IS 0020
Program Design and Software Tools
Introduction to C++ Programming

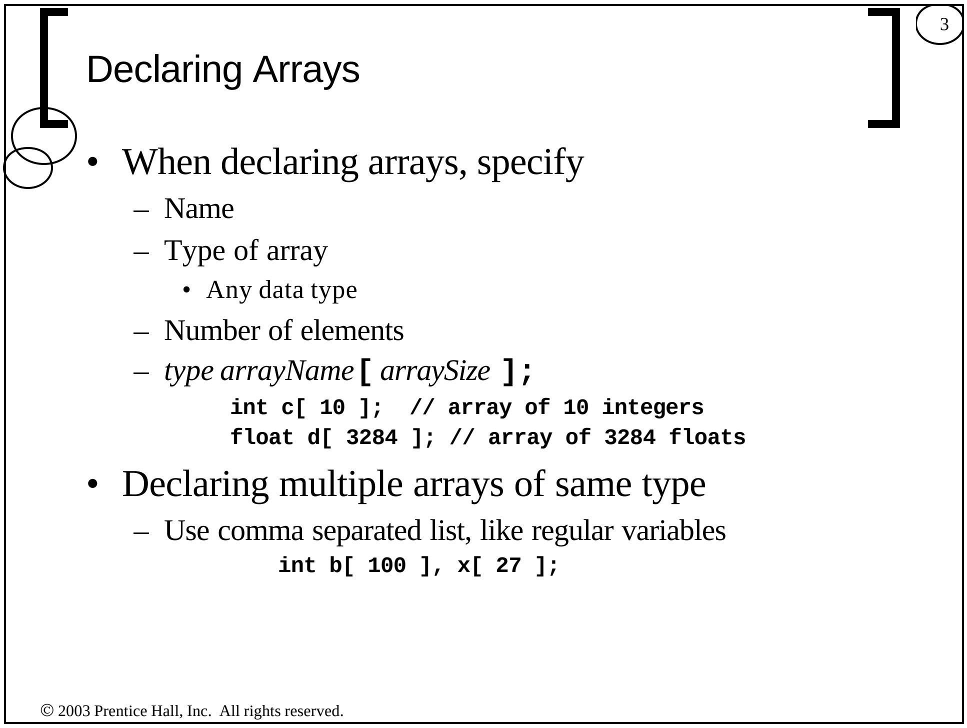
Lecture 3
Arrays & Pointers

Jan 13, 2004

Arrays

- Array
  - Consecutive group of memory locations
  - Same name and type (**`int`**, **`char`**, etc.)
- To refer to an element
  - Specify array name and position number (index)
  - Format: arrayname[position number]
  - First element at position 0
- N-element array c

  `c[ 0 ]`, **`c[ 1 ]`** … **`c[ n - 1 ]`**
  - Nth element as position N-1



3

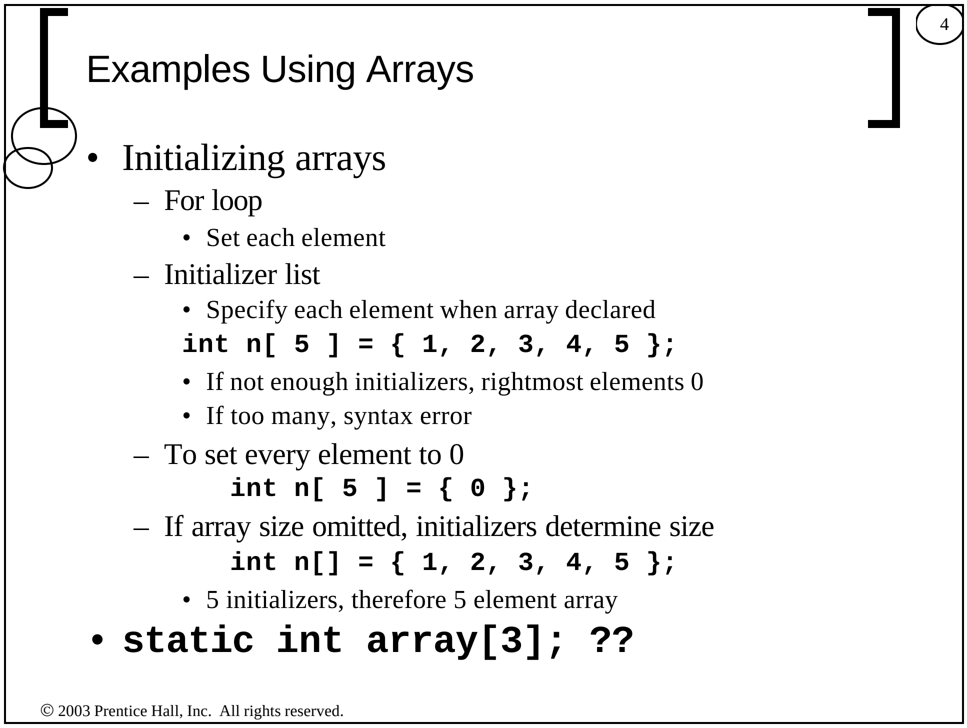
Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
    - Any data type
  - Number of elements
  - *type arrayName* `[` *arraySize* `];`

```
int c[ 10 ];  // array of 10 integers
float d[ 3284 ]; // array of 3284 floats
```

- Declaring multiple arrays of same type
  - Use comma separated list, like regular variables

```
int b[ 100 ], x[ 27 ];
```



