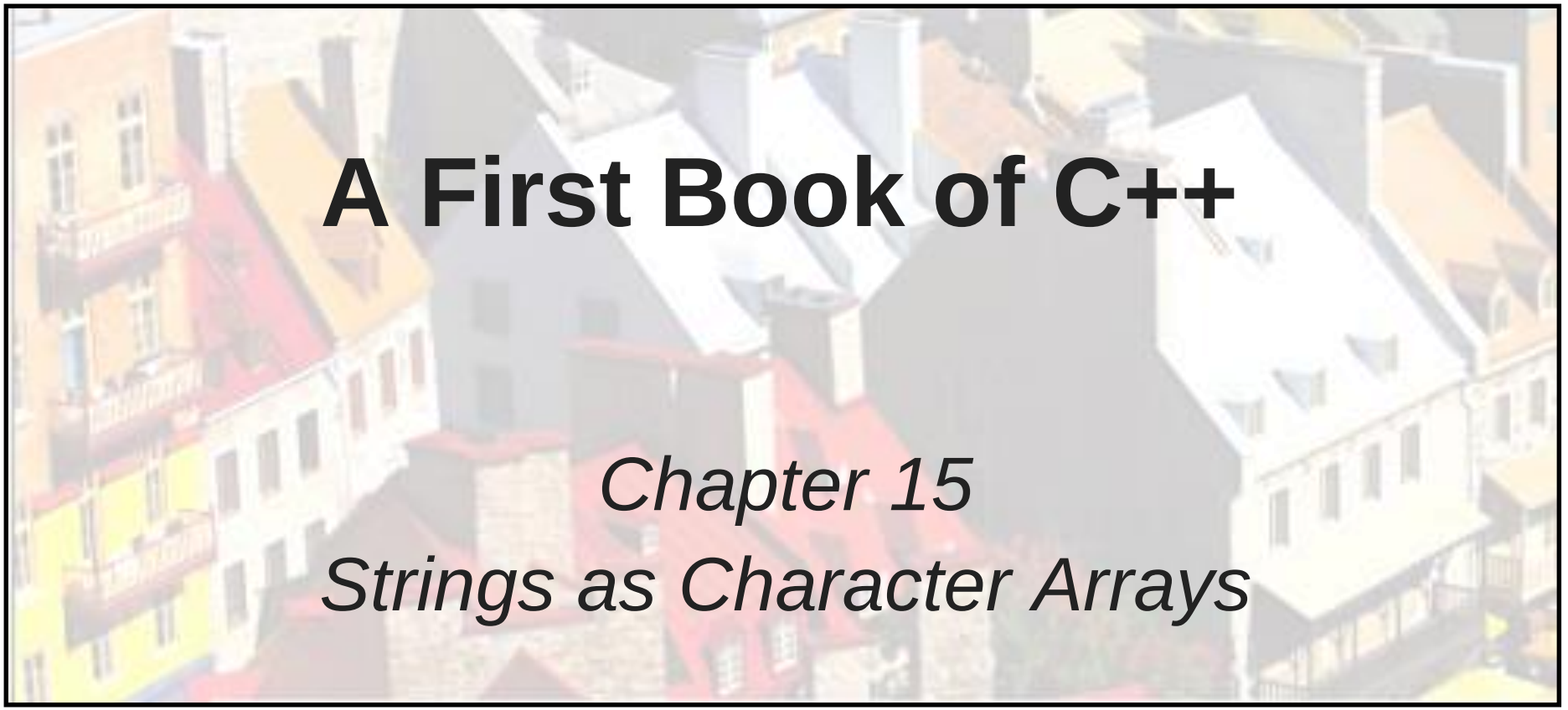

A First Book of C++

Chapter 15

Strings as Character Arrays

Objectives

- In this chapter, you will learn about:
  - C-String Fundamentals
  - Pointers and C-String Library Functions
  - C-String Definitions and Pointer Arrays
  - Common Programming Errors

C-String Fundamentals

- **Character strings (C-strings)**: using an array of characters that is terminated by a sentinel value (the escape sequence `'\0'`)
- C-strings can be created in a number of ways:
  - `char test[5] = "abcd";`
  - `char test[] = "abcd";`
  - `char test[5] = {'a', 'b', 'c', 'd', '\0'};`
  - `char test[] = {'a', 'b', 'c', 'd', '\0'};`

C-String Fundamentals (cont'd.)

