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
    \[
    \text{int } \ast\text{myPtr;}
    \]
    declares pointer to \texttt{int}, pointer of type \texttt{int } \ast
  – Multiple pointers require multiple asterisks
    \[
    \text{int } \ast\text{myPtr1, } \ast\text{myPtr2;}
    \]
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 << "\n\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
27
28 } // 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;

    // 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

Pass number by value; result returned by cubeByValue
// 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

cubeByValue receives parameter passed-by-value
Cubes and returns local variable n
```cpp
// 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 << endl;
    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
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}

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

cubeByReference receives address of int variable, i.e., pointer to an int

Modify and access int variable using indirection operator *
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
}

// 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 ] );
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.

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
5.7 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]`
5.7 Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements between two addresses
    \[
    \text{vPtr2} = \text{v}[2]; \\
    \text{vPtr} = \text{v}[0]; \\
    \text{vPtr2} - \text{vPtr} == 2
    \]

- 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 \textbf{void} (type \textbf{void *} )
    \[
    \begin{align*}
    \text{• Generic pointer, represents any type} \\
    \text{• No casting needed to convert pointer to \textbf{void} pointer} \\
    \text{• \textbf{void} pointers cannot be dereferenced}
    \end{align*}
    \]
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";
```cpp
for ( int offset1 = 0; offset1 < 4; offset1++ )
    cout << "*(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 << "*(bPtr + " << offset2 << ") = "
    << *( bPtr + offset2 ) << '\n';

return 0; // indicates successful termination
```

Using array name and pointer/offset notation.

Using pointer subscript notation.

Using 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
```cpp
#include <iostream>

using namespace std;  

int main()
{
    char string1[] = "Hello";
    char string2[] = "Good Bye";
    char string3[];
    // copy s2 to s1 using array notation
    void copy1( char *s1, const char *s2 )
    {
        for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
            ;   // do nothing in body
    }
    // copy s2 to s1 using pointer notation
    void copy2( char *s1, const char *s2 )
    {
        for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
            ;   // do nothing in body
    }

    copy1( string1, string2 );
    copy2( string3, string2 );

    cout << string1 << endl;
    cout << string3 << endl;
    return 0;
}
```

Use array subscript notation to copy string in \texttt{s2} to character array \texttt{s1}.

Use pointer notation to copy string in \texttt{s2} to character array in \texttt{s1}.

Increment both pointers to point to next elements in corresponding arrays.
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 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.</td>
</tr>
<tr>
<td><code>size_t strlen( const char *s );</code></td>
<td>Determines the length of string s. 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
```cpp
// Fig. 5.28: fig05_28.cpp
// Using strcpy and strncpy.
#include <iostream>
#include <cstring>   // prototypes for strcpy and strncpy

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;
}
```

// <cstring> contains prototypes for strcpy and strncpy.

Copy entire string in array **x** into array **y**.

Copy first 14 characters of that terminating null character.

Append terminating null character.
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 \texttt{strcpy}
Copied first 14 characters using \texttt{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>
#include <cstring> // prototypes for strcat and strncat

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

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;

    cout << "\ns1 = " << s1 << "s2 = " << s2;

    // concatenate first 6 characters of s1 to s3
    strncat( s3, s1, 6 ); // places '\0' after last character
    // concatenate first 6 characters of s1 to s3
    strncat( s3, s1, 6 ); // places '\0' after last character
}
cout << "\n\nAfter strcat(s1, s2):
\ns1 = Happy New Year
\ns2 = New Year\n\nAfter strncat(s3, s1, 6):
\ns1 = Happy New Year
\ns3 = Happy\n\nAfter strcat(s3, s1):
\ns1 = Happy New Year
\ns3 = Happy Happy New Year

Append s1 to s3.
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”
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
// Fig. 5.30: fig05_30.cpp
// Using strcmp and strncmp.
#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 << " s2 = " << s2 << "s3 = " << s3
         << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
         << "\n\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
         << "\n\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
}
cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 ) << strncmp( s1, s3, 6 ) << \n
strncmp(s1, s3, 7) = " << setw( 2 ) << strncmp( s1, s3, 7 ) << \n
strncmp(s3, s1, 7) = " << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;

return 0; // indicates successful termination

} // end main

s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

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

strncmp(s1, s3, 6) =  0
 strncmp(s1, s3, 7) =  1
 strncmp(s3, s1, 7) = -1
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