Lecture 26:
Recursion, Arrays, and Vectors

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 11th, 2010

6.20 Example Using Recursion: Fibonacci Series

- The Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- The series occurs in nature and, in particular, describes a form of spiral.
- The ratio of successive Fibonacci numbers converges on a constant value of 1.618….
- This number, too, frequently occurs in nature and has been called the golden ratio or the golden mean.

6.20 Example Using Recursion: Fibonacci Series (cont.)

- The Fibonacci series can be defined recursively as follows:
  - `fibonacci(0) = 0`
    `fibonacci(1) = 1`
    `fibonacci(`*`n`*`) = `*`fibonacci(n - 1) + fibonacci(n - 2)`*
- The program of Fig. 6.30 calculates the *nth Fibonacci number recursively by using function* `fibonacci`.

```
1  // Fig. 6.30: fig06_30.cpp
2  // Testing the recursive fibonacci function.
3  #include <iostream>
4  using namespace std;
5
6  unsigned long fibonacci( unsigned long ); // function prototype
7
8  int main()
9  {
10    // calculate the fibonacci values of 0 through 10
11    for ( int counter = 0; counter <= 10; counter++ )
12       cout << "fibonacci( " << counter << " ) = "
13          << fibonacci( counter ) << endl;
14
15    // display higher fibonacci values
16    cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
17    cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
18    cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
19 } // end main
20
```

Fig. 6.30 | Demonstrating function `fibonacci`. (Part 1 of 2.)

```
21  // recursive function fibonacci
22  unsigned long fibonacci( unsigned long number )
23  {
24     if ( ( number == 0 ) || ( number == 1 ) ) // base cases
25        return number;
26     else // recursion step
27        return fibonacci( number - 1 ) + fibonacci( number - 2 );
28  } // end function fibonacci
```

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465
```

Fig. 6.30 | Demonstrating function `fibonacci`. (Part 2 of 2.)

6.20 Example Using Recursion: Fibonacci Series (cont.)

- Figure 6.31 shows how function `fibonacci` would evaluate `fibonacci(3)`.
- This figure raises some interesting issues about the order in which C++ compilers evaluate the operands of operators.
- This is a separate issue from the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and associativity.
- Most programmers simply assume that operands are evaluated left to right.
- C++ does not specify the order in which the operands of most operators (including +) are to be evaluated.
- Therefore, you must make no as-sumption about the order in which these calls execute.

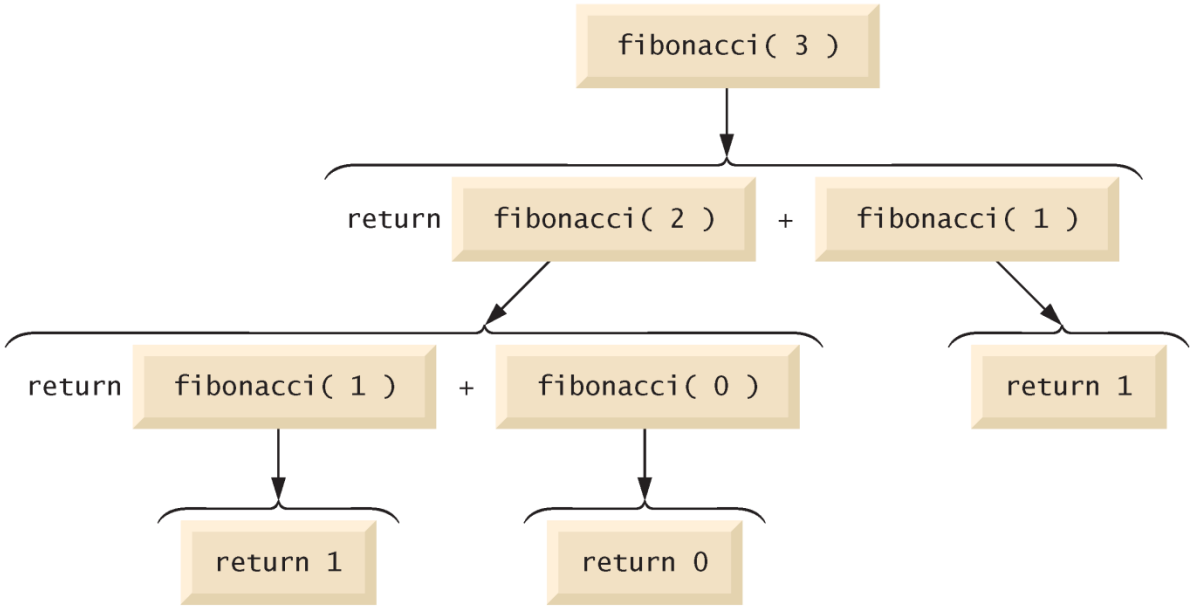

Fig. 6.31 | Set of recursive calls to function `fibonacci`.

Performance Tip 6.8

Avoid Fibonacci-style recursive programs that result in an exponential "explosion" of calls.

6.21 Recursion vs. Iteration

- Both iteration and recursion are based on a control statement: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.
- Iteration and recursion both involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.

6.21 Recursion vs. Iteration (cont.)

- Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion produces simpler versions of the original problem until the base case is reached.
- Both it-eration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.

