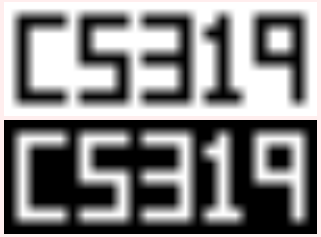


CS319: Scientific Computing (with C++)

Files and Streams

Week 5: 9am and 4pm, 8 February 2017

Annotated

Labs and stuff

(a) The **lab 3** assignment is due Monday at 9am. The purpose of this exercise, it to ensure participation and facilitate feedback. With that in mind:

- Submitting faulty code is better than submitting nothing;
- Try to submit early, and ask for feed-back. If there is potential for improvement, you can re-submit.
- **The purpose of these exercises is learning**.

(b) Have a look at the sample solution to Lab 2 to get a sense for how commenting and **indent style** can improve clarity. I try to be consistent in my indent style (based on *Allman*-style, though with smaller indents). If you develop your own style, and are consistent, you'll write better code more efficiently. For more, see
`https://en.wikipedia.org/wiki/Indent_style`

(c) Use the compiler to help you code. For example, turn all on usual warnings (`-Wall`) to identify possible bugs in the code. Also: gdb, valgrind.

(d) Next Monday we will start **Lab 4**; you will have two weeks to complete that assignment.

myprog > out.txt ... myprog 2> err.txt

All of the C++ programs we have looked at so far took their input from the *standard input stream*, which is usually the keyboard. Example:

```
cout << "Enter an inteter: ";
cin >> i;
```

Although the *standard input stream* can be redirected to be, for example, a file (easily done on Linux), it is usually necessary to open a file **from within the program** and take the data from there. The data is then processed and written to a new file.

To achieve either of these tasks in C++, we create a **file stream** and use it just as we would `cin` or `cout`.

We'll start by looking at a simple example:

(i) open a file,
(ii) count the number of characters,
(iii) save this number to a new file.

Once we have the basic idea, we'll take a closer look at each operation (opening, reading, writing).

When working with files, we need to include the *fstream* header file.

To **read** from a file, declare an object of type `ifstream`; to **write** to a file, declare an object of type `ofstream`.

Open the file by calling the `open()` member function.

To read a single character, can use *InFile.get()*

01CountChars.cpp

```
// File:  01CountChars.cpp
// Author:  Niall Madden
// Date:  08/02/2017
// First example using file streams.
#include <iostream>
#include <fstream>
#include <cstdlib>

int main(void )
{
  std::ifstream InFile;
  std::ofstream OutFile;
  char c;

  std::cout << "Processing the contents of"
      << " CPlusPlusTerms.txt" << std::endl;
  std::cout << " See file Output.txt for"
      << " more information." << std::endl;
  InFile.open("CPlusPlusTerms.txt");
  OutFile.open("Output.txt");

  int i=0;
  InFile.get( c );
```

close a file

If there are no more characters left in the input stream, then `InFile.eof()` evaluates as *true*.

Use the steam objects just as you would use `cin` or `cout`:

`InFile >> data` or
`OutFile << data`.

Close the files:
`InFile.close()`,
`OutFile.close()`

01CountChars.cpp (continued)

```
  while( ! InFile.eof() ) {
    i++;
    InFile.get( c );
  }

  OutFile << "CPlusPlusTerms.txt contains "
    << i << " characters \n";

  InFile.close();
  OutFile.close();

  return(0);
}
```

The method `open` works differently for `ifstream` and `ofstream`:

- `InFile.open()` Opens an existing file for reading,
- `OutFile.open()` Opens a file for writing. If it already exists, its contents are overwritten.

The first argument to `open()` contains the file name, and is an array of characters. More precisely, it is of type `const char*`.

For example, we could have opened the input file in the last example with:

```
char InFileName[20]="CPlusPlusTerms.txt";
...
std::cout << "Processing the contents of "
          << InFileName << std::endl;
...
InFile.open(InFileName);
```

Note that this `char` array is **not** the same as a `string`. The precursor to C++, C, handled strings this way, so they are known as **C-style strings**.

If we do want to use C++ style strings (and we probably do), we have to do it as follows. In this example we'll prompt the user to enter the file name.

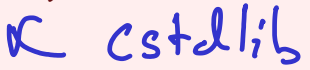
```
std::ifstream InFile;
std::string InFileName;
std::cout << "Input the name of a file: " << std::endl;
std::cin >> InFileName;
InFile.open(InFileName.c_str())
```

C++ string

C-style string.

