C++ Lab 04 - File I/O, Arrays and Vectors

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Ten Short CPP Labs

IAP 2024

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1 Lab Four Overview and Objectives

This lab covers two topics that tie together to produce a nice utility function we will use in later labs. We cover C-style *arrays* and their more modern counterpart *vectors* commonly used C++. Along the way we introduce file input/output (I/O) to give us the power of reading to and from files. In the end we build a simple utility for reading the contents of a file into a vector of strings.

- C-Style Arrays
- C-Style File Input and Formatted Output
- C++ Vectors

2 C-Style Arrays

The C language provides a robust mechanism for representing and using *arrays*, which are contiguous blocks of memory holding elements of the same data type. This is a fundamental concept underpinning C and C++ and commonly found in most if not all programming languages. C++ now provides alternative data structures such as the `vector` which is simpler to use and less prone to programmer error. So an array in typical modern C++ code is more likely to be implemented using a `vector` and for this reason we refer to simple memory arrays as "C-style" arrays.

Since C-style arrays are still commonly found in legacy and even newer C++ code, we take a look at these before introducing and advocating the use of the `vector` later in the lab. We will do a short exercise with C-style arrays, but first, read the page(s) below on C-style arrays.

- http://www.cplusplus.com/doc/tutorial/arrays
- http://www.tutorialspoint.com/cplusplus/cpp_arrays.htm

2.1 Exercise 1: Build a Simple Array from Command Line Arguments

Write a program to read in any number of integers from the command line, constructing an array of such integers, and printing them out afterwards. Your array should be exactly the size of the number of integers read in from the command line. You can assume all arguments on the command line are valid integers.

Call your file `arrays.cpp` and build it to the executable `arrays`. When your program runs, it should be invocable from the command line with:

```
$ ./arrays 1 2 3 88 43 26
Total amount of integers provided: 6
numbers[0]: 1
numbers[1]: 2
numbers[2]: 3
numbers[3]: 88
numbers[4]: 43
numbers[5]: 26
```

The solution to this exercise is in Section 5.1.

2.2 Exercise 2: Analyze and Modify a Simple Array

Continuing with the first exercise, write a program to read in any number of integers from the command line, constructing an array of such integers, and printing them out afterwards. Your array should be exactly the size of the number of integers read in from the command line. Following this scan the array and determine the smallest number and build a new array of size N-1 removing at most one element of the first array equal to the smallest element. You can assume all arguments on the command line are valid integers.

Call your file `array_nosmall.cpp` and build it to the executable `array_nosmall`. When your program runs, it should be invocable from the command line with:

```
$ ./arrays_nosmall 1 88 43 1 7 26
Total amount of integers provided: 6
  numbers[0]: 1
  numbers[1]: 88
  numbers[2]: 43
  numbers[3]: 1
  numbers[4]: 7
  numbers[5]: 26
The new array of integers:
  numbers[0]: 88
  numbers[1]: 43
  numbers[2]: 1
  numbers[3]: 7
  numbers[4]: 26
```

The solution to this exercise is in Section 5.2.

3 C-Style File Input and Formatted Output

The next topic is file I/O, reading from and writing to a file on your computer from within a C or C++ program. Pretty much everything you need for reading and writing can be found in the below five functions, which squarely fall in the realm of C but supported of course in C++. There are several other options and functions to do similar things, but the below is pretty complete. Take a look at them and follow the examples. Pay special attention to the options for the `fopen()` function. Following this are a couple simple exercises.

- `fopen()`
  http://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm
- `fclose()`
  http://www.tutorialspoint.com/c_standard_library/c_function_fclose.htm
- `fprintf()`
  http://www.tutorialspoint.com/c_standard_library/c_function_fprintf.htm
- `feof()`
  http://www.tutorialspoint.com/c_standard_library/c_function_feof.htm
- `fgetc()`
  http://www.tutorialspoint.com/c_standard_library/c_function_fgetc.htm

3.1 Exercise 3: Create and Write to a File with Command Line Args

Write a program that accepts a number of strings on the command line and creates a new file, also specified as a command line argument. The new file will contain one line for each provided command line argument (except for the argument specifying the file name). For this exercise, you will need to know about the `fopen()`, `fclose()`, and `fprintf` functions found on the above web pages, plus some of your knowledge obtained in prior lab exercises for handling command line arguments.

