Chapter 5 - Pointers and Strings

Outline

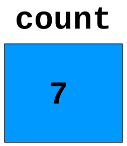
5.1 Introduction
5.2 Pointer Variable Declarations and Initialization
5.3 Pointer Operators
5.4 Calling Functions by Reference
5.5 Using `const` with Pointers
5.6 Bubble Sort Using Pass-by-Reference
5.7 Pointer Expressions and Pointer Arithmetic
5.8 Relationship Between Pointers and Arrays
5.9 Arrays 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

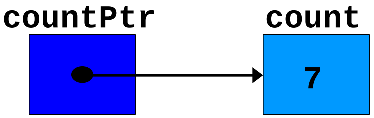
- Pointers
  - Powerful, but difficult to master
  - Simulate pass-by-reference
  - Close relationship with arrays and strings

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

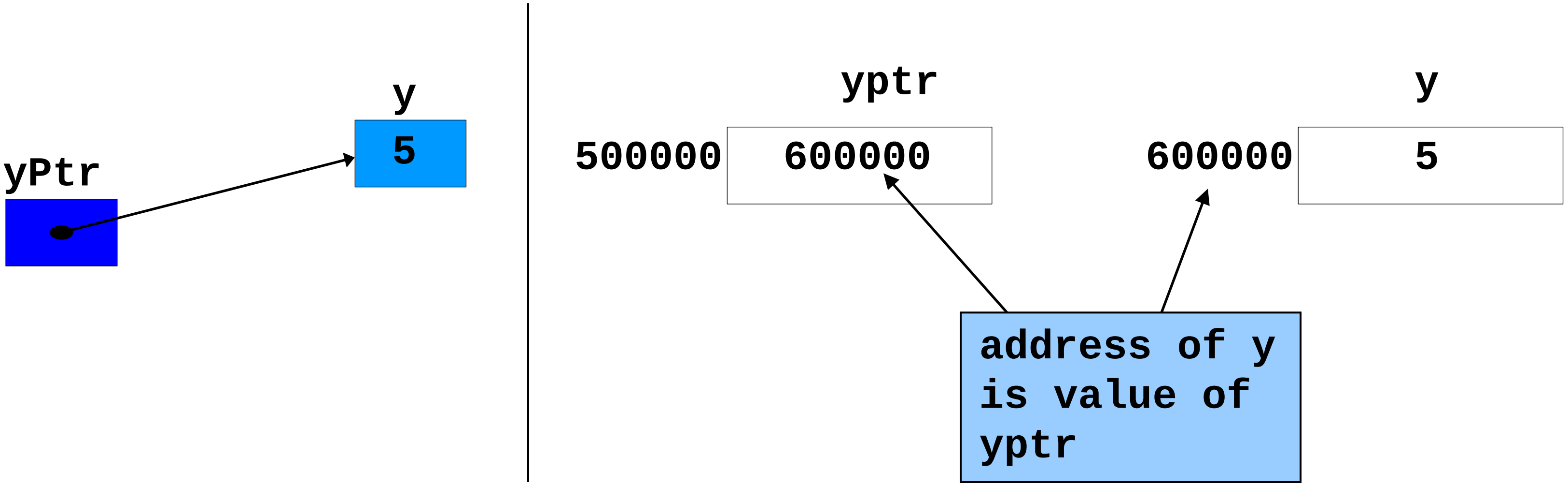
- Can declare pointers to any data type
- Pointer initialization
  - Initialized to **`0`**, **`NULL`**, or address
    - **`0`** or **`NULL`** points to nothing

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)

```
// 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

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

} // end main
```

```
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`**

fig05_04.cpp
(2 of 2)

fig05_04.cpp
output (1 of 1)

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)

```
// 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 "

   // 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 `return`ed by `cubeByValue`

```
// calculate and return cube of integer argument
int cubeByValue( int n )
{
   return n * n * n; // cube local va

} // end function cubeByValue
```

cubeByValue receives parameter passed-by-value

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

Cubes and **return**s local variable **n**

fig05_06.cpp
(2 of 2)

fig05_06.cpp
output (1 of 1)

fig05_07.cpp
(1 of 2)

```
// 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 numbe

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

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

   return 0;  // indicates successful termination

} // end main

