Annotated slides

CS319: Scientific Computing (with C++)

Niall Madden (Niall.Madden@NUIGalway.ie)

Week 8: Linear Systems, and Operator Overloading

9am, 30 March, and 4pm, 31 March, 2021

1. Part 1: Solving Linear Systems (again)
   - Jacobi's method
   - Implementation
2. Part 2: A matrix class
   - MatVec
3. Part 3: Coding Jacobi's method
4. Part 4: Copy Constructors
   - A new constructor
5. Part 5: Operator Overloading
   - Eg 1: Adding two vectors
6. Part 6: The ->, this, and = operators
   - The -> operator
   - The this pointer
   - Overloading =

Tuesday

Wed

See **"extras"** section of today's lectures for more examples of classes and overloading (points, dates, complex numbers); Code for these is in the `Week08/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.

Usual reminders...

	Mon	Tue	Wed	Thu	Fri
9 – 10		LECTURE	✗		
10 – 11		LAB			
11 – 12					
12 – 1					
1 – 2		LAB			
2 – 3					
3 – 4					
4 – 5			LECTURE		

1. Two recorded classes this week: Tuesday at 09.00, and Wednesday at 16.00.
2. **Lab times: Tuesday 10.00-10:50, and 13.00-13.50**. You should try to attend at least one of these.

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

Start of ...

PART 1: Solving Linear Systems (again)

This continues from where we left off in Week 7

Continuing from Part 4 of Week 7.

Our eventual goal is the solve systems of N simultaneous equations in N unknowns: *find* $x_1, x_2, \dots, x_N$, *such that*

Eg

$6x_1 + 3x_2 - x_3 = 10$

$2x_1 + 7x_2 + x_3 = 1$

$-x_1 - x_2 + 4x_3 = -5$

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

We expressed this as a matrix-vector equation: *Find* $\mathbf{x}$ *such that*

$$A\mathbf{x} = \mathbf{b},$$

where A *is a* $N \times N$ *matrix, and* $\mathbf{b}$ *and* $\mathbf{x}$ *are (column) vector with* N *entries.*

We could do this with **Gaussian Elimination** (or LU-factorization, etc). But instead we use a method that is easier to program: *Jacobi's method.*

$$\begin{pmatrix} 6 & 3 & -1 \\ 2 & 7 & 1 \\ -1 & -1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 1 \\ -5 \end{pmatrix}$$

The idea is to choose an initial "guess", which call $\mathbf{x}^{(0)}$.

The we try to compute an improved guess, called call $\mathbf{x}^{(1)}$.

And we improve that again, to get $\mathbf{x}^{(2)}$.

Eventually, we have a sequence of estimates

$$\{x^{(0)}, x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(k)}, \dots\}$$

If could do this an infinite number of times, then

$$\text{as } k \to \infty, \text{ we get } \mathbf{x}^{(k)} \to \mathbf{x}.$$

But in practice, we just iterate until $\mathbf{x}^{(k)}$ is "close enough" to $\mathbf{x}$.

$$x^{(0)} = \begin{pmatrix} x_1^{(0)} \\ x_2^{(0)} \\ x_3^{(0)} \end{pmatrix}$$

The algorithm (i.e., the method of "improving" the $\mathbf{x}^{(k)}$) comes from the observation that, since (for example)

The 1^{st} Eqn $\longrightarrow$ $$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N = b_1,$$ Rearranged

then

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

So we can be optimistic that if $x_2^{(k)}$, $x_3^{(k)}$, ..., $x_N^{(k)}$ are a good estimates for x_2, x_2, ..., x_N, then

$$x_1^{\boxed{(k+1)}} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)})$$

will be an even better one for x_1.

Here $k = 0, 1, 2, 3, 4, \ldots$

Applying the same idea to the rest of the equations, we get

$$
\begin{aligned}
x_1^{(k+1)} &= \frac{1}{a_{11}}\big(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}\big)\\
x_2^{(k+1)} &= \frac{1}{a_{22}}\big(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)}\big)\\
&\vdots\\
x_N^{(k+1)} &= \frac{1}{a_{NN}}\big(b_N - a_{N,1}x_1^{(k)} - \cdots - a_{N,N-1}x_{N-1}^{(k)}\big)
\end{aligned}
$$

This can be programmed with two (or so) nested `for` loops (which is the topic of Lab 6). But it can also be expressed in a simple way, using matrices and vectors.

