CS319: Scientific Computing

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

Dr Niall Madden

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

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

Outline

News and Updates

- Lab 4: grades for Lab 4 have been posted.
- Class test: grades have been posted.
- Lab 6: Deadline pushed out to 17:00, Thursday 7 March.
- Project topics: deadline pushed out to 17:00, Wed 6 March (today!). Anything submitted after that will score zero.
- Project proposals due **17:00, Tuesday 12 March**. See Slide 9 of niallmadden.ie/2324-CS319/2324-CS319-Projects.pdf
- Submit the proposal at `https://universityofgalway.instructure.com/courses/12359/assignments/65516`

Last week we learned that

- A **class** is a general form of data type that we can create;
- An **object** is an instance of a particular class. E.g,
- A **method** is a member of a class that is a function. E.g.,

Before we continue with writing our own classes, we can now visit some important related topics in C++:

- `string`s
- **input and output streams**
- **files**.

A **string** is a collection of characters representing, for example, a word or a sentence.

In C++, a `char` array can be used to store a string. That approach is called a "C string", since it is inherited from an older language, C.

Such "C strings" are no so easy to work with, so C++ provides its one `string` class.

The `string` class is one that is "built-in" to the C++ language, and can be accessed once the `string` header file is included.

We have used it before, but have not thought of it as a class.

Since it is a class, it has some methods, including:

- `length()` and `size()` which both return the number of characters in the string;
- `substr(i,l)` with returns a substring of length `l`, starting at position `i`.
- `find()` which finds the first occurence on one substring in another.
- `c_str()` return the "C string" version.

Example

Write a short C++ program that defines a `string` containing a sentence, and then extract the first word as another `string`.

00substring.cpp

```
#include <iostream>
#include <string>

int main(void)
{
  std::string
    sentence="Ada Lovelace was the first programmer",
    first_word;
  int space_loc = sentence.find(" ");      // Find first space
  first_word = sentence.substr(0,space_loc); // extract substring

  std::cout << "sentence is: " << sentence << std::endl;
  std::cout << "first word is: **" << first_word << "**\n";
  return(0);
}
```

With numbers, we are used to working with special functions called **operators**, which are usually represented by a mathematical symbol, such as `+`, `-`, `=`, `*`, `/`, etc.

When writing our own `class`, we can overload some of these (more about the details later).

The `string` class overloads several operators:

- Assignment: `=`
- Relational: `==`, `>`, `<`, etc;
- Arithmetic: `+`, `+=`

01string-operators.cpp

```
#include <iostream>
#include <string>

int main(void)
{
  std::string name[3], // array of names
    long_name="";
  name[0]="Augusta";
  name[1]="Ada";
  name[2]="King";

  long_name = name[0] + " " + name[1] + " " + name[2];

  std::cout << "long_name: " << long_name << std::endl;
  return(0);
}
```

`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 fact `cout` and `cin` are **objects** and are manipulated by their **methods**, i.e., public member functions and operators. (We saw this in Week 3)

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, 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<=8; 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
```

- `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`

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.

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 with a simple example.

02CountChars.cpp

(i) This program opens an input file called `CPlusPlusTerms.txt`

(ii) It opens an output file called `Output.txt`

(iii) It counts the number of characters in the input file.

(iv) It writes that result to the output file.

Download the input file from `https://www.niallmadden.ie/2324-CS319`. Save it to the folder containing the executable that you compile.

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.

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 );
```

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);
}
```

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 `char`acters. 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 file name is stored as a **"C string"**.

If we want to use C++ style strings, use the `c_str()` method.
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.

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.

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".

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());
```

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;
```

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.

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 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:**

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.

A vector `class`

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

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;
  unsigned int N;
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);

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

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;
}
```

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;
}
```

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;
}
```

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) );
  }
}
```

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* $x_1, x_2, \ldots, x_N$, *such that*

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

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