CS319: Scientific Computing (with C++)

Operator overloading

Week 8: 9am and 4pm, 01/03/2017

Also: see the `Makefile` in today's list of Examples.

See **"extras"** section of today's lectures for more examples of classes and overloading (points, dates, complex numbers) at `http://www.maths.nuigalway.ie/~niall/CS319/Week08/extras/` ←— link!

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.

extras

Introduction

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) function `templates` allow us to define a function that operates with respect to an arbitrary data-type.

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

Our main goal is to overload the addition (`+`) operator for vectors, and the (`*`) for matrix-vector multiplication. Also minus (-)

Before we do that, however, we will first review our `vector` class from last week, and then extend it with a **copy constructor**.

Recall: vector

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 7)
2  class vector {
3  private:
4     double *entries;
5     unsigned int N;
6  public:
7     vector (unsigned int Size=2);
8     ~vector(void);
9     unsigned int size(void) {return N;};
10    double geti (unsigned int i);
11    void seti (unsigned int i, double x);
12    // To save space, zero() and norm() members not shown
13 };

15 // Code for the constructor from Vector.cpp
16 vector::vector (unsigned int Size){
17    N = Size;
18    entries = new double[Size];
19    for (unsigned int i=0; i<N; i++)
20       entries[i]=0.0;
21 }
```

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

Set $c = \alpha a + \beta b$.

```
1 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...

Recall: vector

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:

By default, the assignment operator, =, is defined for any class. And setting c=a, sets

c.N = a.N and c.entries = a.entries

(ie, the default = operator copies the values of the right operand's data members to those of the left operand.)

Recall: vector

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:

For very simple objects, this is OK. But here entries is a pointer: it stores a memory address. So, c=a, results in c & a sharing a memory address. So, eg, changing c.entries[2] also changes a.entries[2] & vice versa.

Recall: vector

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:

Worse! If, say, c goes out of scope, its destructor is called & the memory in c.entries is deallocated. So a.entries is deallocated too (they are the same). Now when a goes out of scope, it's destructor is called, & the program crashes with a "double delete".

with a call to delete.

The copy constructor

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:

```
1  // copy constructor (from latest version of Vector.cpp)
   // Class definition in Vector.h should also be changed
3  vector::vector (const vector &old_vector)
   {
5     N = old_vector.N;
      entries = new double[N];
7     for (unsigned int i=0; i<N; i++)
         entries[i] = old_vector.entries[i];
9  }
```

VectorO8.cpp.

copy values in entries, not its memory address.

Function name is the same as the class name => is a constructor.

Const <=> argument is not changed.

The copy constructor

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
```

not common

(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.

very common.

Operator Overloading

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`.

here $\alpha = 1$ & $\beta = 1$ by default.

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).

`c = a+b` requires 2 overloaded operators:

assignment (=)

plus (+).

Operator Overloading

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 `#`

Operator Overloading

$c = a*b+d \Leftrightarrow c = (a*b)+d$

not $c = a*(b+d)$.

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.

Operator Overloading

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.

Finished here 10 am

Eg 1: Adding two vectors

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

```
1 vector vector::operator+(vector b) {
2    vector c(N); // Make c the same size of a
3    if (N != b.N)
4       cerr << "vector::+ : cant add two vectors of different size!" << endl;
5    else
6       for (unsigned int i=0; i<N; i++)
7          c.entries[i] = entries[i] + b.entries[i];
8    return(c);
9 }
```

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.

The -> operator

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;
```

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`.

The `this` pointer

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 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;
```

Overloading =

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.

```
// Overload the = operator.  CS319-Week 8
vector &vector::operator=(const vector &b)
{
   if (this == &b)
      return(*this); // Taking care for self-assignment

   delete [] entries; // In case memory was already allocated
   entries = new double[b.N];
   for (unsigned int i=0; i<N; i++)
      entries[i] = b.entries[i];

   return(*this);
}
```

Unary operators

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

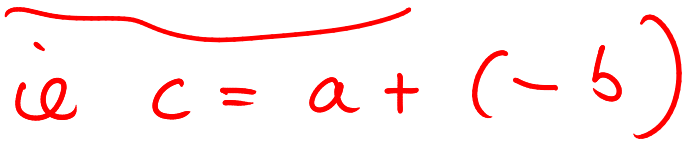
But many C++ operators are **unary**: they take only one argument.

The most common examples of unary operators are `++` and `--`, but for our `vector` class, we'll first over load 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. (Later, we'll look at **postfix** operators, like `a++`, which are a little more complicated).

Out of fun, and lazyness, we will then define the binary minus operator, by using addition and unary minus.



Unary operators

See `Vector08.cpp` for more details

