

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

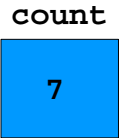
- Pointers
 - Powerful, but difficult to master
 - Simulate pass-by-reference
 - Close relationship with arrays and strings



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;
```



5.2 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

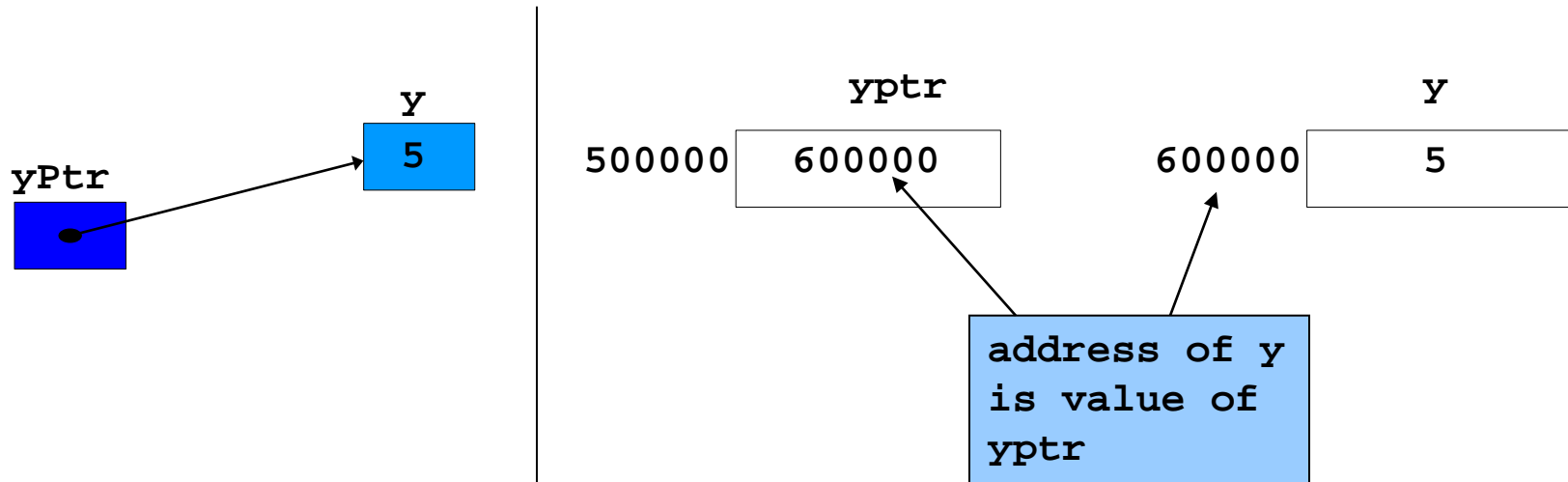
- **&** (address operator)

- 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
 - ***yPtr** returns **y** (because **yPtr** points to **y**).
 - dereferenced pointer is lvalue

```
*yptr = 9;           // assigns 9 to y
```
- ***** and **&** are inverses of each other



**fig05_04.cpp**
(1 of 2)

```
1  // Fig. 5.4: fig05_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10     int a;        // a is an integer
11     int *aPtr;    // aPtr is a pointer to an integer
12
13     a = 7;
14     aPtr = &a;    // aPtr assigned address of a
15
16     cout << "The address of a is " << &a
17           << "\nThe value of aPtr is " << aPtr;
18
19     cout << "\n\nThe value of a is " << a
20           << "\nThe value of *aPtr is " << *aPtr;
21
22     cout << "\n\nShowing that * and & are inverses of "
23           << "each other.\n&*aPtr = " << &*aPtr
24           << "\n*&aPtr = " << *&aPtr << endl;
25
```

* and & are inverses
of each other

**fig05_04.cpp****(2 of 2)****fig05_04.cpp****output (1 of 1)**

```
26     return 0; // indicates successful termination
27
28 }
```

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

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



**fig05_06.cpp**
(1 of 2)

```
1  // Fig. 5.6: fig05_06.cpp
2  // Cube a variable using pass-by-value.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int cubeByValue( int );    // prototype
9
10 int main()
11 {
12     int number = 5;
13
14     cout << "The original value of number is "
15
16     // pass number by value to cubeByValue
17     number = cubeByValue( number );
18
19     cout << "\nThe new value of number is " << number << endl;
20
21     return 0;    // indicates successful termination
22
23 } // end main
24
```

Pass number by value; result
returned by
cubeByValue



fig05_06.cpp
(2 of 2)

fig05_06.cpp
output (1 of 1)

```
25 // calculate and return cube of integer argument
26 int cubeByValue( int n )
27 {
28     return n * n * n; // cube local variable n
29 }
30 // end function cubeByValue
```

cubeByValue receives
parameter passed-by-value

Cubes and **returns**
local variable **n**

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

fig05_07.cpp
(1 of 2)

