Lecture 25:
Strings & Recursion

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 10th, 2010

18.6 string Characteristics (cont.)

- The program declares empty `string` `string1` (line 11) and passes it to function `printStatistics` (line 14).
- Function `printStatistics` (lines 42–48) takes a reference to a `const` `string` as an argument and outputs
  - capacity (using member function `capacity`),
  - maximum size (using member function `max_size`), size (using member function `size`),
  - length (using member function `length`) and
  - whether the `string` is empty (using member function `empty`).
- A size and length of `0` indicate that there are no characters stored in a `string`.
- When the initial capacity is `0` and characters are placed into the `string`, memory is allocated to accommodate the new characters.

Performance Tip 18.1

To minimize the number of times memory is allocated and deallocated, some `string` *class implementations provide a default capacity that is larger than the length of the* `string`.

18.6 `string` Characteristics (cont.)

- Line 30 uses the overloaded += operator to concatenate a 46-character-long string to `string1`.
- The capacity has increased to `63` elements and the length is now `50`.
- Line 35 uses member function `resize` to increase the length of `string1` by `10` characters.
- The additional elements are set to null characters.
- The output shows that the capacity has not changed and the length is now `60`.

18.7 Finding Substrings and Characters in a `string`

- Class `string` provides `const` member functions for finding substrings and characters in a `string`.
- Figure 18.6 demonstrates the find functions.

```
1  // Fig. 18.6: Fig18_06.cpp
2  // Demonstrating the string find member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "noon is 12 pm; midnight is not." );
10    int location;
11
12    // find "is" at location 5 and 24
13    cout << "Original string:\n" << string1
14       << "\n\n(find) \"is\" was found at: " << string1.find( "is" )
15       << "\n(rfind) \"is\" was found at: " << string1.rfind( "is" );
16
17    // find 'o' at location 1
18    location = string1.find_first_of( "misop" );
19    cout << "\n\n(find_first_of) found '" << string1[ location ]
20       << "' from the group \"misop\" at: " << location;
21
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 1 of 3.)

```
22      // find 'o' at location 29
23      location = string1.find_last_of( "misop" );
24      cout << "\n\n(find_last_of) found '" << string1[ location ]
25         << "' from the group \"misop\" at: " << location;
26
27      // find '1' at location 8
28      location = string1.find_first_not_of( "noi spm" );
29      cout << "\n\n(find_first_not_of) '" << string1[ location ]
30         << "' is not contained in \"noi spm\" and was found at: "
31         << location;
32
33      // find '.' at location 12
34      location = string1.find_first_not_of( "12noi spm" );
35      cout << "\n\n(find_first_not_of) '" << string1[ location ]
36         << "' is not contained in \"12noi spm\" and was "
37         << "found at: " << location << endl;
38
39      // search for characters not in string1
40      location = string1.find_first_not_of(
41         "noon is 12 pm; midnight is not." );
42      cout << "\nfind_first_not_of(\"noon is 12 pm; midnight is not.\")"
43         << " returned: " << location << endl;
44   } // end main
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 2 of 3.)

```
Original string:
noon is 12 pm; midnight is not.

(find) "is" was found at: 5
(rfind) "is" was found at: 24

(find_first_of) found 'o' from the group "misop" at: 1

(find_last_of) found 'o' from the group "misop" at: 29

(find_first_not_of) '1' is not contained in "noi spm" and was found at: 8

(find_first_not_of) '.' is not contained in "12noi spm" and was found at: 12

find_first_not_of("noon is 12 pm; midnight is not.") returned: -1
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 3 of 3.)

18.7 Finding Substrings and Characters in a `string` (cont.)

- Line 14 attempts to find `"is"` in `string1` using function `find`.
  - If `"is"` is found, the subscript of the starting location of that string is returned.
  - If the `string` is not found, the value `string::npos` (a `public static` constant defined in class `string`) is returned.
  - This value is returned by the `string` `find`-related functions to indicate that a substring or character was not found in the `string`.

18.7 Finding Substrings and Characters in a `string` (cont.)

- Line 15 uses member function `rfind` to search `string1` backward (i.e., right-to-left).
- If `"is"` is found, the subscript location is returned.
- If the string is not found, `string::npos` is returned.
- [*Note: The rest of the find functions presented in this section return the same type unless otherwise noted.]*

18.7 Finding Substrings and Characters in a `string` (cont.)

- Line 18 uses member function `find_first_of` to locate the first occurrence in `string1` of any character in `"misop"`.
  - The searching is done from the beginning of `string1`.
- Line 23 uses member function `find_last_of` to find the last occurrence in `string1` of any character in `"misop"`.
  - The searching is done from the end of `string1`.
- Line 28 uses member function `find_first_not_of` to find the first character in `string1` not contained in `"noi spm"`.
  - Searching is done from the beginning of `string1`.
- Line 34 uses member function `find_first_not_of` to find the first character not contained in `"12noi spm"`.
  - Searching is done from the end of `string1`.

18.7 Finding Substrings and Characters in a `string` (cont.)

- Lines 40–41 use member function `find_first_not_of` to find the first character not contained in `"noon is 12 pm; midnight is not."`.
  - In this case, the `string` being searched contains every character specified in the string argument.
  - Because a character was not found, `string::npos` (which has the value `-1` in this case) is returned.

18.8 Replacing Characters in a `string`

- Figure 18.7 demonstrates `string` member functions for replacing and erasing characters.
- Line 20 uses `string` member function `erase` to erase everything from (and including) the character in position 62 to the end of `string1`.
- [*Note: Each newline character occupies one element in the `string`.*]