4

Examples Using Arrays

- Initializing arrays
  - For loop
    - Set each element
  - Initializer list
    - Specify each element when array declared

```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

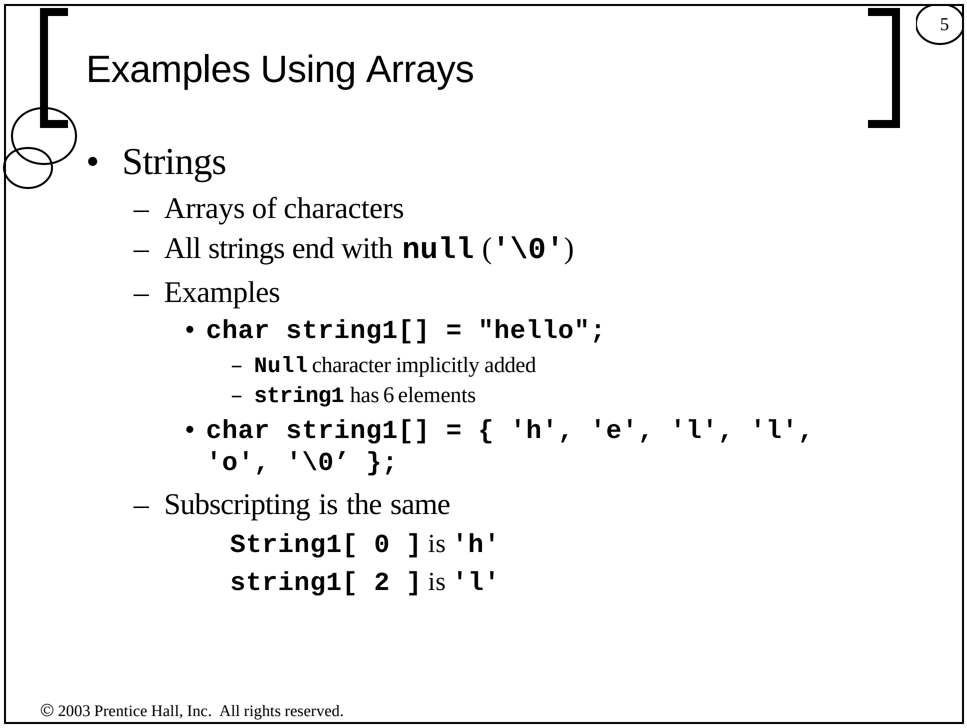
    - If not enough initializers, rightmost elements 0
    - If too many, syntax error
  - To set every element to 0

```
int n[ 5 ] = { 0 };
```

  - If array size omitted, initializers determine size

```
int n[] = { 1, 2, 3, 4, 5 };
```

    - 5 initializers, therefore 5 element array
- **`static int array[3]; ??`**



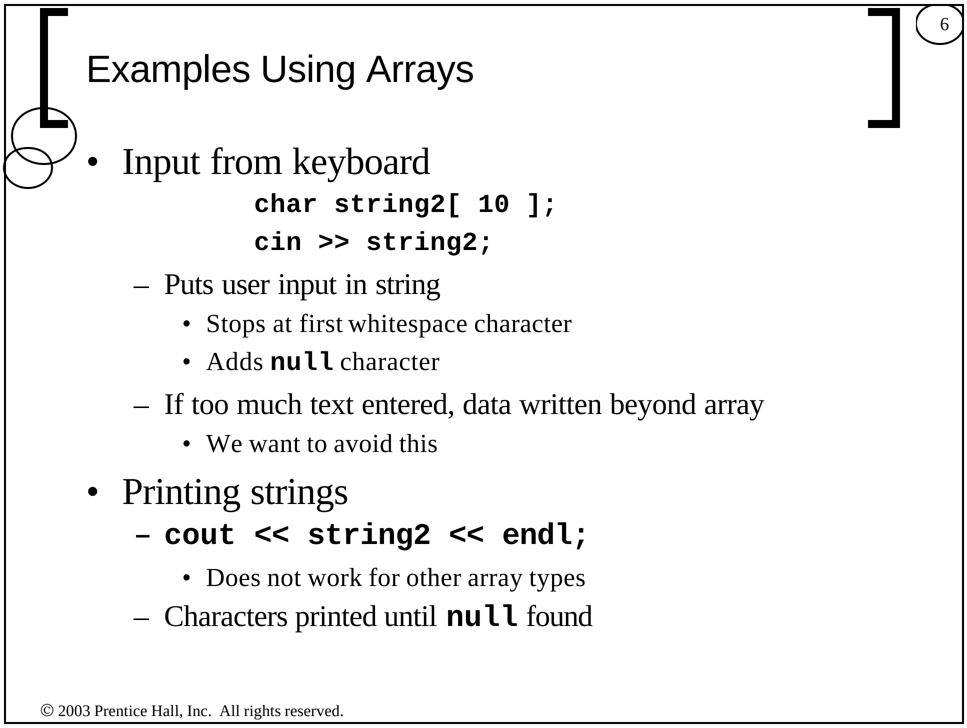
5

Examples Using Arrays

- Strings
  - Arrays of characters
  - All strings end with **`null`** (**`'\0'`**)
  - Examples
    - **`char string1[] = "hello";`**
      - **`Null`** character implicitly added
      - **`string1`** has 6 elements
    - **`char string1[] = { 'h', 'e', 'l', 'l', 'o', '\0’ };`**
  - Subscripting is the same

    `String1[ 0 ]` is **`'h'`**

    `string1[ 2 ]` is **`'l'`**



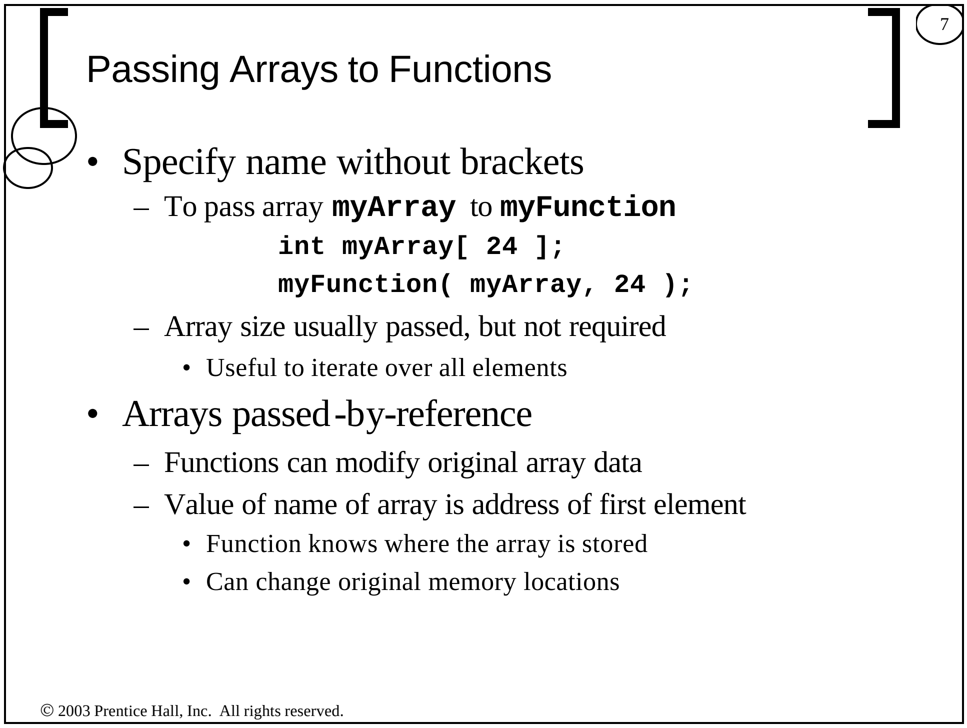
6

Examples Using Arrays

- Input from keyboard

