

**CS319: Scientific Computing**

**Week 10: Vector and Matrix classes**

Dr Niall Madden

**9am** and **4pm**, 13 March, 2024

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

We now want to see another way of accessing the implicitly passed argument. First, though, we need to learn a little more about pointers, and introduce a new piece of C++ notation.

Recall that if, for example, `x` is a `double` and `y` is a pointer to `double`, we can set `y=&x`. So now `y` stores the memory address of `x`. We then access the contents of that address using `*y`.

Now suppose that we have an object of type `Vector` called `v`, and a *pointer to `vector`*, `w`. That is, we have defined

```
Vector v;  
Vector *w;
```

Then we can set `w=&v`. Now accessing the member `N` using `v.N`, will be the same as accessing it as `(*w).N`.

It is important to realise that  $(*w).N$  is **not** the same as  $*w.N$ .

C++ provides a new operator for this situation:  $w \rightarrow N$ , which is equivalent to  $(*w).N$ .

Finished here at 9.50

When writing code for functions, and especially overloaded operators, it can be useful to **explicitly** access the implicitly passed object.

That is done using the `this` pointer, which is a pointer to the object itself.

.....

As we've just noted, since `this` is a pointer, its members are accessed using either `(*this).N` or `this->N`.

We often use the `this` pointer when a function must return the address of the argument that was passed to it. This is the case of the assignment operator.

*a = b*

See `Vector10.cpp` for more details

```
100 // Overload the = operator.
    Vector &Vector::operator=(const Vector &b)
102 {
    if (this == &b)
        return(*this); // Taking care for self-assignment

    delete [] entries; // In case memory was already allocated

    N = b.N;
    entries = new double[b.N];
    for (unsigned int i=0; i<N; i++)
        entries[i] = b.entries[i];
112 return(*this);
}
```

*b is explicit.*

*a is implicit - it is pointed to by 'this'*

# Unary Operators

So far we have discussed just the **binary** operator, `+`. By “**binary**”, we mean it takes **two** arguments.

But many C++ operators are **unary**: they take only one argument; examples include `++` and `--`.

For our `Vector` class, we want to overload the `-` (minus) operator. Note that this can be used in two ways:

- ①  $c = -a$  (unary). — set  $c_i = -a_i$
- ②  $c = a - b$  (binary) — set  $c_i = (a_i - b_i)$

In the first case here, “minus” is an example of a **prefix** operator. (See “Extras” for example of overloading **postfix** operators, like `a++`, which are a little more complicated).

After that we will then define the binary minus operator, by using addition and unary minus.

For the unary “minus” operator, when we write “-a” the object **a** is passed *implicitly*. This is a little different from previous cases, where the object passed implicitly is to the left of the operator.

See Vector10.cpp for more details

```
108 // Overload the unary minus (-) operator. As in b=-a;
110 Vector Vector::operator-(void)
112 {
114     Vector b(N); // Make b the size of a
115     for (unsigned int i=0; i<N; i++)
116         b.entries[i] = -entries[i];
117     return(b);
118 }
```

← implicit .

arg list is “void” because unary “-” takes just one argument, which is implicit.

And now that we have defined this operator, we can define the **binary** minus operator. Now this time when we write “ $a-b$ ”, it is the *left* argument that is implicit.

See Vector10.cpp for more details

```
118 // Overload the binary minus (-) operator. As in  $c=a-b$ 
// This implementation reuses the unary minus (-) operator
Vector Vector::operator-(Vector b)
120 {
    Vector c(N); // Make b the size of a
122     if (N != b.N)
        std::cerr << "Vector:: operator- : dimension mismatch!"
124                 << std::endl;
    else
126         c = *this + (-b);
    return(c);
128 }
```

Use that  $a-b$  is  $a + (-b)$   
defined earlier



# friend functions

In all the examples that we have seen so far, the only functions that may access private data belonging to an object has been a member function/method of that object.

If we need a function that does not belong to the class to be able to access `private` elements, it can be designated a `friend` of the class.

For non-operator functions, there is nothing that complicated about `friends`. However, care must be taken when overloading operators as `friends`.

