Strings

Based on slides from K. N. King and Dianna Xu

Bryn Mawr College
CS246 Programming Paradigm

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

- The backslash character (\)
  
  printf("When you come to a fork in the road, take it. \n  --Yogi Berra");
  
  - In general, the \ character can be used to join two or more lines of a program into a single line.
  
  - When two or more string literals are adjacent, the compiler will join them into a single string.
  
  printf("When you come to a fork in the road, take it. "
  "--Yogi Berra");
  
  This rule allows us to split a string literal over two or more lines.

How String Literals Are Stored

- The string literal "abc" is stored as an array of four characters:
  
  a b c \n
  - The string "\n" is stored as a single null character:
    
    Null character

Operations on String Literals

- We can use a string literal wherever C allows a char * pointer:
  
  char *p;
  
  p = "abc";
  
  This assignment makes p point to the first character of the string.

How String Literals Are Stored

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

• String literals can be subscripted:
  ```c
  char ch;
  ch = "abc"[1]; // ch is 'b'
  ```

• A function that converts a number between 0 and
  15 into the equivalent hex digit:
  ```c
  char digit_to_hex_char(int digit) {
    return "0123456789ABCDEF"[digit];
  }
  ```

Operations on String Literals

• Attempting to modify a string literal causes
  undefined behavior:
  ```c
  char *p = "abc";
  *p = 'd'; // *** WRONG ***
  ```

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

String Literals vs Character Constants

• A string literal containing a single character is not
  the same as a character constant.
  o "a" - represented by a pointer.
  o 'a' - represented by an integer.

• A legal call of printf:
  ```c
  printf("\n");
  ```

• An illegal call:
  ```c
  printf('\n'); // *** WRONG ***
  ```

String Variables

• Any one-dimensional array of characters can be
  used to store a string.

• A string must be terminated by a null character.
  ```c
  #define STR_LEN 80
  char str[STR_LEN+1];
  ```

  - Defining a macro that represents 80 and then adding
    1 separately is a common practice.

Initializing a String Variable

• A string variable can be initialized at the same time
  it’s declared:
  ```c
  char date1[8] = "June 14";
  char date4[] = "June 14";
  char date2[9] = "June 14";
  ```

  - Not a string literal.
  - An abbreviation for an array initializer

Character Arrays vs Character Pointers

• The declaration
  ```c
  char date[] = "June 14";
  ```

  - Array name
  - Characters can be modified
  - declares date to be an array.

• The similar-looking
  ```c
  char *date = "June 14";
  ```

  - Pointer variable
  - String literal – should not be modified.
  - declares date to be a pointer.

• Thanks to the close relationship between arrays and
  pointers, either version can be used as a string.
Character Arrays vs Character Pointers

- char *p; // does not allocate space for a string.
- Using an uninitialized pointer variable as a string is a serious error.
- An attempt at building the string "abc":
  ```c
  char *p;
p[0] = 'a';  /*** WRONG ***/
p[1] = 'b';  /*** WRONG ***/
p[2] = 'c';  /*** WRONG ***/
p[3] = '\0'; // *** WRONG ***
  ```
- Before we can use p as a string, it must point to an array of characters.
  ```c
  char str[STR_LEN+1], *p;
p = str;
  ```

Reading and Writing Strings

- Writing a string
  ```c
  printf
  puts
  ```
- Reading a string
  ```c
  scanf
  gets
  ```
  o read strings one character at a time.
- Reading a string is a bit harder, because the input may be longer than the string variable into which it’s being stored.

```
```c
```
Displaying Characters of a String

```c
int main() {
    char s[] = "01234";
    char *p;
p = s;
    printf("%c\n", s[0]);
    printf("%c\n", *p);
    printf("%c\n", *(p + 0));
    printf("%c\n", s[2]);
    printf("%c\n", *(p + 2));
}
```

printf and puts

```c
puts(str);
```
- After writing a string, `puts` always writes an additional new-line character.

```c
#define BUFLEN 200

int main() {
    char buf[BUFLEN];
    gets(buf);
    puts(buf);
    return 0;
}
```
- `puts` adds `
` to output, equivalent to `printf("%s\n", buf);`

scanf and gets

- The `%s` conversion specification allows `scanf` to read a string into a character array:
  ```c
  scanf("%s", 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.
- `printf` and `puts`

```c
puts(str);
```
- After writing a string, `puts` always writes an additional new-line character.

```c
#define BUFLEN 200

int main() {
    char buf[BUFLEN];
    gets(buf);
    puts(buf);
    return 0;
}
```
- `puts` adds `
` to output, equivalent to `printf("%s\n", buf);`

• The `%s` conversion specification allows `scanf` to read a string into a character array:
  ```c
  scanf("%s", 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.

• 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`: use the conversion specification `%ms` instead of `%s`.
• `gets` is inherently unsafe; `fgets` is a much better alternative.

• Consider the following program fragment:
  ```c
  char sentence[SENT_LEN+1];
  printf("Enter a sentence:\n");
  scanf("%s", sentence);
  ```
• Suppose that the user enters the line
  ```c
  To C, or not to C: that is the question.
  ```
• `scanf` will store the string "To" in `sentence`.
• `gets` will store the string
  ```c
  "To C, or not to C: that is the question."
  ```
  in `sentence`.

- The `%s` conversion specification allows `scanf` to read a string into a character array:
  ```c
  scanf("%s", 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.
- `printf` and `puts`

```c
puts(str);
```
- After writing a string, `puts` always writes an additional new-line character.

```c
#define BUFLEN 200

int main() {
    char buf[BUFLEN];
    gets(buf);
    puts(buf);
    return 0;
}
```
- `puts` adds `
` to output, equivalent to `printf("%s\n", buf);`

• The `%s` conversion specification allows `scanf` to read a string into a character array:
  ```c
  scanf("%s", 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.

• 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`: use the conversion specification `%ms` instead of `%s`.
• `gets` is inherently unsafe; `fgets` is a much better alternative.

- The `%s` conversion specification allows `scanf` to read a string into a character array:
  ```c
  scanf("%s", 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.

• 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`: use the conversion specification `%ms` instead of `%s`.
• `gets` is inherently unsafe; `fgets` is a much better alternative.
Accessing the Characters in a String

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

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

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

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