CS319: Scientific Computing (with C++)

Week 5: Streams and files

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

Usual reminders...

	Mon	Tue	Wed	Thu	Fri
9 – 10		LECTURE	✗		
10 – 11		LAB			
11 – 12					
12 – 1					
1 – 2		LAB			
2 – 3					
3 – 4					
4 – 5			LECTURE		

1. We'll have recorded classes on **Tuesdays** at 9.00 and **Thursdays** at 16.00.
2. **Lab times: Tuesday 10.00-10:50, and 13.00-13.50**. You should try to attend at least one of these.
3. A short introduction to the lab will be recorded.
4. There will be no recorded class next Wednesday (St. Patrick's Day).

CS319 – Week 5
Week 5: Streams and files

Start of ...

PART 1: Review of classes

Part 1: Review of classes

class

In C++, we defined new classed with the `class` keyword.
An instance of the class is called an "*object*".
A `class` combines by data and functions.

Within a class, code and data may be either

- **Private**: accessible only to another part of that object, or
- **Public**: other parts of the program can access it.

Roughly,

- keep data elements `private`,
- make function elements `public`.

Part 1: Review of classes

The basic syntax for defining a class:

```
class class-name {
private:
    ...      // private functions and variables
public:
    ...      // public functions and variables
};
```

class-name becomes a new object type—one can now declare objects to be of type *class-name*.

This is only a declaration. Therefore,

- functions are not defined, though the prototype is given,
- variables are declared but are not initialised,
- the declaration block is delineated by { and }, and terminated with a semicolon.
- use *scope resolution operator* `::` to combine a class name and element/member name.

CONSTRUCTOR

A **Constructor** is a public member function of a class.

- It has the same name as the class.
- It's return type is not specified explicitly.
- It is executed whenever a new instance of that class is created.

Constructors may contain any code you like; but it is good practice to only use them for initialization and, especially **Dynamic memory allocation** (see Part 7 of Week 4).

CS319 – Week 5
Week 5: Streams and files

END OF PART 1

CS319 – Week 5
Week 5: Streams and files

Start of ...

PART 2: Destructors and Constructors

Part 2: Destructors & Constructors

Complementing the idea of a constructor is a **destructor**. This function is called

- for a local object – whenever it goes out of scope,
- for a global object – when the program ends.

The name of the destructor is the same as the class, but preceded by a tilde.

Recall the `MyStack` example from last week:

```
class MyStack {
private:
  char *contents;
  int top;
public:
  MyStack(void );
  ~MyStack(void );
  void push(char c);
  char pop();
};
```

```
MyStack::~MyStack()
{
  delete [] contents;
}
```

The example we had earlier of a constructor was particularly basic, not least because is its parameter list is `void`. More commonly, one passes arguments to the constructor that can be used, e.g.,

- to set the value of a data member;
- dynamically size an array using `new`.

However, one should still provide a default constructor (i.e., one with no arguments), or one with a default argument list.

```
class MyStack
{
private:
  char *contents;
  int top;
public:
  MyStack(void);
  MyStack(unsigned int MyStackSize);
  void push(char c);
  char pop(void );
};
```

```
MyStack::MyStack(void)
{
  top=0;
  contents = new char[MAX_STACK];
}

MyStack::MyStack(unsigned int StackSize)
{
  top=0;
  contents = new char[StackSize];
}
```

CS319 – Week 5
Week 5: Streams and files

END OF PART 2

CS319 – Week 5
Week 5: Streams and files

Start of ...

PART 3: I/O streams as objects

Part 3: I/O streams as objects `cout`, `cin`

`I/O` means "Input/Output. So far, we have taken input from the keyboard, typically using `cin`, and sent output to a terminal window, using `cout`.

These are examples of **streams**: flows of data to or from your program. Moreover, they are examples of **objects** in C++.

In this section, we'll study how to manipulate these streams in C++, including writing to and reading from files.

But first, some more information about `cout` and `cin`.

Part 3: I/O streams as objects `cout, cin`

The objects `cout` and `cin` are objects and are manipulated by their **methods**, i.e., public member functions and operators.

Methods:

- `width(int x)` – minimum number of characters for next output,
- `fill(char x)` – character used to fill with in the case that the width needs to be elongated to fill the minimum.
- `precision(int x)` – sets the number of significant digits for floating-point numbers.