```
1  // Fig. 5.7: fig05_07.cpp
2  // Cube a variable using pass-by-reference
3  // with a pointer argument.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void cubeByReference( int * );    // prototype
10
11 int main()
12 {
13     int number = 5;
14
15     cout << "The original value of number is " << number << endl;
16
17     // pass address of number to cubeByReference
18     cubeByReference( &number );
19
20     cout << "\nThe new value of number is " << number << endl;
21
22     return 0;    // indicates successful termination
23
24 } // end main
25
```

Prototype indicates parameter
is pointer to **int**

Apply address operator **&** to
pass address of number to
cubeByReference

cubeByReference
modified variable
number

```
26 // calculate cube of *nPtr; modifies variable number in main
27 void cubeByReference( int *nPtr )
28 {
29     *nPtr = *nPtr * *nPtr * *nPtr; // cube
30
31 } // end function cubeByReference
```

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

Modify and access **int**
variable using indirection
operator *****

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

fig05_07.cpp
(2 of 2)

fig05_07.cpp
output (1 of 1)

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



fig05_10.cpp
(1 of 2)

```
1  // Fig. 5.10: fig05_10.cpp
2  // Converting lowercase letters to uppercase letters
3  // using a non-constant pointer to non-constant data.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include <cctype>    // prototypes for islower, isupper
10
11 void convertToUppercase( char * );
12
13 int main()
14 {
15     char phrase[] = "characters and $32.98";
16
17     cout << "The phrase before conversion is: " << phrase;
18     convertToUppercase( phrase );
19     cout << "\nThe phrase after conversion is: "
20          << phrase << endl;
21
22     return 0;    // indicates successful termination
23
24 } // end main
25
```

Parameter is nonconstant
pointer to nonconstant data

convertToUppercase
modifies variable phrase



fig05_10.cpp

of 2)

fig05_10.cpp
output (1 of 1)

```

26 // convert string to uppercase letters
27 void convertToUppercase( char *sPtr )
28 {
29     while ( *sPtr != '\0' ) { // current character is not '\0'
30
31         if ( islower( *sPtr ) ) // if character is lowercase
32             *sPtr = toupper( *sPtr ); // convert to uppercase
33
34         ++sPtr; // move sPtr to next element of array
35
36     } // end while
37
38 } // end function convertToUppercase

```

Parameter **sPtr** nonconstant
pointer to nonconstant data

Function **islower** returns
true if character is
lowercase

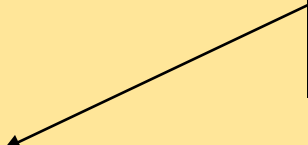
Function **toupper** returns

When operator **++** applied to
pointer that points to array,
memory address stored in
pointer modified to point to
next element of array.

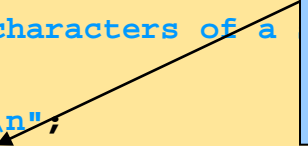
The phrase before conversion is
The phrase after conversion is

fig05_11.cpp
(1 of 2)

```
1  // Fig. 5.11: fig05_11.cpp
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void printCharacters( const char * );
10
11 int main()
12 {
13     char phrase[] = "print characters of a";
14
15     cout << "The string is:\n";
16     printCharacters( phrase );
17     cout << endl;
18
19     return 0; // indicates successful termination
20
21 } // end main
22
```



Parameter is nonconstant
pointer to constant data.



Pass pointer **phrase** to
function
printCharacters.

**fig05_11.cpp****(2 of 2)****fig05_11.cpp****output (1 of 1)**

```
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
```

sPtr is nonconstant pointer
to constant data; cannot
modify character to which

Increment **sPtr** to point to
next character.

The string is:
print characters of a string



fig05_12.cpp
(1 of 1)

fig05_12.cpp
output (1 of 1)

```

1  // Fig. 5.12: fig05_12.cpp
2  // Attempting to modify data through a
3  // non-constant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9      int y;
10
11     f( &y ); // f attempts illegal modification
12
13     return 0; // indicates success
14
15 } // end main
16
17 // xPtr cannot modify the variable
18 // to which it points
19 void f( const int *xPtr )
20 {
21     *xPtr = 100; // error: cannot modify a const object
22
23 } // 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.

```

d:\cpphttp4_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



fig05_13.cpp
(1 of 1)

fig05_13.cpp
output (1 of 1)

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

```
4
5 int main()
```

```
6 {
```

```
7     int x, y;
```

```
8
```

```
9     // ptr is a constant pointer to an int ptr is constant pointer to
```

```
10    // be modified through ptr
```

```
11    // same memory location.
```

```
12    int * const ptr = &x;
```

```
13
```

```
14    *ptr = 7; // allowed: *ptr
```

```
15    ptr = &y; // error: ptr is const; can
```

```
16
```

```
17    return 0; // indicates successful ter
```

```
18
```

```
19 } // end main
```

Can modify **x** (pointed to by

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.