Call your file `fileout.cpp` and build it to the executable `fileout`. When your program runs, it should be invocable from the command line with:

```
$ ./fileout --filename=test.txt one two three four
$ cat test.txt
one
two
three
four
```

The solution to this exercise is in Section 5.3.

3.2 Exercise 4: Reading from a File Specified on the Command Line

Write a program that accepts a given filename on the command line, opens the file for reading, and reads in each line to a string. The string will have the format: `line: [contents]` where `contents` is the raw contents of the line in the file including white space. Your program should then output each string to the terminal. For this exercise, you will need to know about the `fopen()`, `fclose()`, and `feof()` and `fgetc()` functions found on the above web pages, plus some of your knowledge obtained in prior lab exercises for handling command line arguments.

Call your file `filein.cpp` and build it to the executable `filein`. When your program runs, it should be invocable from the command line with:

```
$ ./filein --filename=test.txt
line: [one]
line: [two]
line: [three]
line: [four]
```

The solution to this exercise is in Section 5.4.

4 C++ Vectors

In the situation where one would otherwise use an array in C, code writers in C++ are more likely to use the `vector` class. Even though we haven't covered C++ classes yet in this lab, this shouldn't stop us from seeing how a `vector` can be used in basic operations similar to an array. We will point you to a couple on-line tutorials on the `vector` which treat the topic more thoroughly, but first a description of the minimal things you should know:

4.1 Declaring, building and copying a vector

One the best things about the `vector` is that, unlike a C-style array, you don't need to specify the size when it is declared.

```
int         my_array[100];   // Declaring an C-style array of integers
vector<int> my_vector;       // Declaring an array of integers using a vector
```

The above `my_vector` is initially empty and elements are added always to the end with:

```
my_vector.push_back(22);
my_vector.push_back(17);
my_vector.push_back(43);
```

Note that when the vector is initially declared, it is empty, so the following would not work:

```
vector<int> my_vector;
my_vector[0] = 22;      // error: accessing at an index out-of-bounds
my_vector[1] = 17;      //
my_vector[2] = 43;      //
```

If you would like to declare a `vector` with an initial size and default value for each element, this can be done with:

```
vector<int> my_vector(100, 0);    // my_vector has 100 elements all zero
my_vector[0] = 22;                // Now this is ok. 22 overwrites zero
my_vector[1] = 17;
my_vector[2] = 43;
```

As with most all C++ structures like the `vector` they can copied very simply as you would with fundamental data types like `int` or `double`:

```
vector<int> my_vector(100, 0);    // my_vector has 100 elements all zero
my_vector[0] = 22;                // Now this is ok. 22 overwrites zero
my_vector[1] = 17;
my_vector[2] = 43;

vector<int> his_vector = my_vector;  // his_vector has 100 elements {22,17,43,...,0}
```

Since a `vector` size is not set in stone upon creation, elements may be removed too (from the end) with the `pop_back()` function:

```
vector<int> my_vector;      // my_vector initially has no elements
my_vector.push_back(22);    //
my_vector.push_back(17);
my_vector.push_back(43);    // my_vector now has three element
...
my_vector.pop_back();       // my_vector now has two elements {22,17}
```

4.2 Accesing and Iterating Through Vectors

Accessing and iterating through a `vector` is very similar to array, but since the vector size is not fixed at the outset, we often want to know the size of the vector to prevent accessing an index out of bounds.

```
vector<int> my_vector;      // my_vector initially has no elements
my_vector.push_back(22);    //
my_vector.push_back(17);
my_vector.push_back(43);    // my_vector now has three element

unsigned int total_elements = my_vector.size();
```

Note the *type* returned by the `size()` function is of type `unsigned int` since the size can never be less than zero. Accessing a vector is the same as an array, e.g., `my_vector[8]` refers to the 9th element, and can be used on the left or right side of an assignment operator or passed as a parameter to a function. In fact the whole vector can be passed as a parameter in a function. Iterating through an array is also very similar:

```
vector<int> my_vector;      // my_vector initially has no elements
my_vector.push_back(22);
my_vector.push_back(17);
my_vector.push_back(43);

int total = 0;
for(unsigned int i=0; i<my_vector.size(); i++)
  total += my_vector[i];
```