```
// Overload the unary minus (-) operator.  As in b=-a;
vector vector::operator-(void)
{
  vector b(N); // Make b the size of a
  for (unsigned int i=0; i<N; i++)
    b.entries[i] = -entries[i];
  return(b);
}

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

notice that a is implicit.

b is explicit & a is implicit.

need the 'this' pointer here.

This is tested in 01TestVector.cpp ← link!

Overloading for the matrix class

As with the *vector* class, we would like to overload some functions and arithmetic operators for *matrix*. In the files `Matrix.h` and `Matrix.cpp` examples are given for

Matrix08.h Matrix08.cpp.

- defining the copy constructor;
- overloading the assignment operator.

They follow the same ideas as the corresponding components of the *vector* class.

With those done, we can think about overloading the multiplication operator for *matrix*-*vector* multiplication.

This introduces a few small new complications:

u = A * b
vector matrix,

- 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 C/C++ preprocessor

The preprocessor in C++ is left 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 preproc 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, such use of the `#define` directive is bad practice; but we will use something similar...

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

```
#include <iostream>
#include "Vector08.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 preproc 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, 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 `Vector08.h` in `Matrix08.h`. But then if our main source file includes both `Matrix08.h` and `Vector08.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 `Vector08.h`

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

→ if _VECTOR_H_INCLUDED is not defined,

In another use, we might want the compiler to behave in particular ways for particular operating systems. E.g.,

```
#ifndef linux
system("PAUSE");
#endif
```

Other applications of preprocessor directives include defining parameterized macro (which is like a function), and `#pragma` directives for certain compilers. The latter is used a lot in parallel computing.

Overloading * for MatVec

On the course website, accompanying these notes for Week 8, you can find an extended implementation of the `matrix` class, implemented in `Matrix08.cpp` and `Vector08.cpp`, which includes

- a copy constructor;
- an overloaded assignment operator.

Both these are very similar to the *`vector`* versions, so are not described here.

Instead we focus on overloading `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, and add this line to the definition in `Matrix08.h` :

```
vector operator*(vector b);
```

Overloading * for MatVec

The code from Matrix08.cpp is given below. Compare with MatVec

```
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*w, 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 }
```

Recall the Matrix A is stored as a one dimensional vector of length N^2.

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

Exercise: Convince yourself this works

Overloading * for MatVec

Equipped with this, we can now write a neater version of the `Jacobi` function:

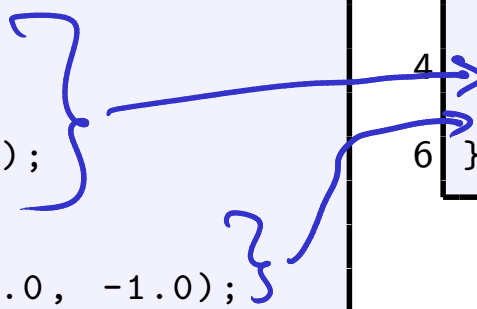
Old Version

```
vector d(N), r(N);
do  {
  count++;
  MatVec(T,x,d);
  VecAdd(d, b, d);
  MatVec(Dinv, d, x);

  MatVec(A, x, r);
  VecAdd(r, b, r, 1.0, -1.0);
}   while ( r.norm() > tol);
```

New Version

```
vector r(N);
do {
  count++;
  x = Dinv*(b+T*x);
  r=b-A*x;
}   while ( r.norm() > tol);
```



Finished here 5pm

`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.

However, it is possible to designate non-member as being *`friends`* of a 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 **Put to** `<<` and **Get from** `>>` operators can only be overloaded as friends.

friend functions

Suppose we add the following line to the class definition in Vector08.h

```
  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 cout directly.