Characters, c-Strings, and the string Class

CS 1: Problem Solving & Program Design Using C++

Objectives

- Perform character checks and conversions
- Knock down the C-string fundamentals
- Point at pointers and C-string library functions
- Discover C-string definitions and pointer arrays
- Look at more common programming errors

Character Checks

Function	Meaning
isalpha()	True if argument is a letter, false otherwise
isalnum()	True if argument is a letter or digit, false otherwise
isdigit()	True if argument is a digit, false otherwise
islower()	True if argument is a lowercase letter, false otherwise
isprint()	True if argument is a printable character, false otherwise
ispunct()	True if argument is a punctuation character, false otherwise
isupper()	True if argument is an uppercase character, false otherwise
isspace()	True if argument is a whitespace character, false otherwise

Character Checks Example

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

int main()
{
		char ch;

		cout << “Input any character:” << endl;
		cin >> ch;
		if (isalpha(ch))
		{
				cout << ch << “ is an alphabetic character.” << endl;
		}
		if (isdigit(ch))
		{
				cout << ch << “ is a digit.” << endl;
		}
```

Character Checks Example (2)

```
        if (islower(ch))
        {
                cout << ch << " is a lowercase character." << endl;
        }
        if (isupper(ch))
        {
                cout << ch << " is an uppercase character." << endl;
        }
        if (isspace(ch))
        {
                cout << ch << " is a whitespace character." << endl;
        }

        return 0;
}
```

Character Conversion: toupper

- toupper: if char argument is lowercase letter, return uppercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
char ch3 = '!';

cout << toupper(ch1);  // displays 'H'

cout << toupper(ch2);  // displays 'E'

cout << toupper(ch3);  // displays '!'
```

Character Conversion: tolower

- tolower: if char argument is uppercase letter, return lowercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
char ch3 = '!';

cout << tolower(ch1);  // displays 'h'

cout << tolower(ch2);  // displays 'e'

cout << tolower(ch3);  // displays '!'
```

C-String Fundamentals

- C++ has two different ways of storing and manipulating strings
  - String class
  - Character strings (C-strings): using an array of characters that is terminated by a sentinel value (the escape sequence '\0')
- Character strings can be manipulated using standard element-by-element array-processing techniques
  - cstring class introduced with latest ANSI/ISO standard

C-String Fundamentals (2)

- String literal (string): a sequence of characters enclosed in double quotes

  “This is a string”

- Strings stored as an 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

C-String Fundamentals (3)

- Individual characters in a string array can be input, manipulated, or output using standard array-handling techniques
- Array-handling techniques can use either subscripts or pointers
- The end-of-string NULL character is useful for detecting the end of the string

C-String Input and Output

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

C-String Input and Output Functions

C++ Routine	Description	Example
cin.getline (str, n, ch)	C-string input from the keyboard	cin.getline (str, 81, '\n');
cin.get ()	Character input from the keyboard	nextChar = cin.get ();
cin.peek ()	Return the next character of the input stream without extracting it from the stream	nextPeek = cin.peek ();
cout.put (charExp)	Place the character on the output stream	cout.put ('A');
cin.putback (charExp)	Push a character back onto the input stream	cin.putback (cKey);
cin.ignore (n, char)	Ignore a maximum of the next n input characters, up to and including the detection of char; if no arguments are specified, ignore the next character on the input stream	cin.ignore (80, '\n'); cin.ignore ();

C-String Input and Output Example

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

int main()
{
        const int MAXCHARS = 81;
        char message[MAXCHARS]; // An array large enough to
        // store a complete line

        cout << "Enter a string : " << endl;
        cin.getline(message, MAXCHARS, '\n');

        cout << "The message entered is " << message << endl;
        cin.ignore();

        return 0;
}
```

C-String Input and Output Example Sample Run

```
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.
```

Notes About the C-String Input and Output Example

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

Notes About the C-String Input and Output Example (2)

- 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
- cout object is used to display the C-string

Reasons for Using a string Class Object

- Automatic bounds checking on every index used to access string elements
- The string class automatically expands and contracts storage as needed
- The string class provides a rich set of methods for operating on a string
- Easy to convert to a C-string using c_str()

Reasons for Using C-Strings

- Programmer has ultimate control over how string is stored and manipulated
- Large number of extremely useful functions exist to input, examine, and process C-strings
- C-strings are an excellent way to explore advanced programming techniques using pointers (Chapter 14)
- You will encounter them throughout your programming career, as they are embedded in almost all existing C++ code
- They are fun to program

