cont.)

- 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" */
```

IMSLab Since 2010

The **`strcat`** (String Concatenation) Function (cont.)

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

```
char str1[6] = "abc";

strcat(str1, "def");    /*** WRONG ***/
```

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

The `strncat` (String Concatenation) Function

- The `strncat` function is a safer but slower version of `strcat`.
- Like `strncpy`, it has a third argument that limits the number of characters it will copy.
- A call of `strncat`:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

- `strncat` will terminate `str1` with a null character, which isn't included in the third argument.

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`.
- 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 (cont.)

- As it compares two strings, strcmp looks at the numerical codes for the characters in the strings.
- Important properties of ASCII:
  - A–Z, a–z, and 0–9 have consecutive codes.
  - All upper-case letters are less than all lower-case letters.
  - Digits are less than letters.
  - Spaces are less than all printing characters.

ASCII value	Character	ASCII value	Character	ASCII value	Character
000	^@	043	+	086	V
001	^A	044	,	087	W
002	^B	045	-	088	X
003	^C	046	.	089	Y
004	^D	047	/	090	Z
005	^E	048	0	091	[
006	^F	049	1	092	\
007	^G	050	2	093	]
008	^H	051	3	094	^
009	^I	052	4	095	_
010	^J	053	5	096	`
011	^K	054	6	097	a
012	^L	055	7	098	b
013	^M	056	8	099	c
014	^N	057	9	100	d
015	^O	158	:	101	e
016	^P	059	;	102	f
017	^Q	060	<	103	g
018	^R	061	=	104	h
019	^S	062	>	105	i
020	^T	063	?	106	j
021	^U	064	@	107	k
022	^V	065	A	108	l
023	^W	066	B	109	m
024	^X	067	C	110	n
025	^Y	068	D	111	o
026	^Z	069	E	112	p
027	^[	070	F	113	q
028	^\	071	G	114	r
029	^]	072	H	115	s
030	^^	073	I	116	t
031	^_	074	J	117	u
032	[space]	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

The `strcmp` (String Comparison) Function (cont.)

- `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`.

    `"abc" < "abd"`

  - All characters of `s1` match `s2`, but `s1` is shorter than `s2`.

    `"abc" < "abcd"`

IMSLab since 2010

Program: Printing a One-Month Reminder List

- The `remind.c` program prints a one-month list of daily reminders.
- The user will enter a series of reminders, with each prefixed by a day of the month.
- When the user enters 0 instead of a valid day, the program will print a list of all reminders entered, sorted by day.
- The next slide shows a session with the program.

Program: Printing a One-Month Reminder List (cont.)

```
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"
```

Program: Printing a One-Month Reminder List (cont.)

- Overall strategy:
  - Read a series of day-and-reminder combinations.
  - Store them in order (sorted by day).
  - Display them.
- `scanf` will be used to read the days.
- `read_line` will be used to read the reminders.

Program: Printing a One-Month Reminder List (cont.)

- The strings will be stored in a two-dimensional array of characters.
- Each row of the array contains one string.
- Actions taken after the program reads a day and its associated reminder:
  - Search the array to determine where the day belongs, using `strcmp` to do comparisons.
  - Use `strcpy` to move all strings below that point down one position.
  - Copy the day into the array and call `strcat` to append the reminder to the day.

IMSLab Since 2010

Program: Printing a One-Month Reminder List (cont.)

- One complication: how to right-justify the days in a two-character field.
- A solution: use `scanf` to read the day into an integer variable, then call `sprintf` to convert the day back into string form.
- `sprintf` is similar to `printf`, except that it writes output into a string.
- The following call writes the value of `day` into `day_str`.

```
sprintf(day_str, "%2d", day);
```

- The following call of `scanf` ensures that the user doesn't enter more than two digits:

```
scanf("%2d", &day);
```

Program: Printing a One-Month Reminder List (cont.)

remind.c

```
#include <stdio.h>
#include <string.h>
#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */
int read_line(char str[], int n);
int main(void)
{
  char reminders[MAX_REMIND][MSG_LEN+3];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;

  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }
    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);
```

Program: Printing a One-Month Reminder List (cont.)

