Chapter 5 - Pointers and Strings

Outline
5.1 Introduction
5.2 Pointer Variable Declarations and Initialization
5.3 Pointer Operators
5.4 Calling Functions by Reference
5.5 Using const with Pointers
5.6 Bubble Sort Using Pass-by-Reference
5.7 Pointer Expressions and Pointer Arithmetic
5.8 Relationship Between Pointers and Arrays
5.9 Arrays of Pointers
5.10 Case Study: Card Shuffling and Dealing Simulation
5.11 Function Pointers
5.12 Introduction to Character and String Processing
  5.12.1 Fundamentals of Characters and Strings
  5.12.2 String Manipulation Functions of the String-Handling Library

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 (indirect reference)

- 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;
    ```
5.3 Pointer Operators

- \& (address operator)
  - Returns memory address of its operand
  - Example
    
    \[
    \begin{align*}
    \text{int } y &= 5; \\
    \text{int } &yPtr; \\
    yPtr &= &y; &// yPtr gets address of \ y \\
    \end{align*}
    \]
  - \yPtr \ "points to" \ y

- * (indirection/dereferencing operator)
  - Returns synonym for object its pointer operand points to
  - Example
    
    \[
    \begin{align*}
    \text{int } y &= 5; \\
    \text{int } *yPtr; \\
    yPtr &= &y; &// yPtr gets address of \ y \\
    \end{align*}
    \]
  - 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”

```
int cubeByValue( int ); // prototype

int main()
{
    int number = 5;
    number = cubeByValue( number );
    cout << "The new value of number is " << number << endl;
    return 0;  // indicates successful termination
}
```

Pass number by value; result returned by cubeByValue

5.4 Calling Functions by Reference

- Pass-by-reference with pointer arguments
  - Simulate pass-by-reference
    - Use pointers and indirection operator
    - Pass address of argument using & operator
  - Arrays not passed with & because array name already pointer
  - * operator used as alias/nickname for variable inside of function

```
int cubeByValue( int ); // calculate and return cube of integer argument

int main()
{
    int number = 5;
    number = cubeByValue( number );
    cout << "The new value of number is " << number << endl;
    return 0;  // indicates successful termination
}
```

Cube and return local variable n
5.5 Using const with Pointers

- **const** qualifier
  - Value of variable should not be modified
  - **const** used when function does not need to change a variable

- Principle of least privilege
  - Award function enough access to accomplish task, but no more

- Four ways to pass pointer to function
  - Nonconstant pointer to nonconstant data
    - Highest amount of access
  - Nonconstant pointer to constant data
  - Constant pointer to nonconstant data
  - Constant pointer to constant data
    - Least amount of access
// convert string to uppercase letters
void convertToUppercase( char *sPtr )
{
while ( *sPtr != '\0' )  // current character is not '\0'
    if ( islower( *sPtr ) )  // if character is lowercase,
        *sPtr = toupper( *sPtr );  // convert to uppercase
    ++sPtr;  // move sPtr to next character in string
}

The phrase before conversion is: characters and $32.98
The phrase after conversion is: CHARACTERS AND $32.98

Parameter sPtr nonconstant pointer to nonconstant data.
Function islower returns true if character is lowercase.
Function toupper returns corresponding uppercase character if original character lowercase; otherwise toupper returns original (uppercase) character.

// Fig. 5.11: fig05_11.cpp
// Printing a string one character at a time using a non-constant pointer to constant data.
#include <iostream>
using std::cout;
using std::endl;

void printCharacters( const char * );

int main()
{
    char phrase[] = "print characters of a string";

    cout << "The string is:
";
    printCharacters( phrase );
    cout << endl;

    return 0;  // indicates successful termination
}

void printCharacters( const char *sPtr )
{
    for ( ; *sPtr != '\0'; sPtr++ )   // no initialization
        cout << *sPtr;
}

The string is:
print characters of a string

Parameter is nonconstant pointer to constant data.
Pass pointer phrase to function printCharacters.

