CS319: Scientific Computing

Week 9: Strings; Files and Streams; A `Vector` class

Dr Niall Madden

9am and **4pm**, 06 March, 2024

CS319
CS319

Slides and examples: https://www.niallmadden.ie/2324-CS319

Outline

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()`

That is, using an ofstream object.

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.
```

beg = "begining".

seek a new location within the file.

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 );
}
```

'\n' is the New Line Character.

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

```
InFile >> c;
```

// as is done with std::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.

One of the complications of working with files, is knowing where to store input files so that your code can find them.

For some, IDEs, this is make additionally complicated by the fact that the compiled version of the program may not be in the same folder as the source code. So you have to work out where that is.

One way that can help, is change the `int main(void)` line to

```
int main(int argc, char * argv[])
{
  std::cout << "This program is running as " << argv[0];
  std::cout << "\nDownload the input file to the same folder";
  std::cout << std::endl;
```

Alternatively, you can try opening a `ofstream` file with a vary particular name, and then search for it.

If using an online compiler, you'll need one that allows multiple files, such as
https://www.jdoodle.com/online-compiler-c++-ide

Portable Bitmap Format (pbm)

Some self-study

We won't go through this section in class: please review in your own time.

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`

Portable Bitmap Format (pbm)

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.

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".

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());
```

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;
```

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;
```

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);
```

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. (called Methods).

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`.

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.
- *scope resolution operator*, `::` , used in function definition.

Review of classes

- A **Constructor** is a public method of a class, that 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.
- A **destructor** is a method that is called on an object whenever it goes out of scope. The name of the destructor is the same as the class, but preceded by a tilde.

Vectors and Matrices

This is a course in Scientific Computing.

Many advanced and general problems in Scientific Computing are based around **vectors** and **matrices**. So one of our goals is to implement C++ classes for such structures, along with standard operations such as matrix-vector multiplication.

Along the way, we'll learn about

- operator overloading;
- `friend` functions and the `this` pointer;
- static variables.
- and much more

Our first step will be to study some problems and applications so that, before we design any classes or algorithms, we'll know what we will use them for. These problems include:

1. Basic analysis of matrices, for example with applications to image processing, graphs and networks.
2. Solution of linear systems of equations, for example with applications to data fitting;
3. Estimation of (certain) eigenvalues, for example with applications to search engine analysis.

Of these problems, probably the most ubiquitous is the solution of (large) systems of simultaneous equations.

That is, we want to solve a linear system of 3 equations in ~~4~~ 3 unknowns: *find* x_1, x_2, x_3, *such that*

$$
\begin{aligned}
3x_1 + 2x_2 + 4x_3 &= 19 \\
x_1 + 2x_2 + 3x_3 &= 14 \\
5x_1 + 1x_2 + 6x_3 &= 25
\end{aligned}
$$

This can be expressed as a **matrix-vector equation:**

$$
\underbrace{\begin{pmatrix} 3 & 2 & 4 \\ 1 & 2 & 3 \\ 5 & 1 & 6 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 19 \\ 14 \\ 25 \end{pmatrix}}_{b}
$$

More generally, the linear system of N equations in N unknowns: *find* $x_1, x_2, \ldots, x_N$, *such that*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

This, as a **matrix-vector equation** is:

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1N} \\ a_{21} & a_{22} & \ldots & a_{2N} \\ \vdots & & \ddots & \vdots \\ a_{N1} & a_{N2} & \ldots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

So, to proceed, we need to be able to represent **vectors** and **matrices** in our codes.

Our first focus will be on defining a class of vectors. Intuitively, we know it needs the following components:

At the very least we need

Data:

Number of element

The value of elements.

Methods:

Check values

Set values.

Later:

1. multiply vector by scalar
2. Add vectors
3. Compute the norm of a vector.

Due to the level of detail in the matrix and vector classes, the following example is divided into three source files:

1. `Vector.h`, the header file which contains the class definition. Include this header file in another source file with:
   `#include "Vector.h"`
   Note that this is **not** `<Vector.h>`
2. `Vector.cpp`, which includes the code for the methods in the *`Vector`* class;
3. `03TestVector.cpp`, a test stub.

In whatever compiler you are using, you'll need to create a project that contains all the files. (Ask Niall for help if needed).