If you are typing the file name, there is a chance you will mis-type it or have it placed in the wrong folder: so **always** check that the file was opened successfully. To do this, use the `fail()` function, which evaluates as `true` if the file was not opened correctly:

```
if (InFile.fail()){
  std::cerr << "Error - cannot open " <<
     InFileName << std::endl;
  exit(1);
}
```



A better approach in this case might be to use a `while` loop, so the user can re-enter the filename. See `02CountCharsV02.cpp`

Recall that if you open an existing file for **output**, its contents are lost. If you wish to **append** data to the end of an existing file, use

To open an existing file and **append** to its contents, use

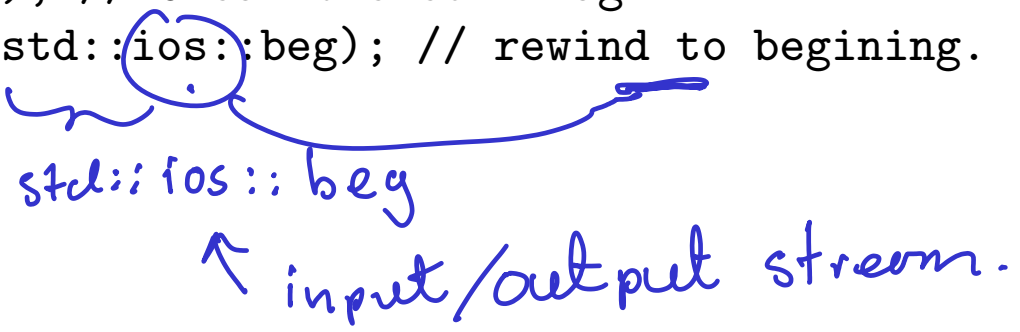
```
OutFile.open("Output.txt", std::ios::app);
```

Other related functions include `is_open()` and, of course, `close()`

Above we also saw that `InFile.eof()` evaluates as *true* if we have reached the end of the (read) file.

Related to this are

```
InFile.clear(); // Clear the eof flag
InFile.seekg(std::ios::beg); // rewind to begining.
```



In the above example, we read a character from the file using `InFile.get(c)`. This read the next character from the InFile stream and stores it in `c`. It will do this for any character, even non-printable ones (such as the newline char). For example, if we wanted to extend our code above to count the number of lines in the file, as well as the number of characters, we could use:

```
std::ifstream InFile;
int CharCount=0, LineCount=0;
...
// Open the file, etc.
InFile.get( c );
while( ! InFile.eof() ) {
  CharCount++;
  if (c == '\n')
     LineCount++;
  InFile.get( c );
}
```

"new line" (ie "end of line")

Alternatively, we could the **stream extraction operator:** `InFile >> c;`
However, this would ignore non-printable characters.

One can also use `get()` to read C-style strings. However, to achieve this task, it can be better to use `getline()`, which allows us to specify a delimiter character: see the "CSV" example later.

Portable Bitmap Format (pbm)

Images analysis and processing is an important sub-field of scientific computing.

There are many different formats: you are probably familar with JPEG/JPG, GIF, PNG, BMP, TIFF, and others. One of the simplest formats is the **Netpbm format**, which you can read about at
`https://en.wikipedia.org/wiki/Netpbm_format`

There are three variabls:

Portable BitMap files represent black-and-white images, and have file extension *.pbm*

Portable GrayMap files represent gray-scale images, and have file extension *.pgm*

Portable PixMap files represent 8-big colour (RGB) images, and have file extension *.ppm*

In this example, we'll focus on `.pbm` files.

Portable Bitmap Format (pbm)

