Pointers and Variable-Length Arrays (C99)

- Pointers are allowed to point to elements of variable-length arrays (VLAs).
- Example:

```
void f(int n) {
  int a[n], *p;
  p = a;
  …
}
```

- When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first.

  A two-dimensional example:

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];
  p = a;
  …
}
```

- Since the type of p depends on n, which isn't constant, p is said to have a ***variably modified type.***
- **A potential problem:**

  The validity of an assignment such as p = a can't always be determined by the compiler.

  The following code will compile but is correct only if m and n are equal:

```
int a[m][n], (*p)[m];
p = a;
```

  If m is not equal to n, any subsequent use of p will cause undefined behavior.

Pointer arithmetic works with VLAs.

- A two-dimensional VLA:

```
int a[m][n];
```

  A pointer capable of pointing to a row of a:

```
int (*p)[n];
```

  A loop that clears column i of a:

```
for(p = a; p < a + m; p++)
  (*p)[i] = 0;
```

- Restriction on variable-length arrays:

  the declaration must be inside the body of a function or in a function prototype.

Chapter 13 Strings

- Strings are arrays of characters in which a special character—the null character—marks the end.
- A ***string literal*** (or string constant) 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
- 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");
```

  Anther way to do it:

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

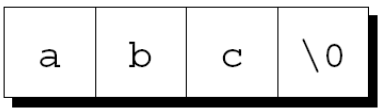
How String Literals Are Stored

- When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string.

  The memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string.

  The null character is a byte whose bits are all zero, so it's represented by the \0 escape sequence.

  For example, the string literal "abc" is stored as an array of four characters:



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

- Since a string literal is stored as an array, the compiler treats it as a pointer of type char *.
- 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.
- 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];
}
```

- A string literal containing a single character isn't the same as a character constant.
  - "a" is represented by a *pointer*.
  - 'a' is represented by an *integer*.
- A legal call of printf:

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

  An illegal call:

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

String Variables

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

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

Initializing a String Variable

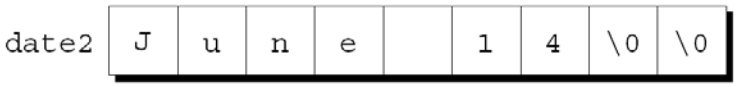
- A string variable can be initialized at the same time it's declared:

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

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



- 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";
```

  The code line above is not good.
- 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 date1[] = "June 14";
  ```

  declares date to be an *array*. The similar-looking

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

  declares date to be a *pointer*.

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

  – In the array version, the characters stored in date1 can be modified. In the pointer version, date2 points to a string literal that shouldn't be modified.

  – In the array version, date1 is an array name. In the pointer version, date2 is a variable that can point to other strings.

- The declaration

  ```
  char *p;
  ```

  does not allocate space for a string.

  Before we can use p as a string, it must point to an array of characters.

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

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

- Another possibility is to make p point to a dynamically allocated string.

- Using an uninitialized pointer variable as a string is a serious error.

  Example: 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.

Reading and Writing Strings

- **Writing** a string is easy using either printf or 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.

- 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

- 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*.

- 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** a string is a bit harder. To read a string in a single step, we can use either **scanf** or **gets**.

  An alternative is to read strings one character at a time.

- 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.

- 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.
- 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.
```

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

  char *fgets(char *str, int n, FILE *stream)
  reads a line from the specified stream and stores it into the string pointed to by **str**. It stops when either **(n-1)** characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
- 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

- 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) read n characters and 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 read.
- 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;//No. of characters stored
}
```

- ch has int type rather than char type because getchar returns an int value.
- Before returning, read_line puts a null character at the end of the string.

  Functions 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.

Accessing the Characters in a String

- Since strings are 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].

  An example we saw earlier:

```
char digit_to_hex_char(int digit){
     return "0123456789ABCDEF"[digit];
}
```

- Another example: 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;
}
```

- 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;
}
```

- 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. 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.

Using the C String Library

- Common string operations include copying strings, comparing strings, concatenating strings, selecting substrings.
- 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 in C.

- 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:

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

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

```
if(str1 == str2);
```

- This statement compares str1 and str2 as *pointers*.

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

The strcpy function

- 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.
- **String copy**: 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).

- EXAMPLE:

```
. . .
strcpy(str2, "abcd");//str2 now has "abcd"
strcpy(str1, str2); // str1 now has "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.

- Calling the **strncpy** function is a safer, albeit slower, way to copy a string.

  strncpy has a third argument that limits the number of characters that will be copied.

  A call of strncpy that copies str2 into str1:

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

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
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).
- examples:

```
strcpy(str1, "abc");
strcat(str1, "def");//has "abcdef"

strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);//has "abcdef"

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

- 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** 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.