```

Prototype indicates parameter is pointer to **`int`**

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

`cubeByReference` modified variable **`number`**

```
// calculate cube of *nPtr; modifies variable number in main
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  // cube

} // end function cubeByReference
```

```
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 *

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)

```
// 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 islo

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

```

Parameter is nonconstant pointer to nonconstant data

`convertToUppercase` modifies variable phrase

Outline

05_10.cpp
f 2)

fig05_10.cpp
output (1 of 1)

```
// convert string to uppercase letters
void convertToUppercase( char *sPtr )
{
   while ( *sPtr != '\0' ) {   // current character is not '\0'

      if ( islower( *sPtr ) )  // if character i
         *sPtr = toupper( *sPtr );  // convert to uppercase

      ++sPtr;  // move sPt

   } // end while

} // end function convertT
```

```
The phrase before conversion i
The phrase after conversion is
```

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.

fig05_11.cpp
(1 of 2)

```
// 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

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

   return 0;  // indicates successful termination

} // end main

```

Parameter is nonconstant pointer to constant data.

Pass pointer **`phrase`** to function **`printCharacters`**.

```
// sPtr cannot modify the character to which it points,
// i.e., sPtr is a "read-only" pointer
void printCharacters( const char *sPtr )
{
   for ( ; *sPtr != '\0'; sPtr++ )   // no initialization
      cout << *sPtr;

} // end function printCharacters
```

```
The string is:
print characters of a string
```

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

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

fig05_11.cpp (2 of 2)

fig05_11.cpp output (1 of 1)

Outline

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 s
14
15  } // end main
16
17  // xPtr cannot modify the va
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:\cpphtp4_examples\ch05\Fig05_12.cpp(21) : error C2166:
   l-value specifies const object
```

5.5 Using const with Pointers

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

Outline

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
10     // be modified through pt
11     // same memory location.
12     int * const ptr = &x;
13
14     *ptr = 7;  // allowed: *p
15     ptr = &y;  // error: ptr is const; can
16
17     return 0;  // indicates successful ter
18
19  } // end main
```

ptr is constant pointer to

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:\cpphtp4_examples\ch05\Fig05_13.cpp(15) : error C2166:
   l-value specifies const object
```

Outline

fig05_14.cpp
(1 of 1)

```
// 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 co
   // ptr always points to the same loc
   // at that location cannot be modified.
   const int *const ptr = &x

   cout << *ptr << endl;

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

   return 0;  // indicates successful termination

} // 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.

Outline

```
d:\cpphtp4_examples\ch05\Fig05_14.cpp(19) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch05\Fig05_14.cpp(20) : error C2166:
   l-value specifies const object
```

_14.cpp
ut (1 of 1)

Line 19 generates compiler error by attempting to modify

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 ];

```

Outline

fig05_15.cpp
(1 of 3)

```
   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 bubbl
void bubbleSort( int *array, const int s
{
   // 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 ] );
```

Declare as
(rather tha
to indicate
bubbleSort receives
single-subscripted array.

Receives size of array as argument; declared **const** to ensure **size** not modified.

Outline

fig05_15.cpp
(3 of 3)

fig05_15.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

    (size of 1 element) * (number of elements)
  - If **`sizeof( int ) = 4`**, then

```
int myArray[10];
cout << sizeof(myArray);
```

    will print 40
- **`sizeof`** can be used with
  - Variable names
  - Type names
  - Constant values

fig05_16.cpp
(1 of 2)

```
// 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
        << sizeof( array );

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

   return 0;  // indicates successful termination

} // end main

```

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

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

```
// return size of ptr
size_t getSize( double *ptr )
{
   return sizeof( ptr );

} // 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_16.cpp (2 of 2)

fig05_16.cpp output (1 of 1)

```
// 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;

```

fig05_17.cpp
(1 of 2)