```
d:\cpphttp4_examples\ch05\Fig05_13.cpp(15) : error C2166:
  l-value specifies const object
```

fig05_14.cpp
 (1 of 1)

```

1  // Fig. 5.14: fig05_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10     int x = 5, y;
11
12     // ptr is a constant pointer to a constant integer.
13     // ptr always points to the same location in memory.
14     // at that location cannot be modified.
15     const int *const ptr = &x;
16
17     cout << *ptr << endl;
18
19     *ptr = 7; // error: *ptr is a constant; cannot assign new value
20     ptr = &y; // error: ptr is const; cannot assign new address
21
22     return 0; // indicates successful termination
23
24 } // end main

```

ptr is constant pointer to integer constant.

Cannot modify **x** (pointed to)

Cannot modify **ptr** to point to new address since **ptr** is constant.



```
d:\cpphttp4_examples\ch05\Fig05_14.cpp(19) : error C2166:
```

```
l-value specifies const object
```

```
d:\cpphttp4_examples\ch05\Fig05_14.cpp(20) : error C2166:
```

```
l-value specifies const object
```

Line 19 generates compiler error by attempting to modify a const object (1 of 1)

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 **&**



**fig05_15.cpp**
(1 of 3)

```
1  // Fig. 5.15: fig05_15.cpp
2  // This program puts values into an array, sorts the values into
3  // ascending order, and prints the resulting array.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 void bubbleSort( int *, const int );    // prototype
14 void swap( int * const, int * const ); // prototype
15
16 int main()
17 {
18     const int arraySize = 10;
19     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
20
21     cout << "Data items in original order\n";
22
23     for ( int i = 0; i < arraySize; i++ )
24         cout << setw( 4 ) << a[ i ];
25
```



fig05_15.cpp (2 of 3)

```

26  bubbleSort( a, arraySize ); // sort the array
27
28  cout << "\nData items in ascending order\n";
29
30  for ( int j = 0; j < arraySize; j++ )
31      cout << setw( 4 ) << a[ j ];
32
33  cout << endl;
34
35  return 0; // indicates successful termination
36
37 } // end main
38
39 // sort an array of integers using bubbleSort
40 void bubbleSort( int *array, const int size )
41 {
42     // loop to control passes
43     for ( int pass = 0; pass < size - 1; pass++ )
44
45         // loop to control comparisons during each pass
46         for ( int k = 0; k < size - 1; k++ )
47
48             // swap adjacent elements if they are out of order
49             if ( array[ k ] > array[ k + 1 ] )
50                 swap( &array[ k ], &array[ k + 1 ] );

```

Declare as `const` to indicate that `size` receives size of array as argument; declared **const** to ensure **size** not modified.

`bubbleSort` receives single-subscripted array.

fig05_15.cpp
(3 of 3)

fig05_15.cpp

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

```
51
52 } // end function bubbleSort
53
54 // swap values at memory locations to which
55 // element1Ptr and element2Ptr point
56 void swap( int * const element1Ptr, int * const element2Ptr )
57 {
58     int hold = *element1Ptr;
59     *element1Ptr = *element2Ptr;
60     *element2Ptr = hold;
61
62 } // end function swap
```

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
$$(\text{size of 1 element}) * (\text{number of elements})$$
- 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



**fig05_16.cpp**
(1 of 2)

```
1  // Fig. 5.16: fig05_16.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  size_t getSize( double * ); // prototype
10
11 int main()
12 {
13     double array[ 20 ];
14
15     cout << "The number of bytes in the array "
16          << sizeof( array );
17
18     cout << "\nThe number of bytes returned by getSize is "
19          << getSize( array ) << endl;
20
21     return 0; // indicates successful termination
22
23 } // end main
24
```

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

Function **getSize** returns number of bytes used to store **array** address.



fig05_16.cpp
(2 of 2)

fig05_16.cpp
output (1 of 1)

```
25 // return size of ptr
26 size_t getSize( double *ptr )
27 {
28     return sizeof( ptr );
29
30 } // end function getSize
```

Operator **sizeof** returns
number of bytes of pointer.

The number of bytes in the array is 16
The number of bytes returned by getSize is 4

**fig05_17.cpp**
(1 of 2)

```
1  // Fig. 5.17: fig05_17.cpp
2  // Demonstrating the sizeof operator.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10     char c;
11     short s;
12     int i;
13     long l;
14     float f;
15     double d;
16     long double ld;
17     int array[ 20 ];
18     int *ptr = array;
19
```




Operator **sizeof** can be
used on

Operator **sizeof** can be
used on type name.

