Introduction to C++ - lectures 7-8

Martin Robinson

2019

Lecture 7 — Operator overloading

We will design a class of vectors in such a way that:

1. objects of this class behave like a new data type; and
2. code similar in style to Matlab may be written using objects of this class.

We will define operations on and between objects of the class of vectors, and between these objects and data types such as `int` and `float`

We want to be able to write code such as

```
Vector u(3), v(3); // vectors of length 3
Matrix A(3,3); // matrix of size 3 by 3
v = A * u;
u = gmres(A, v)
```

Constructors for vectors

We want the declaration

```
Vector u(3);
```

to create a vector of size 3

The following file should be saved as `Vector.hpp`

```
#ifndef VECTORDEF
#define VECTORDEF
// a simple class of vectors
class Vector
{
public:
  // construct vector of given length
  Vector(int sizeVal);
private:
  // data stored in vector
  std::vector<double> mData;

};
#endif
```

The following file should be saved as `Vector.cpp`

```
#include "Vector.hpp"

// constructor that creates vector of given size with
// double precision entries all initially set to zero
Vector::Vector(int sizeVal):
    mData(sizeVal,0)
{}
```

Operator overloading

We want to write code such as

```
v(0) = 1.0;
w = u + v;
```

where `u`, `v`, `w` are defined to be objects of the class `Vector`

We have to define within the class what is meant by the operators `()`, `+` and `=` in this context

This can be achieved by *overloading* the `()`, `+` operator and the `=` operator for the class of vectors

First we will provide access to the data in the vector class using the `()` operator

Overloading the ( ) operator

We may overload the ( ) operator in order to access elements of an array. The function required to do this is

```
double& Vector::operator()(int i)
{
    return data[i];
}
```

and the following line must be included in the file `Vector.hpp`

```
double& operator()(int i);
```

We can now access the first element of `u` by writing `u(0)`

Note the appearance of the symbol `&` on the previous slide This indicates that the operator returns a reference

This allows us to use terms such as `u(1)` on the left hand side of expressions such as `u(1) = 2.0`

Aside: we can also overload the square brackets operator with

```
double& operator[](int i);
```

Binary operators

To write code such as

```
w = u + v;
```

then the lines

```
Vector& operator=(const Vector& rVec);
friend Vector operator+(const Vector& rVec1, const Vector& rVec2);
```

should be added within the class description in the file `Vector.hpp`, and then the following two functions should be included in the file `Vector.cpp`

Assignment operator (Option 1)

Implemented using an index loop

```
Vector& Vector::operator=(const Vector& rVec)
{
  for (int i=0; i<v.mData.size(); i++)
  {
    mData[i] = rVec.mData[i];
  }
  return *this;
}
```

We return a reference to the current object, in order that assignments can be "chained" together. i.e.

```
u = v = w;
```

Assignment operator (Option 2)

Implemented using the STL

```
Vector& Vector::operator=(const Vector& rVec)
{
    std::copy(rVec.mData.begin(),rVec.mData.end(),
              mData.begin());
    return *this;
}
```

Addition operator (Option 1)

```
Vector operator+(const Vector& rVec1,
                 const Vector& rVec2)
{
  Vector w(rVec1.mData.size());
  for (int i=0; i<rVec1.mData.size(); i++)
  {
    w.mData[i] = rVec1.mData[i] + rVec2.mData[i];
  }
  return w;
}
```

We must return a new object as the result of the addition. **BUT**, what if the `Vector` is large! This is a problem for linear algebra libraries, normally solved through expression templates. This is beyond the scope of this course

The binary operators - and * can be overloaded in a similar way as to +

When overloading * we first have to define what `u * v` means for a vector, i.e. do we mean the scalar product or the vector product?

We can also also overload * to define multiplication between an array of double precision numbers and a double precision number

For example, if `a` is a double precision floating point variable, and `u` is an array of double precision floating point numbers we can define what is meant by the operator * in the case `a * u`

```
Vector operator*(double a, const Vector& rVec)
{
  Vector w(rVec.mData.size());
  for (int i=0; i<rVec.mData.size(); i++)
  {
    w.mData[i] = a * rVec.mData[i];
  }
  return w;
}
```

after the following line has been included into the file `Vector.hpp`

```
friend Vector operator*(double a, const Vector& rVec);
```

Binary operators without ‘friend’

In overloading operator+ we have used an external friend function rather than a local method because it feels natural to be adding two objects.

```
Vector operator+(const Vector& rVec1,
                 const Vector& rVec2)
{
  Vector w(rVec1.mData.size());
  for (int i=0; i<rVec1.mData.size(); i++)
  {
    w.mData[i] = rVec1.mData[i] + rVec2.mData[i];
  }
  return w;
}
```

However, it is more efficient (in terms of characters typed) to write a binary operator as a member of a class.

```
Vector Vector::operator+(const Vector& rOther)
{
  Vector w(mData.size());
  for (int i=0; i<mData.size(); i++)
  {
    w.mData[i] = mData[i] + rOther.mData[i];
  }
  return w;
}
```

Both operators would be instantiated as `a = b + c`, but in the case of the second style, it would be run as an internal method of `b` (`mData` evaluates to `b`'s `mData`).