```
   cout << "sizeof c = " << sizeof c
        << "\tsizeof(char) = " << sizeof( char )
        << "\nsizeof s = " << sizeof s
        << "\tsizeof(short) = " << sizeof( short )
        << "\nsizeof i = " << sizeof i
        << "\tsizeof(int) = " << sizeof( int )
        << "\nsizeof l = " << sizeof l
        << "\tsizeof(long) = " << sizeof( long )
        << "\nsizeof f = " << sizeof f
        << "\tsizeof(float) = " << sizeof( float )
        << "\nsizeof d = " << sizeof d
        << "\tsizeof(double) = " << sizeof( double )
        << "\nsizeof ld = " << sizeof ld
        << "\tsizeof(long double) = " << sizeof( long double )
        << "\nsizeof array = " << sizeof array
        << "\nsizeof ptr = " << sizeof ptr
        << endl;

   return 0;  // indicates successful termination

} // end main
```

Operator **sizeof** can be used on

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

```
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
```

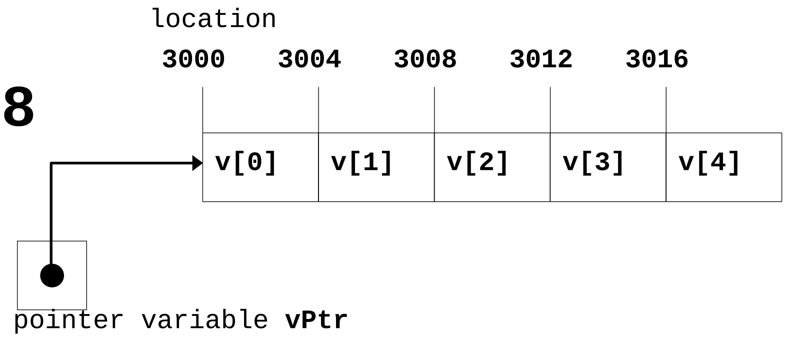
**fig05_17.cpp
output (1 of 1)**

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 **`int`**s
  - **`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 ]`**



location

3000	**3004**	**3008**	**3012**	**3016**
v[0]	v[1]	v[2]	v[3]	v[4]

pointer variable **vPtr**

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)

```cpp
// 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:\n"
        << "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.

fig05_20.cpp
(2 of 2)

```
   for ( int offset1 = 0; offset1 < 4; offset1++ )
      cout << "*(b + " << offset1 << ") = "
           << *( b + offset1 ) << '\n';

   // output array b using bPtr and array su
   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

} // end main
```

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
```

fig05_20.cpp
output (1 of 1)

fig05_21.cpp
(1 of 2)

```cpp
// 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
```

Outline

5_21.cpp
f 2)

```

} // end main

// 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

} // end function copy1

// copy s2 to s1 using pointer no
void copy2( char *s1, const char
{
   for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
      ;   // do nothing in body

} // end function copy2
```

```
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.

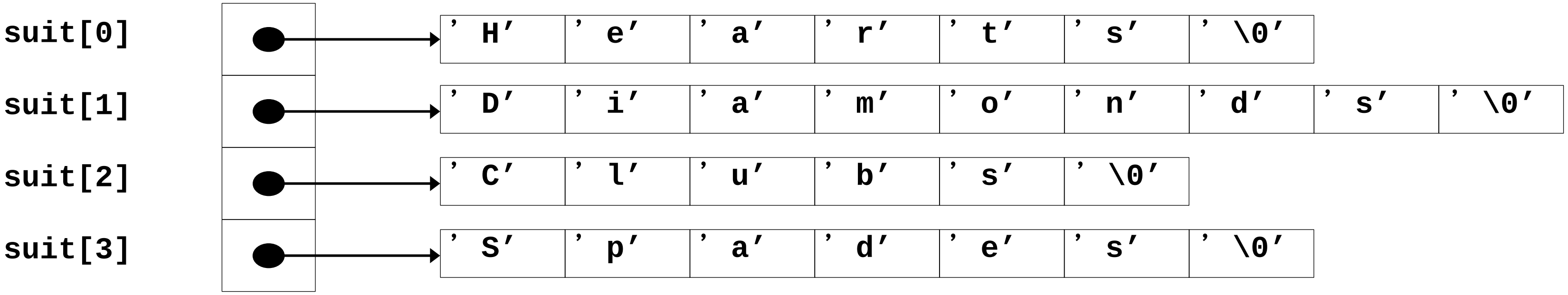
fig05_21.cpp output (1 of 1)

