CMPE 180-92

Data Structures and Algorithms in C++

September 7 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

SILICON VALLEY'S
FIRST CHOICE
FOR NEW
ENGINEERING HIRES
— Silicon Valley Business Journal, 2013

San José State
UNIVERSITY

Assignment #2: Sample Solution

Streams

- I/O (input/output) for a program can be considered a stream of characters.
  - Represented in a program by a stream variable.
- An input stream into your program can be
  - characters typed at the keyboard
  - characters read from a file
- An output stream from your program can be
  - characters displayed on the screen
  - characters written to a file

File I/O

- In order for a program to read from a data file, it must first connect a stream variable to the file.

```
#include <fstream>
using namespace std;
...
ifstream in_stream;   // input  file stream variable
ofstream out_stream;  // output file stream variable
...
in_stream.open("infile.dat");    // connect to the input file
out_stream.open("outfile.dat");  // connect to the output file
...
// Read three integer values from the input file.
int value1, value2, value3;
in_stream >> value1 >> value2 >> value3;

// Write to the output file.
out_stream << "Value #1 is " << value1
           << " and Value #2 is " << value2 << endl;
```

File I/O, *cont'd*

- Close a stream when you're done with reading or writing it.

```
in_stream.close();
out_stream.close();
```

- Closing a stream releases the associated file for use by another program.

Stream Name vs. File Name

- Do not confuse the name of a program's stream variable with the name of the file.
  - The stream variable's name internal to the program.
  - The file's name is external to the program.
- Calling a stream's `open` method connects the stream to the file.
- A stream is an object.
  - `open` and `close` are functions we can call on the object.

We'll learn about C++ classes and objects later.

Formatting Output

- Formatting a value that is being output includes
  - determining the width of the output field
  - deciding whether to write numbers in fixed-point notation or in scientific notation
  - setting how many digits after the decimal point

- To format output to `cout`, call its member functions:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

  - Use fixed-point notation instead of scientific notation.
  - Always include the decimal point in the output.
  - Only two significant digits are required in the output.

Output Manipulators

- Manipulator function `setw` sets the width of an output field.

- Manipulator function `setprecision` sets the number of places after the decimal point.

- Embed calls to manipulators in output statements.
  - Examples:

```
#include <iomanip>
using namespace std;
...
cout << "Value 1 = " << setw(10) << value1 << end;
cout << "$" << setprecision(2) << amount << endl;
```

Passing Streams to Functions

- Pass stream objects to functions only via call-by-reference.
  - Example:

```
void copyFile(ifstream& source,
              ofstream& destination);
```

Character I/O

- Recall that the operator `>>` used on `cin` skips blanks.

- To read all characters from an input stream, including blanks, use the `get` method:

```
char ch;
...
cin.get(ch);
```

- Use the `put` method to output any character to an output stream.

Predefined Character Functions

- Some very useful Boolean functions that test a character:
  - `isupper(ch)`
  - `islower(ch)`
  - `isalpha(ch)`
  - `isdigit(ch)`
  - `isspace(ch)`
  - `toupper(ch)`
  - `tolower(ch)`

The **eof** Function

- Boolean function **eof** tests whether or not an input stream has read the entire file.
  - eof = end of file
  - Example:

    ```
    if (in_stream.eof()) ...
    ```

- Function **eof** returns true only after an attempt was made to read past the end of file.

Quiz

Break

Arrays

- An array variable can have multiple values.
- All values must be the same data type.

- Declare an array variable by indicating how many elements.
  - Example: `int a[6];`

- Use subscripts to access array elements.
- Subscript values for an array can range from 0 ... n-1 where n is equal to the number of elements in the array.

Initialize an Array

- You can initialize an array when you declare it:

```
int ages[] = {12, 9, 7, 2};
```

  - If you initialize an array this way, you can leave off the array size.

- You can initialize the array with assignments:

```
int ages[4];
ages[0] = 12;
ages[1] = 9;
ages[2] = 7;
ages[3] = 2;
```

- Or with a loop:

```
int ages[4];
for (int i = 0; i < 4; i++) ages[i] = 0;
```

Array Function Parameters

- To pass an entire array to a function, indicate that a parameter is an array with `[]`.
  - Example:

```
void sort(double a[], int size);
```

- Also pass the array size.
- Arrays are implicitly passed by reference.
- Make the array parameter `const` to indicate that the function does not change the array.
  - Example:

```
double average(const double a[], int size);
```

Assignment #3.a. Prime Numbers

- Use the Sieve of Eratosthenes to generate an array of prime numbers under 100:

```
Primes:

  .  2  3  .  5  .  7  .  .  .
 11  . 13  .  .  . 17  . 19  .
  .  . 23  .  .  .  .  . 29  .
 31  .  .  .  .  . 37  .  .  .
 41  . 43  .  .  . 47  .  .  .
  .  . 53  .  .  .  .  . 59  .
 61  .  .  .  .  . 67  .  .  .
 71  . 73  .  .  .  .  . 79  .
  .  . 83  .  .  .  .  . 89  .
  .  .  .  .  .  . 97  .  .  .
```

  - See: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Multidimensional Arrays

- A multidimensional array is an array of arrays.
  - Example: A two-dimensional array:

```
char page[30][100];
```

  - Each element of `page` is itself an array of 100 characters.

- Use multiple subscripts to access an element of a multidimensional array.
  - Example: `page[i][j]` to access the `j`th character of the `i`th row.
  - What is `page[k]`?

Assignment #3.b. Spirals

- Print a sequence of integers in a counter-clockwise spiral that is enclosed in a square matrix *n*-by-*n*.
  - The 2-dimensional array has *n* rows and *n* columns.