```
1  // Fig. 6.32: fig06_32.cpp
2  // Testing the iterative factorial function.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; counter++ )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
```

Fig. 6.32 | Iterative factorial solution. (Part 1 of 2.)

```
17  // iterative function factorial
18  unsigned long factorial( unsigned long number )
19  {
20     unsigned long result = 1;
21
22     // iterative factorial calculation
23     for ( unsigned long i = number; i >= 1; i-- )
24        result *= i;
25
26     return result;
27  } // end function factorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 6.32 | Iterative factorial solution. (Part 2 of 2.)

Software Engineering Observation 6.15

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

Performance Tip 6.9

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

7.7 Searching Arrays with Linear Search

- Often it may be necessary to determine whether an array contains a value that matches a certain key value.
  - Called searching.
- The linear search compares each element of an array with a search key (line 36).
  - Because the array is not in any particular order, it's just as likely that the value will be found in the first element as the last.
  - On average, therefore, the program must compare the search key with half the elements of the array.
- To determine that a value is not in the array, the program must compare the search key to every element of the array.

```
1  // Fig. 7.18: fig07_18.cpp
2  // Linear search of an array.
3  #include <iostream>
4  using namespace std;
5
6  int linearSearch( const int [], int, int ); // prototype
7
8  int main()
9  {
10    const int arraySize = 100; // size of array a
11    int a[ arraySize ]; // create array a
12    int searchKey; // value to locate in array a
13
14    for ( int i = 0; i < arraySize; i++ )
15       a[ i ] = 2 * i; // create some data
16
17    cout << "Enter integer search key: ";
18    cin >> searchKey;
19
20    // attempt to locate searchKey in array a
21    int element = linearSearch( a, searchKey, arraySize );
22
```

Fig. 7.18 | Linear search of an array. (Part 1 of 3.)

```
23      // display results
24      if ( element != -1 )
25         cout << "Found value in element " << element << endl;
26      else
27         cout << "Value not found" << endl;
28   } // end main
29
30   // compare key to every element of array until location is
31   // found or until end of array is reached; return subscript of
32   // element if key is found or -1 if key not found
33   int linearSearch( const int array[], int key, int sizeOfArray )
34   {
35      for ( int j = 0; j < sizeOfArray; j++ )
36         if ( array[ j ] == key ) // if found,
37            return j; // return location of key
38
39      return -1; // key not found
40   } // end function linearSearch
```

```
Enter integer search key: 36
Found value in element 18
```

Fig. 7.18 | Linear search of an array. (Part 2 of 3.)

```
Enter integer search key: 37
Value not found
```

Fig. 7.18 | Linear search of an array. (Part 3 of 3.)

7.8 Sorting Arrays with Insertion Sort

- Sorting data
  - placing the data into some particular order such as ascending or descending
  - an intriguing prob-lem that has attracted some of the most intense research efforts in the field of computer science.

Performance Tip 7.3

Simple algorithms can perform poorly. Their virtue is that they're easy to write, test and debug. More complex algorithms are sometimes needed to realize optimal performance.

7.8 Sorting Arrays with Insertion Sort (cont.)

- Insertion sort—a simple, but inefficient, sorting algorithm.
- The first iteration of this algorithm takes the second element and, if it's less than the first element, swaps it with the first element (i.e., the program *inserts the second element in front of the first element).*
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
- At the i^{th} *iteration of this algorithm, the first i elements in the original array will be sorted.*

```
1  // Fig. 7.19: fig07_19.cpp
2  // This program sorts an array's values into ascending order.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int arraySize = 10; // size of array a
10    int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
11    int insert; // temporary variable to hold element to insert
12
13    cout << "Unsorted array:\n";
14
15    // output original array
16    for ( int i = 0; i < arraySize; i++ )
17       cout << setw( 4 ) << data[ i ];
18
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 1 of 3.)