C-String Processing

- C-strings can be manipulated by using either standard library functions or standard array-processing techniques
  - Library functions presented in the next section
- First look at processing a string in a character-by-character fashion
  - Will allow us to understand how standard library functions are constructed and to create our own library functions
  - Example: strcopy() copies contents of string2 to string1

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;
}
```

Main Features of 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

Character-by-Character Input

- C-strings can be entered and displayed using character-by-character techniques
- We can use cin.get() to accept a string one character at a time
  - Replace cin.getline() function
  - 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 from a few slides back
  - Two modifications necessary before converting to a pointer version…

Possible Modifications of strcopy ()

- Modification 1: eliminate (string2[I] != '\0') test from while statement
  - This statement 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

Pointer and C-String Library Function 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 included with all C++ compilers
- These functions and routines provide for C-string assignment, comparison and other operations

Commonly Used Library Functions: strcpy ()

- 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

Commonly Used Library Functions: strcat ()

- strcat(): appends a string expression onto the end of a C-string variable
  - Example: strcat(dest_string, " there World!")

Commonly Used Library Functions: strlen ()

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

Commonly Used Library Functions: strcmp ()

- 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 found, strings are equal
  - If a difference is found, string with the first lower character is considered smaller string
  - Example: "Hello" is greater than "Goodbye" (first 'H' in Hello greater than first 'G' in Goodbye)

Character Routines

- Character-handling routines: 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 Routines

- Used to convert 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

String Conversion Routines

Prototype	Description	Example
int atoi (stringExp)	Convert an ASCII string to an integer; conversion stops at the first non-integer character	atoi (“1234”);
double atof (stringExp)	Convert an ASCII string to a double-precision number; conversion stops at the first character that cannot be interpreted as a double	atof (“12.34”);
char [] itoa (stringExp)	Convert an integer to an ASCII string; the space allocated for the returned string 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 (2)