Code – width

```
for (int i=65; i<123; i++)
{
  std::cout.width(8);
  std::cout << i;
  std::cout.width(3);
  std::cout << (char) i;
  if ( (i%5) == 4)
    std::cout << std::endl;
}
```

Output

```
      65  A      66  B      67  C ...
      70  F      71  G      72  H ...
      75  K      76  L      77  M ...
      80  P      81  Q      82  R ...
      85  U      86  V      87  W
      90  Z      91  [      92  \
      95  _      96  `      97  a
     100  d     101  e     102  f
     105  i     106  j     107  k
     110  n     111  o     112  p
     115  s     116  t     117  u
     120  x     121  y     122  z
```

Code – width, fill

```
std::cout.fill('0');
for (int i=0; i<8; i++)
{
  std::cout.width(6);
  std::cout << rand()%200000 <<std::endl;
}
```

Output

```
089383
130886
092777
036915
147793
038335
085386
160492
```

Code – precision

```
double Pi=3.1415926535;
for (int i=1; i<=10; i++)
{
  std::cout.precision(i);
  std::cout << "Pi (correct to "<< i << " digits) is "
            << Pi << std::endl;
}
```

Output

```
Pi (correct to 1 digits) is 3
Pi (correct to 2 digits) is 3.1
Pi (correct to 3 digits) is 3.14
Pi (correct to 4 digits) is 3.142
Pi (correct to 5 digits) is 3.1416
Pi (correct to 6 digits) is 3.14159
Pi (correct to 7 digits) is 3.141593
Pi (correct to 8 digits) is 3.1415927
Pi (correct to 9 digits) is 3.14159265
Pi (correct to 10 digits) is 3.141592654
```

Part 3: I/O streams as objects — manipulators

- `setw` – like `width`
- `left` – Left justifies output in field width. Used after `setw(n)`.
- `right` – right justify.
- `endl` – inserts a newline into the stream and calls flush.
- `flush` – forces an output stream to write any buffered characters
- `dec` – changes the output format of number to be in decimal format
- `oct` – octal format
- `hex` – hexadecimal format
- `showpoint` – show the decimal point and some zeros with whole numbers

Others: setprecision(n), fixed, scientific, boolalpha, noboolalpha, ...

Need to include `iomanip`

All of the C++ programs we have looked at so far took their input from the *standard input stream*: this was usually the keyboard.

Example:

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

Although, for example, the *standard input stream* can be redirected to a file, it is usually necessary to open a file **from within the program** and take the data from there.

The same is true for writing to a file.

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

CS319 – Week 5
Week 5: Streams and files

END OF PART 3

CS319 – Week 5
Week 5: Streams and files

Start of ...

PART 4 (i): Files

Part 4 (i): Files

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

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

Although the *standard input stream* can be redirected to be, for example, a file (easily done on a Mac and 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.

Part 4 (i): Files

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

```
#include <iostream>
#include <fstream>
#include <cstdlib>

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

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

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

Part 4 (i): Files

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

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

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

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

  return(0);
}
```

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**.

Part 4 (ii): Files

open a file

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())
```

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 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 );
}
```

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.

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.

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*

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)

CS319.pbm

```
P1
25 9
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
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
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
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
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".

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).

03FlipPBM.cpp

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

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

Part 5: Portable Bitmap Format (pbm)

03FlipPBM.cpp

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

Part 5: Portable Bitmap Format (pbm)

03FlipPBM.cpp

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

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

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

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

Part 5: Portable Bitmap Format (pbm)

03FlipPBM.cpp

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

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

Part 5: Portable Bitmap Format (pbm)

CS319 – Week 5
Week 5: Streams and files

END OF PART 5

CS319 – Week 5
Week 5: Streams and files

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 prototypes:

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);
```

The (bubble) `Sort` functions:

`04FunctionTemplate.cpp`

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

```cpp
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

```
int main(void )
{
  int Numbers[8];
  char Letters[8];

  for (int i=0; i<8; i++)
    Numbers[i]=rand()%40;

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

04FunctionTemplate.cpp

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

  Sort(Numbers, 8);
  Sort(Letters, 8);

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

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