Chapter 5 - Pointers and Strings

Outline

5.1 Introduction
5.2 Pointer Variable Declarations and Initialization
5.3 Pointer Operators
5.4 Call Functions by Reference
5.6 Bubble Sort Using Pass-by-Reference
5.7 Pointer Expressions and Pointer Arithmetic
5.8 Relationship Between Pointers and Arrays
5.12 Introduction to Character and String Processing

5.1 Introduction

• Pointers
  – Powerful, but difficult to master
  – Simulate pass-by-reference
  – Close relationship with arrays and strings

5.2 Pointer Variable Declarations and Initialization

• Pointer variables
  – Contain memory addresses as values
  – Normally, variable contains specific value (direct reference)
  – Pointers contain address of variable that has specific value
  – Indirection
    – Referencing value through pointer

• Pointer declarations
  – * indicates variable is pointer
  – int *myPtr;
  – declares pointer to int, pointer of type int *
  – Multiple pointers require multiple asterisks
  – int *myPtr1, *myPtr2;

• Can declare pointers to any data type

• Pointer initialization
  – Initialized to 0, NULL, or address
  – 0 or NULL points to nothing

5.3 Pointer Operators

• & (address operator)
  – Returns memory address of its operand
  – Example
  – int y = 5;
  – int *yPtr;
  – yPtr = &y; // yPtr gets address of y
  – yPtr “points to” y

• * (indirection/dereferencing operator)
  – Returns synonym for object its pointer operand points to
  – *yPtr returns y (because yPtr points to y).
  – Dereferenced pointer is lvalue
  – *yptr = 9; // assigns 9 to y
  – * and & are inverses of each other
5.4 Calling Functions by Reference

- 3 ways to pass arguments to function
  - Pass-by-value
  - Pass-by-reference with reference arguments
  - Pass-by-reference with pointer arguments
- return can return one value from function
- Arguments passed to function using reference arguments
  - Modify original values of arguments
  - More than one value “returned”
### 5.6 Bubble Sort Using Pass-by-Reference

- **Implement** `bubbleSort` **using pointers**
  - Want function `swap` to access array elements
    - Individual array elements: scalars
      - Passed by value by default
    - Pass by reference using address operator `&`

```cpp
void swap( int * const element1Ptr, int * const element2Ptr )
// element1Ptr and element2Ptr point to items in different memory locations
// swap values at memory locations to which they point

int number = 5;

// pass address of number to cubeByReference
void cubeByReference( int *nPtr )
// calculate cube of *nPtr; modifies variable number in main

int main( )
// with a pointer argument.
// Cube a variable using pass-by-reference
// Fig. 5.7: fig05_07.cpp
{
    int arraySize = 10;
    int *array = new int [ arraySize ];
    int number = 5;
    for ( int i = 0; i < arraySize; i++ )
    {
        array [ i ] = number;
    } // end for

    bubbleSort( array, arraySize );  // sort the array

    for ( int j = 0; j < arraySize; j++ )
    {
        cout << setw( 4 ) << array [ j ];
    } // end for

    cout << endl;

    cout << "Data items in ascending order
";
    for ( int j = 0; j < arraySize; j++ )
    {
        cout << setw( 4 ) << array [ j ];
    } // end for

    cout << endl;

    cout << "Data items in original order
";
    for ( int j = 0; j < arraySize; j++ )
    {
        cout << setw( 4 ) << array [ j ];
    } // end for

    cout << "The original value of number is " << number;

    cubeByReference( &number );  // pass address of number to cubeByReference
    cout << "The new value of number is " << number << endl;

    return 0;  // indicates successful termination
} // end function main
```

5.7 Pointer Expressions and Pointer Arithmetic

- Pointer arithmetic
  - Increment/decrement pointer (+ or --)
  - Add/subtract an integer to/from a pointer (+ or +=, - or -=)
  - Pointers may be subtracted from each other
  - Pointer arithmetic meaningless unless performed on pointer to array

- 5 element int array on a machine using 4 byte ints
  - vPtr points to first element v[0], which is at location 3000
  - vPtr = 3000
  - vPtr += 2; sets vPtr to 3008
  - vPtr points to v[2]

- Subtracting pointers
  - Returns number of elements between two addresses

- Pointer assignment
  - Pointer can be assigned to another pointer if both of same type
  - If not same type, cast operator must be used
  - Exception: pointer to void (type void *)
  - Generic pointer, represents any type
  - No casting needed to convert pointer to void pointer
  - void pointers cannot be dereferenced

5.8 Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like constant pointer
  - Pointers can do array subscripting operations

- Accessing array elements with pointers
  - Element b[n] can be accessed by *(bPtr + n)
  - Called pointer/offset notation
  - Addresses
    - *b[3] same as bPtr + 3
    - Array name can be treated as pointer
      - b[3] same as *(b + 3)
    - Pointers can be subscripted (pointer/subscript notation)
      - bPtr[3] same as b[3]