```
20 cout << "sizeof c = " << sizeof c
21     << "\tsizeof(char) = " << sizeof( char )
22     << "\nsizeof s = " << sizeof s
23     << "\tsizeof(short) = " << sizeof( short )
24     << "\nsizeof i = " << sizeof i
25     << "\tsizeof(int) = " << sizeof( int )
26     << "\nsizeof l = " << sizeof l
27     << "\tsizeof(long) = " << sizeof( long )
28     << "\nsizeof f = " << sizeof f
29     << "\tsizeof(float) = " << sizeof( float )
30     << "\nsizeof d = " << sizeof d
31     << "\tsizeof(double) = " << sizeof( double )
32     << "\nsizeof ld = " << sizeof ld
33     << "\tsizeof(long double) = " << sizeof( long double )
34     << "\nsizeof array = " << sizeof array
35     << "\nsizeof ptr = " << sizeof ptr
36     << endl;
37
38     return 0; // indicates successful termination
39
40 }
```

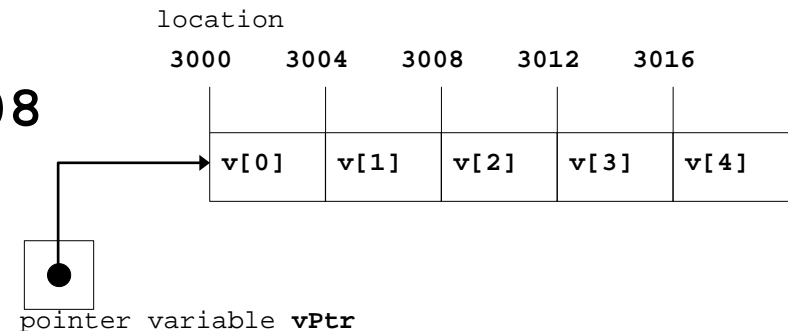


fig05_17.cpp
output (1 of 1)

```
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]**



5.7 Pointer Expressions and Pointer Arithmetic

- Subtracting pointers

- Returns number of elements between two addresses

```
vPtr2 = v[ 2 ];  
vPtr  = v[ 0 ];  
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 `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]`



**fig05_20.cpp**
(1 of 2)

```
1  // Fig. 5.20: fig05_20.cpp
2  // Using subscripting and pointer notations with arrays.
3
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  int main()
10 {
11     int b[] = { 10, 20, 30, 40 };
12     int *bPtr = b;    // set bPtr to point to array b
13
14     // output array b using array subscript notation
15     cout << "Array b printed with:\n"
16          << "Array subscript notation\n";
17
18     for ( int i = 0; i < 4; i++ )
19         cout << "b[" << i << "] = " << b[ i ] << '\n';
20
21     // output array b using the array name and
22     // pointer/offset notation
23     cout << "\nPointer/offset notation where "
24          << "the pointer is the array name\n";
25
```

Using array subscript notation.

fig05_20.cpp
(2 of 2)

```
26 for ( int offset1 = 0; offset1 < 4; offset1++ )
27     cout << "*(b + " << offset1 << ") = "
28         << *( b + offset1 ) << '\n';
29
30 // output array b using bPtr and array subscript notation
31 cout << "\nPointer subscript notation\n";
32
33 for ( int j = 0; j < 4; j++ )
34     cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
35
36 cout << "\nPointer/offset notation\n";
37
38 // output array b using bPtr and pointer/offset notation
39 for ( int offset2 = 0; offset2 < 4; offset2++ )
40     cout << "*(bPtr + " << offset2 << ") = "
41         << *( bPtr + offset2 ) << '\n';
42
43 return 0; // indicates successful termination
44
45 } // end main
```

Using array name and
pointer/offset notation.

Using pointer subscript
notation.

Using **bPtr** and
pointer/offset notation.

**fig05_20.cpp**
output (1 of 1)

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
```

**fig05_21.cpp**
(1 of 2)

```
1  // Fig. 5.21: fig05_21.cpp
2  // Copying a string using array notation
3  // and pointer notation.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void copy1( char *, const char * ); // prototype
10 void copy2( char *, const char * ); // prototype
11
12 int main()
13 {
14     char string1[ 10 ];
15     char *string2 = "Hello";
16     char string3[ 10 ];
17     char string4[] = "Good Bye";
18
19     copy1( string1, string2 );
20     cout << "string1 = " << string1 << endl;
21
22     copy2( string3, string4 );
23     cout << "string3 = " << string3 << endl;
24
25     return 0; // indicates successful termination
```



fig05_21.cpp
of 2)

fig05_21.cpp
output (1 of 1)

```

26
27 } // end main
28
29 // copy s2 to s1 using array notation
30 void copy1( char *s1, const char *s2 )
31 {
32     for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
33         ; // do nothing in body
34
35 } // end function copy1
36
37 // copy s2 to s1 using pointer notation
38 void copy2( char *s1, const char *s2 )
39 {
40     for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
41         ; // do nothing in body
42
43 } // end function copy2

