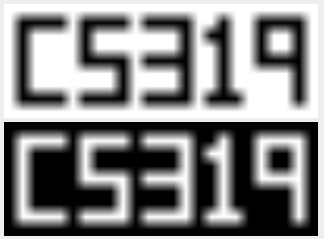
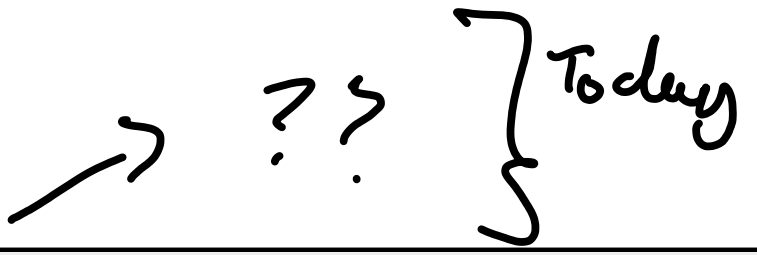
Wednesday, 10th March @ 4pm



CS319: Scientific Computing (with C++)

Week 5: Streams and files

9am, 09 March, and 4pm, 10 March, 2021



CS319 – Week 5
Week 5: Streams and files

Start of ...

PART 4 (ii): Files

This section is split into two parts.
Part 4–(i) was recorded Tuesday, 9 March
Part 4–(ii) was recorded Wednesday, 10 March

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.

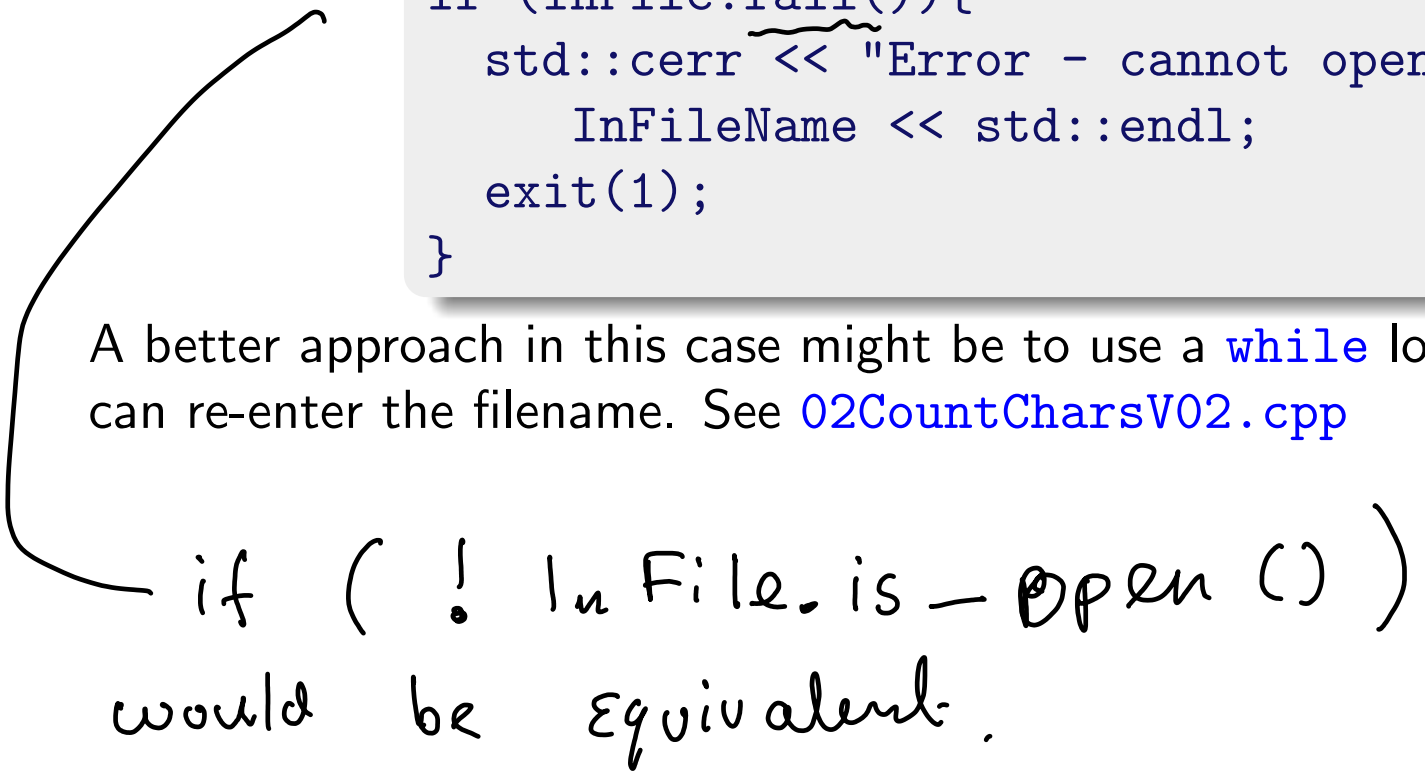
```
std::ifstream InFile;
std::string InFileName;
std::cout << "Input the name of a file: " << std::endl;
std::cin >> InFileName;
InFile.open(InFileName.c_str())
```

