# Chapter 13

# Strings


Copyright © 2008 W. W. Norton & Company.
All rights reserved.



## Introduction

- This chapter covers both
  - string *constants* (or *literals*, as they are called in the C standard)
  - 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


Copyright © 2008 W. W. Norton & Company.
All rights reserved.

## 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
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Continuing 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");
```

- In general, the \ character (*directly followed by* <cr>) can be used to join two or more lines of a program into a single line


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Continuing a String Literal

- There is a better way to deal with long string literals
- 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");
```

Or

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



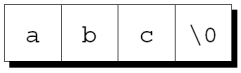
## How String Literals Are Stored

- 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 is represented by the `\0` escape sequence


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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:



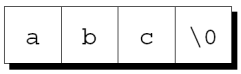
Copyright © 2008 W. W. Norton & Company.
All rights reserved.



## 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 passes to the `printf` function the ***address*** of `"abc"` (a pointer to where the letter `a` is stored in memory):

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





Copyright © 2008 W. W. Norton & Company.
All rights reserved.

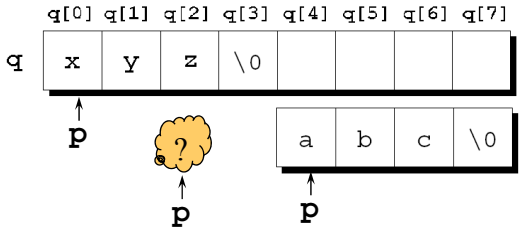
## Operations on String Literals

- 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
- What do you think about the following code?

```
char *p, q[8]= "xyz";

p =  q;
q = "abc"; /* WRONG */


p = "abc";
```





Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Operations on String Literals

- 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];
}
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Operations on String Literals

- Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';      /*** WRONG ***/
p[0] = 'd';    /*** WRONG ***/
p = "d";       /*** OK ***/
```

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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*.
- *How many bytes are needed for each of them?*
  *Do not forget the NULL character*
- A legal call of `printf`:

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

- An illegal call:

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## String Variables

- Any one-dimensional array of characters can be used to store a string
- A string ***must be*** terminated by a null character
  - Finding the length of a string requires searching for the null character


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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 (`STR_LEN`) 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## String Variables

- 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`


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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 array initializer abbreviation, i.e.,
```
char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4'};
```

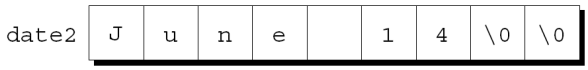

Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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`:




It is similar to:

```
char date2[9] = {'J', 'u', 'n', 'e', ' ', '1', '4'};
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

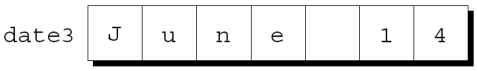


## Initializing a String Variable

- An initializer for a string variable can not 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:




It is similar to:

```
char date3[7] = {'J', 'u', 'n', 'e', ' ', '1', '4'};
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Character Arrays versus Character Pointers

- The declaration

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

  declares `date1` to be an *array*,
- The similar-looking

```
char *date2 = "June 14";
```

  declares `date2` to be a *pointer*
- Thanks to the close relationship between arrays and pointers, either version can be used as a string


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Character Arrays versus Character Pointers

- However, there are ***significant*** differences between the two versions of `date1` and `data2`
  - In the array version, the characters stored in `date1` can be modified
  - In the pointer version, `date2` points to a string literal that can not be modified
  - In the array version, `date1` is an array name, hence it can not point to any other string
  - In the pointer version, `date2` is a variable that can point to other strings


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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 (*we will talk about it later on*)


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Character Arrays versus Character Pointers

- Using an uninitialized pointer variable as a string is a ***serious error***
- A ***WRONG*** 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` has not been initialized, this causes undefined behavior


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Writing and Reading 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Writing Strings Using `printf` and `puts`

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

```
char str[] = "Are we having fun yet?";
printf("%.6s\n", str);
```

will print