- Start with a given value in the center of the matrix.
  - The starting value is not necessarily 1.

- Arrange subsequent values in a counter-clockwise spiral that grows outward until it fills the matrix.

Assignment #3.b. Spirals, *cont'd*

- Example spirals
  - Size 5, starting value 1:

```
17  16  15  14  13
18   5   4   3  12
19   6   1   2  11
20   7   8   9  10
21  22  23  24  25
```

  - Size 9, starting value 11:

```
75  74  73  72  71  70  69  68  67
76  47  46  45  44  43  42  41  66
77  48  27  26  25  24  23  40  65
78  49  28  15  14  13  22  39  64
79  50  29  16  11  12  21  38  63
80  51  30  17  18  19  20  37  62
81  52  31  32  33  34  35  36  61
82  53  54  55  56  57  58  59  60
83  84  85  86  87  88  89  90  91
```

C Strings

- Traditional C programs used arrays of characters to represent strings:

```
char greeting[] = "Hello, world!";
```

- A C string is always terminated by the null character `\0`.

- Therefore, the array size was one greater than the number of characters in the string.
  - The `greeting` character array above has size 14.

C Strings, *cont'd*

- You cannot assign a string value to a C string array variable:
  - Illegal:

```
greeting = "Good-bye!";
```

- Instead, you use the **`strcpy`** ("string copy") function:

```
strcpy(greeting, "Good-bye!");
```

- **Warning:** Do not copy past the end of the destination string!

C Strings, *cont'd*

- To compare two C strings, use the `strcmp` ("string compare") function:

```
strcmp(str1, str2);
```

- It returns:
  - a negative value if `str1` comes alphabetically before `str2`
  - zero if they contain the same characters
  - a positive value if `str1` comes alphabetically after `str2`.

The Standard **string** Class

- C++ programs use the standard **string** class:

```
#include <string>
using namespace std;
```

- You can initialize **string** variables when you declare them:

```
string noun, s1, s2, s3;
string verb("go");
```

- You can assign to **string** variables:

```
noun = "computer";
```

The Standard `string` Class, *cont'd*

- String concatenation:

```
s1 = s2 + " and " + s3;
```

- String comparisons with `== != < <= > >=`
  - Lexicographic comparisons as expected.

- Strings automatically grow and shrink in size.
  - A string keeps track of its own size.

- Use the member function `at` to safely access a character of a string: `s1.at(i)`
  - `s1[i]` is dangerous if you go beyond the length.

The Standard **`string`** Class, *cont'd*

- Many useful member functions :
  - `str.length()`
  - `str.at(i)`
  - `str.substr(position, length)`
  - `str.insert(pos, str2)`
  - `str.erase(pos, length)`
  - `str.find(str1)`
  - `str.find(str1, pos)`
  - `str.find_first_of(str1, pos)`
  - `str.find_first_not_of(str1, pos)`

Vectors

- A vector is a kind of array whose length can dynamically grow and shrink.
  - Vectors are part of the C++ Standard Template Library (STL).

An array on steroids!

- Like an array, a vector has a base type, and all its elements are of that type.

- Different declaration syntaxes from arrays:

```
vector<double> salaries;
vector<bool> truthTable(10);
vector<int> ages = {12, 9, 7, 2};
```

Vectors, *cont'd*

- Index into a vector like an array: `ages[2]`

- Use with a standard for loop:

```
for (int i = 0; i < ages.size(); i++)
{
    cout << ages[i] << endl;
}
```

- Or with a ranged for loop:

```
for (int age : ages)
{
    cout << age << endl;
}
```

Vectors, *cont'd*

- Append new values to the end of a vector:

```
salaries.push_back(100000.0);
salaries.push_back(75000.0);
salaries.push_back(150000.0);
salaries.push_back(200000.0);
```

- Vector assignment: **`v1 = v2`**;
  - Element-by-element assignment of values.
  - The size of **`v1`** can change to match the size of **`v2`**.

Vectors, *cont'd*

- Size of a vector: The current number of elements that the vector contains: `v.size()`

- Capacity of a vector: The number of elements for which memory is currently allocated: `v.capacity()`
  - Change the size: `v.resize(24)`
  - Explicitly set the capacity: `v.reserve(32)`
  - Bump up the capacity by 10: `v.reserve(v.size() + 10)`

Assignment #3.c. Prime Spirals

- Repeat Assignment #3.b, except use vectors instead of arrays.
- Instead of printing the numbers in the spiral, print dots and hashes instead.
  - Print a hash (**#**) if the position corresponds to a prime number.
  - Print a dot (**.**) if the position corresponds to a composite number.
- Curious patterns may emerge in the matrix!

Assignment #3.c. Prime Spirals, *cont'd*

- Example
  - Size 25, starting at 11:

```
#.........#.....#.#.....#
.....#...#.......#.......
....#.......#...#........
.......#.............#...
........#.....#.....#.#..
.#.#.........#.#...#.....
#...#.....#.#...........#
.........#.#...#.#.#.#.#.
....#.#...#.#...#........
.#...#.#.#...#.#.....#...
..#...........#...#...#..
.............#...#.....#.
#.#.....#.#.#.....#......
...........#.#.#.....#.#.
#...#.#...#.....#...#.#..
.#.#.....#.....#.........
........#.....#.....#...#
...#.........#...#.#...#.
....#.#.....#.....#......
.......#...#.#...#.......
#...#.........#.......#..
...#.....#.......#.......
..#.#.....#...#.....#...#
.....#.#.................
..#...#.#...........#....
```

Assignment #3.c. Prime Spirals, *cont'd*

- Are there patterns in the prime numbers?