```
1  // Fig. 18.7: Fig18_07.cpp
2  // Demonstrating string member functions erase and replace.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     // compiler concatenates all parts into one string
10    string string1( "The values in any left subtree"
11       "\nare less than the value in the"
12       "\nparent node and the values in"
13       "\nany right subtree are greater"
14       "\nthan the value in the parent node" );
15
16    cout << "Original string:\n" << string1 << endl << endl;
17
18    // remove all characters from (and including) location 62
19    // through the end of string1
20    string1.erase( 62 );
21
22    // output new string
23    cout << "Original string after erase:\n" << string1
24       << "\n\nAfter first replacement:\n";
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 1 of 3.)

```
25
26     int position = string1.find( " " ); // find first space
27
28     // replace all spaces with period
29     while ( position != string::npos )
30     {
31        string1.replace( position, 1, "." );
32        position = string1.find( " ", position + 1 );
33     } // end while
34
35     cout << string1 << "\n\nAfter second replacement:\n";
36
37     position = string1.find( "." ); // find first period
38
39     // replace all periods with two semicolons
40     // NOTE: this will overwrite characters
41     while ( position != string::npos )
42     {
43        string1.replace( position, 2, "xxxxx;;yyy", 5, 2 );
44        position = string1.find( ".", position + 1 );
45     } // end while
46
47     cout << string1 << endl;
48  } // end main
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 2 of 3.)

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

Original string after erase:
The values in any left subtree
are less than the value in the


After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the


After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 3 of 3.)

18.8 Replacing Characters in a `string` (cont.)

- Lines 26–33 use `find` to locate each occurrence of the space character.
- Each space is then replaced with a period by a call to `string` member function `replace`.
- Function `replace` takes three arguments:
  - the subscript of the character in the `string` at which replacement should begin,
  - the number of characters to replace and
  - the replacement string.

18.8 Replacing Characters in a `string` (cont.)

- Lines 37–45 use function `find` to find every period and another overloaded function `replace` to replace every period and its following character with two semicolons.
- The arguments passed to this version of `replace` are
  - the subscript of the element where the replace operation begins,
  - the number of characters to replace,
  - a replacement character string from which a substring is selected to use as replacement characters,
  - the element in the character string where the replacement substring begins and
  - the number of characters in the replacement character string to use.

18.9 Inserting Characters into a `string`

- Class `string` provides member functions for inserting characters into a `string`.
- Figure 18.8 demonstrates the `string` `insert` capabilities.
- Line 19 uses `string` member function `insert` to insert `string2`'s content before element 10 of `string1`.
- Line 22 uses `insert` to insert `string4` before `string3`'s element 3.
  - The last two arguments specify the starting and last element of `string4` that should be inserted.
  - Using `string::npos` causes the entire `string` to be inserted.