```
    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      strcpy(reminders[j], reminders[j-1]);

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);
    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}
```

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

The `strtok` (String Token) Function

- Prototype for the `strtok` function:

```
char* strtok(char *str, const char *sep);
```

- `strtok` isolates sequential tokens in the string `str`. These tokens are separated by at least one of the characters in `sep`.
- The first time that `strtok()` is called, `str` should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead.
- The separator string, `sep`, must be supplied each time, and may change between calls.

Program: Separating Each Sentence

sep_sentence.c

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

int main()
{
    char str[80] = "Hi, I am Biden. Nice to meet you. Bye.";
    char * token;

    token = strtok(str, ".");
    while(token != NULL) {
        printf("%s\n", token);
        token = strtok(NULL, ".");
    }

    return 0;

}
```

```
$ ./sep_sentence
Hi, I am Biden
 Nice to meet you
 Bye
```

10.5 Arrays of Strings

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 (cont.)

- 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

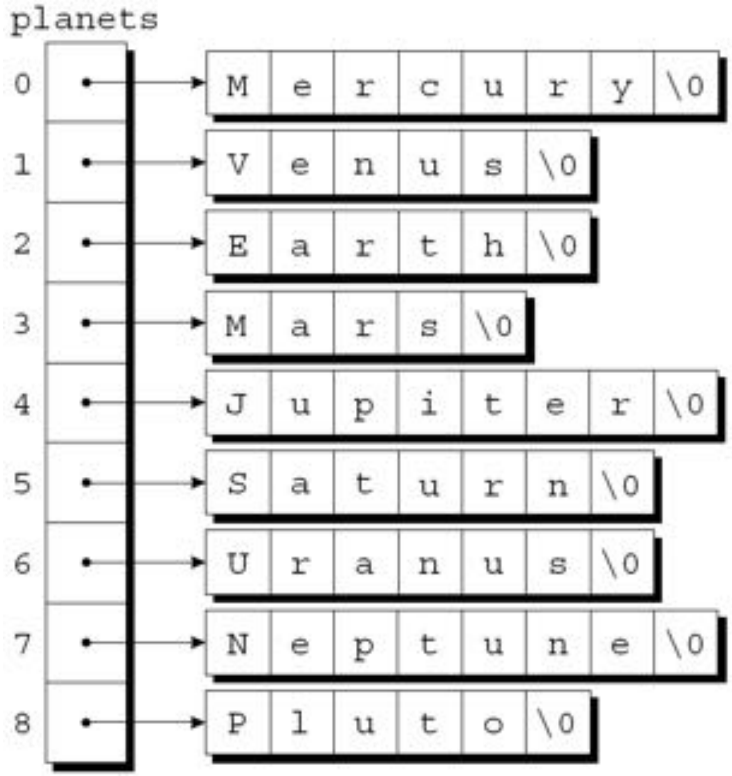
Arrays of Strings (cont.)

- 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 (cont.)

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



Arrays of Strings (cont.)

- 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]);
```

IMSLab Since 2010

Command-Line Arguments

- When we run a program, we'll often need to supply it with information.
- This may include a file name or a switch that modifies the program's behavior.
- Examples of the UNIX `ls` command:

```
ls
ls -l
ls -l remind.c
```

Command-Line Arguments (cont.)

- Command-line information is available to all programs, not just operating system commands.
- To obtain access to ***command-line arguments***, `main` must have two parameters:

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

- Command-line arguments are called ***program parameters*** in the C standard.

Command-Line Arguments (cont.)

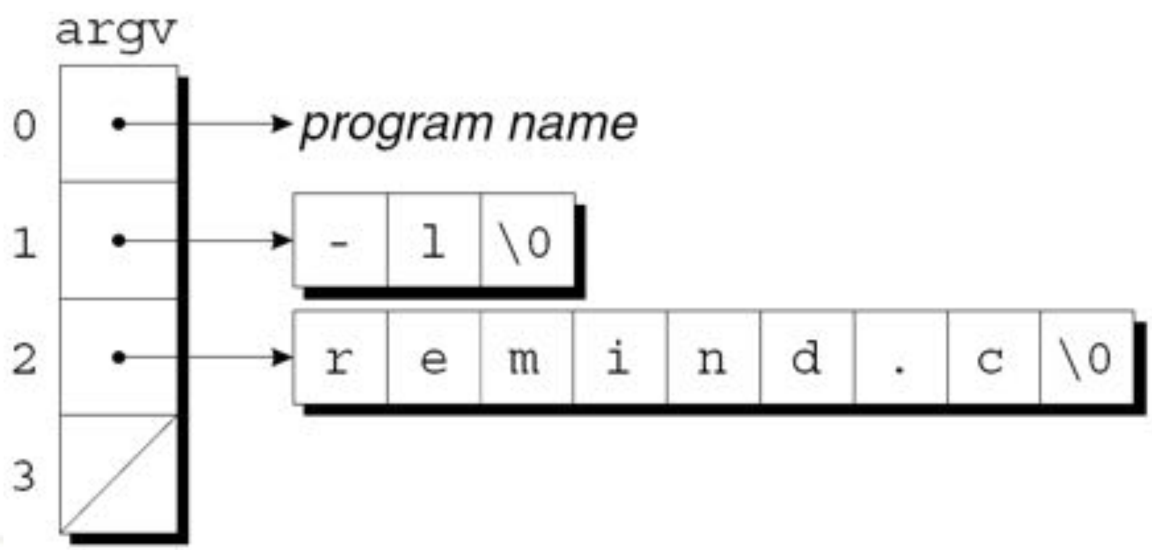
- `argc` ("argument count") is the number of command-line arguments.
- `argv` ("argument vector") is an array of pointers to the command-line arguments (stored as strings).
- `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.
- `argv[argc]` is always a ***null pointer***—a special pointer that points to nothing.
  - The macro `NULL` represents a null pointer.

Command-Line Arguments (cont.)

- If the user enters the command line

  `ls -l remind.c`

  then `argc` will be 3, and `argv` will have the following appearance:



Command-Line Arguments (cont.)

- Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn.

- One way to write such a loop is to use an integer variable as an index into the `argv` array:

```
int i;
for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

- Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly:

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

Program: Checking Planet Names

- The `planet.c` program illustrates how to access command-line arguments.
- The program is designed to check a series of strings to see which ones are names of planets.
- The strings are put on the command line:

```
planet Jupiter venus Earth fred
```

- The program will indicate whether each string is a planet name and, if it is, display the planet's number:

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

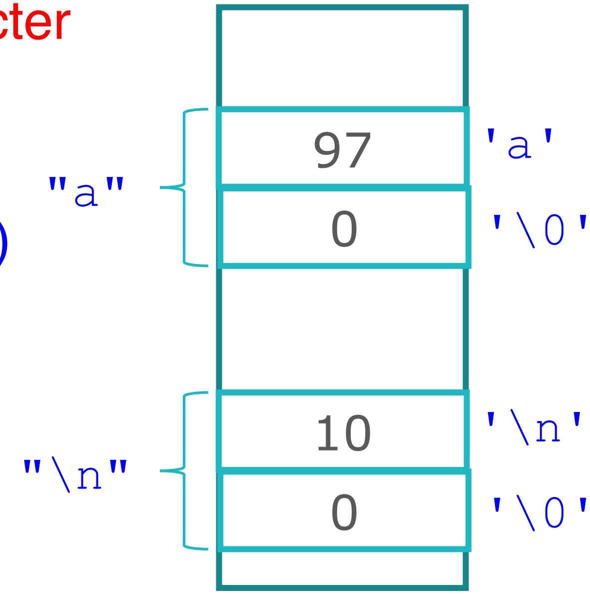
Program: Checking Planet Names (cont.)