```
char string2[ 10 ];
cin >> string2;
```

  - Puts user input in string
    - Stops at first whitespace character
    - Adds **`null`** character
  - If too much text entered, data written beyond array
    - We want to avoid this
- Printing strings
  - **`cout << string2 << endl;`**
    - Does not work for other array types
  - Characters printed until **`null`** found

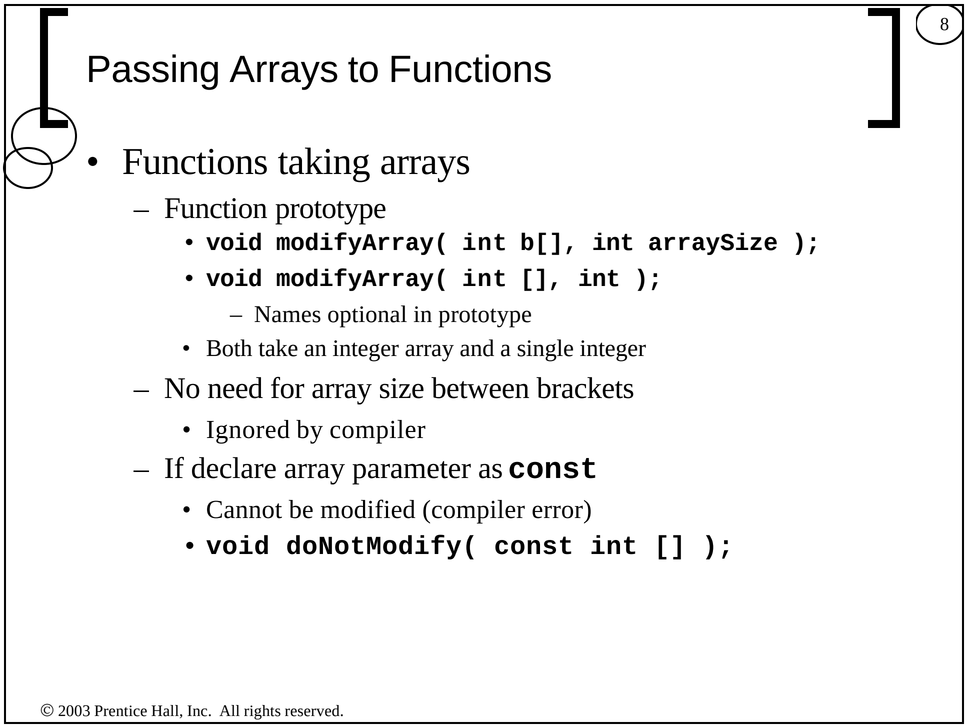


7

Passing Arrays to Functions

- Specify name without brackets
  - To pass array **`myArray`** to **`myFunction`**

    ```
    int myArray[ 24 ];
    myFunction( myArray, 24 );
    ```
  - Array size usually passed, but not required
    - Useful to iterate over all elements
- Arrays passed-by-reference
  - Functions can modify original array data
  - Value of name of array is address of first element
    - Function knows where the array is stored
    - Can change original memory locations



8

Passing Arrays to Functions

- Functions taking arrays
  - Function prototype
    - **`void modifyArray( int b[], int arraySize );`**
    - **`void modifyArray( int [], int );`**
      - Names optional in prototype
    - Both take an integer array and a single integer
  - No need for array size between brackets
    - Ignored by compiler
  - If declare array parameter as **`const`**
    - Cannot be modified (compiler error)
    - **`void doNotModify( const int [] );`**

Sorting Arrays

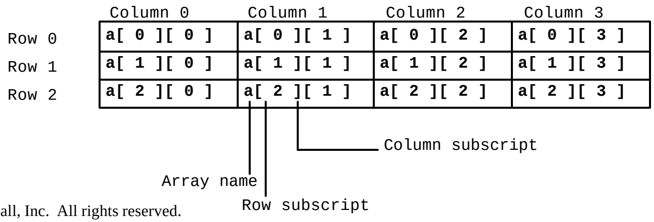
- Example:
  - Go left to right, and exchange elements as necessary
    - One pass for each element
  - Original: 3 4 2 7 6
  - Pass 1: 3 2 4 6 7 (elements exchanged)
  - Pass 2: 2 3 4 6 7
  - Pass 3: 2 3 4 6 7 (no changes needed)
  - Pass 4: 2 3 4 6 7
  - Pass 5: 2 3 4 6 7
  - Small elements "bubble" to the top (like 2 in this example)

Multiple-Subscripted Arrays

- Multiple subscripts
  - **a[i][j]**
  - Tables with rows and columns
  - Specify row, then column
  - “Array of arrays”
    - **a[0]** is an array of 4 elements
    - **a[0][0]** is the first element of that array

- To initialize
  - Default of **0**
  - Initializers grouped by row in braces

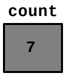
```
int b[ 2 ][ 2 ]={ { 1, 2 }, { 3, 4 } };
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

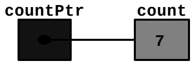


	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

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

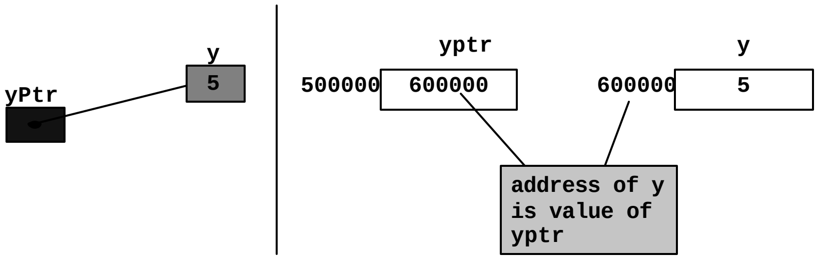
```
int *myPtr1, *myPtr2;
```

Pointer Operators

- `&` (address operator)
  - Returns memory address of its operand
  - Example

```
int y = 5;
int *yPtr;
yPtr = &y;
```

  - `yPtr` “points to” `y`
  - `*` - indirection/ dereferencing operator)
  - `*yPtr` returns `y` dereferenced pointer is lvalue
    - `*yptr = 9  ??`



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
- Arguments passed to function using reference arguments
  - Modify original values of arguments
  - More than one value “returned”

```
int Cube(int *x) {  …}
```

Function call: `Cube(&a)`

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

Outline

```
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 in
10     // be modified through p
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
```

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

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.

fig05_13.cpp
(1 of 1)

fig05_13.cpp
output (1 of 1)

Outline

```
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 c
13     // ptr always points to the same lo
14     // at that location cannot be modified.
15     const int *const ptr = &
16
17     cout << *ptr << endl;
18
19     *ptr = 7;  // error: *pt                          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.

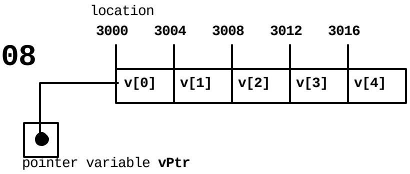
fig05_14.cpp
(1 of 1)

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 ]`**

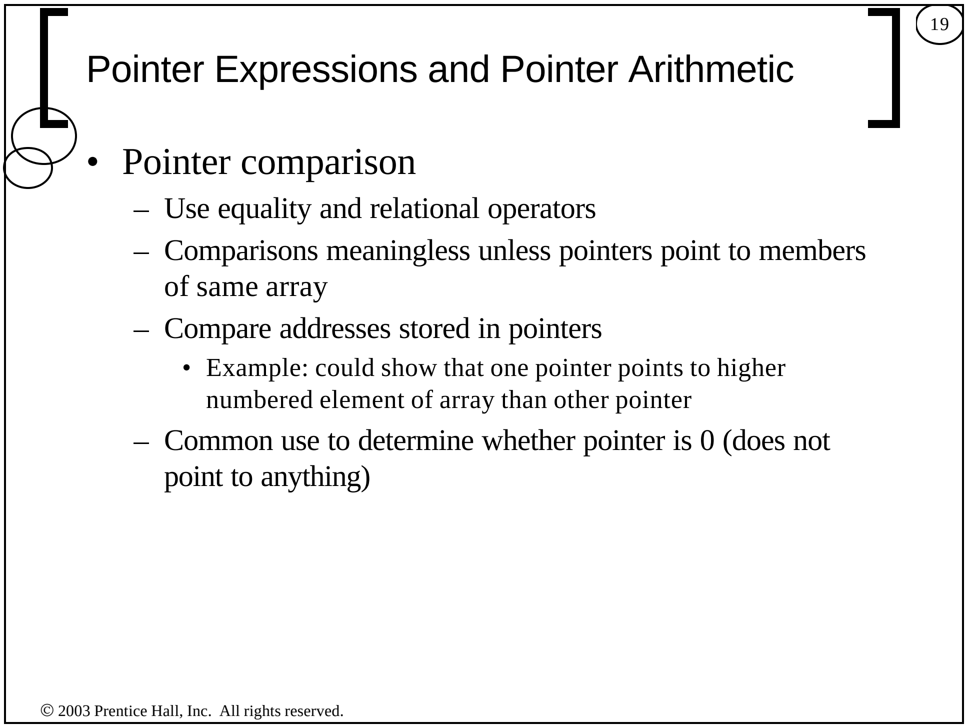


Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements between two addresses

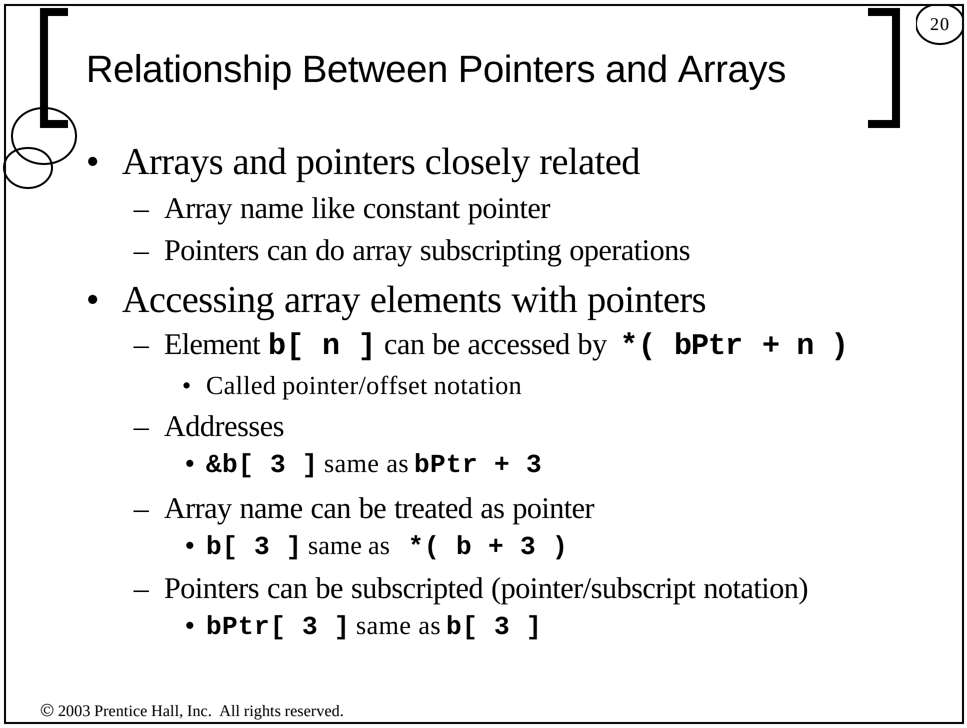
```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
vPtr2 - vPtr == 2
```

- 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



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)



Relationship Between Pointers and Arrays

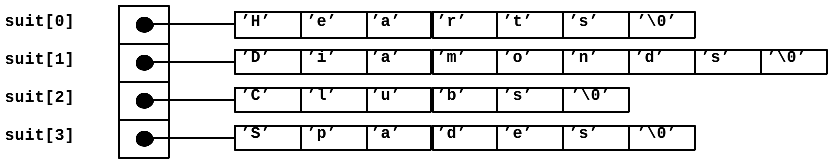
- Arrays and pointers closely related
  - – Array name like constant pointer
  - – Pointers can do array subscripting operations
- Accessing array elements with pointers
  - – Element **`b[ n ]`** can be accessed by **`*( bPtr + n )`**
    - Called pointer/offset notation
  - – Addresses
    - **`&b[ 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 ]`**

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

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

Outline

fig05_25.cpp
(1 of 5)

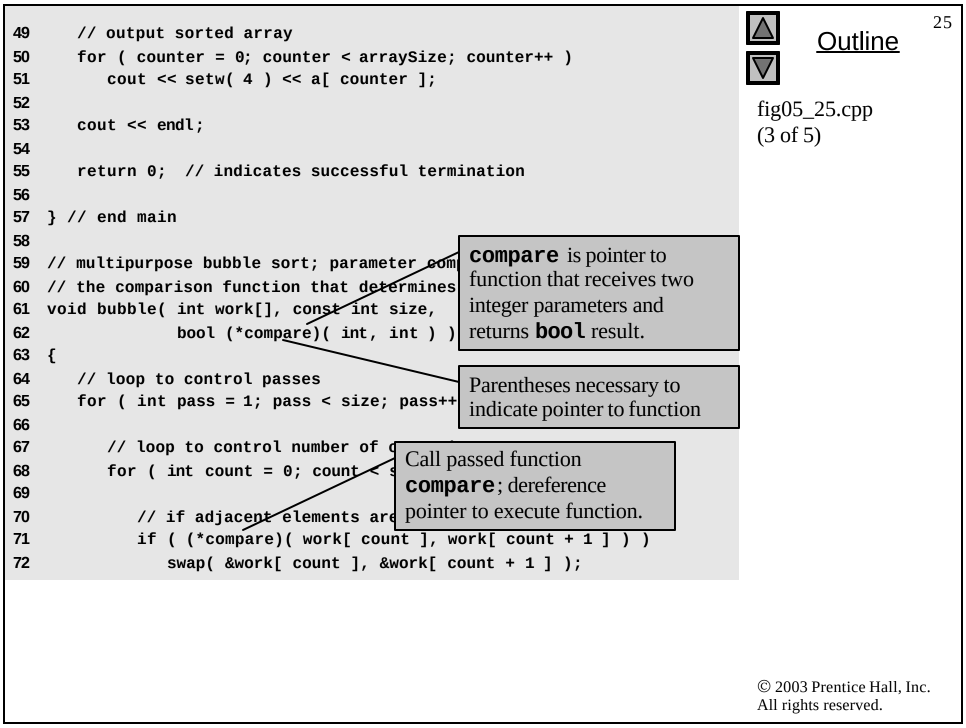
```
// Fig. 5.25: fig05_25.cpp
// Multipurpose sorting program using function pointers.
#include <iostream>