CS319.pbm

```
1  P1
   25 9
3  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0 0 1 1 1 1 0
5  0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 0
   0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1 0
7  0 1 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 1 0 0 1 1 1 1 0
   0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0
9  0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0
   0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0 0 1 0
11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

- The first line is the "magic number". Here "P1" means that it is a PBM format ASCII (i.e, plain-text) file.
- The second line has two integer representing the number of columns and rows of pixels in the image, respectively.
- The remaining lines store the matrix of pixel values: 0 is "white", and 1 is "black".

The file 03FlipPBM.cpp shows how to read such an image, and output its negative. (See notes from class).

CSV files

For our next example of working with files, we'll write a program that reads data from a "*C*omma *S*eparated *V*alues" (*CSV*) file.

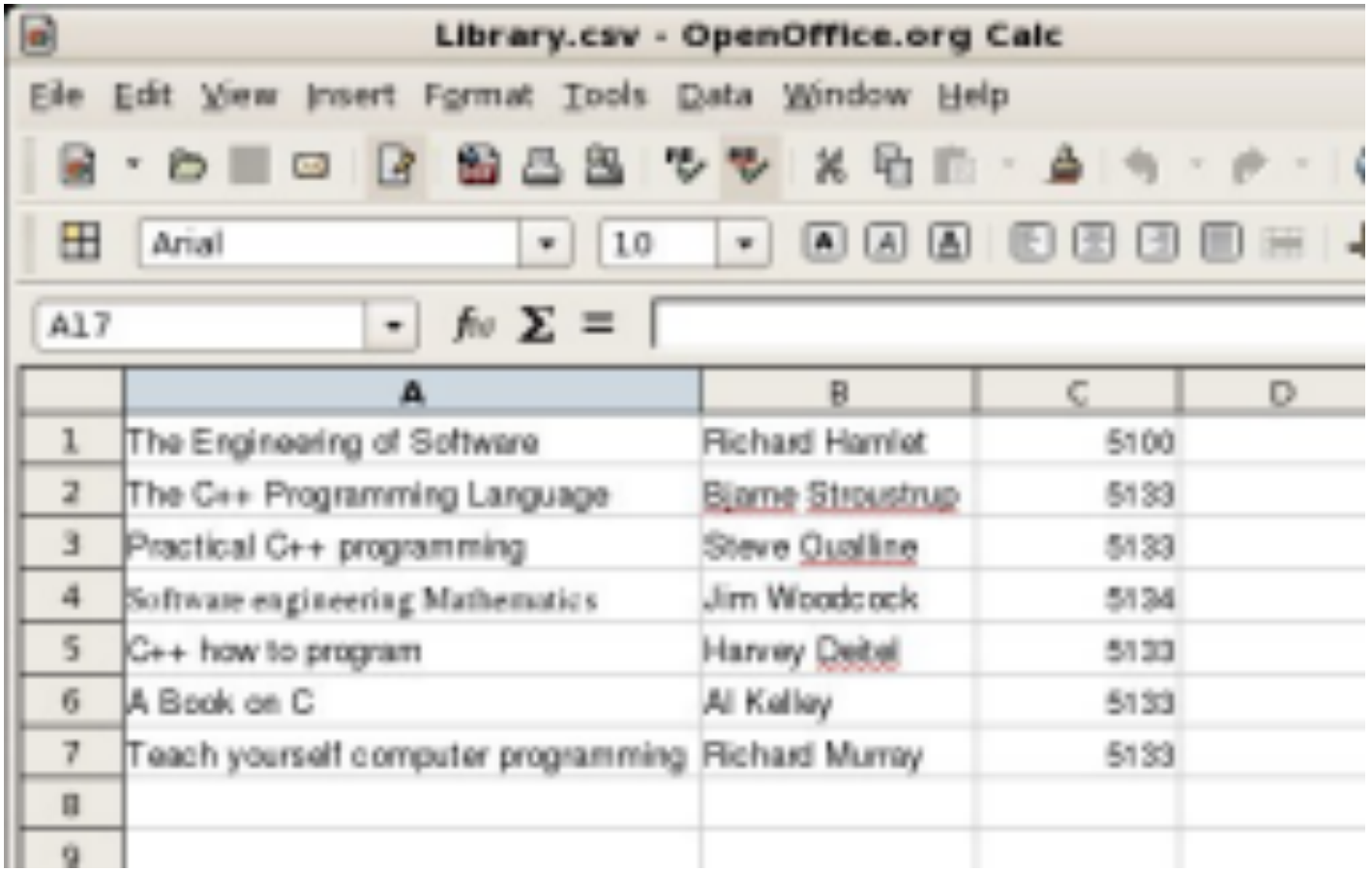
There are many accountancy and spread-sheet packages available. It is necessary for them to be able to share data. Therefore, even though they all have their own file format, they must be able to read and write a neutral data type. This is often *CSV*.

Your favourite data-handling system (e.g., Excel, LibreOffice,...) can read and write these files.

In a CSV file, the contents of cells from the same row are simply separated by commas. Any addition information, such as font type, text alignment, formulae, etc., is lost.

CSV files

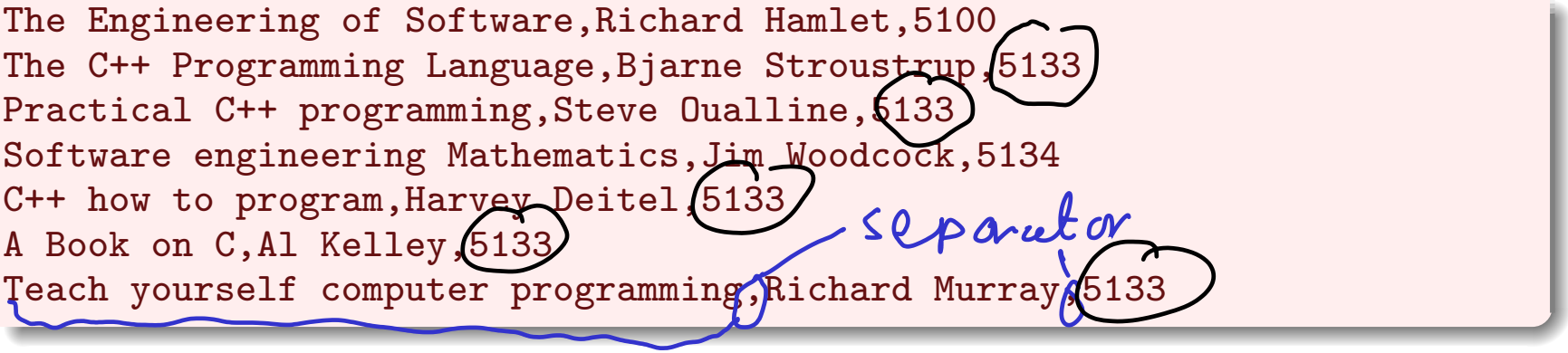
The example we'll look at is based on some library data:



CSV files

When this is saved to a CSV file we get