// Fig. 5.12: fig05_12.cpp
// Attempting to modify data through a non-constant pointer to constant data.
void f( const int * );  // prototype

int main()
{
    int y;

    f( &y );   // f attempts illegal modification

    return 0;  // indicates successful termination
}

void f( const int *xPtr )
{
    *xPtr = 100;  // error: cannot modify a const object
}

d:\cpphtp4_examples\ch05\Fig05_12.cpp(21) : error C2166 : l-value specifies const object
5.5 Using `const` with Pointers

- **`const` pointers**
  - Always point to same memory location
  - Default for array name
  - Must be initialized when declared

```cpp
// Fig. 5.13: fig05_13.cpp
// Attempting to modify a constant pointer to non-constant data.

#include <iostream>

using namespace std;

int main()
{
    int x, y;

    int * const ptr = &x;

    *ptr = 7; // allowed: *ptr is not const

    ptr = &y;  // error: ptr is const; cannot assign new address

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

This code attempts to modify a constant pointer to non-constant data, which is not allowed. The output is as follows:

```
1 d:\cpphtp4_examples\ch05\Fig05_13.cpp(15) : error C2166: l-value specifies const object
ptr is constant pointer to integer.
Can modify x (pointed to by ptr) since x not constant.
Cannot modify ptr to point to new address since ptr is constant.
Line 15 generates compiler error by attempting to assign new address to constant pointer.
```

```cpp
// Fig. 5.14: fig05_14.cpp
// Attempting to modify a constant pointer to constant data.

#include <iostream>

using namespace std;

int main()
{
    int x = 5, y;

    const int *const ptr = &x;

    *ptr = 7;  // error: *ptr is const; cannot assign new value

    ptr = &y;  // error: ptr is const; cannot assign new address

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

This code attempts to modify a constant pointer to a constant integer, which is also not allowed. The output is as follows:

```
1 d:\cpphtp4_examples\ch05\Fig05_14.cpp(19) : error C2166: l-value specifies const object
Line 19 generates compiler error by attempting to modify constant object.
2 d:\cpphtp4_examples\ch05\Fig05_14.cpp(20) : error C2166: l-value specifies const object
Line 20 generates compiler error by attempting to assign new address to constant pointer.
```
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 &

```
// Fig. 5.15: fig05_15.cpp
// This program puts values into an array, sorts the values into
// ascending order, and prints the resulting array.
#include <iostream>
#include <iomanip>

void bubbleSort( int * const, int );   // prototype
void swap( int * const, int * const );  // prototype

int main()
{
    const int arraySize = 10;
    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

    cout << "Data items in original order
";
    for ( int i = 0; i < arraySize; i++ )
        cout << setw( 4 ) << a[ i ];

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

    cout << "Data items in ascending order
";
    for ( int j = 0; j < arraySize; j++ )
        cout << setw( 4 ) << a[ j ];

    cout << endl;
    return 0;  // indicates successful termination
}

// sort an array of integers using bubble sort
void bubbleSort( int *array, const int size )
{
    // loop to control passes
    for ( int pass = 0; pass < size - 1; pass++ )
        // loop to control comparisons during each pass
        for ( int k = 0; k < size - 1; k++ )
            // swap adjacent elements if they are out of order
            if ( array[ k ] > array[ k + 1 ] )
                swap( array[ k ], array[ k + 1 ] );
}

// swap values at memory locations to which
element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}
```

Pass arguments by reference, allowing function to swap values at memory locations.
5.6 Bubble Sort Using Pass-by-Reference

- **sizeof**
  - Unary operator returns size of operand in bytes
  - For arrays, `sizeof` returns
    - (size of 1 element) * (number of elements)
  - If `sizeof(int) = 4`, then
    int myArray[10];
    cout << sizeof(myArray);
  will print 40

- **sizeof** can be used with
  - Variable names
  - Type names
  - Constant values

