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
    ```cpp
    int *myPtr;
    ```
    declares pointer to `int`, pointer of type `int *`
  - Multiple pointers require multiple asterisks
    ```cpp
    int *myPtr1, *myPtr2;
    ```
5.2 Pointer Variable Declarations and Initialization

- 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`
5.3 Pointer Operators

• \* (indirection/dereferencing operator)
  – Returns synonym for object its pointer operand points to
  – \*y\_Ptr returns y (because y\_Ptr points to y).
  – dereferenced pointer is lvalue
    \*yptr = 9;       // assigns 9 to y

• \* and \& are inverses of each other
// Fig. 5.4: fig05_04.cpp
// Using the & and * operators.
#include <iostream>

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

int main()
{
    int a;     // a is an integer
    int *aPtr; // aPtr is a pointer to an integer
    a = 7;
aPtr = &a;  // aPtr assigned address of a

    cout << "The address of a is " << &a
    << "\nThe value of aPtr is " << aPtr;

    cout << "\n\nThe value of a is " << a
    << "\nThe value of *aPtr is " << *aPtr;

    cout << "\nShowing that * and & are inverses of each other.\n&*aPtr = " << &*aPtr
    << "\n*&aPtr = " << *&aPtr << endl;

* and & are inverses of each other
The address of a is 0012FED4
The value of aPtr is 0012FED4

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012FED4
*&aPtr = 0012FED4

* and & are inverses; same result when both applied to aPtr

return 0;  // indicates successful termination

}  // end main
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”
5.4 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
// Fig. 5.6: fig05_06.cpp
// Cube a variable using pass-by-value.
#include <iostream>

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

int cubeByValue( int ); // prototype

int main()
{
    int number = 5;

    cout << "The original value of number is " << number << endl;
    // pass number by value to cubeByValue
    number = cubeByValue( number );

    cout << "\nThe new value of number is " << number << endl;
    return 0; // indicates successful termination
}
// end main
// calculate and return cube of integer argument
int cubeByValue(int n)
{
    return n * n * n; // cube local variable n
}
// end function cubeByValue

The original value of number is 5
The new value of number is 125
// Fig. 5.7: fig05_07.cpp
// Cube a variable using pass-by-reference
// with a pointer argument.
#include <iostream>

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

void cubeByReference( int * );    // prototype

int main()
{
    int number = 5;

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

    // pass address of number to cubeByReference
    cubeByReference( &number );

    cout << "\nThe new value of number is " << number << endl;

    return 0;    // indicates successful termination
}  // end main
// calculate cube of *nPtr; modifies variable number in main
cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
// end function cubeByReference

The original value of number is 5
The new value of number is 125

Modify and access int variable using indirection operator *
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
// Fig. 5.10: fig05_10.cpp
// Converting lowercase letters to uppercase letters
// using a non-constant pointer to non-constant data.
#include <iostream>

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

#include <cctype>  // prototypes for islower and toupper

void convertToUppercase( char * );

int main()
{
    char phrase[] = "characters and $32.98";
    cout << "The phrase before conversion is: " << phrase;
    convertToUppercase( phrase );
    cout << "\nThe phrase after conversion is: "
         << phrase << endl;
    return 0;  // indicates successful termination
}  // end main
// 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
    } // end while
} // end function convertToUppercase

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
When operator ++ applied to
pointer that points to array,
memory address stored in
pointer modified to point to
next element of array.
// 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:\n";
    printCharacters( phrase );
    cout << endl;
    return 0; // indicates successful termination
} // end main
23 // sPtr cannot modify the character to which it points, 
24 // i.e., sPtr is a "read-only" pointer
25 void printCharacters( const char *sPtr )
26 {
27     for ( ; *sPtr != '\0'; sPtr++) // no initialization
28         cout << *sPtr;
29 }
30 } // end function printCharacters

The string is:
print characters of a string
// 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
}

// xPtr cannot modify the value to which it points
void f( const int *xPtr )
{
    *xPtr = 100; // error: cannot modify a const object
}

// end function f

// Parameter is nonconstant pointer to constant data.

// Pass address of int variable y to attempt illegal modification.

// Attempt to modify const object pointed to by xPtr.