```
The Engineering of Software,Richard Hamlet,5100
The C++ Programming Language,Bjarne Stroustrup,5133
Practical C++ programming,Steve Oualline,5133
Software engineering Mathematics,Jim Woodcock,5134
C++ how to program,Harvey Deitel,5133
A Book on C,Al Kelley,5133
Teach yourself computer programming,Richard Murray,5133
```



We'll look at how to write a C++ program that can open this file and read data from it.

We shall assume that we know the structure of the file. In particular, we'll assume we know how many columns there are, and what they contain.

CSV files

04Library_CSV.cpp

```
6  #include <iostream>
   #include <string>
8  #include <iomanip>
   #include <fstream>
10 #include <cstdlib> // For EXIT_FAILURE and atoi

12 int main(void)
   {
14   std::ifstream InFile;
     std::string InFileName="Library.csv";
16   char str_tmp[100];

18   std::string *Author, *Title;
     int *CallNumber;
20
     InFile.open(InFileName.c_str());
22
     if (InFile.fail())
24   {
       std::cerr << "Error - can't open " << InFileName << std::endl;
26     exit(EXIT_FAILURE);
     }
```

Also (check opens in excel).

pointers for DMA.

Don't know yet how many records.

CSV files

04Library_CSV.cpp

```
30  // First count the number of entries
    char c;
    int Lines=0;
32  InFile.get(c);
    while(!InFile.eof())
34  {
      if (c=='\n')
36      Lines++;
      InFile.get(c);
38  }
    std::cout << "There are " << Lines << " in " << InFileName << std::endl;
40
    Author = new std::string [Lines];
42  Title = new std::string [Lines];
    CallNumber = new int [Lines];
```

read one char at a time

similar to 02 CountChars....

CSV files

04Library_CSV.cpp

```
45    InFile.clear(); // Clear the eof flag
46    InFile.seekg(std::ios::beg); // rewind to beginning.