planet.c

```
#include <stdio.h>
#include <string.h>
#define NUM_PLANETS 9
int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;
  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

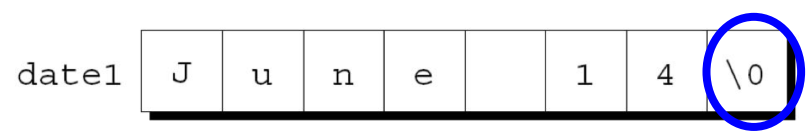
A Quick Review to This Lecture



- Strings are arrays of characters in which the null character marks the end (an extra byte is required).
- String Literals
  - String Literals: `"a"` (represented by a *pointer* (*char **))
  - Character Constants: `'a'` (represented by an *integer*)
  - String literals can be subscripted:

    char ch = “abc”[1];
- String Variables:
  - Be sure to leave room for the null character

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



  - Length can be omitted

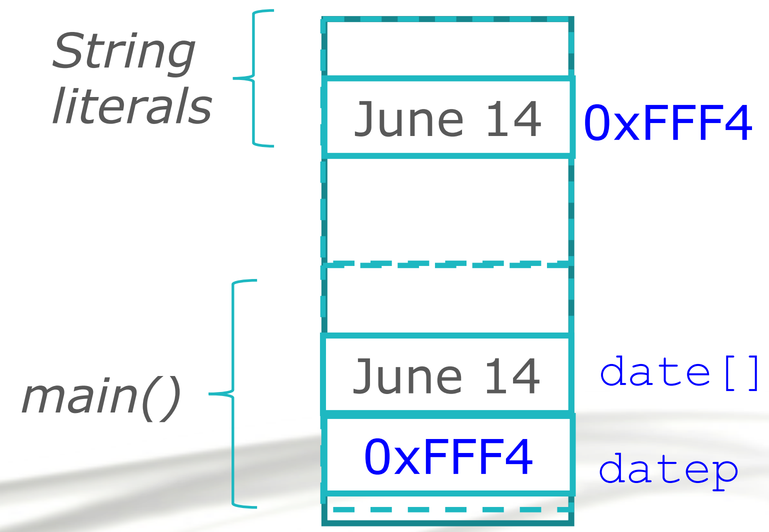
    ```
    char date1[] = "June 14";
    ```

A Quick Review to This Lecture (cont.)

- Character array vs. character pointer

Character array	Character pointer
`char date[] = "June 14";`	`char *datep = "June 14";`
Allocate space	Does not allocate space
Characters can be modified	Characters can not be modified

- Writing strings: `printf` / `puts`
- Reading strings: `scanf` / `gets` / `fgets`
- Reading a character: `getchar`



A Quick Review to This Lecture (cont.)

- `printf("%m.ps”, str);`
  - `%m.ps`: display first *p* characters in a field of size *m* (right justified)
  - `%-m.ps`: display first *p* characters in a field of size *m* (left justified)
- `puts(str);   //` always writes an additional new-line character

A Quick Review to This Lecture (cont.)

- `scanf("%ns", str);`
  - `%ns`: *n* indicates the maximum number of stored characters
  - skips leading white spaces, **reads**, and stops reading while encountering a white space or *n* characters are read
- `gets(str);`
  - Do not skip leading white spaces, **reads**, and stops reading only when encountering a new-line character
- `fgets(str, n, stdin);`

A Quick Review to This Lecture (cont.)

- C String Library (`#include <string.h>`)
  - `strcpy` / `strncpy` copies the string `s2` into the string `s1`, and returns `s1`

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

```
strcpy(str2, "abcd");
```

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

  - `strlen` returns the length of a string `s`, not including the null character.

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

```
len = strlen("abc"); // 3
```

  - `strcat` /`strncat` appends the string `s2` to the end of string `s1`, and returns `s1`

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

```
strcat(str1, "def");
```

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

A Quick Review to This Lecture (cont.)

- `strcmp` compares the strings `s1` and `s2`, returning a negative value, zero or a positive value if `s1` is less than, equal to, or greater than `s2`.

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

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

`s1` is less than `s2` if 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`.

```
"abc" < "abd"
```

- All characters of `s1` match `s2`, but `s1` is shorter than `s2`.

```
"abc" < "abcd"
```

- `sprintf` is similar to `printf`, but writes output into a string.

```
sprintf(day_str, "%2d", day);
```

IMSLab Since 2010

A Quick Review to This Lecture (cont.)

- `strtok` isolates sequential tokens in the string `str`. These tokens are separated by at least one of the characters in `sep`.

```
char* strtok(char *str, const char *sep);
```

- The first time that `strtok()` is called, `str` should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead.
- The separator string, `sep`, must be supplied each time, and may change between calls.

```
token = strtok(str, ".");
while(token != NULL) {
    printf("%s\n", token);
    token = strtok(NULL, ".");
}
```

IMSLab Since 2010

A Quick Review to This Lecture (cont.)

- To obtain access to ***command-line arguments***, `main` must have two parameters:

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

- If the user enters the command line

```
ls -l remind.c
```

then `argc` will be 3, and `argv` will have the following appearance: