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

Lecture 2
Arrays & Pointers
May 17, 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

Arrays

• Array elements like other variables
  – Assignment, printing for an integer array c
    
    c[ 0 ] = 3;
    cout << c[ 0 ];
  
  • Can perform operations inside subscript
    
    c[ 5 - 2 ] same as c[3]

Declaring Arrays

• When declaring arrays, specify
  – Name
  – Type of array
  – Array 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 a[ 100 ], b[ 27 ];

© 2003 Prentice Hall, Inc. All rights reserved.
Examples Using Arrays

- **Initializing arrays**
  - For loop
    - Set each element
  - Initializer list
    - Specify each element when array declared
    ```c
    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 same value
      ```c
      int n[5] = { 0 };
      ```
    - If array size omitted, initializers determine size
      ```c
      int n[] = { 1, 2, 3, 4, 5 };
      ```
      - 5 initializers, therefore 5 element array

- **Strings**
  - Arrays of characters
  - All strings end with null (\0)
  - Examples
    ```c
    char string1[] = "hello";
    ```
    - Null character implicitly added
    ```c
    char string1[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
    ```
    - Subscripting is the same
      ```c
      String1[0] = 'h'
      ```
      ```c
      String1[2] = 'l'
      ```

- **Input from keyboard**
  ```c
  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**
  ```c
  cout << string2 << endl;
  ```
  - Does not work for other array types
  - Characters printed until null found

- **Recall static storage**
  - If static, local variables save values between function calls
  - Visible only in function body
  - Can declare local arrays to be static
    - Initialized to zero
    ```c
    static int array[3];
    ```
  - If not static
    - Created (and destroyed) in every function call
Passing Arrays to Functions

- Specify name without brackets
  - To pass array `myArray` to `myFunction`
    ```c
    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
  - Individual array elements passed-by-value
    - Like regular variables
    - `square(myArray[3]);`

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

Pointers

- Pointers
  - Powerful, but difficult to master
  - Simulate pass-by-reference
  - Close relationship with arrays and strings
- Can declare pointers to any data type
- Pointer initialization
  - Initialized to 0, NULL, or address
    - 0 or NULL points to nothing

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
    ```c
    int *myPtr;
    ```
    declares pointer to int, pointer of type int *
  - Multiple pointers require multiple asterisks
    ```c
    int *myPtr1, *myPtr2;
    ```
Pointer Operators

- \& (address operator)
  - Returns memory address of its operand
  - Example
    
    ```
    int y = 5;
    int *yPtr;
    yPtr = &y;    // yPtr gets address of y
    ```
  - \&y “points to” \(y\)

- \* (indirection/dereferencing operator)
  - Returns synonym for object its pointer operand points to
  - Example
    
    ```
    int yPtr = y;  // yPtr points to y
    *yPtr = 9;    // assigns 9 to y
    ```
  - \* and \& are inverses of each other

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”

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

int main()
{
    int x, y;

    // ptr is a constant pointer to an integer that can
    // be modified through ptr, but ptr always points to the
    // same memory location.
    int * const ptr = &x;

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

    return 0; // indicates successful termination
}

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

// Fig. 5.14: fig05_14.cpp
// Attempting to modify a constant pointer to constant data.

#include <iostream>

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

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

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

    cout << *ptr << endl;

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

    return 0;  // indicates successful termination
} // end main
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
  - Each element of `suit` points to `char *` (a string)
  - Array does not store strings, only pointers to strings

Function Pointers

- Calling functions using pointers
  - Assume parameter:
    - `bool ( *compare ) ( int, int )`
  - Execute function with either
    - `*( *compare ) ( int1, int2 )`
      - Dereference pointer to function to execute
  OR
    - `compare( int1, int2 )`
      - Could be confusing
        - User may think `compare` name of actual function in program
// Fig. 5.25: fig05_25.cpp
// Multipurpose sorting program using function pointers.
#include <iostream>

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

#include <iomanip>

using std::setw;

// prototypes
void bubble( int[], const int, bool (*)( int, int ) );
void swap( int* const, int* const );
bool ascending( int, int );
bool descending( int, int );

int main()
{
    const int arraySize = 10;
    int order;
    int counter;
    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

    cout << "Enter 1 to sort in ascending order,
     Enter 2 to sort in descending order: ";
    cin >> order;
    cout << "
Data items in original order
";

    // 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 << "
Data items in ascending order
";
    }

    // sort array in descending order; pass function descending
    // as an argument to specify descending sorting order
    else {
        bubble( a, arraySize, descending );
        cout << "
Data items in descending order
";
    }

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

    return 0;  // indicates successful termination
}

// multipurpose bubble sort; parameter compare is a pointer to
// the comparison function that determines sorting order
void bubble( int work[], const int size, bool (*compare)( int, int ) )
{
    // loop to control passes
    for ( int pass = 1; pass < size; pass++ )
    {
        // loop to control number of comparisons per pass
        for ( int count = 0; count < size - 1; count++ )
            // if adjacent elements are out of order, swap them
            if ( (*compare)( work[ count ], work[ count + 1 ] ) )
                swap( &work[ count ], &work[ count + 1 ] );
    }
}

// swap values at memory locations to which
// element1Ptr and element2Ptr point
void swap( int* const element1Ptr, int* const element2Ptr )
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}

// determine whether elements are out of order
// for an ascending order sort
bool ascending( int a, int b )
{
    return b < a;   // swap if b is less than a
}

// determine whether elements are out of order
// for a descending order sort

bool descending( int a, int b )
{
    return b > a;   // swap if b is greater than a
}

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1
Data items in original order
2 6 4 8 10 12 89 68 45 37
Data items in ascending order
2 4 6 8 10 12 37 45 68 89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2
Data items in original order
2 6 4 8 10 12 89 68 45 37
Data items in descending order
89 68 45 37 12 10 8 6 4 2

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

Array initialized with names of three functions; function names are pointers.
void function2( int b )
{
    cout << "You entered " << b 
    << " so function2 was called

";
} // end function2

void function3( int c )
{
    cout << "You entered " << c 
    << " so function3 was called

";
} // 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

- **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' );
    ```