c_str() method return a 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.

↑ End-of-file

Related to this are

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

↑ "input/output stream"

In the above example, we read a character from the file using `InFile.get(c)`. This reads 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 );
}
```

ie
End of
Part 4-(i).

new line
char (like endl).

Alternatively, we could the **stream extraction operator:**

```
InFile >> c;
```

(similar to cin)

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.

CS319 – Week 5
Week 5: Streams and files

END OF PART 4

CS319 – Week 5
Week 5: Streams and files

Start of ...

PART 5: Portable Bitmap Format (pbm)

We'll introduce this image format as a motivation for working with files.

Part 5: Portable Bitmap Format (pbm)

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

pdf / eps

There are many different formats: you are probably familiar 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 variants:

Portable BitMap files represent black-and-white images, and have file extension *.pbm* { we'll do this one }

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.

Part 5: Portable Bitmap Format (pbm)

magic number.

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
```

25 cols + 9 rows

Image.

- 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".

Part 5: Portable Bitmap Format (pbm)

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

is swap 0 for 1 & 1 for 0.

03FlipPBM.cpp

```
  std::ifstream InFile;
16 std::ofstream OutFile;
  std::string InFileName, OutFileName;

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

[The pbm file for the CS319 image is on BB].

Part 5: Portable Bitmap Format (pbm)

03FlipPBM.cpp

```
   while (InFile.fail() )
24 {
     std::cout << "Cannot open " << InFileName << " for reading."
26       << std::endl;
     std::cout << "Enter another file name : ";
28   std::cin >> InFileName;
     InFile.open(InFileName.c_str());
30 }
   std::cout << "Successfully opened " << InFileName << std::endl;
```

Handling a mis-typed filename.

Part 5: Portable Bitmap Format (pbm)

03FlipPBM.cpp

```
   // Open the output file
34 OutFileName = "Negative_"+InFileName;
   OutFile.open(OutFileName.c_str());

   std::string line;
38 // Read the "P1" at the start of the file
   InFile >> line;
40 OutFile << "P1" << std::endl;

42 // Read the number of columns and rows
   unsigned int rows, cols;
44 InFile >> cols >> rows;
   OutFile << cols << " " << rows << std::endl;

   std::cout << "read: cols=" << cols << ", rows="
48           << rows << std::endl;
```

Part 5: Portable Bitmap Format (pbm)

03FlipPBM.cpp

```
50  for (unsigned int i=0; i<rows; i++)
    {
52    for (unsigned int j=0; j<cols; j++)
      {
54      int pixel;
        InFile >> pixel;
56      OutFile << 1-pixel << " ";
      }
58    OutFile << std::endl;
    }
60  InFile.close();
    OutFile.close();