48    for (int i=0; i< Lines; i++)
49    {
50        InFile.get(str_tmp, 99, ',');
51        Title[i] = str_tmp;
52        InFile.ignore();

54        InFile.get(str_tmp, 99, ',');
55        Author[i] = str_tmp;
56        InFile.ignore();

58        InFile.get(str_tmp, 99, '\n');
59        CallNumber[i] = atoi(str_tmp);
60        InFile.ignore();
61    }
```

Read the file again

read until comma, but do not read it

C-string

at most 99 chars.

ignore the comma

"ascii to integer"

book name, author name, call \n

↑ new line.

CSV files

We'll check if it worked by outputting a subset of the data.

04Library_CSV.cpp

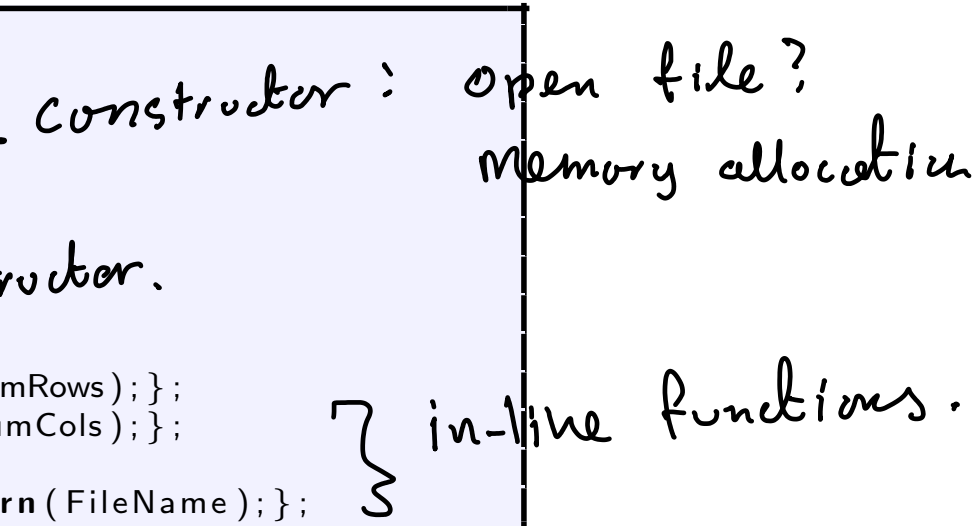
```
63    std::cout << "Here are the 5133 books: " << std::endl;
64    for (int i=0; i<Lines; i++)
65    {
66       if (CallNumber[i] == 5133)
67         std::cout << std::setw(20) << Author[i] << ": "
68              << Title[i] << std::endl;
69    }

71    delete [] Author;
72    delete [] Title;
73    delete [] CallNumber;
74    InFile.close();
75    return(0);
```

A class for CSV files

In the spirit of OOP, we can create our own objects that represent CSV files. I propose a class with the following definition

05CSV_Class.cpp

```
13 class CSVfile {
14 private:
15   std::ifstream csvfile;
16   std::string FileName;
17   int NumRows;
18   int NumCols;
19 public:
20   CSVfile(void);
21   ~CSVfile(void);
22   int CountRows(void) {return(NumRows);};
23   int CountCols(void) {return(NumCols);};
24   std::string getcell(void);
25   std::string name(void)   {return(FileName);};
26   bool eof(void)       {return(csvfile.eof());};
27 };
```



Discussion:

- What should the methods do?
- What private elements do we need?

A class for CSV files

05CSV_Class.cpp

```
29  CSVfile::CSVfile(void)
30  {
31    std::cout << "Enter the file name : ";
32    std::cin >> FileName; // "Library.csv";
33    csvfile.open(FileName.c_str());
34    if (csvfile.fail())
35    {
36      std::cerr << "Error - can't open " << FileName << std::endl;
37      exit(EXIT_FAILURE);
38    }

40    // First count the number of rows and cols
41    NumRows=0;
42    NumCols=0;

44    char c;
45    csvfile.get(c);
46    // Read the first line
47    while((c!='\n') && (!eof()) )
48    {
49      if (c==',')
50        NumCols++;
51      csvfile.get(c);
52    }
53    NumCols++;
54    NumRows++;
```

Constructor

} prompt user for file name (good?)

} already defined (private elements).

Don't know how many rows & cols in file

counting commas and new line, on first line only (assuming every line has some number of cells.

A class for CSV files

05CSV_Class.cpp

```
55    // Now read the rest
56    csvfile.get(c);
57    while( !eof() )
58    {
59      if (c=='\n')
60        NumRows++;
61      csvfile.get(c);
62    }
63    csvfile.clear(); // Clear the eof flag
64    csvfile.seekg(std::ios::beg); // rewind to beginning.
65  }
```

for the rest of the file, just count new lines.