- Array of characters terminated by a special end-of-string marker called the `NULL` character
  - This character is a sentinel marking the end of the string
  - The `NULL` character is represented by the escape sequence,`\0`
- Individual characters in a string array can be input, manipulated, or output using standard array-handling techniques

C-String Input and Output

- Inputting and displaying a string requires a standard library function or class method:
  - **`cin`** and **`cout`** (standard input and output streams)
  - String and character I/O functions (Table 15.1)
    - Require the `iostream` header file
- Character input methods are not the same as methods defined for the `string` class having the same name
- Character output methods are the same as for `string` class

Table 15.1 String and Character I/O Methods (Require the Header File `iostream`)

C++ Method	Description	Example
`cin.getline(str,n,ch)`	Inputs a C-string `(str)` from the keyboard, up to a maximum of *n* characters, that's terminated by the character `ch` (typically the newline character, `'\n'`)	`cin.getline(str, 81, '\n');`
`cin.get()`	Extracts the next character from the input stream	`nextKey = cin.get();`
`cin.peek()`	Returns the next character from the input stream *without* extracting the character from the stream	`nextKey = cin.peek();`
`cout.put(charExp)`	Places the character value of `charExp` on the output stream	`cout.put('A');`
`cin.putback(charExp)`	Pushes the character value of `charExp` back onto the input stream	`cin.putback(cKey);`
`cin.ignore(n, char)`	Ignores a maximum of the next *n* input characters, up to and including the detection of `char`; if no arguments are specified, ignores the next single character on the input stream	`cin.ignore(80,'\n');cin.ignore();`

C-String Input and Output (cont'd.)

Program 15.1

```
#include <iostream>
using namespace std;

int main ()
{
  const int MAXCHARS  =  81;
  char message [MAXCHARS]; // an array of characters with
                           // enough storage for a complete line
  cout << "Enter a string:\n";
  cin.getline(message,MAXCHARS, '\n');
  cout << "The string just entered is:\n"
       << message << endl;

  return 0;
}
```

C-String Input and Output (cont'd.)

- Program 15.1 illustrates using `cin.getline()` and `cout` to input and output a string entered at the user's terminal
  - Sample run of Program 15.1:

```
Enter a string:
This is a test input of a string of characters.
The string just entered is:
This is a test input of a string of characters.
```

C-String Input and Output (cont'd.)

- The `cin.getline()` method in Program 15.1 continuously accepts and stores characters into character array named `message`
  - Input continues until:
    - Either 80 characters are entered
    - The ENTER key is detected

C-String Input and Output (cont'd.)

- In Program 15.1, all characters encountered by `cin.getline()`, except newline character, are stored in `message` array
- Before returning, `cin.getline()` function appends a `NULL` character, `'\0'`, to the stored set of characters (Figure 15.2)
- `cout` object is used to display the C-string

C-String Processing

- C-strings can be manipulated by using either standard library functions or as subscripted array variables
  - Library functions are presented in the next section
- First look at processing a string in a character-by-character fashion
  - Example: `strcopy()` copies contents of `string2` to `string1`

C-String Processing (cont'd.)

- Function `strcopy()`

```
// copy string2 to string1
void strcopy(char string1[], char string2[])
{
  int i = 0;
  while ( string2[i] != '\0')
  {
    string1[i] = string2[i];
    i++;
  }
  string1[i] = '\0';
  return;
}
```

C-String Processing (cont'd.)

- Main features of function `strcopy()`
  - The two strings are passed to `strcopy` as arrays
  - Each element of `string2` is assigned to the equivalent element of `string1` until end-of-string marker is encountered
  - Detection of `NULL` character forces termination of the `while` loop that controls the copying of elements
  - Because `NULL` character is not copied from `string2` to `string1`, the last statement in `strcopy()` appends an end-of-string character to `string1`

C-String Processing (cont'd.)

- C-strings can be processed by using character-by-character techniques
- Program 15.3 uses `cin.get()` to accept a string one character at a time
  - Code lines 8 – 14 replace `cin.getline()` function used in Program 10.1
  - Characters will be read and stored in `message` array, provided:
    - Number of characters is less than 81
    - Newline character is not encountered

Pointers and C-String Library Functions

- Pointers are very useful in constructing functions that manipulate C-strings
- When pointers are used in place of subscripts to access individual C-string characters, resulting statements are more compact and efficient
- Consider `strcopy()` function (slide 12)
  - Two modifications are necessary before converting to a pointer version…

Pointers and C-String Library Functions (cont'd.)