In particular:

- ▶ All arguments are passed explicitly to `friend` functions/operators.
- ▶ Certain operators, particularly the **insertion/put-to** `<<` and **extraction/get-from** `>>` operators can only be overloaded as friends.

In last week's version of the **Vector** class, we could output its elements using the **print()** method. E.g.:

```
Vector v;  
v.zero()  
std::cout << "v  has values  ";  
v.print();
```

But it would be much more convenient just to do

```
std::cout << "v  has values  " << v;
```

But the **insertion** operator was not defined for our class.

We can fix that, by overloading it. However, the **<<** operator belongs to **std::cout**, not to **Vector**. So it cannot access its **entries** member.

Here is how we resolve this...

We add the following line to the definition of the **Vector** class.

```
1 friend std::ostream &operator<<(std::ostream &, Vector &v);
```

Output stream: cout is an object of this type.

And then we define:

```
1 std::ostream &operator<<(std::ostream &output, Vector &v)
  {
3   output << "[";
   for (unsigned int i=0; i<v.size()-1; i++)
5     output << v.entries[i] << ",";
   output << v.entries[v.size()-1] << "]";

   return(output);
9 }
```

Usually, is cout

Now we can display a vector using **std::cout** directly.

As a friend function, all arguments are explicit.

# Preprocessor Directives

Our next step is to define a `Matrix` class, and overload some of the associated operators. One of those is the multiplication (“times”) operator `*` for matrix-vector multiplication.

With those done, we can think about overloading the multiplication operator for `Matrix-Vector` multiplication.

This introduces a few small new complications:

- ▶ the return type is different from the class type;
- ▶ if we use multiple source files, how do we know where exactly to place the `#include` directives?

So, before we can proceed, we need to take a short detour to consider **preprocessor** directives.

The preprocessor in C++ is a hang-over over from early versions of C. Originally, that language did not have a construct for defining constants and including header files. To get around this, an early version of C introduced the **preprocessor**. This is a program that

- ▶ reads and modifies your source code by checking for any lines that being with a hash symbol (**#**);
- ▶ carries out any operations required by these lines;
- ▶ forms a new source code that is then compiled.

We usually don't get to see this new file, though you can view it by compiling with certain options (with **g++**, this is **-E**).

The preprocessor is *separate* from the compiler, and has its own syntax.

The simplest preprocessor directive is `#define`. This is used for defining global constants, and doing a simple search-and-replace. For example,

```
#define SIZE 10
```

will find every instance of the word (well, token, really) *SIZE* and replaces it with *10*.

In general, this use of the `#define` directive to define identifiers to be used like “global variables” is not very good practice. However, it can be very useful as a way of checking if a piece of code has already been compiled.

The most familiar preprocessor is `#include`, e.g.,

```
#include <iostream>
#include "Vector10.h"
```

This tells the preprocessor to take the named file(s) and insert them into the current file.

If the name is contained in angle brackets, as in `<iostream>`, this means the preprocessor will look in “the usual place” – where the compiler is installed on your system.

If the named file is in quotes, it looks in the current directory/folder, or in the specified location.

Finally, we have **conditional compilation**.

Suppose we want to write a member function for the *Matrix* class that involves the *Vector* class.

So we need to include *Vector10.h* in *Matrix10.h*. But then if our main source file includes both *Matrix10.h* and *Vector10.h* we could end up defining it twice.

To get around this we use *conditional compilation*.

In the files we can have such lines as the following in *Vector10.h*

```
#ifndef _VECTOR_H_INCLUDED
#define _VECTOR_H_INCLUDED
// stuff goes here
#endif
```

*#ifndef means  
"if not defined".*



# A matrix class

We now write a `class` implementation for a `matrix`, along with the associated functions.

We'll first consider how the matrix data is stored. The most natural approach might seem to be to construct a two dimensional array. This can be done as follows:

```
double **entries = new double *[N];  
for (int i=0; i<N; i++)  
    entries[i] = new double N;
```

A simpler, faster approach is to store the  $N^2$  entries of the matrix in a single, one-dimensional, array of length  $N^2$ , and then take care how the access is done:

$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \rightarrow [a \ b \ c \ d \ e \ f \ g \ h \ i]$

# A matrix class

## Matrix10.h

```
12 class Matrix {
13 private:
14     double *entries; // note this is one-dimensional.
15     unsigned int N;
16 public:
17     Matrix (unsigned int Size=2);
18     Matrix (const Matrix &m); // Copy constructor
19     ~Matrix(void);
20
21     Matrix &operator=(const Matrix &B); // assignment operator
22
23     unsigned int size(void) {return (N);};
24     double getij(unsigned int i, unsigned int j);
25     void setij(unsigned int i, unsigned int j, double x);
26
27     Vector operator*(Vector u); // Define later!
28     void print(void);
29 };
```

# A matrix class

First we'll look at the code for the constructor, to verify that the data is stored just as an 1D array:

from `Matrix10.cpp`

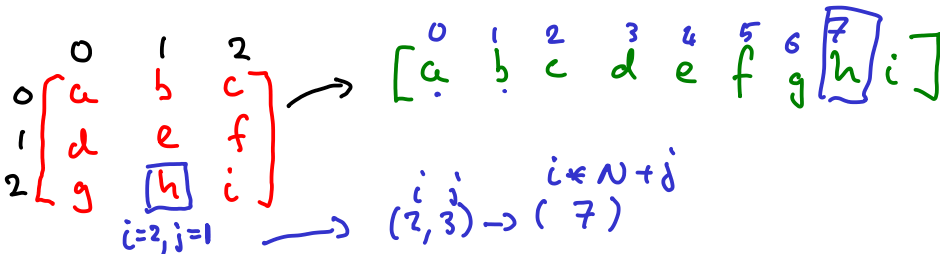
```
12 // Basic constructor. See below for copy constructor.  
12 Matrix::Matrix (unsigned int Size)  
13 {  
14     N = Size;  
15     entries = new double [N*N];  
16 }
```

# A matrix class

Next we'll look at the `setij()` member, to see how indexing works.

from `Matrix10.cpp`

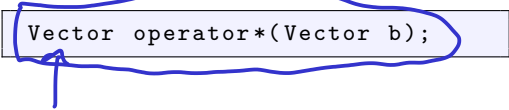
```
24 void Matrix::setij (unsigned int i, unsigned int j, double x)
26 {
    if (i<N && j<N)
26     entries[i*N+j]=x;
    else
28     std::cerr << "Matrix::setij(): Index out of bounds.\n";
}
```



Other components of the `Matrix` class are similar to the corresponding functions for the `Vector` class, such as the assignment operator, and the copy constructor.

So, we'll just focus on overloading the `operator*` for multiplication of a vector by a matrix:  $c = A * b$ , where  $A$  is an  $N \times N$  matrix, and  $c$  and  $b$  are vectors with  $N$  entries.

Since the left operand is a matrix, we'll make this operator a member of the `Matrix` class; its header has the line:



```
Vector operator*(Vector b);
```

Return type

The code from `Matrix10.cpp` is given below.

```
84 // Overload the operator multiplication (*) for a Matrix-Vector
// product. Matrix is passed implicitly as "this", the Vector is
86 // passed explicitly. Will return v=(this)*u
Vector Matrix::operator*(Vector u)
88 {
    Vector v(N); // v = A*u, where A is the implicitly passed Matrix
    90 if (N != u.size())
        std::cerr << "Error: Matrix::operator* - dimension mismatch"
        << std::endl;
    92 else
    94     for (unsigned int i=0; i<N; i++)
    96     {
        double x=0;
        for (unsigned int j=0; j<N; j++)
        98         x += entries[i*N+j]*u.geti(j);
        v.seti(i,x);
    100     }
    102 return(v);
}
```

## Preview of Lab 8

In Lab 7 you had to write a function that implemented the `Jacobi` method. For Lab 8, you'll rewrite it using the `Vector` and `Matrix` classes.

In addition, you'll implement the Gauss-Seidel method. For that, you'll have to implement the “backslash” operator, for solving  $Ax = y$ , where  $A$  is triangular.