Example

See video or annotated slides

$$6x_1 + 3x_2 - x_3 = 10$$
$$2x_1 + 7x_2 + x_3 = 1$$
$$-x_1 - x_2 + 4x_3 = -5$$

Take $x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

So
$$x_1^{(1)} = \frac{1}{6}\left(10 - 3x_2^{(0)} + x_3^{(0)}\right) = 10/6$$
$$x_2^{(1)} = \frac{1}{7}\left(1 - 2x_1^{(0)} - x_3^{(0)}\right) = 1/7$$
$$x_3^{(1)} = \frac{1}{4}\left(-5 + x_1^{(0)} + x_2^{(0)}\right) = -5/4$$

So $x^{(1)} = \begin{pmatrix} 10/6 \\ 1/7 \\ -5/4 \end{pmatrix}$

See video or annotated slides

Take $x^{(1)} = \begin{pmatrix} 10/6 \\ 1/7 \\ -5/4 \end{pmatrix}$

So $x_1^{(2)} = \frac{1}{6}\left(10 - 3x_2^{(1)} + x_3^{(1)}\right) =$

$\frac{1}{6}\left(10 - 3\left(\frac{1}{7}\right) - \frac{5}{4}\right) = 1.3864$

similarly compute $x_2^{(2)}$ and $x_3^{(2)}$

There is also a "Matrix View" of this.

See video or annotated slides

- The initial system is $Ax = b$.

We can write the Jacobi iteration in matrix form too.

Let $D = \text{diag}(A) = \begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}$

Let $T = D - A = \begin{pmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{pmatrix}$

The Jacobi iteration is

$$x^{(k+1)} = D^{-1}\left(b + Tx^{(k)}\right).$$

Note: D^{-1}

$$\begin{pmatrix} 1/a_{11} & 0 & 0 \\ 0 & 1/a_{22} & 0 \\ 0 & 0 & 1/a_{33} \end{pmatrix}$$

Now that we know the method, let us summarise the steps, so as to work out what standard operations on vectors and matrices we need.

We expressed the problem as a matrix-vector equation: *Find* $\mathbf{x}$ *such that*

$$A\mathbf{x} = \mathbf{b},$$

where A *is a* $N \times N$ *matrix, and* $\mathbf{b}$ *and* $\mathbf{x}$ *are (column) vector with* N *entries.*

We then derived **Jacobi's method**: choose $\mathbf{x}^{(0)}$ and set

$$x^{(k+1)} = D^{-1}(b + Tx^{(k)}).$$

where $D = \mathrm{diag}(A)$ and $T = D - A$.

Looking at this we see that the fundamental operations are: **vector addition** and **matrix-vector multiplication**.

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

END OF PART 1

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

Start of ...

PART 2: A matrix class

Part 2: A matrix class

Since we already have Vector class from last week, our next step is to write a class implementation for a matrix, along with the associated functions.

Then we need to define a function to multiply a matrix by vector.

First though, we consider the matrix representation. The most natural approach might seem to be to construct a two dimensional array. This can be done as follows (see Lab 4):

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

An array of arrays.

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:

In C++, if we want, we can just represent a matrix A is a "big vector"

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \text{programmed as} \quad \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \\ a_{33} \end{pmatrix}$$

Part 2: A matrix class

This weeks files also have an updated "Matrix08.h".

Matrix.h

```
// File:  Matrix.h (W07.1)
// Author:  Niall Madden (NUI Galway) Niall.Madden@NUIGalway.ie
// Date:  Week of 2021-CS319)
// What:  Implementation of "Matrix":  a class of square matrices
// See also:  Matrix.cpp and 02TestMatrix.cpp

class Matrix {
private:
  double *entries;
  unsigned int N;
public:
  Matrix (unsigned int Size=2);
  ~Matrix(void) { delete [] entries; };

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

  void print(void);
};
```

(compare with Vector.h)

where we'l store values

number of rows and cols

constructor + destructor.

For returning an entry in the matrix

Part 2: A matrix class

from Matrix.cpp

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

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

} constructor.

Just check that i & j are valid (ie, in the range $0, \dots, N-1$

Part 2: A matrix class

from Matrix.cpp

```
double Matrix::getij (unsigned int i, unsigned int j)
{
  if (i<N && j<N)
    return(entries[i*N+j]);
  else
  {
    std::cerr << "Matrix::getij(): Index out of bounds."
              << std::endl;
    return(0);
  }
}

void Matrix::print (void)
{
// std::cout << "Matrix is of size " << M << "-by-"
// << N << std::std::endl;
  for (unsigned int i=0; i<N; i++)
  {
    for (unsigned int j=0; j<N; j++)
      std::cout << "[" << entries[i*N+j] << "]";
    std::cout << std::endl;
  }
```

So a_{ij} is stored in entries[i*N+j];

We'll test this by implementing matrix-vector multiplication function:

02TestMatrix.cpp

```
// File:      01TestMatrix.h (Set v=A*u)
// Author:    Niall Madden (Niall.Madden@NUIGalway.ie)
// Date:      Week 8 of 2021-CS319)
// What:      Test the implementation Matrix class
```

```
void MatVec(Matrix &A, Vector &u, Vector &v)
{
  unsigned int N;
  N = A.size();
  if ( (N != u.size()) || ( N != v.size() ) )
    std::cerr << "dimension mismatch in MatVec " << std::endl;
  else
    for (unsigned int i=0; i<N; i++)
    {
      double x=0;
      for (unsigned int j=0; j<N; j++)
        x += A.getij(i,j)*u.geti(j);
      v.seti(i,x);
    }
}
```

MatVec is a Matrix-Vector Product.

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

END OF PART 2

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

Start of ...

PART 3: Coding Jacobi's Method

Recorded Tue, 30 March (~ 9.45).

Part 3: Coding Jacobi's method

Now we can implement Jacobi's method. The specific example coded, we will solve $N = 3$ equations whose matrix representation is

$$9x_1 + 3x_2 + 3x_3 = 15 \tag{1}$$
$$3x_1 + 9x_2 + 3x_3 = 15 \tag{2}$$
$$3x_1 + 3x_2 + 9x_3 = 15 \tag{3}$$

This problem is constructed so that the solution is $x_1 = x_2 = x_3 = 1$.

Have a look at the `main()` function in `03Jacobi.cpp` to see how the problem is set up, and how the Jacobi solver is called. Here we will focus on that solver.

Soln : $x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

$A = \begin{pmatrix} 9 & 3 & 3 \\ 3 & 9 & 3 \\ 3 & 3 & 9 \end{pmatrix}$ $\qquad b = \begin{pmatrix} 15 \\ 15 \\ 15 \end{pmatrix}$

Part 3: Coding Jacobi's method

See `02Jacobi.cpp` for more details

```
// Use Jacobi's method to solve Ax=b,
// On entry :  x is the initial guess
// On exit :  x is the estimate for the solution
void Jacobi(Matrix &A, Vector &b, Vector &x,
            unsigned int &count, double tol)
{
  unsigned int N=A.size();
  count=0;
  if ( (N != b.size()) || (N != x.size() ) )
    std::cout << "Jacobi: error - A must be the same size as b,x"
              << std::endl;
```

— ie $x^{(0)}$

— $x^{(k)}$

making sure A, b & x all have the same N.

See 02Jacobi.cpp for more details

```
Matrix Dinv(N), T(N);    // The diagonal and off-diagonal matrices
for (unsigned int i=0; i<N; i++)
  for (unsigned int j=0; j<N; j++)
    if (j != i)
    {
      T.setij(i,j, -A.getij(i,j));
      Dinv.setij(i,j,0.0);
    }
    else
    {
      T.setij(i,j, 0.0);
      Dinv.setij(i,j, 1.0/A.getij(i,j));
    }
```

(not diagonal)

(ie $j == i$, so diagonal entry)

setting up Dinv & T.

Dinv $\sim D^{-1}$ where $D = \mathrm{diag}(A)$

$T = D - A$.

Part 3: Coding Jacobi's method

d

$x^{(k+1)} = D^{-1}(b + Tx^{(k)})$

See 02Jacobi.cpp for more details

```
// Now implement the algorithm:
Vector d(N), r(N);
do
{
  count++;
  MatVec(T,x,d);         // Set d=T*x
  VecAdd(d, b, d);       // set d=b+d (so d=b+T*x)
  MatVec(Dinv, d, x); // set x = inverse(D)*(b+T*x)

  MatVec(A, x, r);       // set r=A*x
  VecAdd(r, b, r, 1.0, -1.0); // set r=b-A*r

}   while ( r.norm() > tol);
```

(number of iterations.)

This is the actual Jacobi Method.

Since $Ax = b$, we know $b - Ax = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

So if $b - Ax^{(k)}$ is "small", we have a good approximation

Of course, the above code would be a lot neater, and much more readable, if we were able to write, for example, r=A*x instead of MatVec(A,x,r) ....

Exercise (8.1)

Write a method `Matrix::norm()` that returns the "Entry-wise" 2-norm of a matrix (also called the **Frobenius** *or Hilbert–Schmidt norm) :*

$$\|A\|_p = \Big(\sum_{i=1}^{n}\sum_{j=1}^{n}|A_{i,j}|^p\Big)^{1/p}.$$

and the max-norm:

$$\|A\|_0 = \max_{i,j}|A_{i,j}|.$$

End of Part 3.