// Error produced when attempting to compile.
5.5 Using \texttt{const} with Pointers

- \texttt{const} pointers
  - Always point to same memory location
  - Default for array name
  - Must be initialized when declared
// Fig. 5.13: fig05_13.cpp
// Attempting to modify a constant pointer to
// non-constant data.

ing main()
{
    int x, y;

    // ptr is a constant pointer to an integer that can
    // be modified through ptr, but ptr always points to the
    // same memory location.
    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
} // end main
// Fig. 5.14: fig05_14.cpp
// Attempting to modify a constant pointer to constant data.
#include <iostream>

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

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

    // ptr is a constant pointer to a constant integer.
    // ptr always points to the same location; the integer
    // at that location cannot be modified.
    const int *const ptr = &x;

    cout << *ptr << endl;

    *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
} // end main
Line 19 generates compiler error by attempting to modify constant 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>

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

#include <iomanip>

using std::setw;

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\n";

    for ( int i = 0; i < arraySize; i++ )
        cout << setw( 4 ) << a[ i ];
bubbleSort ( a, arraySize );  // sort the array

cout << "\nData items in ascending order\n";

for ( int j = 0; j < arraySize; j++ )
    cout << setw( 4 ) << a[ j ];

cout << endl;

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

// sort an array of integers using bubble sort algorithm
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 ] );
```cpp
// end function bubbleSort

// 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;
}
// end function swap
```

Pass arguments by reference, allowing function to swap values at memory locations.

Data items in original order

2  6  4  8  10  12  89  68  45  37

Data items in ascending order

2  4  6  8  10  12  37  45  68  89
5.6 Bubble Sort Using Pass-by-Reference

• `sizeof`
  – Unary operator returns size of operand in bytes
  – For arrays, `sizeof` returns
    \[(\text{size of 1 element}) \times (\text{number of elements})\]
  – 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
// 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 );
   cout << "\nThe number of bytes returned by getSize is " << getSize( array ) << endl;

   return 0; // indicates successful termination
}

Operator sizeof applied to an array returns total number of bytes in array.

Function getSize returns number of bytes used to store array address.
25    // return size of ptr
26    size_t getSize( double *ptr )
27    {
28        return sizeof( ptr );
29    } // end function getSize

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

Operator \texttt{sizeof} returns
number of bytes of pointer.
// 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;
}
cout << "sizeof c = " << sizeof c \\
<< "\ts sizeof(char) = " << sizeof( char ) \\
<< "\tn sizeof s = " << sizeof s \\
<< "\ts sizeof(short) = " << sizeof( short ) \\
<< "\tn sizeof i = " << sizeof i \\
<< "\ts sizeof(int) = " << sizeof( int ) \\
<< "\tn sizeof l = " << sizeof l \\
<< "\ts sizeof(long) = " << sizeof( long ) \\
<< "\tn sizeof f = " << sizeof f \\
<< "\ts sizeof(float) = " << sizeof( float ) \\
<< "\tn sizeof d = " << sizeof d \\
<< "\ts sizeof(double) = " << sizeof( double ) \\
<< "\tn sizeof ld = " << sizeof ld \\
<< "\ts sizeof(long double) = " << sizeof( long double ) \\
<< "\tn sizeof array = " << sizeof array \\
<< "\ts sizeof ptr = " << sizeof ptr \\
<< endl;

return 0; // indicates successful termination

} // end main
```cpp
sizeof c = 1  sizeof(char) = 1
sizeof s = 2  sizeof(short) = 2
sizeof i = 4  sizeof(int) = 4
sizeof l = 4  sizeof(long) = 4
sizeof f = 4  sizeof(float) = 4
sizeof d = 8  sizeof(double) = 8
sizeof ld = 8  sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```
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 ]

<table>
<thead>
<tr>
<th>location</th>
<th>3000</th>
<th>3004</th>
<th>3008</th>
<th>3012</th>
<th>3016</th>
</tr>
</thead>
</table>

pointer variable vPtr
5.7 Pointer Expressions and Pointer Arithmetic

• Subtracting pointers
  – Returns number of elements between two addresses
    
    \[
    \begin{align*}
    \text{vPtr2} &= \text{v[ 2 ]}; \\
    \text{vPtr} &= \text{v[ 0 ]}; \\
    \text{vPtr2} - \text{vPtr} &= 2
    \end{align*}
    \]

• 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 \texttt{void} (type \texttt{void *})
    • Generic pointer, represents any type
    • No casting needed to convert pointer to \texttt{void} pointer
    • \texttt{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 $b[n]$ 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 $b[3]$
// 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\n";

    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 << "\nPointer/offset notation where "
    "the pointer is the array name\n";

    Using array subscript notation.
```cpp
for ( int offset1 = 0; offset1 < 4; offset1++ )
    cout << "\n*(b + " << offset1 << ") = "
        << *(b + offset1) << '\n';

// output array b using bPtr and array subscript notation
cout << "\nPointer subscript notation\n";

for ( int j = 0; j < 4; j++ )
    cout << "bPtr[" << j << "] = " << bPtr[j] << '\n';

cout << "\nPointer/offset notation\n";

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

return 0; // indicates successful termination

} // end main
```

Using array name and pointer/offset notation.

Using pointer subscript notation.

Using \texttt{bPtr} and pointer/offset notation.
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
// Fig. 5.21: fig05_21.cpp
// Copying a string using array notation
// and pointer notation.
#include <iostream>

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

void copy1( char *, const char * );  // prototype
void copy2( char *, const char * );  // prototype

int main()
{
    char string1[10];
    char *string2 = "Hello";
    char string3[10];
    char string4[] = "Good Bye";

    copy1( string1, string2 );
    cout << "string1 = " << string1 << endl;

    copy2( string3, string4 );
    cout << "string3 = " << string3 << endl;

    return 0;  // indicates successful termination
fig05_21.cpp
(output) (1 of 1)

26 } // end main
27
28 // copy s2 to s1 using array notation
29 void copy1( char *s1, const char *s2 )
30 {  
31      for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )  
32         ; // do nothing in body  
33  } // end function copy1
34
35 // copy s2 to s1 using pointer notation
36 void copy2( char *s1, const char *s2 )
37 {  
38      for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )  
39         ; // do nothing in body  
40  } // end function copy2
41
string1 = Hello  
string3 = Good Bye

Use array subscript notation to copy string in s2 to character array s1.

Use pointer notation to copy string in s2 to character array in s1.

Increment both pointers to point to next elements in corresponding arrays.
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)

- Place 1-52 into the array to specify the order in which the cards are dealt
5.10 Case Study: Card Shuffling and Dealing Simulation

- Pseudocode for shuffling and dealing simulation

First refinement
- Initialize the suit array
- Initialize the face array
- Initialize the deck array
- Shuffle the deck
- Deal 52 cards

Second refinement
- For each of the 52 cards
  - Place card number in randomly selected unoccupied slot of deck
- For each slot of the deck array
  - Find card number in deck array and print face and suit of card

Third refinement
- Choose slot of deck randomly
- While chosen slot of deck has been previously chosen
  - Choose slot of deck randomly
  - Place card number in chosen slot of deck
- For each slot of the deck array
  - If slot contains card number
    - Print the face and suit of the card
// Fig. 5.24: fig05_24.cpp
// Card shuffling dealing program.
#include <iostream>

using std::cout;
using std::left;
using std::right;

#include <iomanip>

using std::setw;

#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
    const char *suit[ 4 ] =
    { "Hearts", "Diamonds", "Clubs", "Spades" };

    suit array contains pointers to char arrays.
// initialize face array
const char *face[13] =
{
    "Ace", "Deuce", "Three", "Four",
    "Five", "Six", "Seven", "Eight",
    "Nine", "Ten", "Jack", "Queen", "King" 
};

// 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

} // end main
// 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++ ) {

        // choose new random location until unoccupied slot found
        do {
            row = rand() % 4;
            column = rand() % 13;
        } while ( wDeck[ row ][ column ] != 0 ); // end do/while

        // place card number in chosen slot of deck
        wDeck[ row ][ column ] = card;

    } // end for

} // end function shuffle

Current position is at randomly selected row and column.
```cpp
// 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 ? '\n' : '\t' );
                }
        // end if
    // end function deal
}
```
<table>
<thead>
<tr>
<th>Card</th>
<th>Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nine of Spades</td>
<td>Seven of Clubs</td>
</tr>
<tr>
<td>Five of Spades</td>
<td>Eight of Clubs</td>
</tr>
<tr>
<td>Queen of Diamonds</td>
<td>Three of Hearts</td>
</tr>
<tr>
<td>Jack of Spades</td>
<td>Five of Diamonds</td>
</tr>
<tr>
<td>Jack of Diamonds</td>
<td>Three of Diamonds</td>
</tr>
<tr>
<td>Three of Clubs</td>
<td>Six of Clubs</td>
</tr>
<tr>
<td>Ten of Clubs</td>
<td>Nine of Diamonds</td>
</tr>
<tr>
<td>Ace of Hearts</td>
<td>Queen of Hearts</td>
</tr>
<tr>
<td>Seven of Spades</td>
<td>Deuce of Spades</td>
</tr>
<tr>
<td>Six of Hearts</td>
<td>Deuce of Clubs</td>
</tr>
<tr>
<td>Ace of Clubs</td>
<td>Deuce of Diamonds</td>
</tr>
<tr>
<td>Nine of Hearts</td>
<td>Seven of Diamonds</td>
</tr>
<tr>
<td>Six of Spades</td>
<td>Eight of Diamonds</td>
</tr>
<tr>
<td>Ten of Spades</td>
<td>King of Hearts</td>
</tr>
<tr>
<td>Four of Clubs</td>
<td>Ace of Spades</td>
</tr>
<tr>
<td>Ten of Hearts</td>
<td>Four of Spades</td>
</tr>
<tr>
<td>Eight of Hearts</td>
<td>Eight of Spades</td>
</tr>
<tr>
<td>Jack of Hearts</td>
<td>Ten of Diamonds</td>
</tr>
<tr>
<td>Four of Diamonds</td>
<td>King of Diamonds</td>
</tr>
<tr>
<td>Seven of Hearts</td>
<td>King of Spades</td>
</tr>
<tr>
<td>Queen of Spades</td>
<td>Four of Hearts</td>
</tr>
<tr>
<td>Nine of Clubs</td>
<td>Six of Diamonds</td>
</tr>
<tr>
<td>Deuce of Hearts</td>
<td>Jack of Clubs</td>
</tr>
<tr>
<td>King of Clubs</td>
<td>Three of Spades</td>
</tr>
<tr>
<td>Queen of Clubs</td>
<td>Five of Clubs</td>
</tr>
<tr>
<td>Five of Hearts</td>
<td>Ace of Diamonds</td>
</tr>
</tbody>
</table>
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
    OR
    • `compare( int1, int2 )`
      – Could be confusing
        • User may think `compare` name of actual function in program
// Fig. 5.25: fig05_25.cpp
// Multipurpose sorting program using function pointers.
#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 };

Parameter is pointer to function that receives two integer parameters and returns bool result.
cout << "Enter 1 to sort in ascending order, \n" << "Enter 2 to sort in descending order: ";
cin >> order;
cout << "\nData items in original order\n";

// 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 << "\nData items in ascending order\n";
}

// sort array in descending order; pass function descending
// as an argument to specify descending sorting order
else {
    bubble( a, arraySize, descending );
    cout << "\nData items in descending order\n";
}
// output sorted array
for ( counter = 0; counter < arraySize; counter++ )
    cout << setw(4) << a[counter];

cout << endl;

return 0; // indicates successful termination
}

// multipurpose bubble sort; parameter compare is 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 comparisons per pass
        for ( int count = 0; count < size - 1; count++ )
            // if adjacent elements are out of order, swap them
            if ( (*compare)( work[count], work[count + 1] ) )
                swap( &work[count], &work[count + 1] );

compare is pointer to function that receives two integer parameters and returns bool result.
Parentheses necessary to indicate pointer to function
Call passed function compare; dereference pointer to execute function.
// 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;
}

// determine whether elements are out of order
// for an ascending order sort

bool ascending( int a, int b )
{
    return b < a;  // swap if b is less than a
}

} // end function swap

} // end function bubble
```cpp
// determine whether elements are out of order
// for a descending order sort
bool descending( int a, int b )
{
    return b > a;    // swap if b is greater than a
}
```

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
2   6   4   8  10  12  89  68  45  37
Data items in ascending order
2   4   6   8  10  12  37  45  68  89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
2   6   4   8  10  12  89  68  45  37
Data items in descending order
89  68  45  37  12  10   8   6   4   2
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
```
// Fig. 5.26: fig05_26.cpp
// Demonstrating an array of pointers to functions.
#include <iostream>

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

// function prototypes
void function1( int );
void function2( int );
void function3( int );

int main()
{
    // initialize array of 3 pointers to functions that each
    // take an int argument and return void
    void (*f[3])(int) = { function1, function2, function3 };

    int choice;

    cout << "Enter a number between 0 and 2, 3 to end: ";
    cin >> choice;
}
```
```c++
// 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;
}

cout << "Program execution completed."

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

Call chosen function by dereferencing corresponding element in array.
void function2( int b )
{
    cout << "You entered " << b
    << " so function2 was called\n\n";
}
// end function2

void function3( int c )
{
    cout << "You entered " << c
    << " so function3 was called\n\n";
}
// end function3

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

• 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
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’ };
5.12.1 Fundamentals of Characters and Strings

• 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

- 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)
### 5.12.2 String Manipulation Functions of the String-handling Library

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strcpy( char *s1, const char *s2 );</code></td>
<td>Copies the string <code>s2</code> into the character array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncpy( char *s1, const char *s2, size_t n );</code></td>
<td>Copies at most <code>n</code> characters of the string <code>s2</code> into the character array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strcat( char *s1, const char *s2 );</code></td>
<td>Appends the string <code>s2</code> to the string <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncat( char *s1, const char *s2, size_t n );</code></td>
<td>Appends at most <code>n</code> characters of string <code>s2</code> to string <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>int strcmp( const char *s1, const char *s2 );</code></td>
<td>Compares the string <code>s1</code> with the string <code>s2</code>. The function returns a value of zero, less than zero or greater than zero if <code>s1</code> is equal to, less than or greater than <code>s2</code>, respectively.</td>
</tr>
</tbody>
</table>
### 5.12.2 String Manipulation Functions of the String-handling Library

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int strncmp(const char *s1, const char *s2, size_t n);</code></td>
<td>Compares up to <code>n</code> characters of the string <code>s1</code> with the string <code>s2</code>. The function returns zero, less than zero or greater than zero if <code>s1</code> is equal to, less than or greater than <code>s2</code>, respectively.</td>
</tr>
<tr>
<td><code>char *strtok(char *s1, const char *s2);</code></td>
<td>A sequence of calls to <code>strtok</code> breaks string <code>s1</code> into “tokens”—logical pieces such as words in a line of text—delimited by characters contained in string <code>s2</code>. The first call contains <code>s1</code> as the first argument, and subsequent calls to continue tokenizing the same string contain <code>NULL</code> as the first argument. A pointer to the current to-ken is returned by each call. If there are no more tokens when the function is called, <code>NULL</code> is returned.</td>
</tr>
<tr>
<td><code>size_t strlen(const char *s);</code></td>
<td>Determines the length of string <code>s</code>. The number of characters preceding the terminating null character is returned.</td>
</tr>
</tbody>
</table>
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
// Fig. 5.28: fig05_28.cpp
// Using strcpy and strncpy.
#include <iostream>
#include <cstring>

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

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;

    // copy first 14 characters of x into z
    strncpy( z, x, 14 ); // does not copy null character
    z[14] = '\0'; // append '\0' to z's contents

    cout << "The string in array z is: " << z << endl;
}
```cpp
return 0; // indicates successful termination

} // end main
```

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

String to copy
Copied string using `strcpy`
Copied first 14 characters using `strncpy`
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
// Fig. 5.29: fig05_29.cpp
// Using strcat and strncat.
#include <iostream>
using std::cout;
using std::endl;
#include <cstring>
// prototypes for strcat and strncat

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

    cout << "s1 = " << s1 << "\ns2 = " << s2;
    strcat( s1, s2 ); // concatenate s2 to s1
    cout << "\n\nAfter strcat(s1, s2):
\ns1 = " << s1 << "\ns2 = " << s2;

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

    cout << "\n\nAfter strncat(s3, s1, 6):
\ns3 = " << s3 << "\ns1 = " << s1;
}

cout << "\n\nAfter strcat(s3, s1): \nns1 = " << s1 << "\nns3 = " << s3 << endl;

return 0;  // indicates successful termination

} // end main

s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
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 Interchage”
    • EBCDIC
      – “Extended Binary Coded Decimal Interchange Code”
5.12.2 String Manipulation Functions of the String-handling Library

- Comparing strings
  - \texttt{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
  - \texttt{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
```
#include <iostream>

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

#include <iomanip>
using std::setw;

#include <cstring> // prototypes for strcmp and strncmp.

int main()
{
    char *s1 = "Happy New Year";
    char *s2 = "Happy New Year";
    char *s3 = "Happy Holidays";

    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\nns3 = " << s3 << "\n
strcmp(s1, s2) = " << setw(2) << strcmp(s1, s2) << "\n
strcmp(s1, s3) = " << setw(2) << strcmp(s1, s3) << "\n
strcmp(s3, s1) = " << setw(2) << strcmp(s3, s1);"
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) = 0
strcmp(s1, s3) = 1
strcmp(s3, s1) = -1

strn cmp(s1, s3, 6) = 0
strn cmp(s1, s3, 7) = 1
strn cmp(s3, s1, 7) = -1
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>

using namespace std;

#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: \n";
    cout << "The tokens are: \n";

    // begin tokenization of sentence
    tokenPtr = strtok( sentence, " ");
}

// <cstring> contains prototype for strtok.

First call to strtok begins tokenization.
// continue tokenizing sentence until tokenPtr becomes NULL
while ( tokenPtr != NULL ) {
    cout << tokenPtr << '\n';
    tokenPtr = strtok( NULL, " " ); // get next token
}

} // end while

cout << "\nAfter strtok, sentence = " << sentence << endl;

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

Subsequent calls to \texttt{strtok} with \texttt{NULL} as first argument to indicate continuation.
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
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>

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

#include <cstring> // prototype for strlen

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

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

    return 0; // indicates successful termination
}

// end main
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6