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[0] ’H’ ’e’ ’a’ ’r’ ’t’ ’s’ ’\0’

suit[1] ’D’ ’i’ ’a’ ’m’ ’o’ ’n’ ’d’ ’s’ ’\0’

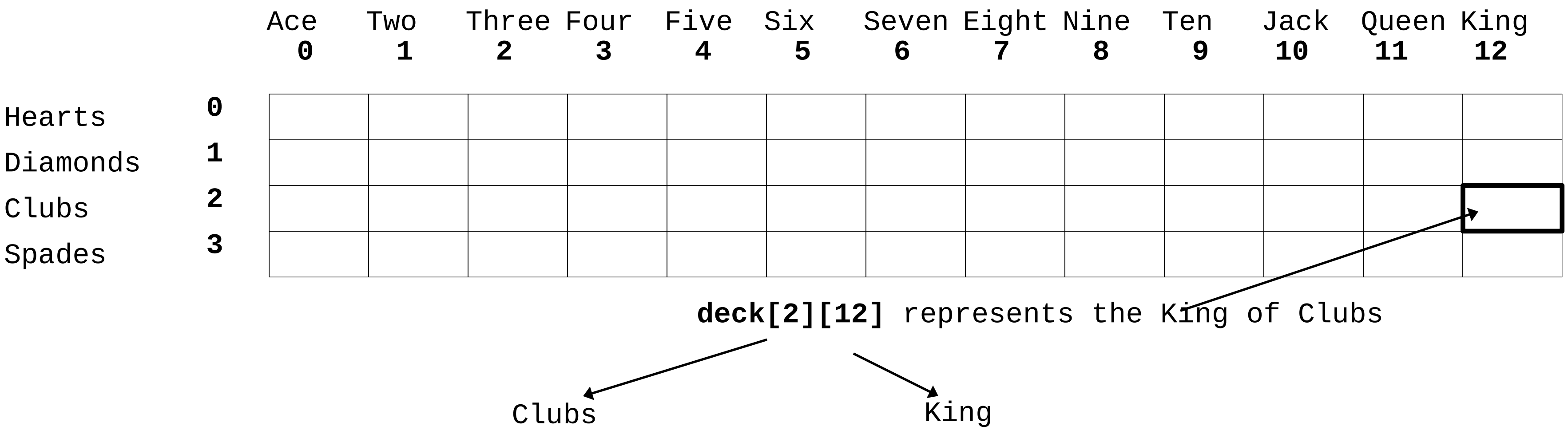
suit[2] ’C’ ’l’ ’u’ ’b’ ’s’ ’\0’

suit[3] ’S’ ’p’ ’a’ ’d’ ’e’ ’s’ ’\0’

  - **`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)



