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
    - 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 ];
```

Examples Using Arrays

5

- 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 same value

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

  - If array size omitted, initializers determine size

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

    - 5 initializers, therefore 5 element array

Examples Using Arrays

6

- 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'
```


Examples Using Arrays

7

- 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

Examples Using Arrays

8

- 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

```
static int array[3];
```

- If not static
  - Created (and destroyed) in every function call

Passing Arrays to Functions

9

- 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

Passing Arrays to Functions

10

- 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]);**

Passing Arrays to Functions

11

- 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

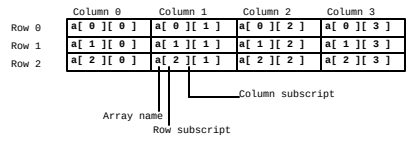
12

- 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

13

- 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



Multiple-Subscripted Arrays

14

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

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
                    Row 0      Row 1
```

1	2
3	4

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

1	0
3	4

Pointers

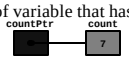
15

- 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

16

- 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;
```

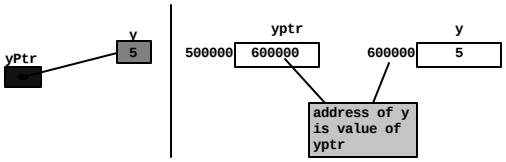
Pointer Operators

17

- **&** (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**



Pointer Operators

18

- ***** (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

Calling Functions by Reference

19

- 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

20

- **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

```
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 ptr,
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.

Outline

fig05_13.cpp (1 of 1)

fig05_13.cpp output (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 co
13    // ptr always points to the same loc
14    // at that location cannot be modified.
15    const int *const ptr = &x
16
17    cout << *ptr << endl;
18
19    *ptr = 7;  // error: *ptr
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.

value

Outline

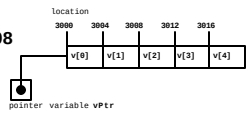
fig05_14.cpp (1 of 1)

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

25

- 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

26

- 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 ]`**

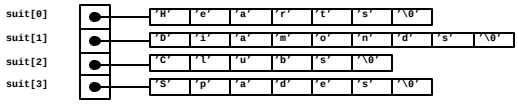
Arrays of Pointers

27

- 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

Function Pointers

28

- 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)

```
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.

Outline

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 agrument to specify descending sorting order
44    else {
45       bubble( a, arraySize, descending );
46       cout << "\nData items in descending order\n";
47    }
48
```


Outline

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 comp
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 c
68       for ( int count = 0; count < s
69
70          // if adjacent elements are
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.

Outline

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
```

```
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
```


Outline

fig05_25.cpp
(5 of 5)

fig05_25.cpp
output (1 of 1)

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


```
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 fu
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.

Outline

fig05_26.cpp
(1 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.

Outline

fig05_26.cpp
(2 of 3)

```
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.
```

Outline

fig05_26.cpp
(3 of 3)

fig05_26.cpp
output (1 of 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

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'**)

Fundamentals of Characters and Strings

41

- **`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' );
```


String Manipulation Functions of the String-handling Library

42

- 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

43

`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

44

`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.