- Modification 1: eliminate `(string2[i] != '\0')` test from `while` statement
  - This statement is only false when end-of-string character is encountered
  - Test can be replaced by `(string2[i])`
- Modification 2: include assignment inside test portion of `while` statement
  - Eliminates need to terminate copied string with `NULL` character

Pointers and C-String Library Functions (cont'd.)

- Pointer version of `strcopy()`

```
void strcopy(char *string1, char *string2)
{
   while (*string1 = *string2)
   {
      string1++;
      string2++;
   }
   return;
}
```

Library Functions

- C++ does not provide built-in operations for complete arrays (such as array assignments)
- Assignment and relational operations are not provided for C-strings
- Extensive collections of C-string handling functions and routines are included with all C++ compilers (Table 15.2)
  - These functions and routines provide for C-string assignment, comparison, and other operations

Library Functions (cont'd.)

- Four most commonly used C-string library functions:
  - `strcpy()`: copies a source C-string expression into a destination C-string variable
    - Example: `strcpy(string1, "Hello World!")` copies source string literal "Hello World!" into destination C-string variable `string1`

Library Functions (cont'd.)

- `strcat()`: appends a string expression onto the end of a C-string variable
  - Example:

```
strcat(dest_string, " there World!")
```

- `strlen()`: returns the number of characters in its C-string parameter (not including `NULL` character)
  - Example: value returned by `strlen("Hello World!")` is 12

Library Functions (cont'd.)

- `strcmp()`: compares two C-string expressions for equality
  - When two C-strings are compared, individual characters are compared a pair at a time
  - If no differences are found, strings are equal
  - If a difference is found, string with the first lower character is considered the smaller string
  - Example:
    - `"Hello"` is greater than `"Good Bye"` (first `'H'` in `Hello` greater than first `'G'` in `Good Bye`)

Character-Handling Functions

- Provided by C++ compilers in addition to C-string manipulation functions
- Prototypes for routines are contained in header file `cctype`; should be included in any program that uses them

Conversion Functions

- Used to convert C-strings to and from integer and double-precision data types
- Prototypes for routines contained in header file `cstdlib`;
  - `cstdlib` should be included in any program that uses these routines

Conversion Functions (cont'd.)

Table 15.4 String Conversion Functions (Require the Header File `cstdlib`)

Function Prototype	Description	Example
`int atoi(stringExp)`	Converts `stringExp` (an ASCII string) to an integer. Conversion stops at the first non-integer character.	`atoi("1234")`
`double atof(stringExp)`	Converts `stringExp` (an ASCII string) to a double-precision number. Conversion stops at the first character that can't be interpreted as a `double`.	`atof("12.34")`
`char[] itoa(integerExp)`	Converts `integerExp` (an integer) to a character array. The space allocated for the returned characters must be large enough for the converted value.	`itoa(1234)`

C-String Definitions and Pointer Arrays

- The definition of a C-string automatically involves a pointer
- Example: Definition `char message1[80];`
  - Reserves storage for 80 characters
  - Automatically creates a pointer constant, `message1`, that contains the address of `message1[0]`
  - Address associated with the pointer constant cannot be changed
    - It must always “point to” the beginning of the created array

C-String Definitions and Pointer Arrays (cont'd.)

- Also possible to create C-string using a pointer
  - Example: Definition `char *message2;` creates a pointer to a character
  - `message2` is a true pointer variable
- Once a pointer to a character is defined, assignment statements, such as `message2 = "this is a string";`, can be made
  - `message2`, which is a pointer, receives address of the first character in the string

C-String Definitions and Pointer Arrays (cont'd.)

- Main difference in the definitions of `message1` as an array and `message2` as a pointer is the way the pointer is created
- `char message1[80]` explicitly calls for a fixed amount of storage for the array
  - Compiler creates a pointer constant
- `char *message2` explicitly creates a pointer variable first
  - Pointer holds the address of a C-string when the C-string is actually specified

C-String Definitions and Pointer Arrays (cont'd.)

- Defining `message2` as a pointer to a character allows C-string assignments

  `message2 = "this is a string";` is valid

- Similar assignments not allowed for C-strings defined as arrays

  `message1 = "this is a string";` is not valid

- Both definitions allow initializations using string literals such as:

```
char message1[80] = "this is a string";
char *message2 = "this is a string";
```

C-String Definitions and Pointer Arrays (cont'd.)