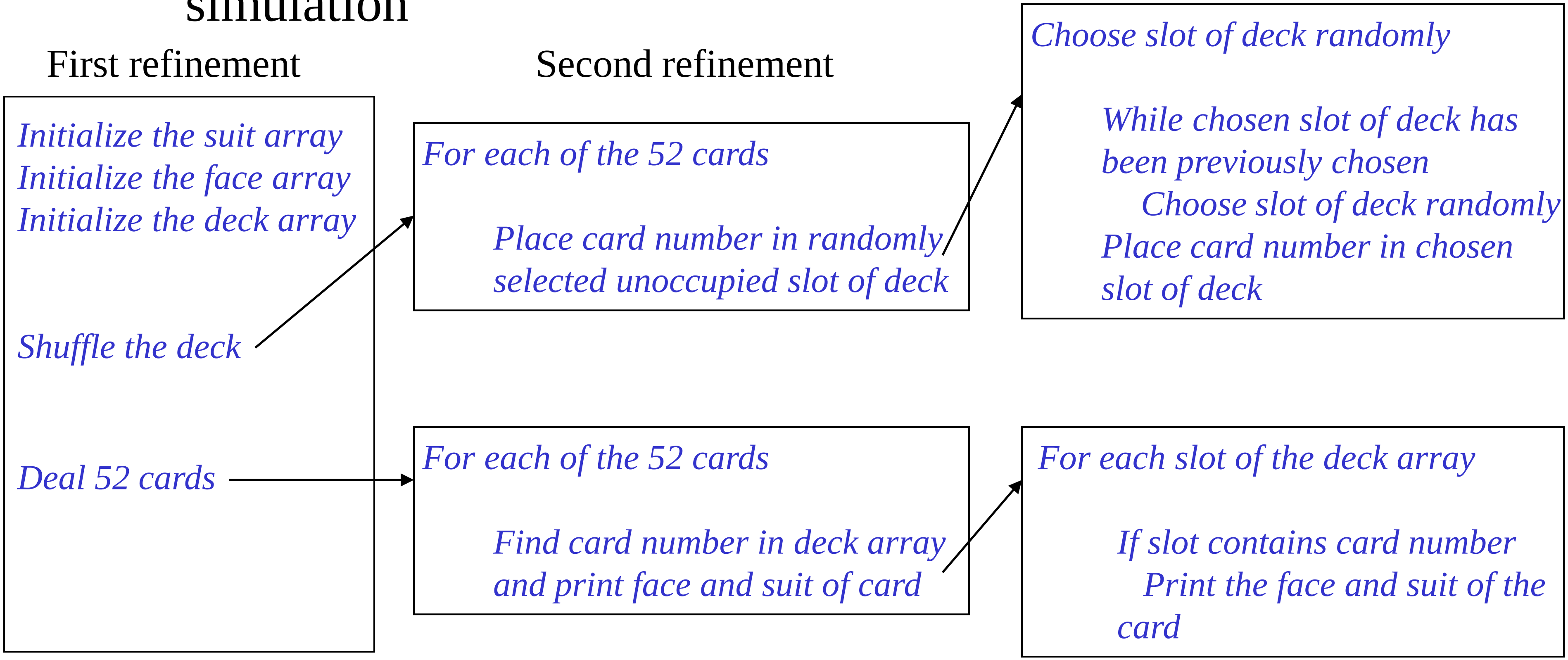
		Ace **0**	Two **1**	Three **2**	Four **3**	Five **4**	Six **5**	Seven **6**	Eight **7**	Nine **8**	Ten **9**	Jack **10**	Queen **11**	King **12**
Hearts	**0**													
Diamonds	**1**													
Clubs	**2**													
Spades	**3**													

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

  - 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)

```
// 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.

fig05_24.cpp
(2 of 4)

```
   // 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

```

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

fig05_24.cpp
(3 of 4)

```
// 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 unoccu
      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.

fig05_24.cpp
(4 of 4)

```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
            if ( wDeck[ row ][ column ] == card ) {
               cout << setw( 5 ) << right << wFace[ colu
                    << " of " << setw( 8 ) << left
                    << wSuit[ row ]
                    << ( card % 2 == 0 ? '\n' : '\t' );

            } // end if

} // end function deal
```

Cause face to be output right

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

```
 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
```

Outline

fig05_24.cpp
output (1 of 1)

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)

```
// 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 agrument to specify descending sorting order
   else {
      bubble( a, arraySize, descending );
      cout << "\nData items in descending order\n";
   }

```

fig05_25.cpp
(2 of 5)

Outline

fig05_25.cpp
(3 of 5)

```
   // output sorted array
   for ( counter = 0; counter < arraySize; counter++ )
      cout << setw( 4 ) << a[ counter ];

   cout << endl;

   return 0;  // indicates successful termination

} // end main

// multipurpose bubble sort; parameter comp
// the comparison function that determines
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 c
      for ( int count = 0; count < s

         // if adjacent elements are
         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.

fig05_25.cpp
(4 of 5)