---

```cpp
// Fig. 5.16: fig05_16.cpp
// Sizeof operator when used on an array name
// returns the number of bytes in the array.
#include <iostream>
using std::cout;
using std::endl;

size_t getSize( double * );  // prototype

int main()
{
    double array[ 20 ];

    cout << "The number of bytes in the array is "
         << sizeof( array ) << endl;

    cout << "The number of bytes returned by getSize is "
         << getSize( array ) << endl;

    return 0;  // indicates successful termination
}

size_t getSize( double *ptr )
{
    return sizeof( ptr );
}
```

The number of bytes in the array is 160
The number of bytes returned by getSize is 4

---

```cpp
// Fig. 5.17: fig05_17.cpp
// Demonstrating the sizeof operator.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    char c;
    short s;
    int i;
    long l;
    float f;
    double d;
    long double ld;
    int array[ 20 ];
    int *ptr = array;
    ```
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
    - vPtr2 - vPtr == 2
- 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
5.7 Pointer Expressions and Pointer Arithmetic

- Pointer comparison
  - Use equality and relational operators
  - Comparisons meaningless unless pointers point to members of same array
  - Compare addresses stored in pointers
  - Example: could show that one pointer points to higher numbered element of array than other pointer
  - Common use to determine whether pointer is 0 (does not point to anything)

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 \( \text{b}[n] \) can be accessed by \( *\text{(bPtr + n)} \)
  - Called pointer/offset notation
  - Addresses
    - \( \text{bPtr}[3] \) same as \( *\text{bPtr} + 3 \)
    - Array name can be treated as pointer
      - \( \text{b}[3] \) same as \( *\text{b} + 3 \)
      - Pointers can be subscripted (pointer/subscript notation)
        - \( \text{bPtr}[3] \) same as \( \text{b}[3] \)

Outline

// Fig. 5.20: fig05_20.cpp
// Using subscripting and pointer notations with arrays.

#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:
" << "Array subscript notation
";
  for ( int i = 0; i < 4; i++ )
    cout << "b[" << i << "] = " << b[i] << \\
         "\n";

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

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

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

  return 0;  // indicates successful termination
} // end main
**5.9 Arrays of Pointers**

Arrays can contain pointers

- Commonly used to store array of strings
  ```c
  ```
- Each element of `suit` points to `char *` (a string)
- Array does not store strings, only pointers to strings

- `suit` array has fixed size, but strings can be of any size
5.10 Case Study: Card Shuffling and Dealing Simulation

- Card shuffling program
  - Use an array of pointers to strings, to store suit names
  - Use a double scripted array (suit by value)

  ```
  Deck[2][12]
  ```

- Place 1-52 into the array to specify the order in which the cards are dealt

- Place 1-52 into the array to specify the order in which the cards are dealt

Pseudocode for shuffling and dealing simulation

```c
for each of the 52 cards
  place card number in randomly selected unoccupied slot of deck
for each of the 52 cards
  find card number in deck array and print face and suit of card
```

Outline

```c
// Fig. 5.24: fig05_24.cpp
// Card shuffling dealing program.
#include <iostream>
#include <iomanip>
#include <cstdlib>  // prototypes for rand and srand
#include <ctime>    // prototype for time

// prototypes
void shuffle( int [ ] [ 13 ] );
void deal( const int [ ] [ 13 ], const char [ ] [ ], const char [ ] [ ] );

int main()
{
  // initialize suit array

  // initialize face array

  // initialize deck array
  int deck[ 4 ][ 13 ] = { 0 };