Unary operators

The unary operators - and + may also be overloaded in a similar manner to binary operators: add the line

```
friend Vector operator-(const Vector& rVec);
```

to the list of public members of `Vector` in the file `Vector.hpp`, and add the function on the following slide to the file `Vector.cpp`

```
Vector operator-(const Vector& rVec)
{
  Vector w(v.mData.size());
  for (int i=0; i<v.mData.size(); i++)
  {
    w.mData[i] = -v.mData[i];
  }
  return w;
}
```

Overloading the output operator

C++ does not know how to print out a vector, unless you tell it how. (If you print a pointer, then a memory address will be printed.)

Overload the << operator in the hpp file:

```
class Vector
{
private:
  int mSize;
  double* x;
public:
  Vector(int);
  ...
  friend std::ostream&
    operator<<(std::ostream& output, const Vector& rVec);
```

The implementation might look like this:

```
// std::cout << "a_vector = " << a_vector << "\n";
// appears as: a_vector = (10, 20)
std::ostream& operator<<
   (std::ostream& output, const Vector& rVec)
{
  output << "(";
  for (int i=0; i<rVec.mSize; i++)
  {
    output << rVec.mData[i];
        if (i != rVec.mData.size()-1)
      output << ", ";
    else
      output << ")";
  }
  return output; // for multiple << operators.
```

or. . . .

```
// std::cout << "a_vector = " << a_vector << "\n";
// appears as: a_vector = (10, 20)
std::ostream& operator<<
   (std::ostream& output, const Vector& rVec)
{
  output << "(";
  std::copy(v.begin(),
            --v.end(),
            std::ostream_iterator<T>(output, ", "));
  output << v.back() << ")";
  return output; // for multiple << operators.
}
```

Tip 1: plugging C++ into Matlab

You may need to interface C++ with Matlab (or with Gnu Octave)

- to get the speed of compiled code in a critical place
- to use an external library written in C++

This is possible with a Matlab executable file (Mex)

In the simplest case the C++ code is a single file containing a function called `mexFunction` with a specific signature

The function `mexFunction` takes points to arrays for output and input

This is compiled with `mex` (a wrapper compiler to g++) and a `.mex` file is produced

The `.mex` file is treated like a `.m` file by Matlab

An example file `myFunc.cpp`

```cpp
#include "mex.h"
#include <iostream>
void
mexFunction(int nlhs, mxArray *plhs[], int nrhs,
    const mxArray *prhs[])
{
  mxArray *v = mxCreateDoubleMatrix(1, 1, mxREAL);
  double *data = mxGetPr(v);
  *data = 3.142;
  std::cout<<"Num args = "<<nrhs<<" \n";
  plhs[0] = v;
}
```

Recall that you can write generic functions using templates

```
template <typename T>
T get_min (T a, T b) {
    if (a<b) {
        return a;
    }
    return b; // When a>=b
}

void main() {
   std::cout << get_min<int>(10,-2) << "\n";
   double ans = get_min<double>(22.0/7.0, M_PI);
}
```

Template specialisation

You can provide specialisations for template arguments. Explicit (full) specialisation is when *all* template arguments are specialised

```
template <typename T>
bool is_int(T a) {
    return false;
}

template <>
bool is_int<int> (T a) {
    return true;
}
```

Class Template

It is possible to template both functions and classes

```
template<unsigned DIM>
class DoubleVector {
   std::array<double,DIM>;
   public:
   double& operator[](int pos) {
      assert(pos<DIM);
      return(mData[pos]);
   }
};

int main() {
   DoubleVector<5> a;
   a[0] = 10;
```

Partial specialisation

Class templates can also be *partially* specialised

```
// primary template
template<class T1, class T2, int I>
class A {};

// partial specialization where T2 is a pointer to T1
template<class T, int I>
class A<T, T*, I> {};

// partial specialization where T1 is a pointer
template<class T, class T2, int I>
class A<T*, T2, I> {};

// partial specialization where T1 is int, I is 5
```

For example: using partial specialisation to determine if class T is a pointer

```
template <typename T>
struct is_pointer { static const bool value = false; };

template <typename T>
struct is_pointer<T*> { static const bool value = true; };

void main(void) {
    assert(is_pointer<int*>::value);
}
```

This, and many more, already implemented for you in C++ `type_traits` standard library

Template instantiation

Before you use, or *instantiate*, a templated class the compiler has nothing to compile. The templated class is simply a generic *template*.

```
template <typename T>
struct Vect3 {
    T x,y,z;
    T norm();
};
```

For a normal class you would probably define the `norm` function in a `.cpp` file which is compiled separately. For a templated class the definition must occur in a header (e.g. `.hpp`) file

```
template <typename T>
T Vect3<T>::norm() {
    return sqrt(pow(x,2) + pow(y,2) + pow(z,2));
```

When you instantiate a class, say in a main .cpp file, the compiler fills in the template arguments and can then compile the class

```
void main(void) {
    Vect3<double> v;
    Vect3<float> v;
    Vect3<int> v;
}
```

Note, the above will generate three different classes.

Tip 2: plugging C++ into Python

Python is a useful dynamic (scripting) language for scientific computing.

Many different ways exist to interface C++ to Python

- Boost Python (www.boost.org)
- Swig (www.swig.org)
- Cython (cython.org)
- Using Python API (docs.python.org/2/extending/extending.html) and `distutils`

You can also interleave C and C++ code in Python scripts

- Scipy.Weave (www.scipy.org)

Boost Python

Use the Boost Python library to wrap your C++ functions or classes for Python

An example file `hello.cpp`

```
char const* greet()
{
   return "hello, world";
}

#include <boost/python.hpp>

BOOST_PYTHON_MODULE(hello)
{
    using namespace boost::python;
    def("greet", greet);
}
```