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

Outline

Extras

See **"extras"** section of today's lectures for more examples of classes and overloading (points, dates, complex numbers); Code for these is in the `Week10/extras/` folder on the repository/website.

These slides do not include all issues concerning operator overloading. Among the topics omitted are:

- ▶ overloading the unary `++` and `--` operators. There are complications because they work in both prefix and postfix form.
- ▶ Overloading the ternary operator: `? :`
- ▶ **Important:** overloading the `[]` operator.

News and Updates

▶ Lab 6: grades will be posted soon (if not already).

▶ Project proposals. Deadline for draft versions was **17:00, Tuesday 12 March**. **Check canvas for feedback**. Should discuss further in labs.

▶ next step is to schedule presentations in Week 12. Indicate your availability here:
https://forms.office.com/e/bKBzktxdPs

Overview

▶ This week is all about extending the `Vector` class from Week 9. Since the code for the header and definition are different from last week's version, the new files have different names: `Vector10.h`, `Vector10.cpp`.

▶ I've also provided a program to test these, called `01TestVectorOperators.cpp`. To run that, you'll need a project that contains all three files. If you are using an online compiler, I suggest trying `online-cpp.com`. For this specific example, try `https://www.online-cpp.com/znZjLKN8hl`

▶ An early version of the notes for Week 9 included a simple `Matrix` class. I've removed that, and instead developed the `Vector` class fully. (For reasons...)

In Week 9, was saw how, for example, the + operator was "overloaded" to allow us to "add" (i.e., concatenate) two strings. We want to see how overload operators for classes that we write so that, for example, we can use the + operator to add two vectors.

That is, we want to study **Operator Overloading**. But to get this to work, we need to study **copy constructor**s.

This is a technical area of C++ programming, but is unavoidable.

As we already know, **constructor** is a method associated with a class that is called automatically whenever an object of that class is declared.

But there are time when objects are *implicitly* declared, such as when passed (by value) to a function.

Since this will happen often, we need to write special constuctors to handle it.

Last week we defined a class for vectors:

- It stores a vector of *N* doubles in a dynamically assigned array called *entries*;
- The constructor takes care of the memory allocation.

```cpp
 1  // From Vector.h (Week 9)
    class Vector {
 3  private:
      double *entries;
 5    unsigned int N;
    public:
 7    Vector (unsigned int Size=2);
      ~Vector(void);

      unsigned int size(void) {return N;};
11    double geti (unsigned int i);
      void seti (unsigned int i, double x);
13    // print(), zero() and norm() not shown
    };

    // Code for the constructor from Vector.cpp
17  Vector::Vector (unsigned int Size) {
      N = Size;
19    entries = new double[Size];
    }
```

We then wrote some functions that manipulate vectors, such as
`AddVec` in `Week09/03TestVector.cpp`

```
void VecAdd (Vector &c, Vector &a, Vector &b,
             double alpha=1.0, double beta=1.0);
```

Note that the `Vector` arguments are passed by reference...

What would happen if we tried the following, seemingly reasonable
piece of code?

```
Vector x(4);
x.zero(); // sets entries of a all to 0
Vector y=x; // should define a new vector, with a copy of x
```

This will cause problems for the following reasons:

The solve this problem, we should define our own **copy constructor**. A **copy constructor** is used to make an exact copy of an existing object. Therefore, it takes a single parameter: the address of the object to copy. For example:

See Vector10.cpp for more details

```
20  // copy constructor
    Vector::Vector (const Vector &old_Vector)
22  {
      N = old_Vector.N;
24    entries = new double[N];
      for (unsigned int i=0; i<N; i++)
26      entries[i] = old_Vector.entries[i];
    }
```

The **copy constructor** can be called two ways:

(a) *explicitly*, .e.g,

```
Vector V(2);
V.seti(0)=1.0; V.seti(1)=2.0;
Vector W(V); // W is a copy V
```

(b) *implicitly*, when ever an object is passed by value to a function. If we have not defined our own copy constructor, the default one is used, which usually causes trouble.

In this section, we'll study **"Operator overloading"** .

Our main goal is to overload the addition (+) and subtraction (−) operators for vectors.

Last week, we wrote a function to add two `Vector`s: `AddVec`.

It is called as `AddVec(c,a,b)`, and adds the contents of vectors a and b, and stores the result in c.

It would be much more natural redefine the standard **addition** and **assignment** operators so that we could just write `c=a+b`. This is called **operator overloading**.

To overload an operator we create an **operator function** – usually as a member of the class. (It is also possible to declare an operator function to be a `friend` of a class – it is not a member but does have access to private members of the class. More about `friend`s later).

