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.

```
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,
2             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:

Calling y=x "copies" x to y.
That is, it sets y.N to be x.N (which is 4).
Also, it sets y.entries to be x.entries
That is - they now store the same memory address. So, eg, setting x.entries[2] = -20, would set y.entries[2] = -20.

However, the situation is even worse!

If at a later point we call the destructor for y, then:

```
delete y.entries [];
```

So x.entries is deleted too!

Furter, when x's destructor is called we get a "double delete" error.

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];
   }
```

} Copying values.

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 `+` : That is: x+y*z always means x+(y*z)
- 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.

Also: <

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

When we call x+y, then in the code below: x is implicit, e.g. 'N' means x.N

See Vector10.cpp for more details

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

Any y is explicit: b.N means y.N.

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`.

Finished here at 9.50