```

} // end function bubble

// 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

// 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 ascending

```

Outline

```
// 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

} // 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
```

fig05_25.cpp
(5 of 5)

fig05_25.cpp
output (1 of 1)

5.11 Function Pointers

- Arrays of pointers to functions
  - Menu-driven systems
  - Pointers to each function stored in array of pointers to functions
    - All functions must have same return type and same parameter types
  - Menu choice → subscript into array of function pointers

fig05_26.cpp
(1 of 3)

```cpp
// 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 fu
   // 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;

```

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

```
   // 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 compl

   return 0;  // indicates successful termination

} // end main

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

} // end function1

```

Call chosen function by dereferencing corresponding element in array.

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

```
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.
```

fig05_26.cpp
(3 of 3)

fig05_26.cpp
output (1 of 1)

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

char *strcpy(char *s1, const char *s2);	Copies the string **s2** into the character array **s1**. The value of **s1** is returned.
char *strncpy(char *s1, const char *s2, size_t n);	Copies at most **n** characters of the string **s2** into the character array **s1**. The value of **s1** is returned.
char *strcat(char *s1, const char *s2);	Appends the string **s2** to the string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
char *strncat(char *s1, const char *s2, size_t n);	Appends at most **n** characters of string **s2** to string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
int strcmp(const char *s1, const char *s2);	Compares the string **s1** with the string **s2**. The function returns a value of zero, less than zero or greater than zero if **s1** is equal to, less than or greater than **s2**, respectively.

5.12.2 String Manipulation Functions of the String-handling Library

`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to **`n`** characters of the string **`s1`** with the string **`s2`**. The function returns zero, less than zero or greater than zero if **`s1`** is equal to, less than or greater than **`s2`**, respectively.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to **`strtok`** breaks string **`s1`** into “tokens”—logical pieces such as words in a line of text—delimited by characters contained in string **`s2`**. The first call contains **`s1`** as the first argument, and subsequent calls to continue tokenizing the same string contain **`NULL`** as the first argument. A pointer to the current to-ken is returned by each call. If there are no more tokens when the function is called, **`NULL`** is returned.
`size_t strlen( const char *s );`	Determines the length of string **`s`**. The number of characters preceding the terminating null character is returned.

◀ ▶

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)

```
// Fig. 5.28: fig05_28.cpp
// Using strcpy and strncpy.
#include <iostream>

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

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

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

   strcpy( y, x );  // copy contents of x into y

   cout << "The string in array x is: "
        << "\nThe string in array y is:

   // copy first 14 characters of x
   strncpy( z, x, 14 );  // does not cop
   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.

```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`**.

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)

```
// 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

```

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

Append **s2** to **s1**.

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

```
   cout << "\n\nAfter strncat(
        << "\ns3 = " << s3;

   strcat( s3, s1 );  // concatenate s1 to s3
   cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
        << "\ns3 = " << s3 << endl;

   return 0;  // indicates successful termination

} // end main
```

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

```
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
```

fig05_29.cpp (2 of 2)

fig05_29.cpp output (1 of 1)

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
  - `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)

```cpp
// 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 << "\ns2 = " << s2
        << "\ns3 = " << s3 << "\n\nstrcmp(s1
        << setw( 2 ) << strcmp( s1, s2 )
        << "\nstrcmp(s1, s3) = " << setw( 2 )
        << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
        << setw( 2 ) << strcmp( s3, s1 );
```

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

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

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

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

Outline

```

   cout << "\n\nstrncmp(s1, s3, 6) = " <
        << strncmp( s1, s3, 6 ) << "\nstrn
        << setw( 2 ) << strncmp( s1, s3, 7 )
        << "\nstrncmp(s3, s1, 7) = "
        << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;

   return 0;  // indicates successful termination

} // end main
```

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)

```
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)

```
// Fig. 5.31: fig05_31.cpp
// Using strtok.
#include <iostream>

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

#include <cstring>  // prototype for strtok

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

   cout << "The string to be tokeni
        << "\n\nThe tokens are:\n\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 = " <

   return 0;  // indicates successful term

} // end main
```

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

fig05_31.cpp
(2 of 2)

```
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
```

fig05_31.cpp
output (1 of 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

```cpp
// 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
```

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

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

fig05_32.cpp
(1 of 1)

```
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)