See `Vector.h` for more details

```
// File: Vector.h (Version W07.1)
// Author: Niall Madden ¡Niall.Madden@UniversityOfGalway.ie¿
// Date: Week 9 of 2324-CS319
// What: Header file for vector class
// See also: Vector.cpp and 03TestVector.cpp
class Vector {
private:
  double *entries;  // array for storing data.
  unsigned int N;   // number of entries in the vector.
public:
  Vector(unsigned int Size=2);
  ~Vector(void);

  unsigned int size(void) {return N;};
  double geti(unsigned int i);
  void seti(unsigned int i, double x);  set v[i] = x

  void print(void);
  double norm(void);  // Compute the 2-norm of a vector
  void zero(void);  // Set entries of vector to zero.
};
```

note the use of the "scope resolution operator"

Vector.cpp

```
Vector::Vector(unsigned int Size)
{
  N = Size;
  entries = new double[Size];
}

Vector::~Vector()
{
  delete [] entries;
}

void Vector::seti(unsigned int i, double x)
{
  if (i<N)
    entries[i]=x;
  else
    std::cerr << "Vector::seti(): Index out of bounds."
              << std::endl;
}
```

Constructor

destructor.

Vector.cpp continued

```
double Vector::geti(unsigned int i)
{
  if (i<N)
    return(entries[i]);
  else {
    std::cerr << "Vector::geti(): Index out of bounds."
              << std::endl;
    return(0);
  }
}

void Vector::print(void)
{
  for (unsigned int i=0; i<N; i++)
    std::cout << "[" << entries[i] << "]" << std::endl;
}
```

A vector class

Vector.cpp continued

```cpp
double Vector::norm(void)
{
  double x=0;
  for (unsigned int i=0; i<N; i++)
    x+=entries[i]*entries[i];
  return (sqrt(x));
}

void Vector::zero(void)
{
  for (unsigned int i=0; i<N; i++)
    entries[i]=0;
}
```

Computes
$\sqrt{x_1^2 + x_2^2 + \cdots x_N^2}$

Here is a simple implementation of a function that computes
$c = \alpha a + \beta b$

See `03TestVector.cpp` for more details

```
// c = alpha*a + beta*b where a,b are vectors; alpha, beta are scalars
void VecAdd (vector &c, vector &a, vector &b,
        double alpha, double beta)
{
  unsigned int N;
  N = a.size();

  if ( (N != b.size()) )
    std::cerr << "dimension mismatch in VecAdd " << std::endl;
  else
  {
    for (unsigned int i=0; i<N; i++)
      c.seti(i, alpha*a.geti(i)+beta*b.geti(i) );
  }
}
```

c is output .

$c_i = \alpha a_i + \beta b_i$

Solving Linear Systems

We now move towards learning about **matrices**. When implementing the class, we will learn about

- operator overloading;
- `friend` functions and the `this` pointer;
- static variables.
- and much more

One of the most ubiquitous problems in scientific computing is the solution of (large) systems of simultaneous equations. That is, we want to solve a linear system of N equations in N unknowns: *find* $x_1, x_2, \dots, x_N$, *such that*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

There are several classic approaches:

1. Gaussian Elimination;
2. Related: LU- and Cholesky factorisation;
3. Stationary Iterative schemes such as **Jacobi's method**, **Gauss-Seidel** and Successive Over Relaxation (SOR);
4. Krylov subspace methods, of which Conjugate Gradients is the best known;
5. Enhancements of the Methods 3 and 4, using preconditioning with, for example, MultiGrid and Incomplete LU-factorisation.

Of the approaches listed above, Jacobi's is by far the simplest to implement, and so is the one we will study first.

See annotated slides.

(For much more details, see

- the notes from Lab 7: `https://www.niallmadden.ie/2324-CS319/lab7/CS319-lab7.pdf`
- an implementation from Lab 7 that does **not** use classes/objects: `https://www.niallmadden.ie/2324-CS319/lab7/Jacobi-Lab7.cpp`

Idea: To solve

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

(For much more details, see

- the notes from Lab 7: `https://www.niallmadden.ie/2324-CS319/lab7/CS319-lab7.pdf`
- an implementation from Lab 7 that does **not** use classes/objects: `https://www.niallmadden.ie/2324-CS319/lab7/Jacobi-Lab7.cpp`

Guess x_2 & x_3 and set

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3).$$

If the guess for x_2 & x_3 is correct, so too is x_1

If not we find x_1 is still a "good" estimate. Now repeat for x_2 & then x_3.