```
Are we
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Writing Strings Using `printf` and `puts`

- The `%ms` 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.ps` causes the first *p* characters of a string to be displayed in a field of size *m*


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Writing Strings Using `printf` and `puts`

- `printf` is not 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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 is no need to put the `&` operator in front of `str`
- When `scanf` is called, it
  - skips white space
  - 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*


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Reading Strings Using `scanf` and `gets`

- `scanf` will not 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`:
  - Does not 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Reading Strings Using `scanf` and `gets`

- 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`


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Reading Strings Using `scanf` and `gets`

- 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`


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Reading Strings Using `scanf` and `gets`

- As they read characters into an array, `scanf` and `gets` have no way to detect when it is 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Reading Strings Character by Character

- 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?


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Reading Strings Character by Character

- Suppose we need a function that
  1) does not skip white-space characters,
  2) stops reading at the first new-line character (which is not stored in the string), and
  3) discards extra characters (i.e., discard anything more than `n` characters)
  4) returns the number of stored characters
- A prototype for the function:

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Reading Strings Character by Character

- `read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there is 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Reading Strings Character by Character

- Before returning, `read_line`, a ***null*** character has been put 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 are writing our own input function, we must take on that responsibility


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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]`


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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;
}
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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;
}
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Accessing the Characters in a String

- 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 is ***no difference*** between the two
  - *Does the form of the parameter (`s[]` or `*s`) affect what can be supplied as an argument?*

    ***No***


Copyright © 2008 W. W. Norton & Company.
All rights reserved.



## 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 are restricted in the same ways as arrays
  - In particular, they can not be copied or compared using operators


Copyright © 2008 W. W. Norton & Company.
All rights reserved.

## 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";
```

In this *initializing* context, = is not the assignment operator


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Using the C String Library

- Attempting to compare strings using a relational or equality operator is *legal* but *will not 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`


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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)


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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" */
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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 does not fit, undefined behavior occurs


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## The `strncpy` (String Copy) Function

- Calling the `strncpy` function is a ***safer but 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));
```

- *Is there a problem with the above statement?*


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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);
```

- `Strncpy` will terminate `str1` with a null character, which is not included in the third argument


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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 */
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

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

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## The `strcat` (String Concatenation) Function

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

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## The `strcat` (String Concatenation) Function

- `strcat(str1, str2)` causes undefined behavior if the `str1` array is not 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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 concatenation
- A call of `strncat`:

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

- `strncat` will terminate `str1` with a null character, which is not included in the third argument


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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:
- equal to `0`, if `s1` is equal to `s2`
- less than `0`, if `s1` is less than `s2`
- greater than `0`, if `s1` is greater than `s2`


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## 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`


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## The `strcmp` (String Comparison) Function

- As it compares two strings, `strcmp` looks at the numerical codes for the characters in the strings
- Important properties of ASCII:
  - 0–9, A–Z, and a–z, 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## The `strcmp` (String Comparison) Function

- Testing whether `str1` is greater 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? */
  …
```

- Testing whether `str1` is equal to `str2`:

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

or

```
if (!strcmp(str1, str2))     /* is str1 == str2? */
```

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Program: 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"
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Program: Printing a One-Month Reminder List

- 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Program: Printing a One-Month Reminder List

- 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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## Program: Printing a One-Month Reminder List

- *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 call

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

  writes the value of `day` into `day_str`


Copyright © 2008 W. W. Norton & Company. All rights reserved.



## Program: Printing a One-Month Reminder List

- The following call of `scanf` ensures that the user does not enter more than two digits:

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


Copyright © 2008 W. W. Norton & Company. All rights reserved.

## remind.c

```
/* Prints a one-month reminder list */

#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;
    }
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.



```
    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN); /* See Slides 35–37 */

    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;
}
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

```
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;
}
```


Copyright © 2008 W. W. Norton & Company.
All rights reserved.