```

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.

```

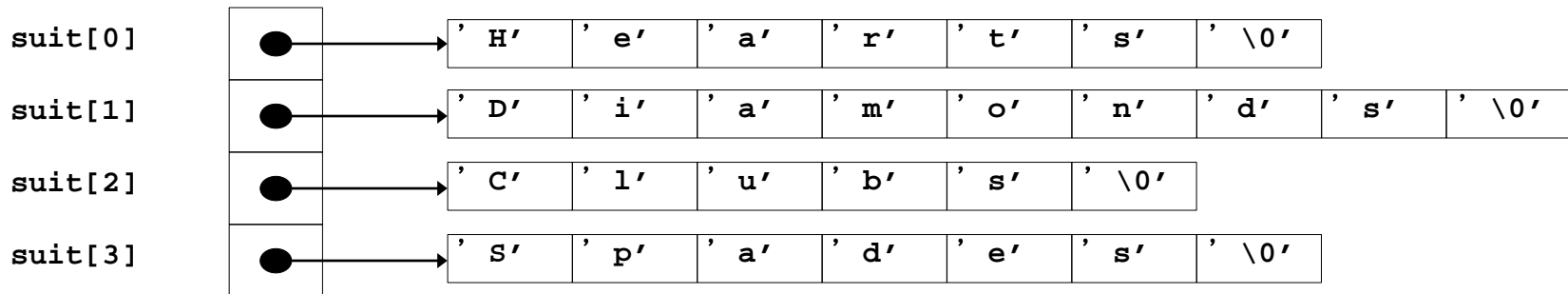
string1 = Hello
string3 = Good Bye

```

5.9 Arrays of Pointers

- Arrays can contain pointers
 - Commonly used to store array of strings


```
char *suit[ 4 ] = { "Hearts", "Diamonds",  
                  "Clubs", "Spades" };
```
 - 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)

		Ace	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Jack	Queen	King
		0	1	2	3	4	5	6	7	8	9	10	11	12
Hearts	0													
Diamonds	1													
Clubs	2													
Spades	3													

`deck[2][12]` represents the King of Clubs

Clubs

King

- 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 of the 52 cards

*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*



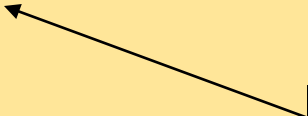
**fig05_24.cpp**
(1 of 4)

```
1  // Fig. 5.24: fig05_24.cpp
2  // Card shuffling dealing program.
3  #include <iostream>
4
5  using std::cout;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 #include <cstdlib>    // prototypes for rand and srand
14 #include <ctime>     // prototype for time
15
16 // prototypes
17 void shuffle( int [][] [ 13 ] );
18 void deal( const int [][] [ 13 ], const char *[], const char *[] );
19
20 int main()
21 {
22     // initialize suit array
23     const char *suit[ 4 ] =
24         { "Hearts", "Diamonds", "Clubs", "Spades" };
25
```

suit array contains pointers
to **char** arrays.

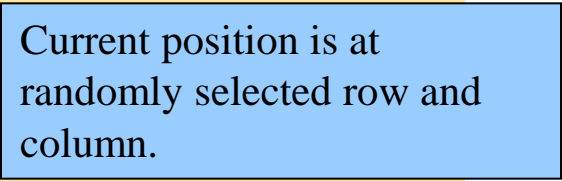
fig05_24.cpp
(2 of 4)

```
26 // initialize face array
27 const char *face[ 13 ] =
28     { "Ace", "Deuce", "Three", "Four",
29       "Five", "Six", "Seven", "Eight",
30       "Nine", "Ten", "Jack", "Queen", "King" };
31
32 // initialize deck array
33 int deck[ 4 ][ 13 ] = { 0 };
34
35 srand( time( 0 ) ); // seed random number generator
36
37 shuffle( deck );
38 deal( deck, face, suit );
39
40 return 0; // indicates successful termination
41
42 } // end main
43
```



face array contains pointers
to **char** arrays.


```
44 // shuffle cards in deck
45 void shuffle( int wDeck[][ 13 ] )
46 {
47     int row;
48     int column;
49
50     // for each of the 52 cards, choose slot of deck randomly
51     for ( int card = 1; card <= 52; card++ ) {
52
53         // choose new random location until unoccupied
54         do {
55             row = rand() % 4;
56             column = rand() % 13;
57         } while ( wDeck[ row ][ column ] != 0 ); // end do/while
58
59         // place card number in chosen slot of deck
60         wDeck[ row ][ column ] = card;
61
62     } // end for
63
64 } // end function shuffle
65
```