```
19       // insertion sort
20       // loop over the elements of the array
21       for ( int next = 1; next < arraySize; next++ )
22       {
23          insert = data[ next ]; // store the value in the current element
24
25          int moveItem = next; // initialize location to place element
26
27          // search for the location in which to put the current element
28          while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
29          {
30             // shift element one slot to the right
31             data[ moveItem ] = data[ moveItem - 1 ];
32             moveItem--;
33          } // end while
34
35          data[ moveItem ] = insert; // place inserted element into the array
36       } // end for
37
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 2 of 3.)

```
38      cout << "\nSorted array:\n";
39
40      // output sorted array
41      for ( int i = 0; i < arraySize; i++ )
42         cout << setw( 4 ) << data[ i ];
43
44      cout << endl;
45   } // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 3 of 3.)

7.9 Multidimensional Arrays

- Arrays with two dimensions (i.e., subscripts) often represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, we must specify two subscripts.
  - By convention, the first identifies the element's row and the second identifies the element's column.
- Often called two-dimensional arrays or 2-D arrays.
- Arrays with two or more dimensions are known as multidimensional arrays.
- Figure 7.20 illustrates a two-dimensional array, a.
  - The array contains three rows and four columns, so it's said to be a 3-by-4 array.
  - In general, an array with *m rows and n columns is called an m-by-n array*.

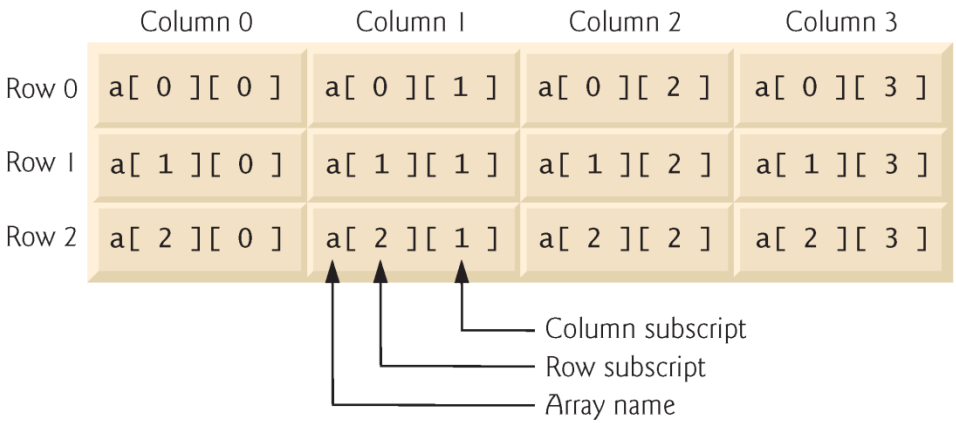

Fig. 7.20 | Two-dimensional array with three rows and four columns.

7.9 Multidimensional Arrays (cont.)

- A multidimensional array can be initialized in its declaration much like a one-dimensional array.
- The values are grouped by row in braces.
- If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.
- Figure 7.21 demonstrates initializing two-dimensional arrays in declarations.

```
1  // Fig. 7.21: fig07_21.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4  using namespace std;
5
6  void printArray( const int [][ 3 ] ); // prototype
7  const int rows = 2;
8  const int columns = 3;
9
10 int main()
11 {
12    int array1[ rows ][ columns ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13    int array2[ rows ][ columns ] = { 1, 2, 3, 4, 5 };
14    int array3[ rows ][ columns ] = { { 1, 2 }, { 4 } };
15
16    cout << "Values in array1 by row are:" << endl;
17    printArray( array1 );
18
19    cout << "\nValues in array2 by row are:" << endl;
20    printArray( array2 );
21
22    cout << "\nValues in array3 by row are:" << endl;
23    printArray( array3 );
24 } // end main
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 1 of 3.)

```
25
26 // output array with two rows and three columns
27 void printArray( const int a[][ columns ] )
28 {
29    // loop through array's rows
30    for ( int i = 0; i < rows; i++ )
31    {
32       // loop through columns of current row
33       for ( int j = 0; j < columns; j++ )
34          cout << a[ i ][ j ] << ' ';
35
36       cout << endl; // start new line of output
37    } // end outer for
38 } // end function printArray
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 2 of 3.)

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0

Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 3 of 3.)

7.11 Introduction to C++ Standard Library Class Template vector

- C++ Standard Library class template `vector` represents a more robust type of array featuring many additional capabilities.
- C-style pointer-based arrays have great potential for errors and are not flexible
  - A program can easily "walk off" either end of an array, because C++ does not check whether subscripts fall outside the range of an array.
  - Two ar-rays cannot be meaningfully compared with equality operators or relational operators.
  - When an array is passed to a general-purpose function designed to handle arrays of any size, the size of the array must be passed as an additional argument.
  - One array cannot be assigned to another with the assignment operator(s).
- Class template `vector` allows you to create a more powerful and less error-prone alternative to arrays.

7.11 Introduction to C++ Standard Library Class Template `vector` (cont.)

- The program of Fig. 7.25 demonstrates capabilities provided by class template `vector` that are not available for C-style pointer-based arrays.
- Standard class template `vector` is defined in header `<vector>` and belongs to namespace `std`.