A vector may be "reset" or emptied with the `clear()` function:

```
vector<int> my_vector;      // my_vector initially has no elements
my_vector.push_back(22);
my_vector.push_back(17);
my_vector.push_back(43);    // my_vector has three elements

my_vector.clear();          // my_vector again has no elements
```

Lastly, it is often useful to know the last element of a vector regardless of the index of the last element. For this, use the `back()` function:

```
vector<int> my_vector;      // my_vector initially has no elements
my_vector.push_back(22);
my_vector.push_back(17);
my_vector.push_back(43);    // my_vector has three elements

int val = my_vector.back(); // val is set to 43, my_vector still has three elements
```

4.3 Exercise 5: Build a Vector from Command Line Arguments

Write a program, similar to Excercise 2, to read in any number of integers from the command line, but constructing this time a `vector` of such integers, and printing them out afterwards. As before, scan the vector and determine the smallest number and build a new vector of size N-1 removing at most one element of the first vector equal to the smallest element. You can assume all arguments on the command line are valid integers.

Call your file `vectors_nosmall.cpp` and build it to the executable `vectors_nosmall`. When your program runs, it should be invocable from the command line with similar output:

```
$ ./vectors_nosmall 1 88 43 1 7 26
Total amount of integers provided: 6
  numbers[0]: 1
  numbers[1]: 88
  numbers[2]: 43
  numbers[3]: 1
  numbers[4]: 7
  numbers[5]: 26
The new array of integers:
  numbers[0]: 88
  numbers[1]: 43
  numbers[2]: 1
  numbers[3]: 7
  numbers[4]: 26
```

The solution to this exercise is in Section 5.5.

4.4 Exercise 6: Build a Vector of Strings Input from a File

Completion of this exercise is the primary goal of this lab. This exercise asks you to build a utility function we will use in later labs, that essentially reads the contents of a file into a vector of strings. We have touched on all the components of this through prior exercises so far, so this should just pull things together.

Write a program that reads in one argument from the command line (`--filename=file`), and reads in the contents of this file into a vector of strings. Afterwards, all non-empty strings will be written to the terminal on a separate line. An empty line should be indicated as empty.

Your program should consist of three files, `filebuff_main.cpp` containing your `main()` function and `FileBuffer.h` and `FileBuffer.cpp` containing your utility function. When your program runs, it should be invocable from the command line with:

```
$ cat test.txt      // Assuming a test file with five lines, one of them blank
one
two
three

four
$ ./filebuff --filename=test.txt
line: [one]
line: [two]
line: [three]
line: Empty!
line: [four]
```

The solution to this exercise is in Section 5.6.

4.5 Links to On-Line Tutorials on Vectors for Further Info

Here are a few good links for further reading. Feel free to explore and try some of the things you see.

- This link provides an example of a matrix with two vectors:
  http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html
- This link also touches on *algorithms* available for operation on vectors, such as sorting and randomizing:
  http://www.mochima.com/tutorials/vectors.html
- Here you can see all the functions defined on a vector in the menu on the lefthand side of the page:
  http://www.cplusplus.com/reference/vector/vector/vector/

4.6 Behind-the-Scenes Info about Vectors Worth Knowing Now

Some of the web links above touch on these issues, but in case you didn't read them or want further emphasis, we discuss a few of these ideas here.

Memory allocation in vectors:

The issue of allocating memory is largely hidden from the user when using a vector. A vector still uses a contiguous block of memory just like an array, but the user typically isn't concerned how the vector obtains or grows this memory. A vector has a *capacity* which is always greater or equal to the number of elements in the array. The vector will grow the capacity as needed, behind the scenes without the user needing to be concerned with the when or how. A good discussion can be found here:

- vector capacity: http://www.cplusplus.com/reference/vector/vector/capacity/

At times the user may want to think about vector capacity. If you are building a big vector by pushing one element at a time onto the end, the vector capacity will up-size periodically along the way by grabbing a bigger chunk of memory and copying the existing contents into the new memory before proceeding. However, if you know this vector will contain say a million elements, you can set

the capacity right at the time of creation so no re-sizing ever needs to be done. If this operation is a core operation (repeated many times over and over in a short period) in your program, then the reservation of vector capacity can make the difference in having a program that runs sufficiently fast.