Wed, 31 March, 2021

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

Start of ...

PART 4: Copy Constructors

Toward : operator overloading

In the next section, we will introduce the idea of **Operator Overloading**. But to get this to work, we need to study **copy constructors**.

This is a very 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~~ times 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.

method that has the same name as the class.

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.

```
// From Vector.h (Week 7)
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);
  // print(), zero() and norm() not shown
};

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

→ constructor

Here "entries" is pointer to type double — stores memory address of the start of the arrays

We then wrote some functions that manipulate vectors, such as AddVec in Week07/01TestVector.cpp

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

— set c = alpha * a + beta * b

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

that is

1. & before the variable name.

2. the function get the actual variable's address, not just a copy

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

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

This will cause problems for the following reasons:

We would like to think it sets

c.entries[0] = a.entries[0]

c.entries[1] = a.entriess[1]

c.entries[2] = a.entries[2]

c.entries[3] = a.entriess[3]

But actually, it sets

c.entries = a.entries.

That is, they both point to some memory address.

This is a problem

1. changing c.entries[k] changes a.entries[k] and vice versa.
2. If c (for example) goes out of scope, its destructor is called, deleting the memory allocated c.entries. So a.entries gets deleted too!
3. When a goes out of scope, a.entries is deleted (again!). which causes program to crash.

Double delete

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 Vector08.cpp for more details

```
// copy constructor (Version W08.1)
// Class definition in Vector08.h has also changed
Vector::Vector (const Vector &old_Vector)
{
  N = old_Vector.N;
  entries = new double[N];
  for (unsigned int i=0; i<N; i++)
    entries[i] = old_Vector.entries[i];
}
```

old_Vector not changed.

argument is an object of the same class.

notice we don't have to give the object's name – it is implicit.

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.

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

END OF PART Part 4

Const is a modifier to a argument entry in a function that asserts that that argument won't be changed.

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

Start of ...

PART 5: Operator Overloading

Recall that, along with **Encapsulation** (Classes) and **Inheritance** (deriving new classes from old), **Polymorphism** is one of the pillar ideas of Object-Oriented Programming

So far we have seen two forms of **Polymorphism**:

(a) we may have two functions with the same name but different argument lists. This is **function overloading**.

(b) `template`s allow us to define a function or class that with arbitrary data-types, which are not specified until used. In the case of a `class template` it is specified when an object of that class is defined.

We'll cover another form of polymorphism today: **"Operator overloading"** .

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

Yesterday

~~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 `#`

Eg

+ or =

- Almost all C++ operators can be overloaded:

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

but not `.` `::` `.*` `?`

Note that, for example, − (minus) can be both
- binary, as in $c = a - b$
- unary, as in $c = -a$.

Also ++ can be a++ or ++a.

- 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. (Eg if a&b are ints, a+b is what it always way.)
- 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. (same as copy constructor)

$Z = A*B+C$ is $Z = (A*B)+C$

and not $Z = A*(B+C)$.

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, `Vector08.h`:

```
Vector operator+(Vector b);
```

Then to `Vector08.cpp`, we add the code

Return type

See `Vector08.cpp` for more details

belongs to the Vector class.

```
// 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 size!"
              << std::endl;
  else
    for (unsigned int i=0; i<N; i++)
      c.entries[i] = entries[i] + b.entries[i];
  return(c);
}
```

Set $c = a + b$ (but a is implicit

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.

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

END OF PART Part 5

CS319 – Week 8
Week 8: Linear Systems, and Operator Overloading

Start of ...

PART 6: The "pointy", `this`, and = operators

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.

First, remember 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;
```

so (*w) = v

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`.

However, 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`.

The following simple (but contrived) demonstration shows why we might like to use the `this` pointer.

The (original) constructor for our Vector class took an integer argument called Size:

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

Suppose, for no good reason, we wanted to call the passed argument N: Then we would get the code below:

```
// Silly, broken version
Vector::Vector (unsigned int N)
{
   N = N;
   entries = new double[N];
}
```

Surprisingly, this will compile, but will give bizarre results (before crashing). To get around this we must distinguish between the local, passed, variable N, and the class member N.

This can be done by changing the offending line to:

```
this->N = N;
```

A much better use of this is for the case where a function must return the address of the argument that was passed to it. This is the case of the assignment operator.

See Vector08.cpp for more details

(assignment, as in c = b

```
// Overload the = operator.
Vector &Vector::operator=(const Vector &b)
{
  if (this == &b)
```

So here this is a pointer to c.

```
    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];
```

← copy.

```
  return(*this);
}
```

(not to be confused with equality c==b),