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.

Outline

fig05_25.cpp
(2 of 5)

```
   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";
   }

```



25

Outline

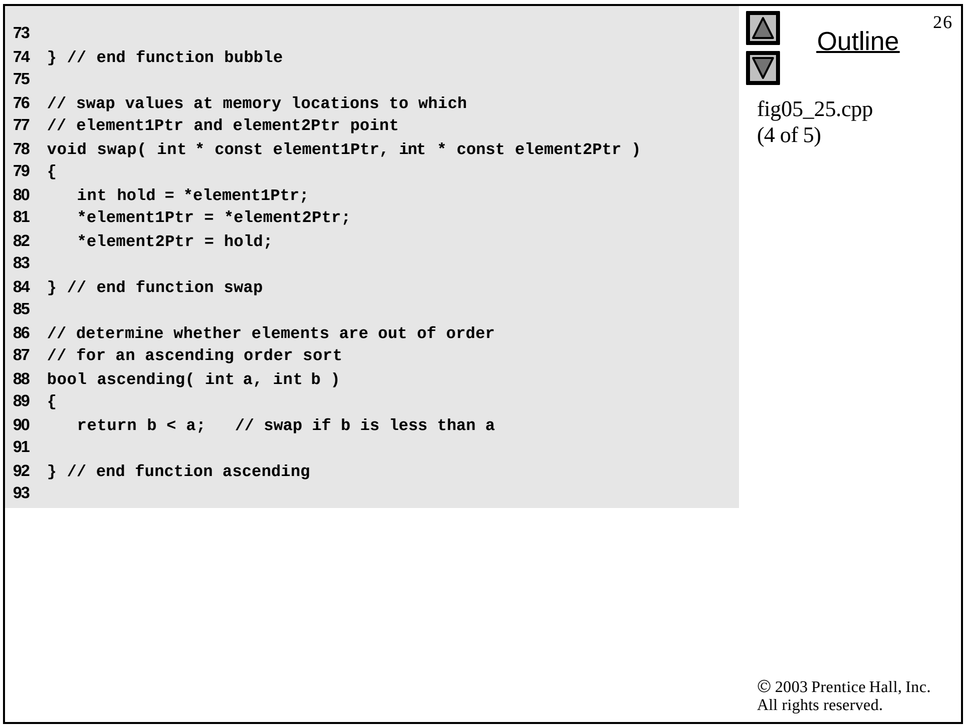
fig05_25.cpp
(3 of 5)

```
   // 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.



26

Outline

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

fig05_25.cpp
(5 of 5)