A few things to remember about `vector` capacity:

- There is no way to set capacity at declaration. It is a two step process. First the declaration, then the reservation of capacity:

```
vector<int> vector_a;
vector_a.reserve(1000000);        // Has size 0. Capacity exactly 1,000,000
...
vector<int> vector_b(1000000);    // Has size 1,000,000. Capacity >= 1,000,000
```

- Invoking the `clear()` function doesn't change it's capacity, but sets the size to zero.
- You can know the capacity by invoking the `capacity()` function.

4.7 A Preview of other STL Containers

The `vector` is but one type of *container* available in C++. Below are a few short descriptions of other useful containers we may explore in later labs. These are all from the cplusplus.com website.

- *deque* (usually pronounced like "deck") is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).
- *Maps* are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order. In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key.
- *Sets* are containers that store unique elements following a specific order. In a set, the value of an element also identifies it (the value is itself the key), and each value must be unique. The value of the elements in a set cannot be modified once in the container, but they can be inserted or removed from the container.
- *Lists* are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions. List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

5 Solutions to Exercises

5.1 Solution to Exercise 1

```
/*-------------------------------------------------------------------------*/
/*  FILE:  arrays.cpp  (Fourth C++ Lab Exercise 1)                         */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/arrays.cpp                  */
/*  BUILD: g++ -o arrays arrays.cpp                                        */
/*  RUN:   ./arrays 23 11 98 32 2 23                                       */
/*-------------------------------------------------------------------------*/

#include <iostream>      // For use of the cout function
#include <cstdlib>       // For use of the atoi function

using namespace std;

int main(int argc, char **argv)
{
  int total_numbers = argc - 1;
  if(total_numbers == 0) {
    cout << "Usage: ./arrays NUMBER [MORENUMBERS]" << endl;
    return(1);
  }

  // Initialize the array
  int numbers[total_numbers];

  // Get the numbers from the command line arguments
  for(int i=1; i<argc; i++)
    numbers[i-1] = atoi(argv[i]);

  // Output the raw numbers provided
  cout << "Total amount of integers provided: " << total_numbers << endl;
  for(int i=0; i<total_numbers; i++)
    cout << "numbers[" << i << "]: " << numbers[i] << endl;

  return(0);
}
```

5.2 Solution to Exercise 2

```
/*-------------------------------------------------------------------------*/
/*  FILE:  arrays_nosmall.cpp  (Fourth C++ Lab Exercise 2)                 */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/arrays_nosmall.cpp          */
/*  BUILD: g++ -o arrays_nosmall arrays_nosmall.cpp                        */
/*  RUN:   ./arrays_nosmall 23 11 98 32 2 23                               */
/*-------------------------------------------------------------------------*/

#include <iostream>      // For use of the cout function
#include <cstdlib>       // For use of the atoi function

using namespace std;

int main(int argc, char **argv)
{
  int total_numbers = argc - 1;
  if(total_numbers == 0) {
    cout << "Usage: ./arrays NUMBER [MORENUMBERS]" << endl;
    return(1);
  }

  // Initialize the array
  int numbers[total_numbers];

  // Get the numbers from the command line arguments
  for(int i=1; i<argc; i++)
    numbers[i-1] = atoi(argv[i]);

  // Output the raw numbers provided
  cout << "Total amount of integers provided: " << total_numbers << endl;
  for(int i=0; i<total_numbers; i++)
    cout << "  numbers[" << i << "]: " << numbers[i] << endl;

  // Find the smallest number in the array
  int smallest = numbers[0];
  for(int i=1; i<total_numbers; i++) {
    if(numbers[i] < smallest)
      smallest = numbers[i];
  }

  // make a new array one size smaller
  int new_numbers[total_numbers-1];

  // Populate the new array with all but smallest number
  int curr_index = 0;
  bool hit_smallest = false;
  for(int i=0; i<total_numbers; i++) {
    if((numbers[i] == smallest) && !hit_smallest)
      hit_smallest = true;
    else {
      new_numbers[curr_index] = numbers[i];
      curr_index++;
    }
  }

  // Output the new array (minus the smallest of the originals)
  cout << "The new array of integers: " << endl;
  for(int i=0; i<total_numbers-1; i++)
    cout << "  numbers[" << i << "]: " << new_numbers[i] << endl;

  return(0);
}
```