Outline

fig05_20.cpp (1 of 2)

```cpp
#include <iostream>

using std::cout;
using std::endl;

int main()
{

    int b[] = { 10, 20, 30, 40 };
    int *bPtr = b; // set bPtr to point to array b

    // output array b using array subscript notation
    cout << "Array b printed with:
";
    for ( int i = 0; i < 4; i++ )
        cout << "b[" << i << "] = " << b[i] << endl;

    // output array b using the array name and
    // pointer/offset notation
    cout << "Pointer/offset notation where the pointer is the array name
";
    for ( int offset1 = 0; offset1 < 4; offset1++ )
        cout << "*(b + " << offset1 << ") = " << *(b + offset1) << endl;

    // output array b using bPtr and array subscript notation
    cout << "Pointer subscript notation
";
    for ( int j = 0; j < 4; j++ )
        cout << "bPtr[" << j << "] = " << bPtr[j] << endl;

    // output array b using bPtr and pointer/offset notation
    cout << "Pointer/offset notation
";
    for ( int offset2 = 0; offset2 < 4; offset2++ )
        cout << "*(bPtr + " << offset2 << ") = " << *(bPtr + offset2) << endl;

    return 0;  // indicates successful termination
} // end main
```

fig05_20.cpp (2 of 2)

Using array subscript notation.
Using pointer subscript notation.
Using bPtr and pointer/offset notation.
5.12.1 Fundamentals of Characters and Strings

- String assignment
  - Character array
    - char color[] = "blue";
    - Creates 5 element char array color
  - Variable of type char *
    - char *colorPtr = "blue";
    - Creates pointer colorPtr to letter b in string "blue"
  - Alternative for character array
    - char color[] = { 'b', 'l', 'u', 'e', '\0' };

- Character constant
  - Integer value represented as character in single quotes
  - 'z' is integer value of z
  - 122 in ASCII

- String
  - Series of characters treated as single unit
  - Can include letters, digits, special characters +, -, * ...
  - String literal (string constants)
  - Enclosed in double quotes, for example:
    "I like C++"
  - Array of characters, ends with null character '\0'
  - String is constant pointer
  - Pointer to string’s first character
  - Like arrays

- Reading strings
  - Assign input to character array word[ 20 ]
    - cin >> word
  - Reads characters until whitespace or EOF
  - String could exceed array size
    - cin >> setw( 20 ) >> word;
    - Reads 19 characters (space reserved for '\0' )
5.12.1 Fundamentals of Characters and Strings

- `cin.getline`  
  - Read line of text  
  - `cin.getline( array, size, delimiter );`  
  - Copies input into specified `array` until either  
    - One less than `size` is reached  
    - `delimiter` character is input  
  - Example  
    ```
    char sentence[ 80 ];
    cin.getline( sentence, 80, '\n' );
    ```

5.12.2 String Manipulation Functions of the String-handling Library

- Copying strings  
  - `char *strcpy( char *s1, const char *s2 );`  
    - Copies second argument into first argument  
    - First argument must be large enough to store string and terminating null character  
  - `char *strncpy( char *s1, const char *s2, size_t n );`  
    - Specifies number of characters to be copied from string into array  
    - Does not necessarily copy terminating null character  
  - `char *strcat( char *s1, const char *s2 );`  
    - Appends the string `s2` to the string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.  
  - `char *strncat( char *s1, const char *s2, size_t n );`  
    - Appends at most `n` characters of string `s2` to string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.  
  - `int strcmp( const char *s1, const char *s2 );`  
    - Compares the string `s1` with the string `s2`. The function returns a value of zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively.  
  - `int strncmp( const char *s1, const char *s2, size_t n );`  
    - Compares up to `n` characters of the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively.  
  - `size_t strlen( const char *s );`  
    - Determines the length of string `s`. The number of characters preceding the terminating null character is returned.  

```
#include <iostream>
#include <cstring>   // prototypes for strcpy and strncpy

int main()
{
    char x[] = "Happy Birthday to You";
    char y[ 25 ];
    char z[ 15 ];

    strcpy( y, x );  // copy contents of x into y
    cout << "The string in array x is: " << x << "The string in array y is: " << y << endl;

    // copy first 14 characters of x into z
    strncpy( z, x, 14 );  // does not copy null character
    z[ 14 ] = '\0';       // append null character
    cout << "The string in array z is: " << z << endl;
}
```
5.12.2 String Manipulation Functions of the String-handling Library

- Comparing strings
  - Characters represented as numeric codes
  - Strings compared using numeric codes
  - Character codes / character sets
    - **ASCII**
      - “American Standard Code for Information Interchange”
    - **EBCDIC**
      - “Extended Binary Coded Decimal Interchange Code”

- Concatenating strings
  - `char *strcat( char *s1, const char *s2 )`
    - Appends second argument to first argument
    - First character of second argument replaces null character terminating first argument
    - Ensure first argument large enough to store concatenated result and null character
  - `char *strncat( char *s1, const char *s2, size_t n )`
    - Appends specified number of characters from second argument to first argument
    - Appends terminating null character to result

- Comparing strings
  - `int strcmp( const char *s1, const char *s2 )`
    - Compares character by character
    - Returns
      - Zero if strings equal
      - Negative value if first string less than second string
      - Positive value if first string greater than second string
  - `int strncmp( const char *s1, const char *s2, size_t n )`
    - Compares up to specified number of characters
    - Stops comparing if reaches null character in one of arguments
5.12.2 String Manipulation Functions of the String-handling Library

- Determining string lengths
  - `size_t strlen( const char *s )`
    - Returns number of characters in string
      - Terminating null character not included in length