```
1  // Fig. 18.8: Fig18_08.cpp
2  // Demonstrating class string insert member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "beginning end" );
10    string string2( "middle " );
11    string string3( "12345678" );
12    string string4( "xx" );
13
14    cout << "Initial strings:\nstring1: " << string1
15       << "\nstring2: " << string2 << "\nstring3: " << string3
16       << "\nstring4: " << string4 << "\n\n";
17
18    // insert "middle" at location 10 in string1
19    string1.insert( 10, string2 );
20
21    // insert "xx" at location 3 in string3
22    string3.insert( 3, string4, 0, string::npos );
23
```

Fig. 18.8 | Demonstrating the `string` `insert` member functions. (Part 1 of 2.)

```
24      cout << "Strings after insert:\nstring1: " << string1
25         << "\nstring2: " << string2 << "\nstring3: " << string3
26         << "\nstring4: " << string4 << endl;
27   } // end main
```

```
Initial strings:
string1: beginning end
string2: middle
string3: 12345678
string4: xx

Strings after insert:
string1: beginning middle end
string2: middle
string3: 123xx45678
string4: xx
```

Fig. 18.8 | Demonstrating the `string` `insert` member functions. (Part 2 of 2.)

18.10 Conversion to C-Style Pointer-Based char * Strings

- Class `string` provides member functions for converting `string` class objects to C-style pointer-based strings.
- As mentioned earlier, unlike pointer-based strings, `string`s are not necessarily null terminated.
- These conversion functions are useful when a given function takes a pointer-based string as an argument.
- Figure 18.9 demonstrates conversion of `string`s to pointer-based strings.
- The `string` `string1` is initialized to `"STRINGS"`, `ptr1` is initialized to `0` and `length` is initialized to the length of `string1`.
- Memory of sufficient size to hold a pointer-based string equivalent of `string` `string1` is allocated dynamically and attached to `char` pointer `ptr2`.

```
1  // Fig. 18.9: Fig18_09.cpp
2  // Converting to C-style strings.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "STRINGS" ); // string constructor with char* arg
10    const char *ptr1 = 0; // initialize *ptr1
11    int length = string1.length();
12    char *ptr2 = new char[ length + 1 ]; // including null
13
14    // copy characters from string1 into allocated memory
15    string1.copy( ptr2, length, 0 ); // copy string1 to ptr2 char*
16    ptr2[ length ] = '\0'; // add null terminator
17
18    cout << "string string1 is " << string1
19       << "\nstring1 converted to a C-Style string is "
20       << string1.c_str()   << "\nptr1 is ";
21
```

Fig. 18.9 | Converting `string`s to C-style strings and character arrays. (Part 1 of 2.)

```
22      // Assign to pointer ptr1 the const char * returned by
23      // function data(). NOTE: this is a potentially dangerous
24      // assignment. If string1 is modified, pointer ptr1 can
25      // become invalid.
26      ptr1 = string1.data();
27
28      // output each character using pointer
29      for ( int i = 0; i < length; i++ )
30         cout << *( ptr1 + i ); // use pointer arithmetic
31
32      cout << "\nptr2 is " << ptr2 << endl;
33      delete [] ptr2; // reclaim dynamically allocated memory
34   } // end main
```

```
string string1 is STRINGS
string1 converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

Fig. 18.9 | Converting `string`s to C-style strings and character arrays. (Part 2 of 2.)

18.11 Conversion to C-Style Pointer-Based char * Strings (cont.)

- Line 15 uses `string` member function `copy` to copy object `string1` into the `char` array pointed to by `ptr2`.
- Line 16 manually places a terminating null character in the array pointed to by `ptr2`.
- Line 20 uses function `c_str` to obtain a `const char *` that points to a null terminated C-style string with the same content as `string1`.
- The pointer is passed to the stream insertion operator for output.

18.12 Conversion to C-Style Pointer-Based char * Strings (cont.)

- Line 26 assigns the `const char * ptr1` a pointer returned by class `string` member function `data`.
- This member function returns a non-null-terminated C-style character array.
- We do not modify `string string1` in this example.
- If `string1` were to be modified (e.g., the `string`'s dynamic memory changes its address due to a member function call such as `string1.insert( 0, "abcd" );`), `ptr1` could become invalid—which could lead to unpredictable results.
- Lines 29–30 use pointer arithmetic to output the character array pointed to by `ptr1`.
- In lines 32–33, the C-style string pointed to by `ptr2` is output and the memory allocated for `ptr2` is `delete`d to avoid a memory leak.