Current position is at
randomly selected row and
column.

fig05_24.cpp (4 of 4)

```

66 // deal cards in deck
67 void deal( const int wDeck[][ 13 ], const char *wFace[],
68           const char *wSuit[] )
69 {
70     // for each of the 52 cards
71     for ( int card = 1; card <= 52; card++ )
72
73         // loop through rows of wDeck
74         for ( int row = 0; row <= 3; row++ )
75
76             // loop through columns of wDeck for current row
77             for ( int column = 0; column <= 12; column++ )
78
79                 // if slot contains current card, display
80                 if ( wDeck[ row ][ column ] == card ) {
81                     cout << setw( 5 ) << right << wFace[ column ]
82                         << " of " << setw( 8 ) << left
83                         << wSuit[ row ]
84                         << ( card % 2 == 0 ? '\n' : '\t' );
85
86                 } // end if
87
88 } // end function deal

```

Cause face to be output right
Cause suit to be output left
justified in field of 8
characters.



fig05_24.cpp
output (1 of 1)

Nine of Spades	Seven of Clubs
Five of Spades	Eight of Clubs
Queen of Diamonds	Three of Hearts
Jack of Spades	Five of Diamonds
Jack of Diamonds	Three of Diamonds
Three of Clubs	Six of Clubs
Ten of Clubs	Nine of Diamonds
Ace of Hearts	Queen of Hearts
Seven of Spades	Deuce of Spades
Six of Hearts	Deuce of Clubs
Ace of Clubs	Deuce of Diamonds
Nine of Hearts	Seven of Diamonds
Six of Spades	Eight of Diamonds
Ten of Spades	King of Hearts
Four of Clubs	Ace of Spades
Ten of Hearts	Four of Spades
Eight of Hearts	Eight of Spades
Jack of Hearts	Ten of Diamonds
Four of Diamonds	King of Diamonds
Seven of Hearts	King of Spades
Queen of Spades	Four of Hearts
Nine of Clubs	Six of Diamonds
Deuce of Hearts	Jack of Clubs
King of Clubs	Three of Spades
Queen of Clubs	Five of Clubs
Five of Hearts	Ace of Diamonds

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



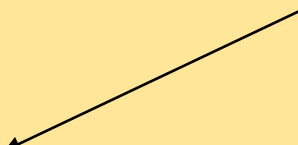
5.11 Function Pointers

- 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



fig05_25.cpp
(1 of 5)

```
1  // Fig. 5.25: fig05_25.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 // prototypes
14 void bubble( int [], const int, bool (*)( int, int ) );
15 void swap( int * const, int * const );
16 bool ascending( int, int );
17 bool descending( int, int );
18
19 int main()
20 {
21     const int arraySize = 10;
22     int order;
23     int counter;
24     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
25
```



Parameter is pointer to
function that receives two
integer parameters and returns
bool result.

**fig05_25.cpp**
(2 of 5)

```
26  cout << "Enter 1 to sort in ascending order,\n"
27      << "Enter 2 to sort in descending order: ";
28  cin >> order;
29  cout << "\nData items in original order\n";
30
31  // output original array
32  for ( counter = 0; counter < arraySize; counter++ )
33      cout << setw( 4 ) << a[ counter ];
34
35  // sort array in ascending order; pass function ascending
36  // as an argument to specify ascending sorting order
37  if ( order == 1 ) {
38      bubble( a, arraySize, ascending );
39      cout << "\nData items in ascending order\n";
40  }
41
42  // sort array in descending order; pass function descending
43  // as an argument to specify descending sorting order
44  else {
45      bubble( a, arraySize, descending );
46      cout << "\nData items in descending order\n";
47  }
48
```

fig05_25.cpp (3 of 5)

```

49 // output sorted array
50 for ( counter = 0; counter < arraySize; counter++ )
51     cout << setw( 4 ) << a[ counter ];
52
53 cout << endl;
54
55 return 0; // indicates successful termination
56
57 } // end main
58
59 // multipurpose bubble sort; parameter compare
60 // the comparison function that determines
61 void bubble( int work[], const int size,
62             bool (*compare)( int, int ) )
63 {
64     // loop to control passes
65     for ( int pass = 1; pass < size; pass++
66
67         // loop to control number of comparisons
68         for ( int count = 0; count < size - 1; count++ )
69
70             // if adjacent elements are out of order
71             if ( (*compare)( work[ count ], work[ count + 1 ] ) )
72                 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.

**fig05_25.cpp**
(4 of 5)