5.3 Solution to Exercise 3

```
/*-------------------------------------------------------------------------*/
/*  FILE:  fileout.cpp  (Fourth C++ Lab Exercise 3)                        */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/fileout.cpp                 */
/*  BUILD: g++ -o fileout fileout.cpp                                      */
/*  RUN:   ./fileout --filename=test.txt one two three                     */
/*-------------------------------------------------------------------------*/

#include <iostream>      // For the cout function
#include <cstdio>              // For the fopen, fclose and fprintf functions

using namespace std;

int main(int argc, char **argv)
{
  string filename;
  for(int i=1; i<argc; i++) {
    string argi = argv[i];
    if(argi.find("--filename=") == 0)
      filename = argi.substr(11);
  }

  if(filename == "") {
    cout << "Usage: fileout --filename=test.txt one two three" << endl;
    return(1);
  }

  FILE *f = fopen(filename.c_str(), "w");
  if(!f) {
    cout << "Unable to open file: " << filename << endl;
    return(1);
  }

  for(int i=1; i<argc; i++) {
    string argi = argv[i];
    if(argi.find("--filename=") != 0)
      fprintf(f, "%s\n", argi.c_str());
  }

  fclose(f);
  return(0);
}
```

5.4 Solution to Exercise 4

```
/*-------------------------------------------------------------------------*/
/*  FILE:  filein.cpp  (Fourth C++ Lab Exercise 4)                         */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/filein.cpp                  */
/*  BUILD: g++ -o filein filein.cpp                                        */
/*  RUN:   ./filein --filename=test.txt                                    */
/*-------------------------------------------------------------------------*/

#include <iostream>      // For the cout function
#include <cstdio>        // For the fopen, fclose, fgetc, feof functions

using namespace std;

int main(int argc, char **argv)
{
  // Find the name of the file to be read
  string filename;
  for(int i=1; i<argc; i++) {
    string argi = argv[i];
    if(argi.find("--filename=") == 0)
      filename = argi.substr(11);
  }

  // If no file specified, produce usage information and exit now.
  if(filename == "") {
    cout << "Usage: filein --filename=test.txt" << endl;
    return(1);
  }

  // Open the specified file for reading and exit with message if failed
  FILE *f = fopen(filename.c_str(), "r");
  if(!f) {
    cout << "Unable to open file: " << filename << endl;
    return(1);
  }

  // Begin reading in file, ending only when end-of-file detected
  string str = "line: [";
  while(1) {
    int c = fgetc(f);

    if(feof(f))
      break;
    if(c == '\n') {
      str += "]";
      cout << str << endl;
      str = "line: [";
    }
    else
      str += c;
  }

  // Make sure we close the file
  fclose(f);
  return(0);
}
```