- Allocation of space for `message1` is different from that for `message2`
- Both initializations cause computer to store the same C-string internally (Figure 15.5)
- `message1` storage:
  - Specific set of 80 storage locations reserved; first 17 locations initialized
  - Different C-strings can be stored, but each string overwrites previously stored characters
    - Same is not true for `message2`

C-String Definitions and Pointer Arrays (cont'd.)

- Definition of `message2` reserves enough storage for one pointer
  - Initialization then causes the string literal to be stored in memory
    - Address of the string's first character (`'t'`) is loaded into the pointer
  - If a later assignment is made to `message2`, the initial C-string remains in memory; new storage locations are allocated to new C-string (Figure 10.6)

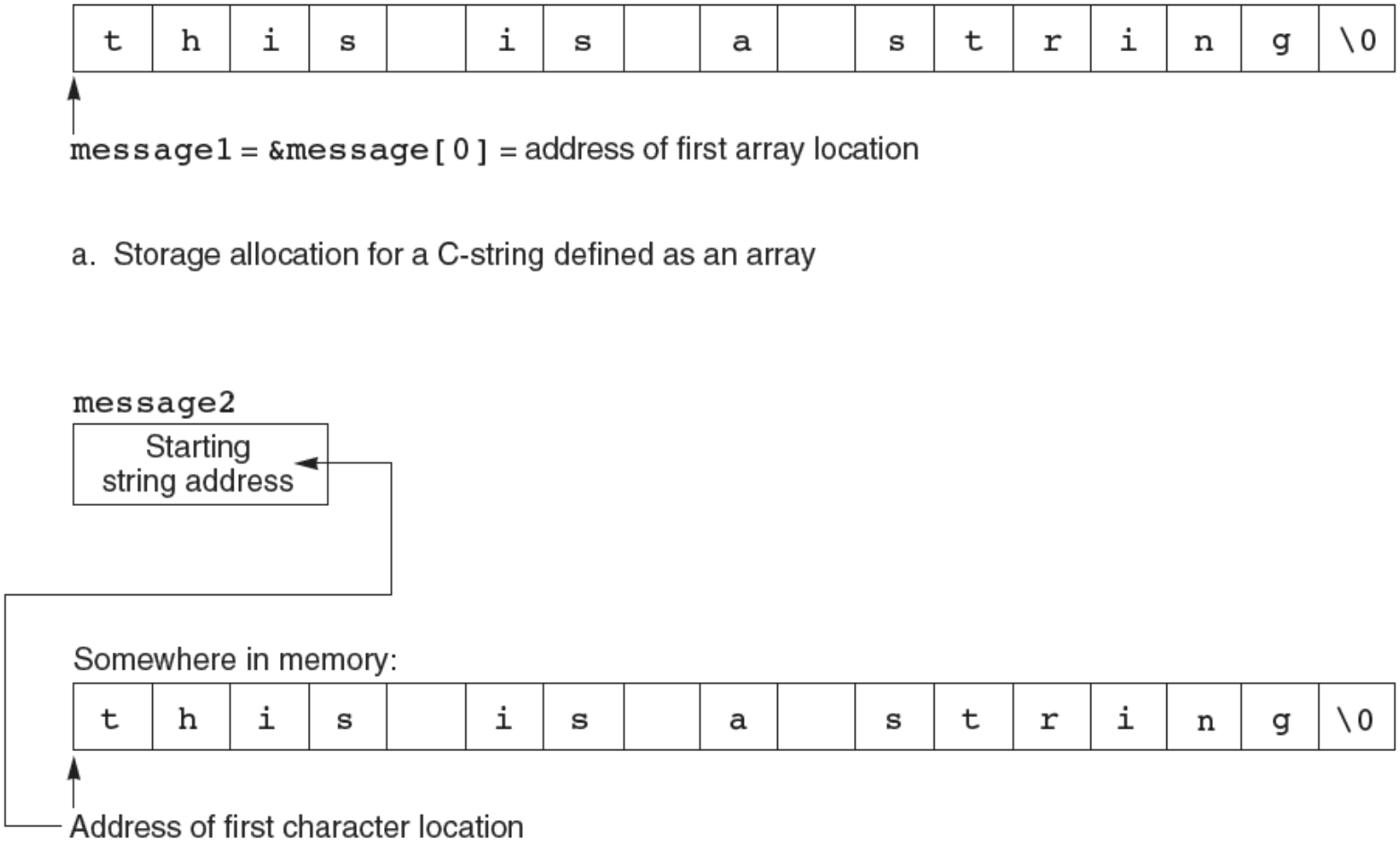

Figure 15.5 C-string storage allocation

C-String Definitions and Pointer Arrays (cont'd.)