- 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 (3)

- 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 (4)

- 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 (5)

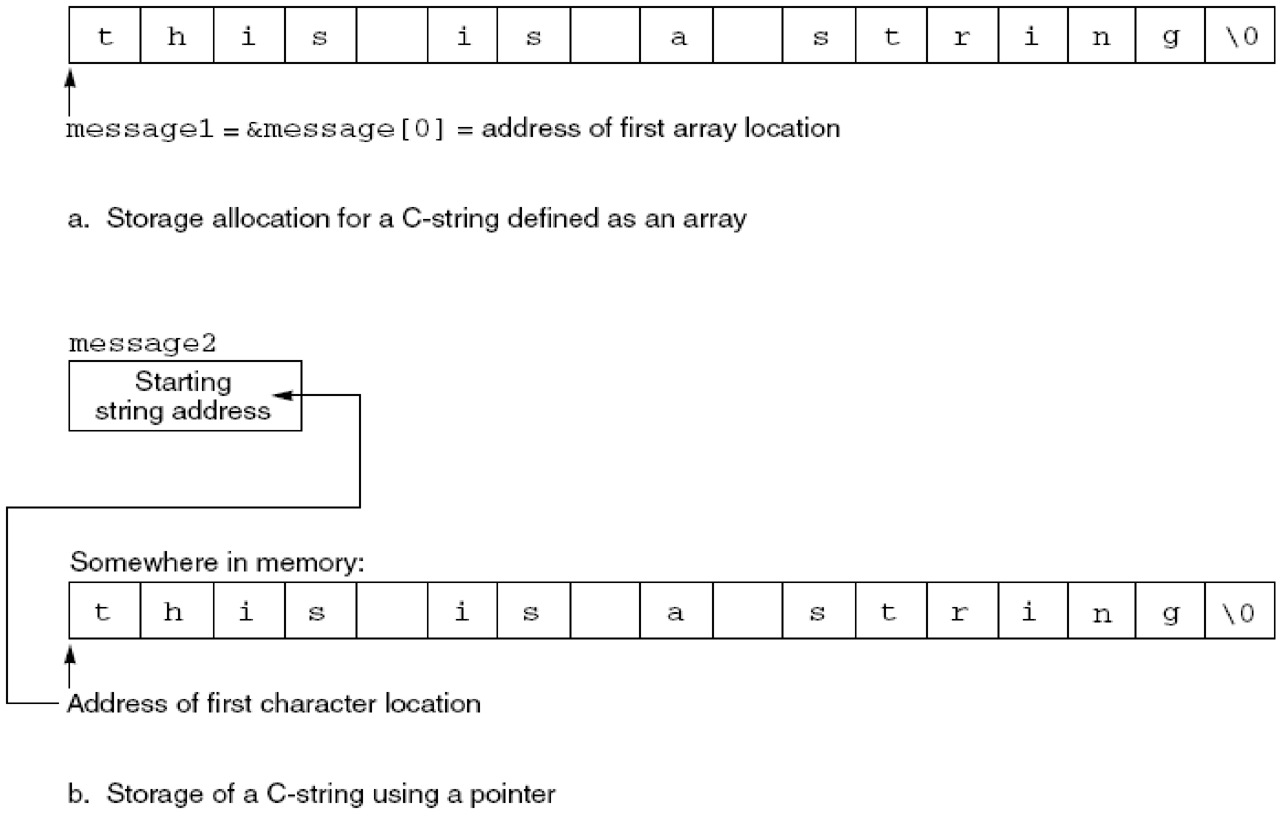
- Allocation of space for message1 different from that for message2
- Both initializations cause computer to store same C-string internally
- message1 storage:
  - Specific set of 80 storage locations reserved; first 17 locations initialized
  - Different C-strings can be stored, but each string overwrites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

C-String Definitions and Pointer Arrays (6)

- 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 allocated to new C-string

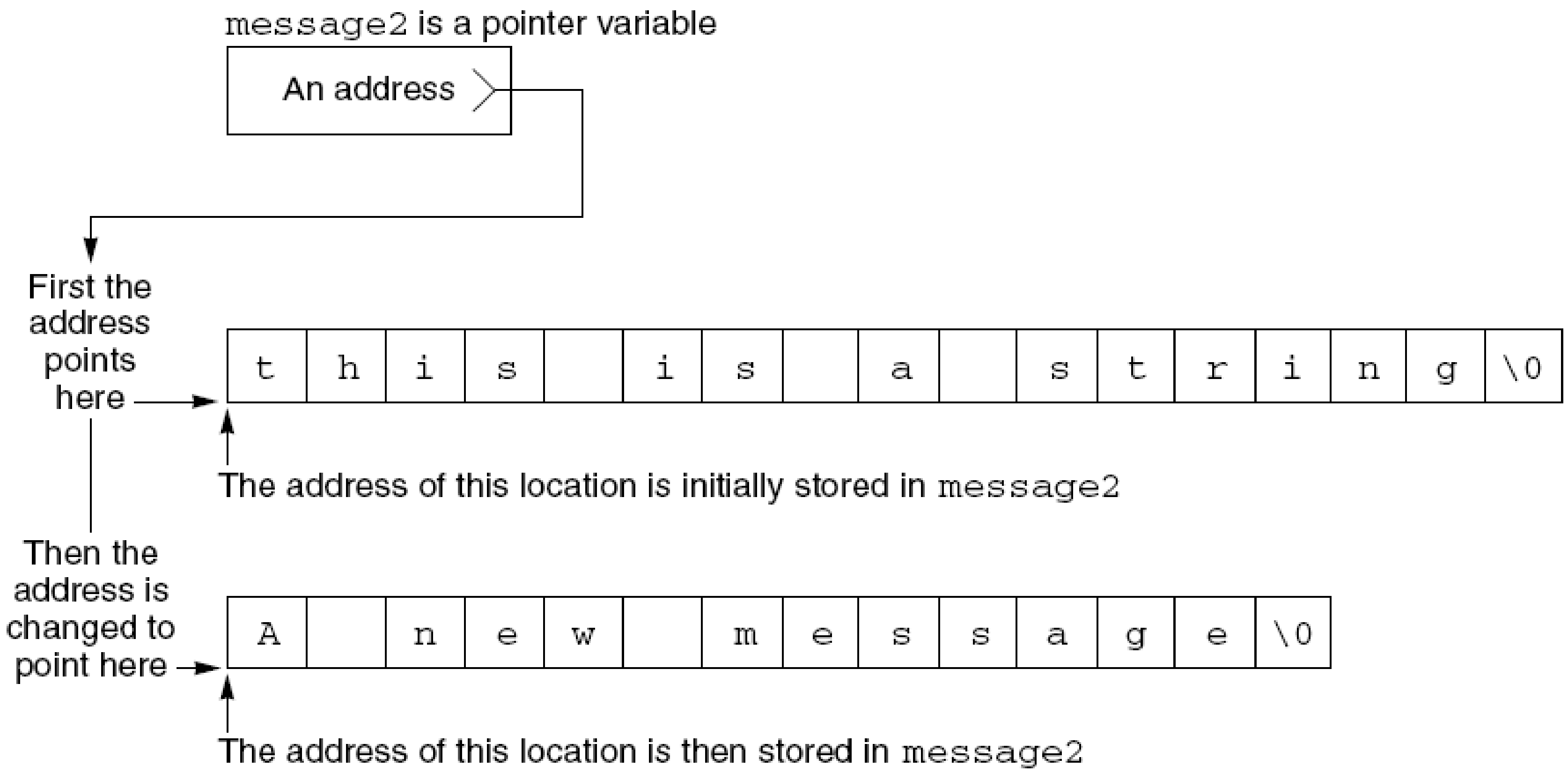
C-String Storage Allocation

FIGURE 10.5 *C-string Storage Allocation*



C-String Storage Allocation (2)

FIGURE 10.6 *Storage Allocation for Figure 10.5*



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

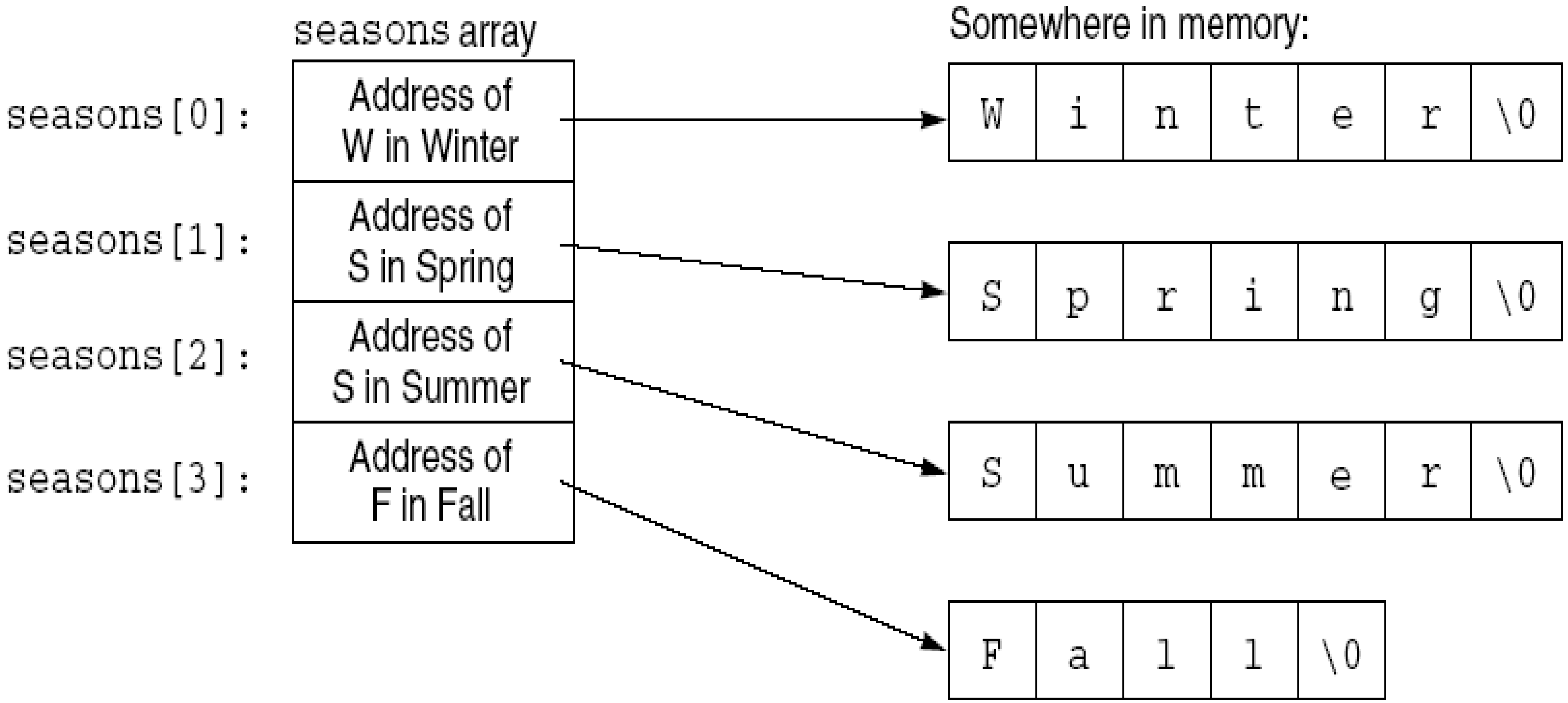
Pointer Arrays (2)

- The seasons array does not contain actual strings assigned to the pointers
  - 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 (3)

FIGURE 10.7 *The Addresses Contained 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 (2)

- 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'};
```