  srand( time( 0 ) );        // seed random number generator
  shuffle( deck );
  deal( deck, face, suit );
  return 0;  // indicates successful termination
}
```

Outline
// shuffle cards in deck
void shuffle( int wDeck[][ 13 ] )
{
    int row;
    int column;
    // for each of the 52 cards, choose slot of deck randomly
    for ( int card = 1; card <= 52; card++ ) {
        do {
            row = rand() % 4;
            column = rand() % 13;
        } while ( wDeck[ row ][ column ] != 0 );
        // place card number in chosen slot of deck
        wDeck[ row ][ column ] = card;
    } // end for
} // end function shuffle

// deal cards in deck
void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] )
{
    // for each of the 52 cards
    for ( int card = 1; card <= 52; card++ )
        // loop through rows of wDeck
        for ( int row = 0; row <= 3; row++ )
            // loop through columns of wDeck for current row
            for ( int column = 0; column <= 12; column++ )
                // if slot contains current card, display card
                if ( wDeck[ row ][ column ] == card ) {
                    cout << setw( 5 ) << right << wFace[ column ]
                         << " of " << setw( 8 ) << left
                         << wSuit[ row ]
                         << ( card % 2 == 0 ? '
                             : '	' );
                } // end if
    } // end function deal

5.11 Function Pointers
• Pointers to functions
  – Contain address of function
  – Similar to how array name is address of first element
  – Function name is starting address of code that defines function
• Function pointers can be
  – Passed to functions
  – Returned from functions
  – Stored in arrays
  – Assigned to other function pointers
5.11 Function Pointers

- Calling functions using pointers
  - Assume parameter:
    - `bool (*compare) (int, int)`
  - Execute function with either
    - `(compare)(int1, int2)`
    - Dereference pointer to function to execute
    - `compare(int1, int2)`
    - Could be confusing
      - User may think `compare` name of actual function in program

```
#include <iostream>

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

#include <iomanip>

using std::setw;

// prototypes
void bubble(int[], const int, bool (*)(int, int));
void swap(int* const, int* const);
bool ascending(int, int);
bool descending(int, int);

int main()
{
  const int arraySize = 10;
  int order;
  int counter;
  int a[arraySize] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

  cout << "Enter 1 to sort in ascending order,
       Enter 2 to sort in descending order: ";
  cin >> order;
  cout << 
  // output original array
  for (counter = 0; counter < arraySize; counter++)
    cout << setw(4) << a[counter];

  // sort array in ascending order; pass function ascending
  // as an argument to specify ascending sorting order
  if (order == 1) {
    bubble(a, arraySize, ascending);
    cout << "Data items in ascending order
    for (counter = 0; counter < arraySize; counter++)
      cout << setw(4) << a[counter];
  }

  // sort array in descending order; pass function descending
  // as an argument to specify descending sorting order
  else {
    bubble(a, arraySize, descending);
    cout << "Data items in descending order
    for (counter = 0; counter < arraySize; counter++)
      cout << setw(4) << a[counter];
  }
  cout << endl;
  return 0;  // indicates successful termination
}
```

```cpp
// multipurpose bubble sort; parameter compare is a pointer to
// the comparison function that determines sorting order
void bubble(int work[], const int size, bool (*compare)(int, int))
{
  // loop to control passes
  for (int pass = 1; pass < size; pass++)
    // loop to control number of
    // adjacent elements to check
    for (int count = 0; count < size - 1; count++)
      if (*compare(work[count], work[count + 1]))
        swap(&work[count], &work[count + 1]);
}
```
5.11 Function Pointers

- Arrays of pointers to functions
  - Menu-driven systems
  - Pointers to each function stored in array of pointers to functions
    - All functions must have same return type and same parameter types
  - Menu choice subscript into array of function pointers
Call chosen function by dereferencing corresponding element in array.

5.12.1 Fundamentals of Characters and Strings

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

5.12.1 Fundamentals of Characters and Strings

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

```cpp
// process user's choice
while ( choice >= 0 && choice < 3 ) {
   // invoke function at location choice in array f
   // and pass choice as an argument
   (*f[ choice ])( choice );
   cout << "Enter a number between 0 and 2, 3 to end: ";
   cin >> choice;
}

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

void function1( int a )
{
   cout << "You entered " << a << " so function1 was called


   return 0;  // end main

void function2( int b )
{
   cout << "You entered " << b << " so function2 was called


   return 0;  // end main

void function3( int c )
{
   cout << "You entered " << c << " so function3 was called


   return 0;  // end main
```

Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
5.12.1 Fundamentals of Characters and Strings

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

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

5.12.2 String Manipulation Functions of the String-handling Library

- String handling library `<cstring>` provides functions to
  - Manipulate string data
  - Compare strings
  - Search strings for characters and other strings
  - Tokenize strings (separate strings into logical pieces)

```
char *strcpy( char *s1, const char *s2 );
```
Copies the string `s2` into the character array `s1`. The value of `s1` is returned.