```
// 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
```

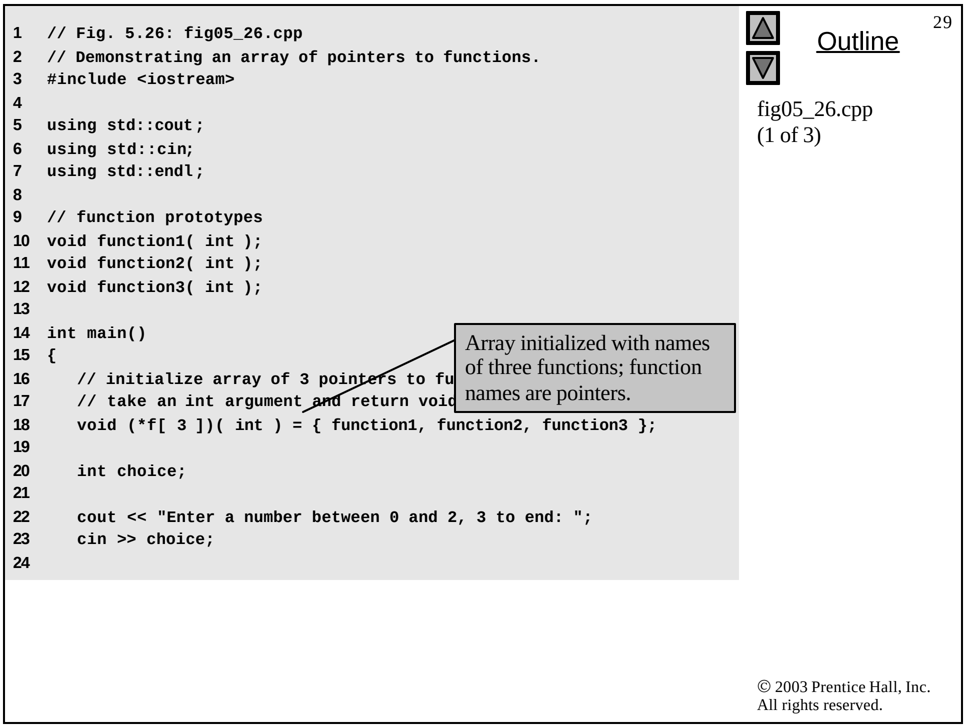
fig05_25.cpp
output (1 of 1)

```
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
```

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

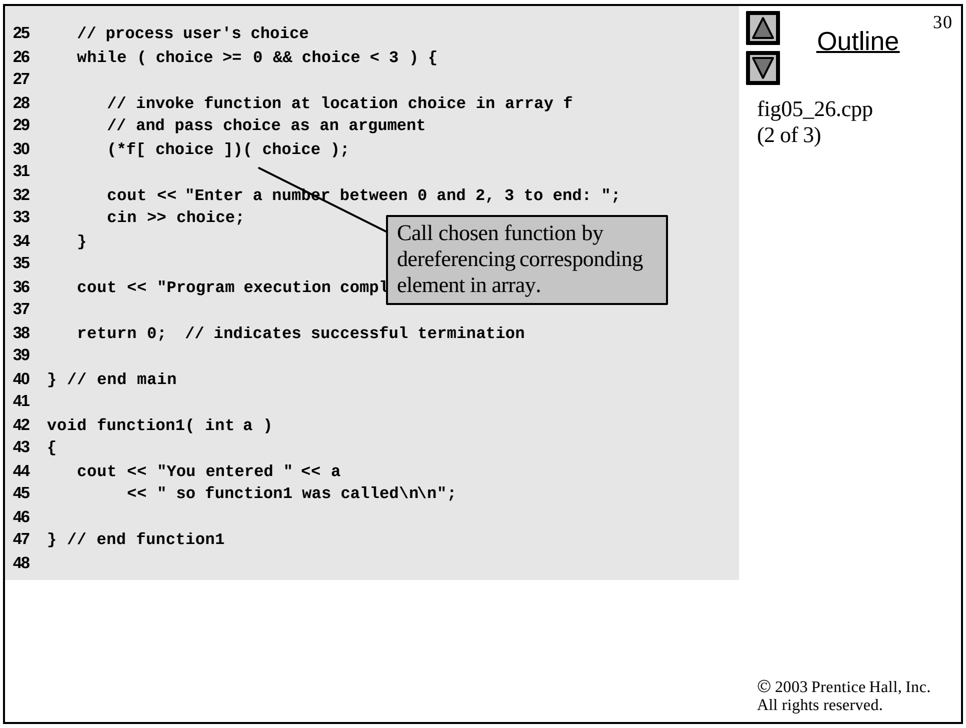


Outline

fig05_26.cpp
(1 of 3)

```
// 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.



Outline

fig05_26.cpp
(2 of 3)

```
   // 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.

Outline

fig05_26.cpp
(3 of 3)

fig05_26.cpp
output (1 of 1)

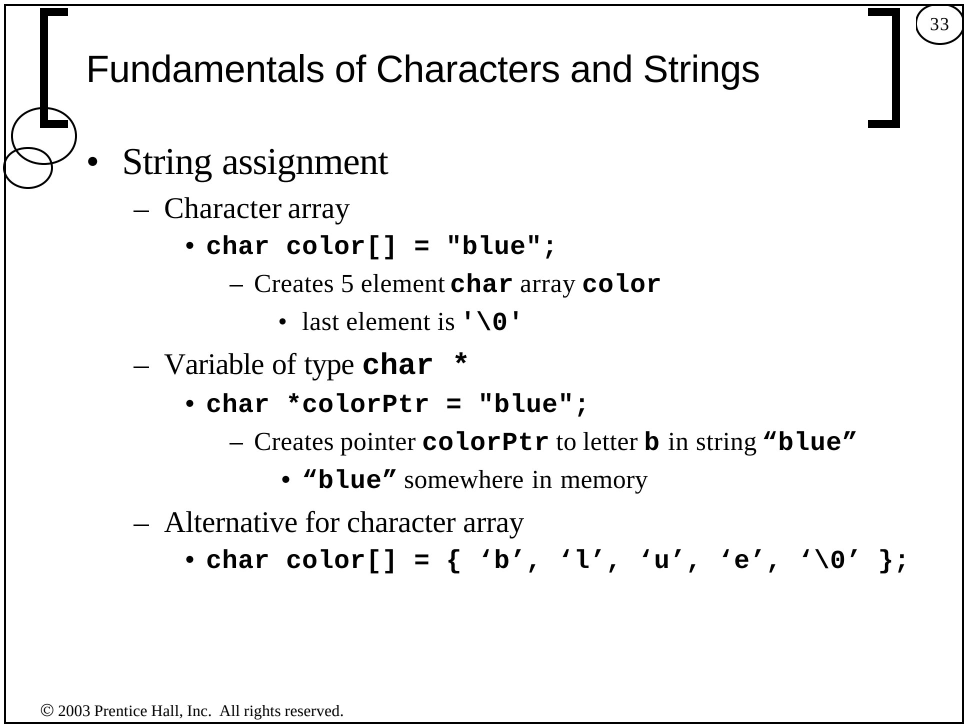
```
void function2( int b )
{
   cout << "You entered " << b
        << " so function2 was called\n\n";