```
73
74 } // end function bubble
75
76 // swap values at memory locations to which
77 // element1Ptr and element2Ptr point
78 void swap( int * const element1Ptr, int * const element2Ptr )
79 {
80     int hold = *element1Ptr;
81     *element1Ptr = *element2Ptr;
82     *element2Ptr = hold;
83
84 } // end function swap
85
86 // determine whether elements are out of order
87 // for an ascending order sort
88 bool ascending( int a, int b )
89 {
90     return b < a;    // swap if b is less than a
91
92 } // end function ascending
93
```



fig05_25.cpp
(5 of 5)

fig05_25.cpp
output (1 of 1)

```
94 // determine whether elements are out of order
95 // for a descending order sort
96 bool descending( int a, int b )
97 {
98     return b > a;    // swap if b is greater than a
99
100 } // end function descending
```

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



**fig05_26.cpp**
(1 of 3)

```
1  // Fig. 5.26: fig05_26.cpp
2  // Demonstrating an array of pointers to functions.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  // function prototypes
10 void function1( int );
11 void function2( int );
12 void function3( int );
13
14 int main()
15 {
16     // initialize array of 3 pointers to functions
17     // take an int argument and return void
18     void (*f[ 3 ])( int ) = { function1, function2, function3 };
19
20     int choice;
21
22     cout << "Enter a number between 0 and 2, 3 to end: ";
23     cin >> choice;
24
```

Array initialized with names
of three functions; function
names are pointers.

**fig05_26.cpp**
(2 of 3)

```
25 // process user's choice
26 while ( choice >= 0 && choice < 3 ) {
27
28     // invoke function at location choice in array f
29     // and pass choice as an argument
30     (*f[ choice ])( choice );
31
32     cout << "Enter a number between 0 and 2, 3 to end: ";
33     cin >> choice;
34 }
35
36 cout << "Program execution compl
37
38 return 0; // indicates successful termination
39
40 } // end main
41
42 void function1( int a )
43 {
44     cout << "You entered " << a
45         << " so function1 was called\n\n";
46
47 } // end function1
48
```

Call chosen function by
dereferencing corresponding
element in array.



fig05_26.cpp
(3 of 3)

fig05_26.cpp
output (1 of 1)

```
49 void function2( int b )
50 {
51     cout << "You entered " << b
52         << " so function2 was called\n\n";
53
54 } // end function2
55
56 void function3( int c )
57 {
58     cout << "You entered " << c
59         << " so function3 was called\n\n";
60
61 } // 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

<code>char *strcpy(char *s1, const char *s2);</code>	Copies the string s2 into the character array s1 . The value of s1 is returned.
<code>char *strncpy(char *s1, const char *s2, size_t n);</code>	Copies at most n characters of the string s2 into the character array s1 . The value of s1 is returned.
<code>char *strcat(char *s1, const char *s2);</code>	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.
<code>char *strncat(char *s1, const char *s2, size_t n);</code>	Appends at most n characters of string s2 to string s1 . The first character of s2 overwrites the terminating null character of s1 . The value of s1 is returned.
<code>int strcmp(const char *s1, const char *s2);</code>	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

<pre>int strncmp(const char *s1, const char *s2, size_t n);</pre>	<p>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.</p>
<pre>char *strtok(char *s1, const char *s2);</pre>	<p>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 to-ken is returned by each call. If there are no more tokens when the function is called, NULL is returned.</p>
<pre>size_t strlen(const char *s);</pre>	<p>Determines the length of string s. The number of characters preceding the terminating null character is returned.</p>



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



fig05_28.cpp (1 of 2)

```

1  // Fig. 5.28: fig05_28.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring>    // prototypes for strcpy and strncpy
9
10 int main()
11 {
12     char x[] = "Happy Birthday to Y";
13     char y[ 25 ];
14     char z[ 15 ];
15
16     strcpy( y, x );    // copy contents of x into y
17
18     cout << "The string in array x is: "
19          << "\nThe string in array y is: "
20
21     // copy first 14 characters of x
22     strncpy( z, x, 14 );    // does not copy terminating null character.
23     z[ 14 ] = '\0';        // append '\0' to z's contents
24
25     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
Append terminating null character.



```
26
27     return 0; // indicates successful termination
28
29 }
```

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 .

fig05_28.cpp
output (1 of 1)

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



fig05_29.cpp (1 of 2)

```

1  // Fig. 5.29: fig05_29.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring>    // prototypes for strcat and strncat
9
10 int main()
11 {
12     char s1[ 20 ] = "Happy ";
13     char s2[] = "New Year ";
14     char s3[ 40 ] = "";
15
16     cout << "s1 = " << s1 << "\ns2 = " << s2;
17
18     strcat( s1, s2 );    // concatenate s2 to s1
19
20     cout << "\n\nAfter strcat(s1, s2)...\n";
21     cout << "\ns2 = " << s2;
22
23     // concatenate first 6 characters of s1 to s3
24     strncat( s3, s1, 6 ); // places '\0' after last character
25

```

<cstring> contains prototypes for `strcat` and `strncat`.

Append `s2` to `s1`.

Append first 6 characters of `s1` to `s3`.



fig05_29.cpp
(2 of 2)

fig05_29.cpp
output (1 of 1)

Append **s1** to **s3**.