The general form of the operator function is:

```
return-type  class-name::operator#(arguments)
{
    ⋮    // operations to be performed.
};
```

return-type of a operator is usually the class for which it is defined, but it can be any type.

Note that we have a new key-word: `operator`.

The operator being overloaded is substituted for the `#` symbol

Almost all C++ operators can be overloaded:

```
+     -     *     /     %     ^     &     |     ~         !
=     <     >     +=    -=    *=    /=    %=    ^=      & =
|=    <<    >>    >>=   <<=   ==    !=    <=    >=      &&
||    ++    --    ->*   ,     ->    []    ()   new   delete
```

but not   .   ::   .*   ?

- ▶ Operator precedence cannot be changed: `*` is still evaluated before `+`
- ▶ The number of arguments that the operator takes cannot be changed, e.g., the `++` operator will still take a single argument, and the `/` operator will still take two.
- ▶ The original meaning of an operator is not changed; its functionality is extended. It follows from this that operator overloading is always relative to a user-defined type (in our examples, a `class`), and not a built-in type such as `int` or `char`.
- ▶ Operator overloading is always relative to a user-defined type (in our examples, a `class`).
- ▶ The assignment operator, `=`, is automatically overloaded, but in a way that usually fails except for very simple classes.

We are free to have the overloaded operator perform any operation we wish, but it is good practice to relate it to a task based on the traditional meaning of the operator. E.g., if we wanted to use an operator to add two matrices, it makes more sense to use `+` as the operator rather than, say, `*`.

We will concentrate mainly on binary operators, but later we will also look at overloading the unary "minus" operator.

. . . .

For our first example, we'll see how to overload `operator+` to add two objects from our `vector` class.

First we'll add the declaration of the operator to the class
definition in the header file, `Vector10.h`:

```
vector operator+(vector b);
```

Then to `Vector10.cpp`, we add the code

See `Vector10.cpp` for more details

```
     // Overload the + operator.
 96  Vector Vector::operator+(Vector b)
     {
 98    Vector c(N);  // Make c the size of a
       if (N != b.N)
100      std::cerr << "vector::+ : cant add two vectors of different si
                   << std::endl;
102    else
         for (unsigned int i=0; i<N; i++)
104        c.entries[i] = entries[i] + b.entries[i];
       return(c);
106  }
```

First thing to notice is that, although + is a binary operator, it seems to take only one argument. This is because, when we call the operator, `c = a + b`   then `a` is passed **implicitly** to the function and `b` is passed **explicitly**.
Therefore, for example, `a.N` is known to the function simply as `N`.

The temporary object `c` is used inside the object to store the result. It is this object that is returned. Neither `a` or `b` are modified.

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->N`, which is equivalent to `(*w).N`.

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.

See `Vector10.cpp` for more details

```cpp
      // Overload the = operator.
100   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;
108     entries = new double[b.N];
        for (unsigned int i=0; i<N; i++)
110       entries[i] = b.entries[i];

112     return(*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:

- ▶ $c = -a$ (unary).
- ▶ $c = a - b$ (binary)

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;
     Vector Vector::operator-(void)
110  {
       Vector b(N);  // Make b the size of a
112    for (unsigned int i=0; i<N; i++)
         b.entries[i] = -entries[i];
114    return(b);
     }
```

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

```
    // Overload the binary minus (-) operator. As in c=a-b
118 // 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 }
```

`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 below 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 `friend`s. However, care must be taken when overloading operators as `friend`s.

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 convinient 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);
```

And the 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 }
```

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

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.

Preprocessor Directives

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
```

A matrix `class`

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

We'll first consider how the matrix is 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:

A matrix class

Matrix10.h

```
12  class Matrix {
    private:
14    double *entries;
      unsigned int N;
16  public:
      Matrix (unsigned int Size=2);
18    Matrix (const Matrix &m);  // Copy constructor
      ~Matrix(void);

      Matrix &operator=(const Matrix &B);  // assignment operator

      unsigned int size(void) {return (N);};
24    double getij (unsigned int i, unsigned int j);
      void setij (unsigned int i, unsigned int j, double x);

      Vector operator*(Vector u);  // Define later!
28    void print(void);
    };
```

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`

```cpp
   // Basic constructor. See below for copy constructor.
12 Matrix::Matrix (unsigned int Size)
   {
14   N = Size;
     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`

```
     void Matrix::setij (unsigned int i, unsigned int j, double x)
24   {
       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);
```

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"
92                   << std::endl;
       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      }
       return(v);
102 }
```

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 to implement the "backslash" operator, for solving $Ax = y$, where A is triangular.