```
char *strncpy( char *s1, const char *s2, size_t n );
```
Copies at most `n` characters of the string `s2` into the character array `s1`. The value of `s1` is returned.

```
char *strncat( char *s1, const char *s2, size_t n );
```
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 *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.

```
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.
5.12.2 String Manipulation Functions of the String-handling Library

- **strlen( const char *s );**
  - Determines the length of string `s`. The number of characters preceding the terminating null character is returned.

- **strtok( char *s1, const char *s2 );**
  - A sequence of calls to `strtok` breaks string `s1` into "tokens"—logical pieces such as words in a line of text—delimited by characters contained in string `s2`. The first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, `NULL` is returned.

- **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.

### Copying strings

- **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

- **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

### Example Code

```cpp
#include <iostream>
#include <cstring>

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 << "\n";
    cout << "The string in array y is: " << y << \n";

    // 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;

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

Output:

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```
5.12.2 String Manipulation Functions of the String-handling Library

- 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

```
#include <cstring>  // prototypes for strcat and strncat

int main()
{
    char s1[20] = "Happy ";
    char s2[] = " New Year ";
    char s3[40] = " ";

    cout << "s1 = " << s1 << " s2 = " << s2;  

    strcat( s1, s2 );  // concatenate s2 to s1
    cout << "

    After strcat(s1, s2):
    s1 = " << s1 << " s2 = " << s2;

    // concatenate first 6 characters of s1 to s3
    strncat( s3, s1, 6 );  // places '\0' after last character
    cout << "

    After strncat(s3, s1, 6):
    s1 = " << s1 << " s3 = " << s3;

    strcat( s3, s1 );  // concatenate s1 to s3
    cout << "

    After strcat(s3, s1):
    s1 = " << s1 << " s3 = " << s3 << endl;

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

- 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”
5.12.2 String Manipulation Functions of the String-handling Library

• 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

• Tokenizing
  - Breaking strings into tokens, separated by delimiting characters
  - Tokens usually logical units, such as words (separated by spaces)
  - "This is my string" has 4 word tokens (separated by spaces)
  - char *strtok( char *s1, const char *s2 )
    • Multiple calls required
      - First call contains two arguments, string to be tokenized and string containing delimiting characters
      - Finds next delimiting character and replaces with null character
      - Subsequent calls continue tokenizing
    • Call with first argument NULL
// Fig. 5.31: fig05_31.cpp
// Using strtok.

#include <iostream>

#include <cstring>  // prototype for strtok

int main()
{
    char sentence[] = "This is a sentence with 7 tokens";
    char *tokenPtr;

    cout << "The string to be tokenized is:
" << sentence
     << "The tokens are:
"
     << "This" << "is" << "a" << "sentence" << "with" << "7" << "tokens"
     << "After strtok, sentence = This";

    tokenPtr = strtok( sentence, " ");
    cout << "First call to strtok begins tokenization."
    << tokenPtr << " 
"
    tokenPtr = strtok( NULL, " "); // get next token
    cout << "Subsequent call to strtok with NULL as first argument to indicate continuation.";

    while ( tokenPtr != NULL )
    {
        cout << tokenPtr << " |
";
        tokenPtr = strtok( NULL, " "); // get next token
    }

    cout << "After strtok, sentence = This"
     << sentence << endl;

    return 0;  // indicates successful termination
}

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
// Fig. 5.32: fig05_32.cpp
// Using strlen.
#include <iostream>
#include <cstring> // prototype for strlen
using std::cout;
using std::endl;

int main()
{
    char *string1 = "abcdefghijklmnopqrstuvwxyz";
    char *string2 = "four";
    char *string3 = "Boston";

    cout << "The length of " << string1 << " is " << strlen(string1) << "\n";
    cout << "The length of " << string2 << " is " << strlen(string2) << "\n";
    cout << "The length of " << string3 << " is " << strlen(string3) << endl;

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