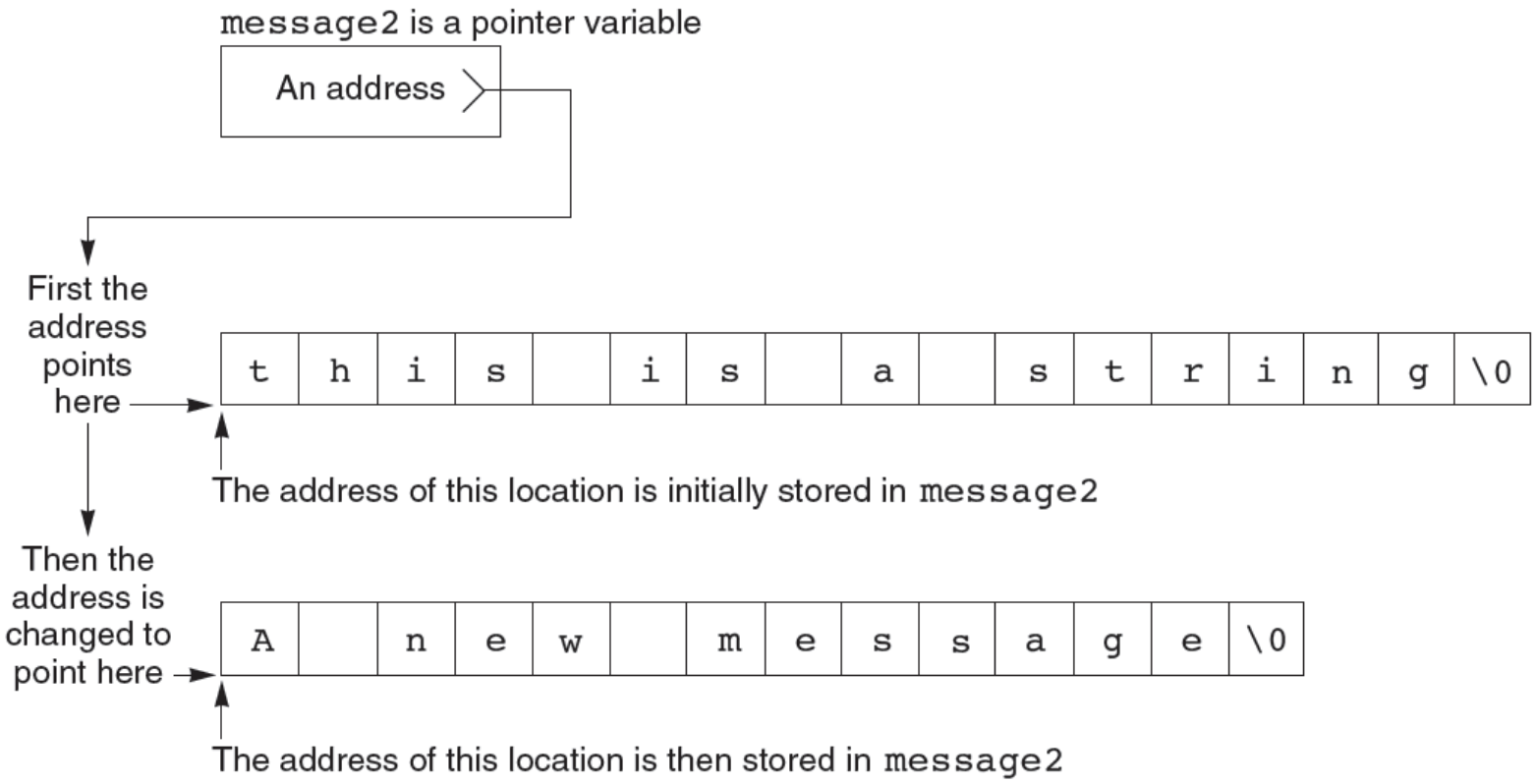

Figure 15.6 Storage allocation for Figure 15.5

Pointer Arrays

- Declaration of an array of character pointers is an extremely useful extension to single string pointer declarations
  - Declaration `char *seasons[4];` creates an array of four elements; each element is a pointer to a character
- Each pointer can be assigned to point to a string using string assignment statements

```
seasons[0] = "Winter";
seasons[1] = "Spring";
seasons[2] = "Summer";
seasons[3] = "Fall";
        // note: string lengths may differ
```

Pointer Arrays (cont'd.)

