COP 1334, Spring 2014

# Tutorial

## Reading Data from a Comma Separated Values (CSV) File


Prof. Adi Zejnilovic
4/27/2014

The following tutorial will walk you through the process of reading data from a comma delimited file.

## CSV

Flat files usually contain so called Comma Separated Values (CSV) information. This format is used to exchange data between applications such as Microsoft Excel and a program written in C++. A CSV file is similar to spreadsheet except that the data is separated by commas. Following is an example of a comma separated file:

```
Aston Martin,V8 Vantage,2014,15.1973
Audi,R8,2014,16.8379
BMW,Z4,2014,34.1594
Lamborghini,Gallardo,2014,15.7180
Porsche,Cayman,2014,23.8000
```

In the above example, it is clear that a data record consists of a car make, model, year manufactured, and miles per gallon (mpg) information.

So how do you programmatically read this information from a file?

## How to Read Data from A CSV File

First of all, you need to create a data structure that can be used to accommodate the information contained in the file. Let's examine the fields that comprise a record:

| Make | Model | Year | MPG |
|---|---|---|---|
| Aston Martin | V8 Vantage | 2.2014 | 15.1973 |

One thing that comes to mind is the following structure:

```
1  struct CarInfo
2  {
3     string make;
4     string model;
5     int    year;
6     double mpg;
7  };
```

The above structure would those suffice if we only needed to keep track of only one car. However, we want to keep track of multiple cars, so we need to declare an array of structures. Since we do not know how many cars we're going to encounter in the file, we need to declare an array large enough to accommodate many cars. Here we can guess the hundred cars for example.

Often times, CSV files may come with a number of records on the first line followed by actual records:

```
5
Aston Martin,V8 Vantage,2014,15.1973
Audi,R8,2014,16.8379
BMW,Z4,2014,34.1594
Lamborghini,Gallardo,2014,15.7180
Porsche,Cayman,2014,23.8000
```

where “5” would represent number of records that follow. In that case, simply read the number of records that follow into an integer variable and proceed to write your programming logic.

For this tutorial, we’re going to assume that the file does not come with the number of records in the first line of input. To simplify the tutorial were going to assume that there are five cars in our input file.

## Using the string Objects to Read the Data

We first present a program that will work with string objects:

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4  #include <fstream>
5
6  using namespace std;
7
8  const int NUM_CARS = 5;
9
10 struct CarInfo
11 {
12     string make;
13     string model;
14     int    year;
15     double mpg;
16 };
17
18 bool getCars(CarInfo [], int );
19 void displayCars(CarInfo [], int );
20
21 int main(int argc, char *argv[])
22 {
23     CarInfo myCars[5];     // declare an array of 5 structures
24
25     if (!getCars(myCars, NUM_CARS))
26     {
27        cout << "Trouble reading from input file.";
28        cout << "\nExiting...";
29        system("PAUSE");
30        return EXIT_FAILURE;
31     }
32     displayCars(myCars, NUM_CARS);
33
34     system("PAUSE");
35     return EXIT_SUCCESS;
36 }
37
38 void displayCars(CarInfo cars[], int size)
39 {
40      for (int i=0; i<size; i++)
```

```
41      {
42         cout << "Make: " << cars[i].make << endl;
43         cout << "Model: " << cars[i].model << endl;
44         cout << "Year: " << cars[i].year << endl;
45         cout << "MPG: " << cars[i].mpg << endl;
46         cout << endl;
47      }
48 }
49
50 bool getCars(CarInfo cars[], int size)
51 {
52     ifstream inFile;
53     string temp;
54     inFile.open("Cars.txt");
55     if (!inFile)
56     {
57         return false;
58     }
59     else
60     {
61         for (int i=0; i<size; i++)
62         {
63               getline(inFile, cars[i].make, ',');
64               getline(inFile, cars[i].model,',');
65               getline(inFile, temp, ',');     // read as a temp string
66               cars[i].year = atoi(temp.c_str()); // convert to integer
67               getline(inFile, temp);
68               cars[i].mpg = atof(temp.c_str());
69         }
70         return true;
71     }
72 }
```

When examining the above code, specifically the getCars function, we can see that in order to successfully treat the data from the file into the array of structures, we need to use the get one function. The get one function has couple of overrides, one of which is demonstrated on line 63. This overrides of the get one function takes in three parameters:

```
getline(inFile, cars[i].make, ',');
```

where the inFile is the stream object for which the characters are extracted – in our case that would be the input file, cars**[i].**make which is a string object that receives the extracted information, and lastly the delimiter (‘,’) – in this case it’s the comma.

This works fine for the first to structure members make and model because they are of data type string. However when we tried to read in the numeric data from the file, we need to read the data into a temporary string object and then convert it to appropriate data type. In case of the structure member year which is an integer, we first read data from the input file into a temp string object (line 65). Next we utilized the atoi function to convert this data to an integer (line 66). Since the atoi function requires a constant character array, we call the c_str() member function.

We repeat the process for the mpg structure member on lines 67 and 68.

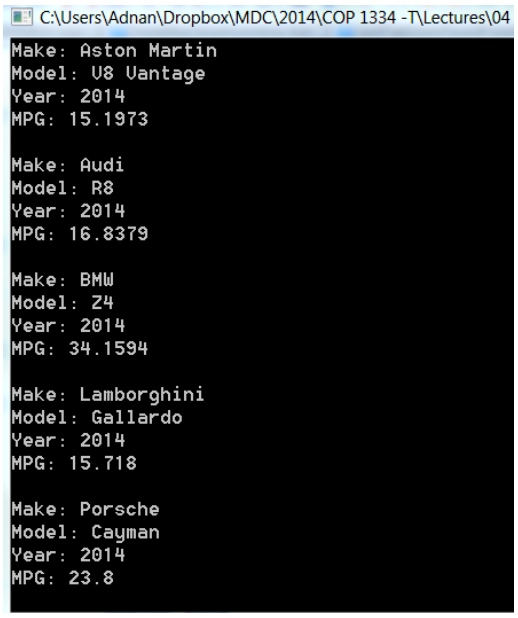


Figure 1. Output

## Using the C-Style String to Read the Data

Next we present the same program with C-Style strings (character arrays) data type for make and model structure members:

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4  #include <fstream>
5
6  using namespace std;
7
8  // Global Constants
9  const int NUM_CARS = 5;
10 const int MAX_SIZE = 50;
11
12 // Programmer defined data types
13 struct CarInfo
14 {
15     char make[MAX_SIZE];
16     char model[MAX_SIZE];
17     int    year;
18     double mpg;
19 };
20
21 // function prototypes
22 bool getCars(CarInfo [], int );
23 void displayCars(const CarInfo [], int );
24
25 int main(int argc, char *argv[])
26 {
27     CarInfo myCars[5];
28
29     if (!getCars(myCars, NUM_CARS))
```

```
30      {
31         cout << "Trouble reading from input file.";
32         cout << "\nExiting...";
33         system("PAUSE");
34         return EXIT_FAILURE;
35      }
36      displayCars(myCars, NUM_CARS);
37
38      system("PAUSE");
39      return EXIT_SUCCESS;
40  }
41
42  void displayCars(const CarInfo cars[], int size)
43  {
44       for (int i=0; i<size; i++)
45       {
46          cout << "Make: " << cars[i].make << endl;
47          cout << "Model: " << cars[i].model << endl;
48          cout << "Year: " << cars[i].year << endl;
49          cout << "MPG: " << cars[i].mpg << endl;
50          cout << endl;
51       }
52  }
53
54  bool getCars(CarInfo cars[], int size)
55  {
56      ifstream inFile;
57      char temp[MAX_SIZE]; // temp char array
58      inFile.open("Cars.txt");
59      if (!inFile)
60      {
61          return false;
62      }
63      else
64      {
65          for (int i=0; i<size; i++)
66          {
67                inFile.getline(cars[i].make, MAX_SIZE, ',');
68                inFile.getline(cars[i].model, MAX_SIZE, ',');
69
70                // read as a temp C-Style string
71                inFile.getline(temp, MAX_SIZE, ',');
72
73                // convert to an integer
74                cars[i].year = atoi(temp);
75
76                // read as a temp C_Style string
77                inFile.getline(temp, MAX_SIZE, '\n');
78
79                // convert to a double
80                cars[i].mpg = atof(temp);
81          }
82          return true;
83      }
84  }
```

While much of the code remains the same (displayCars function), there is a slight change in the way we utilize the getline function to read information from the input file. On line 66, we read the make in the following manner:

```
inFile.getline(cars[i].make, MAX_SIZE, ',');
```

inFile is the stream object from which the data is to be read. The getline function override takes in three parameters:

- cars[i].make - the character array into which the read data is to be copied,
- MAX_SIZE - the maximum length the data that is to be read, and
- delimiter (a comma).

The compiler is going to read either MAX_SIZE characters from the input stream or as many characters as needed until it encounters the delimiter whichever happens first. On online number 77, the delimiter is the newline character. Even though flat files are referred to as the CSV (Comma Separated Values) files – the delimiters don't necessarily have to be commas.

## Conclusion

In this tutorial, we demonstrated how to read the data from a CSV file into an abstract data type. While reading string/character data from the input file is rather easy, reading numeric data is done in two steps: first the data is read into a temporary variable of string/character data type followed by the conversion to appropriate numeric data type.

The delimiters used in CSV files don't necessarily have to be commas. They could be any character – as demonstrated in this tutorial, we used a newline character as one of the delimiters.