5.5 Solution to Exercise 5

```
/*-------------------------------------------------------------------------*/
/*  FILE:  vectors_nosmall.cpp  (Fourth C++ Lab Exercise 5)                 */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/vectors_nosmall.cpp          */
/*  BUILD: g++ -o vectors_nosmall vectors_nosmall.cpp                       */
/*  RUN:   ./vectors_nosmall 23 11 98 32 2 23                               */
/*-------------------------------------------------------------------------*/

#include <iostream>      // For use of the cout function
#include <cstdlib>       // For use of the atoi function
#include <vector>        // For use of the STL vector class

using namespace std;

int main(int argc, char **argv)
{
  vector<int> numbers;
  // Get the numbers from the command line arguments
  for(int i=1; i<argc; i++)
    numbers.push_back(atoi(argv[i]));

  // Check if no numbers provided, exit if so.
  if(numbers.size() == 0) {
    cout << "Usage: ./arrays NUMBER [MORENUMBERS]" << endl;
    return(1);
  }

  // Output the raw numbers provided
  cout << "Total amount of integers provided: " << numbers.size() << endl;
  for(unsigned int i=0; i<numbers.size(); i++)
    cout << "numbers[" << i << "]: " << numbers[i] << endl;

  // Find the smallest number in the array
  int smallest = numbers[0];
  for(unsigned int i=1; i<numbers.size(); i++) {
    if(numbers[i] < smallest)
      smallest = numbers[i];
  }

  // make a new array one size smaller  // make a new vector
  vector<int> new_numbers;
  bool hit_smallest = false;
  for(unsigned int i=0; i<numbers.size(); i++) {
    if((numbers[i] == smallest) && !hit_smallest)
      hit_smallest = true;
    else
      new_numbers.push_back(numbers[i]);
  }

  // Output the new array (minus the smallest of the originals)
  for(unsigned int i=0; i<new_numbers.size(); i++)
    cout << "  numbers[" << i << "]: " << new_numbers[i] << endl;

  return(0);
}
```

5.6 Solution to Exercise 6

```
/*-------------------------------------------------------------------------*/
/*  FILE:  filebuff_main.cpp  (Fourth C++ Lab Exercise 5)                  */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/filebuff_main.cpp           */
/*  BUILD: g++ -o filebuff FileBuffer.cpp filebuff_main.cpp                */
/*  RUN:   ./filebuff --filename=test.txt                                  */
/*-------------------------------------------------------------------------*/

#include <iostream>      // For use of the cout function
#include <vector>        // For use of the STL vector class
#include "FileBuffer.h"

using namespace std;

int main(int argc, char **argv)
{
  // Find the name of the file to be read
  string filename;
  for(int i=1; i<argc; i++) {
    string argi = argv[i];
    if(argi.find("--filename=") == 0)
      filename = argi.substr(11);
  }

  // If no file specified, produce usage information and exit now.
  if(filename == "") {
    cout << "Usage: filein --filename=test.txt" << endl;
    return(1);
  }

  vector<string> lines = fileBuffer(filename);

  cout << "Total Lines: " << lines.size() << endl;
  for(unsigned int i=0; i<lines.size(); i++) {
    if(lines[i].length() > 0)
      cout << "line: [" << lines[i] << "]" << endl;
    else
      cout << "line: Empty!" << endl;
  }

  return(0);
}
```

```
/*--------------------------------------------------------------------*/
/*  FILE:  FileBuffer.h                                               */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/FileBuffer.h           */
/*--------------------------------------------------------------------*/

#ifndef FILE_BUFFER_HEADER
#define FILE_BUFFER_HEADER

#include <vector>
#include <string>

std::vector<std::string> fileBuffer(std::string);

#endif
```

```
/*--------------------------------------------------------------------*/
/*  FILE:  FileBuffer.cpp                                             */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/FileBuffer.cpp         */
/*  BUILD: A utility meant to be built with other source code         */
/*         For example: g++ -o myprog FileBuffer.cpp main.cpp         */
/*--------------------------------------------------------------------*/

#include "FileBuffer.h"
#include <cstdio>

using namespace std;

vector<string> fileBuffer(string filename)
{
  // Create a vector of strings. In ANY case this is the return type
  vector<string> fvector;

  // Check that the file can be read. Return empty vector otherwise
  FILE *f = fopen(filename.c_str(), "r");
  if(!f)
    return(fvector);

  // Create a string buffer to hold each line. Reserve a big size to
  // allow for efficient growth.
  string strbuff;
  strbuff.reserve(50000);

  // Read in the file one character at a time. Stopping only when the
  // file pointer points to the end of file.
  while(1) {
    int intval = fgetc(f);
    char c = (char) intval;
    if(feof(f)) {
      if(strbuff.length() != 0)
        fvector.push_back(strbuff);
      break;
    }
    else if(c == '\n') {
      fvector.push_back(strbuff);
      strbuff.clear();
    }
    else
      strbuff.push_back(c);
  }

  // Make sure we close the file.
  fclose(f);

  return(fvector);
}
```