    std::cout << "Negative of " << InFileName << " written to "
64            << OutFileName << std::endl;
    return(0);
```

(will read an int from file).

so 0 → 1

and 1 → 0.

CS319 – Week 5
Week 5: Streams and files

END OF PART 5

CS319 – Week 5
Week 5: Streams and files + Templates

Start of ...

PART 6: Templates

We'll now start building towards solving the problem of, given a VERY long list of (pass)words, determine which ones occur most frequently.

The source of the data is the infamous **RockYou** password file, a list of over 30,000,000 unencrypted passwords stolen from RockYou in 2009, and now widely available online. The version we'll work with was provided by David Malone from Maynooth University, who used it in an article Investigating the Distribution of Password Choices[1]

The benign reasons for wanting to do this include

- We could use this as a way of testing the security of our own systems;
- Understanding how these attacks are done help up protect against them.

We'll solve part of the problem this week, and finish the rest next week.

[1] David Malone and Kevin Maher. *Investigating the Distribution of Password Choices*. International conference on the World Wide Web (WWW). 19 April 2012.

We have now worked out that we need to do some list sorting. Presently, we'll recap on a sorting function that we used in Week 3. However, it just sorted integers. We'll need to sort list of strings, or perhaps lists of objects belonging to a class we define. So we would like to write a `sort` function that works for **any** datatype.

If we took our old `Sort(int *list, int length)` function (from `Week03/09Sort.cpp`), we could rewrite it for (say) `string`s: `Sort(string *list, int length)`

Most of the source code of the two functions would be identical: we'd just replace several instances of the datatype `int` with `string`.

To avoid this repetition, and to allow us to write functions or class **generic** datatypes, C++ provides `template`s.

Today we will only consider **function templates**. We'll return to the related idea of **class templates** another time.

To perform essentially identical operations for different types of data compactly, use function templates.

- Syntax: `template <typename T>` immediately precedes the function definition. It means that we'll be referring to the generic datatype as `T` in the function definition.
- Write a single function template definition. In it, the generic datatype is named `T`.
- Based on the argument types provided in calls to the function, the compiler automatically creates functions to handle each type of call appropriately.

In the example below, which you can find in detail in `04FunctionTemplate.cpp`, we'll write three functions:

a. `PrintList(MyType *x, int n)`
b. `void Sort(MyType &a, MyType &b)`
c. `void Sort(MyType *x, int n)`

The function declarations: proto-types

04FunctionTemplate.cpp

```
template <typename MyType>
void PrintList(MyType *x, unsigned int n);

template <typename MyType>
void Sort(MyType &a, MyType &b);

template <typename MyType>
void Sort(MyType *list, unsigned int length);
```

Proto-types

The (bubble) `Sort` functions:

04FunctionTemplate.cpp

```
template <typename MyType>
void Sort(MyType &a, MyType &b) {
  if (a>b)
  {
    MyType tmp=a;
    a=b;
    b=tmp;
  }
}
```

This will work for any type for which > is defined, including for classes.

```
template <typename MyType>
void Sort(MyType *x, unsigned int n) {
  for (int i=n-1; i>1; i--)
    for (int k=0; k<i; k++)
      Sort(x[k], x[k+1]);
}
```

04FunctionTemplate.cpp

```
22  int main(void)
    {
24    int Numbers[8];
      char Letters[8];
26
      for (int i=0; i<8; i++)
28      Numbers[i]=rand()%40;

30    for (int i=0; i<8; i++)
        Letters[i]='A'+rand()%26;
```

(will give a "random" number between 0 & 39)

Random upper-case letter.

04FunctionTemplate.cpp

```
    std::cout << "Before sorting:" << std::endl;
34  std::cout << "Numbers: "; PrintList(Numbers, 8);
    std::cout << "Letters: "; PrintList(Letters, 8);
36
    Sort(Numbers, 8);
38  Sort(Letters, 8);

40  std::cout << "After sorting: " << std::endl;
    std::cout << "Numbers: "; PrintList(Numbers, 8);
42  std::cout << "Letters: "; PrintList(Letters, 8);
```

} using Sort in 2 different ways.

Typical output

```
Before sorting:
Numbers:   23    6   17   35   33   15   26   12
Letters:    B    H    C    D    A    R    Z    O
After sorting:
Numbers:    6   12   15   17   23   26   33   35
Letters:    A    B    C    D    H    O    R    Z
```

CS319 – Week 5
Week 5: Streams and files

END OF PART 6

Finished here at 4.50 (!).