} // 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.
```

Fundamentals of Characters and Strings

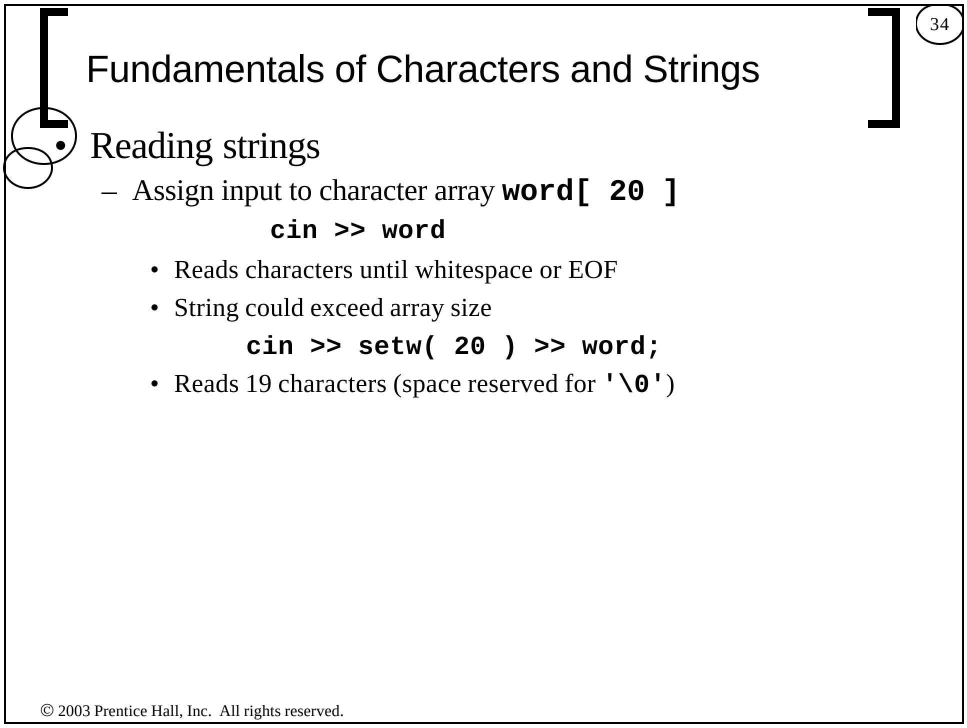
- Character constant
  - Integer value represented as character in single quotes
  - `'z'` is integer value of `z`
    - `122` in ASCII
- String
  - Series of characters treated as single unit
  - Can include letters, digits, special characters `+`, `-`, `*` ...
  - String literal (string constants)
    - 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



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’ };`



Fundamentals of Characters and Strings

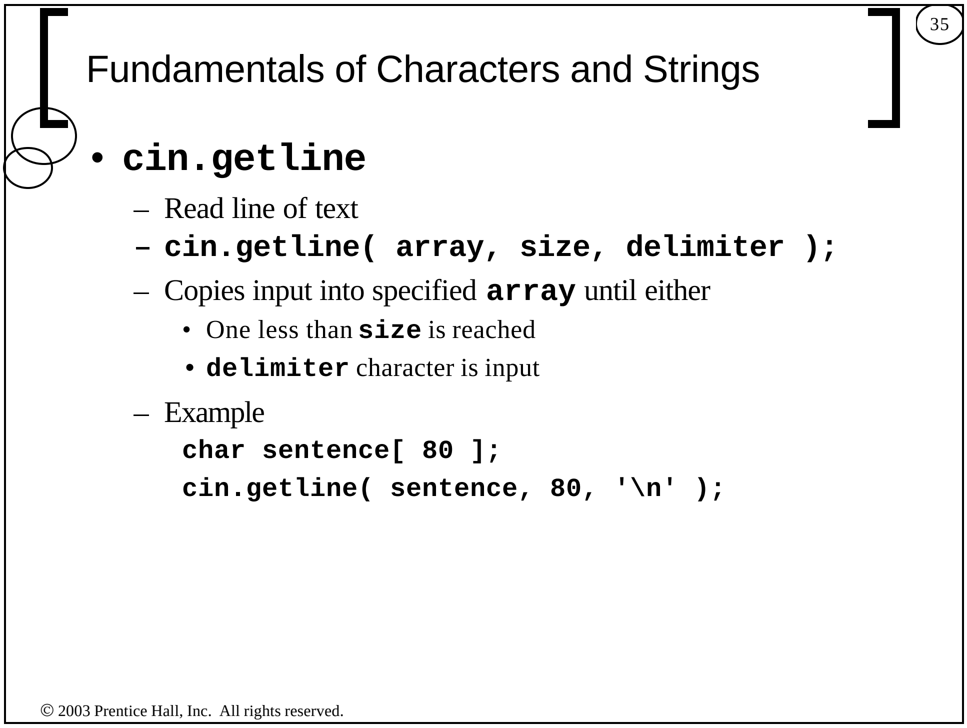
- Reading strings
  - Assign input to character array `word[ 20 ]`

    ```
    cin >> word
    ```

    - Reads characters until whitespace or EOF
    - String could exceed array size

    ```
    cin >> setw( 20 ) >> word;
    ```

    - Reads 19 characters (space reserved for `'\0'`)

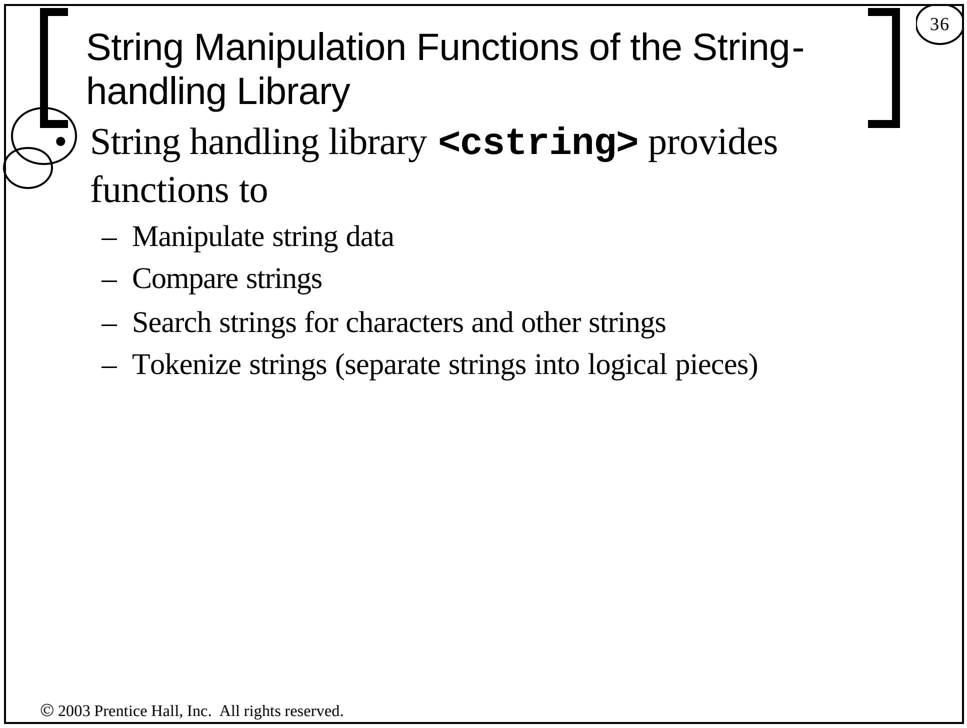


35

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

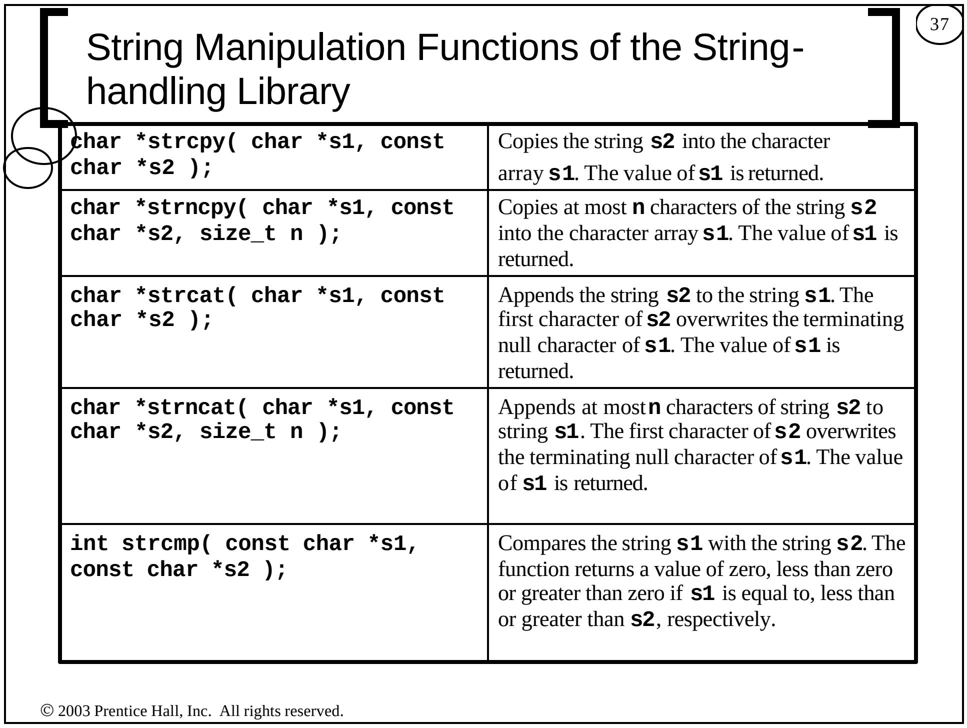
```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```