Common Programming Error 18.4

Not terminating the character array returned by `data` *with a null character can lead to execution-time errors.*

Good Programming Practice 18.1

Whenever possible, use the more robust `string` *class objects rather than C-style pointer-based strings.*

18.13 Iterators

- Class `string` provides iterators for forward and backward traversal of `string`s.
- Iterators provide access to individual characters with syntax that is similar to pointer operations.
- Iterators are not range checked.
- In this section we provide “mechanical examples” to demonstrate the use of iterators.
- We discuss more robust uses of iterators in Chapter 22, Standard Template Library (STL).

18.13 Iterators (cont.)

- Figure 18.10 demonstrates iterators.
- Lines 9–10 declare `string string1` and `string::const_iterator iterator1`.
- A `const_iterator` is an iterator that cannot modify the `string`—in this case the `string` through which it's iterating.
- Iterator `iterator1` is initialized to the beginning of `string1` with the `string` class member function `begin`.
- Two versions of `begin` exist—one that returns an `iterator` for iterating through a non-`const string` and a `const` version that returns a `const_iterator` for iterating through a `const string`.

```
1  // Fig. 18.10: Fig18_10.cpp
2  // Using an iterator to output a string.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "Testing iterators" );
10    string::const_iterator iterator1 = string1.begin();
11
12    cout << "string1 = " << string1
13       << "\n(Using iterator iterator1) string1 is: ";
14
15    // iterate through string
16    while ( iterator1 != string1.end() )
17    {
18       cout << *iterator1; // dereference iterator to get char
19       iterator1++; // advance iterator to next char
20    } // end while
21
22    cout << endl;
23 } // end main
```

Fig. 18.10 | Using an iterator to output a `string`. (Part 1 of 2.)

```
string1 = Testing iterators
(Using iterator iterator1) string1 is: Testing iterators
```

Fig. 18.10 | Using an iterator to output a `string`. (Part 2 of 2.)

18.13 Iterators (cont.)

- Lines 16–20 use iterator `iterator1` to "walk through" `string1`.
- Class `string` member function end returns an `iterator` (or a `const_iterator`) for the position past the last element of `string1`.
- Each element is printed by dereferencing the iterator much as you'd dereference a pointer, and the iterator is advanced one position using operator ++.

18.13 Iterators (cont.)

- Class `string` provides member functions `rend` and `rbegin` for accessing individual `string` characters in reverse from the end of a `string` toward the beginning.
- Member functions `rend` and `rbegin` return `reverse_iterators` or `const_reverse_iterators` (based on whether the `string` is non-`const` or `const`).
- We'll use iterators and reverse iterators more in Chapter 22.

Error-Prevention Tip 18.1

Use `string` *member function* `at` *(rather than iterators) when you want the benefit of range checking.*

Good Programming Practice 18.2

When the operations involving the iterator should not modify the data being processed, use a `const_iterator`. *This is another example of employing the principle of least privilege.*