- The `seasons` array does not contain actual strings assigned to the pointers (Figure 15.7)
  - Strings stored in data area allocated to the program
- Array of pointers contains only the addresses of the starting location for each string
- Initializations of the `seasons` array can also be put within array definition:

```
char *seasons[4] = {"Winter",
                    "Spring",
                    "Summer",
                    "Fall"};
```

Pointer Arrays (cont'd.)

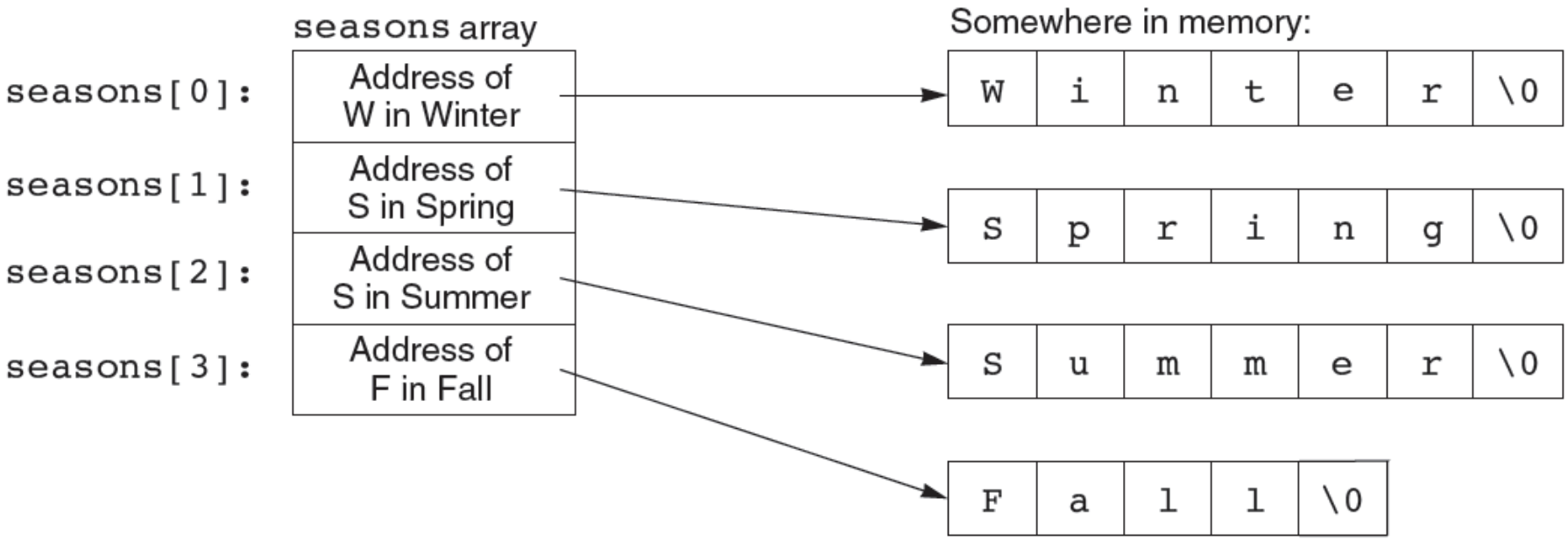

Figure 15.7 The addresses in the `seasons[]` pointers

Common Programming Errors

- Using a pointer to point to a nonexistent data element
- Not providing enough storage for a C-string to be stored
- Misunderstanding of terminology
  - Example: If `text` is defined as `char *text;`
    - Variable `text` is sometimes called a string
    - `text` is not a string; it is a pointer that contains the address of the first character in the C-string

Summary

- A C-string is an array of characters that is terminated by the `NULL` character
- C-strings can always be processed using standard array-processing techniques
- The `cin`, `cin.get()`, and `cin.getline()` routines can be used to input a C-string
- The `cout` object can be used to display C-strings
- Pointer notation and pointer arithmetic are useful for manipulating C-string elements

Summary (cont'd.)

- Many standard library functions exist for processing C-strings as a complete unit
- C-string storage can be created by declaring an array of characters or by declaring and initializing a pointer to a character
- Arrays can be initialized using a string literal assignment of the form:

```
char *arr_name[ ] = "text";
```

  - This initialization is equivalent to:

```
char *arr_name[ ] = {'t','e','x','t','\0'};
```

- A pointer to a character can be assigned a string literal