```
1  // Fig. 7.25: fig07_25.cpp
2  // Demonstrating C++ Standard Library class template vector.
3  #include <iostream>
4  #include <iomanip>
5  #include <vector>
6  using namespace std;
7
8  void outputVector( const vector< int > & ); // display the vector
9  void inputVector( vector< int > & ); // input values into the vector
10
11 int main()
12 {
13    vector< int > integers1( 7 ); // 7-element vector< int >
14    vector< int > integers2( 10 ); // 10-element vector< int >
15
16    // print integers1 size and contents
17    cout << "Size of vector integers1 is " << integers1.size()
18       << "\nvector after initialization:" << endl;
19    outputVector( integers1 );
20
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 1 of 8.)

```
21      // print integers2 size and contents
22      cout << "\nSize of vector integers2 is " << integers2.size()
23         << "\nvector after initialization:" << endl;
24      outputVector( integers2 );
25
26      // input and print integers1 and integers2
27      cout << "\nEnter 17 integers:" << endl;
28      inputVector( integers1 );
29      inputVector( integers2 );
30
31      cout << "\nAfter input, the vectors contain:\n"
32         << "integers1:" << endl;
33      outputVector( integers1 );
34      cout << "integers2:" << endl;
35      outputVector( integers2 );
36
37      // use inequality (!=) operator with vector objects
38      cout << "\nEvaluating: integers1 != integers2" << endl;
39
40      if ( integers1 != integers2 )
41         cout << "integers1 and integers2 are not equal" << endl;
42
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 2 of 8.)

```
43      // create vector integers3 using integers1 as an
44      // initializer; print size and contents
45      vector< int > integers3( integers1 ); // copy constructor
46
47      cout << "\nSize of vector integers3 is " << integers3.size()
48         << "\nvector after initialization:" << endl;
49      outputVector( integers3 );
50
51      // use overloaded assignment (=) operator
52      cout << "\nAssigning integers2 to integers1:" << endl;
53      integers1 = integers2; // assign integers2 to integers1
54
55      cout << "integers1:" << endl;
56      outputVector( integers1 );
57      cout << "integers2:" << endl;
58      outputVector( integers2 );
59
60      // use equality (==) operator with vector objects
61      cout << "\nEvaluating: integers1 == integers2" << endl;
62
63      if ( integers1 == integers2 )
64         cout << "integers1 and integers2 are equal" << endl;
65
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 3 of 8.)

```
66      // use square brackets to create rvalue
67      cout << "\nintegers1[5] is " << integers1[ 5 ];
68
69      // use square brackets to create lvalue
70      cout << "\n\nAssigning 1000 to integers1[5]" << endl;
71      integers1[ 5 ] = 1000;
72      cout << "integers1:" << endl;
73      outputVector( integers1 );
74
75      // attempt to use out-of-range subscript
76      cout << "\nAttempt to assign 1000 to integers1.at( 15 )" << endl;
77      integers1.at( 15 ) = 1000; // ERROR: out of range
78   } // end main
79
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 4 of 8.)

```
80  // output vector contents
81  void outputVector( const vector< int > &array )
82  {
83     size_t i; // declare control variable
84
85     for ( i = 0; i < array.size(); i++ )
86     {
87        cout << setw( 12 ) << array[ i ];
88
89        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
90           cout << endl;
91     } // end for
92
93     if ( i % 4 != 0 )
94        cout << endl;
95  } // end function outputVector
96
97  // input vector contents
98  void inputVector( vector< int > &array )
99  {
100    for ( size_t i = 0; i < array.size(); i++ )
101       cin >> array[ i ];
102 } // end function inputVector
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 5 of 8.)

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 6 of 8.)

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 7 of 8.)

```
Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1.at( 15 )

abnormal program termination
```

Fig. 7.25 | C++ Standard Library class template `vector`. (Part 8 of 8.)

7.11 Introduction to C++ Standard Library Class Template `vector` (cont.)

- By default, all the elements of a `vector` object are set to 0.
- `vector`s can be defined to store any data type.
- `vector` member function `size` obtain the number of elements in the `vector`.
- You can use square brackets, `[]`, to access the elements in a `vector`.
- `vector` objects can be compared with one another using the equality operators.
- You can create a new `vector` object that is initialized with the contents of an existing `vector` by using its copy constructor.

7.11 Introduction to C++ Standard Library Class Template `vector` (cont.)

- You can use the assignment (=) operator with `vector` objects.
- As with C-style pointer-based arrays, C++ does not perform any bounds checking when `vector` elements are accessed with square brackets.
- Standard class template `vector` provides bounds checking in its member function `at`, which "throws an exception" (see Chapter 16, Exception Handling) if its argument is an invalid subscript.

Questions