6.19 Recursion

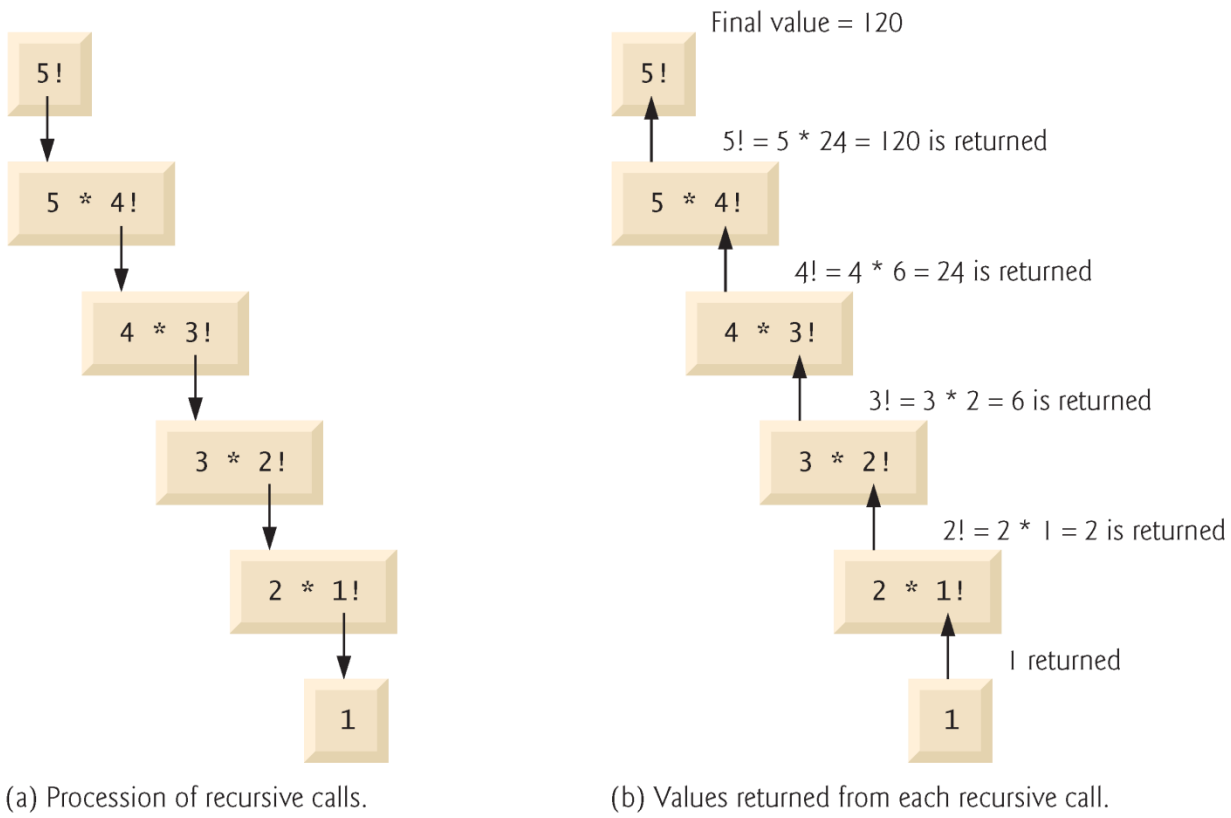
- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- Recursive problem-solving approaches have a number of elements in common.
  - A recursive function is called to solve a problem.
  - The function actu-ally knows how to solve only the simplest case(s), or so-called base case(s).
  - If the func-tion is called with a base case, the function simply returns a result.
  - If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do.
  - This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step.

6.19 Recursion (cont.)

- The recursion step often includes the key-word `return`, because its result will be combined with the portion of the problem the function knew how to solve to form the result passed back to the original caller, possibly `main`.
- The recursion step executes while the original call to the function is still "open," i.e., it has not yet finished executing.
- The recursion step can result in many more such recursive calls.

6.19 Recursion (cont.)

- The factorial of a nonnegative integer *n*, *written n! (and pronounced "n factorial"), is the product*
  - *n · (n - 1) · (n - 2) · … · 1*
- with 1! equal to 1, and 0! defined to be 1.
- The factorial of an integer, `number`, greater than or equal to 0, can be calculated iteratively (nonrecursively) by using a loop.
- A recursive definition of the factorial function is arrived at by observing the follow-ing algebraic relationship:
  - *n! = n · (n - 1)!*



(a) Procession of recursive calls.

(b) Values returned from each recursive call.

Fig. 6.28 | Recursive evaluation of 5!.

```
1  // Fig. 6.29: fig06_29.cpp
2  // Demonstrating the recursive function factorial.
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
17 // recursive definition of function factorial
18 unsigned long factorial( unsigned long number )
19 {
20    if ( number <= 1 ) // test for base case
21       return 1; // base cases: 0! = 1 and 1! = 1
22    else // recursion step
23       return number * factorial( number - 1 );
24 } // end function factorial
```

Fig. 6.29 | Demonstrating the recursive function `factorial`. (Part 1 of 2.)

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

Fig. 6.29 | Demonstrating the recursive function `factorial`. (Part 2 of 2.)

Common Programming Error 6.16

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, causes "infinite" recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

Questions