```

26  cout << "\n\nAfter strncat(
27      << "\ns3 = " << s3;
28
29  strcat( s3, s1 ); // concatenate s1 to s3
30  cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31      << "\ns3 = " << s3 << endl;
32
33  return 0; // indicates successful termination
34
35 } // 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 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



fig05_30.cpp
 (1 of 2)

```

1  // Fig. 5.30: fig05_30.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 #include <cstring> // prototypes for strcmp and strncmp
13
14 int main()
15 {
16     char *s1 = "Happy New Year";
17     char *s2 = "Happy New Year";
18     char *s3 = "Happy Holidays";
19
20     cout << "s1 = " << s1 << "\ns2 = " << s2
21          << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
22          << setw( 2 ) << strcmp( s1, s2 )
23          << "\nstrcmp(s1, s3) = " << setw( 2 )
24          << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
25          << setw( 2 ) << strcmp( s3, s1 );
  
```

<cstring> contains
prototypes for **strcmp** and
strncmp.

Compare **s1** and **s2**.

Compare **s1** and **s3**.

Compare **s3** and **s1**.



Compare up to 7 characters of **s1** and **s3**

Compare up to 7 characters of **s3** and **s1**.

fig05_30.cpp

(2 of 2)

fig05_30.cpp
output (1 of 1)

```

26
27 cout << "\n\nstrncmp(s1, s3, 6) = " << strncmp(s1, s3, 6) << "\n\nstrncmp(s3, s1, 7) = " << strncmp(s3, s1, 7) << endl;
28 << setw( 2 ) << strncmp( s1, s3, 7 ) << "\n\nstrncmp(s1, s3, 6) = " << strncmp(s1, s3, 6) << "\n\nstrncmp(s3, s1, 7) = " << strncmp(s3, s1, 7) << endl;
29 << setw( 2 ) << strncmp( s1, s3, 7 ) << "\n\nstrncmp(s3, s1, 7) = " << strncmp(s3, s1, 7) << endl;
30 << "\n\nstrncmp(s3, s1, 7) = " << strncmp(s3, s1, 7) << endl;
31 << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
32
33 return 0; // indicates successful termination
34
35 } // end main

```

```

33 return 0; // indicates successful termination
34
35 } // end main

```

```

35 } // 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

- 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**



**fig05_31.cpp**
(1 of 2)

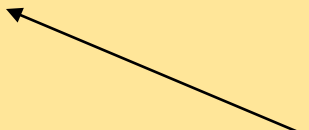
```
1  // Fig. 5.31: fig05_31.cpp
2  // Using strtok.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring> // prototype for strtok
9
10 int main()
11 {
12     char sentence[] = "This is a sentence with 7 tokens";
13     char *tokenPtr;
14
15     cout << "The string to be tokenized is: " << sentence << endl;
16     cout << "\n\nThe tokens are:\n\n";
17
18     // begin tokenization of sentence
19     tokenPtr = strtok( sentence, " " );
20
```

<cstring> contains
prototype for **strtok**.

First call to **strtok** begins
tokenization.

fig05_31.cpp
(2 of 2)

```
21 // continue tokenizing sentence until tokenPtr becomes NULL
22 while ( tokenPtr != NULL ) {
23     cout << tokenPtr << '\n';
24     tokenPtr = strtok( NULL, " " ); // get next token
25
26 } // end while
27
28 cout << "\nAfter strtok, sentence = " <
29
30 return 0; // indicates successful term
31
32 } // end main
```



Subsequent calls to **strtok**
with **NULL** as first argument
to indicate continuation.

**fig05_31.cpp**
output (1 of 1)

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



**fig05_32.cpp**
(1 of 1)

```
1  // Fig. 5.32: fig05_32.cpp
2  // Using strlen.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring> // prototype for strlen
9
10 int main()
11 {
12     char *string1 = "abcdefghijklmnopqrstuvwxyz";
13     char *string2 = "four";
14     char *string3 = "Boston";
15
16     cout << "The length of \"" << string1
17          << "\" is " << strlen( string1 )
18          << "\nThe length of \"" << string2
19          << "\" is " << strlen( string2 )
20          << "\nThe length of \"" << string3
21          << "\" is " << strlen( string3 ) << endl;
22
23     return 0; // indicates successful termination
24
25 }
```

<cstring> contains
prototype for **strlen**.

Using **strlen** to determine
length of strings.



The length of "abcdefghijklmnopqrstuvwxyz" is 26

The length of "four" is 4

The length of "Boston" is 6

fig05_32.cpp
output (1 of 1)