36

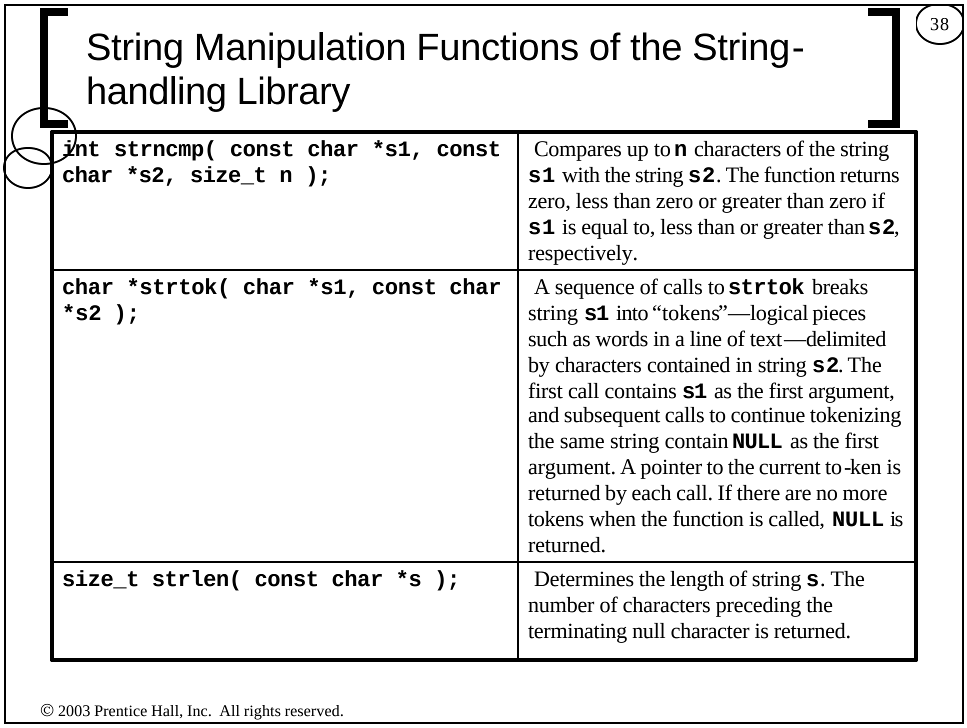
String Manipulation Functions of the String-handling Library

- 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)



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.



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.