Operator Overloading: class `Vector`

Shahram Rahatlou

Sapienza Università di Roma

http://www.roma1.infn.it/people/rahatlou/programmazione++/

Corso di Programmazione++

Roma, 11 May 2009

Today's Lecture

- Final lecture on overloading operators
  - Example of class **`Vector`**
- Dynamic memory allocation for data members
  - constructors and destructors revisited
- Example of static data and functions for classes

Class `Vector`

- Built-in C arrays not satisfactory in many ways
  - No protection against bad usage by users
  - No way to extend an array after its creation
  - No operators to add/subtract arrays
  - No way to find out how large an array is

- In the last few weeks we have seen how custom made classes can be written including overloading operators to treat custom classes as built-in types
  - Use above limitations to develop a **`Vector`** class providing all missing functionalities of C arrays

- First start with defining the interface of such class
  - What does the user expect to have?

Requirements and Interface of `Vector`

```
class Vector {
  public:
  // constructors
    Vector();
    Vector( int size );
    Vector( const Vector& );

    // destructor
    ~Vector();

    // getters
    int size() const;
    const double& operator[](int index) const;

    //operators between Vector
    Vector operator+( const Vector& vec) const;
    Vector operator-( const Vector& vec) const;
    const Vector& operator=( const Vector& );

    // interaction with doubles
    Vector operator*( double scale) const;
    friend Vector operator*(double scale, const Vector& vec);

    // boolean operators
    bool operator==( const Vector& vec) const;
    bool operator!=( const Vector& vec) const;
    bool operator<( const Vector& vec) const;

   // I/O
   friend ostream& operator<<(ostream& os, const Vector& vec);

};
```

```
  private:
     // data members
     // what would you add as data members?
```

What should the default constructor do?

What is the type of each element?

Data Members for **`Vector`**

- Which are the attributes of a **`Vector`**?
  - what characterizes an object of type **`Vector`**
  - what differentiates between two different vectors

- Remember the limitations of C arrays
  - We would like to extend vectors dynamically

- Possible solution:

```
class Vector {
  public:
    // member functions

  private:
    int size_; // size of array
    double  data_[size_]; // actual data
};
```

Problem with Proposed Solution

```
class Vector {
  public:
  // constructors
    Vector();
    Vector( int size );
    Vector( const Vector& );

  private:
    int size_; // size of array
    double  data_[size_]; // actual data

};
```

```
Vector::Vector() {

}

Vector::Vector(int size) {
  size_ = size;
  data_ = double[size];
}


Vector::Vector(const Vector&vec) {
  size_ = vec.size_;
  data_ = vec.data_;
}
```

- Can’t really even get it to compile
  - Actual C++ errors
- Real conceptual errors as well
  - size of the array is not known until the constructor is used!
  - How can data_ be initialized?
- What about dynamic memory allocation?

Dynamic Memory Allocation in Vector

```
#ifndef Vector_h
#define Vector_h
class Vector {
  public:
    // constructors
    Vector();
    Vector( int size );

    int size() const { return size_; }

  private:
    int size_; // size of array
    double*  data_; // pointer to actual data!

};
#endif
```

```
#include "Vector3.h"

Vector::Vector() {
  cout << "Vector::Vector() called" << endl;
  size_ = 0;
  data_ = 0; // null pointer

}

Vector::Vector(int size) {
  cout << "Vector::Vector(" << size
       << ") called" << endl;
  size_ = size;
  data_ = new double[size]; // dynam. alloc.
}
```

```
// app1.cpp
#include <iostream>
using namespace std;
#include "Vector3.h"

int main() {
  Vector v1;
  cout << "v1.size: " << v1.size() << endl;

  Vector v2(3475);
  cout << "v2.size: " << v2.size() << endl;

  return 0;
}
```

```
$ ./app1
Vector::Vector() called
v1.size: 0
Vector::Vector(3475) called
v2.size: 3475
```

Ops! Don't Forget the Destructor!

```
#ifndef Vector_h
#define Vector_h
class Vector {
  public:
    // constructors
    Vector();
    Vector( int size );
    ~Vector();

    int size() const { return size_; }

  private:
    int size_; // size of array
    double*  data_; // pointer to actual data!

};
#endif
```

```
Vector::~Vector() {
  cout << "Vector::~Vector() called" << endl;
   delete[] data_;
}
```

```
$ g++ -o app1 app1.cpp Vector3.cc
$ ./app1
Vector::Vector() called
v1.size: 0
Vector::Vector(3475) called
v2.size: 3475
Vector::~Vector() called
Vector::~Vector() called
```

- Remember! For each **`new`** there should be a **`delete`** somewhere
- **`Vector`** is responsible for dynamically allocated data in its constructors
- **`Vector::~Vector()`** must take care of managing the allocated memory upon destruction of each Vector object

`Vector` Constructors

- Do we really need a default constructor?
- What about default value for `Vector::Vector(int)` ?

```
#ifndef Vector_h
#define Vector_h
class Vector {
  public:
    // constructors
    Vector( int size  = 0 );
    ~Vector();

    int size() const { return size_; }

  private:
    int size_; // size of array
    double*  data_; // pointer to actual data!

};
```

```
Vector::Vector(int size) {
  cout << "Vector::Vector(" << size << ") called" << endl;
  size_ = size;
  data_ = new double[size]; // dynamically allocated memory!
  for(int i=0; i<size; ++i) {
    data_[i] = 0.;
  }
}
```

Initialize elements in the constructor

Access to Elements of Vector

- Overload operator[] to provide access to elements of Vector
  - Same functionality of built-in C arrays

```
class Vector {
  public:

    const double& operator[](int index) const;

}
```

```
const double&
Vector::operator[](int index) const {
  return data_[index];
}
```

- Reading elements works just fine

```
#include "Vector4.h"

int main() {

  Vector v2(3475);
  double x = v2[45];
  cout << "v2[45]: " << x << endl;

  return 0;
}
```

```
$ g++ -o app3 app3.cpp Vector4.cc
$ ./app3
Vector::Vector(3475) called
v2[45]: 0
Vector::~Vector() called
```

- What about assigning values to each element?

Assigning Value to Elements of Vector

- We can't use the overloaded operator[] to assign values to individual elements?
  - Why?

```
class Vector {
  public:
    const double& operator[](int index) const;
}
```

```
// app4.cpp
#include <iostream>
using namespace std;

#include "Vector4.h"

int main() {

  Vector v2(3475);
  v2[45] = 3.4;
  cout << "v2[45]: " << v2[45] << endl;

  return 0;
}
```

```
$ g++ -o app4 app4.cpp Vector4.cc
app4.cpp: In function `int main()':
app4.cpp:10: error: assignment of read-only location
```

- operator[] returns a constant reference to element
  - Client can not modify the return value

- But we do need non-const access to each element!

Overloading operator[] with Different Signatures

- We need to provide a new member function that grants non-const access to each element

```
class Vector {
  public:
    double& operator[](int index);
}
```

```
double& Vector::operator[](int index) {
  return data_[index];
}
```

```
// app4.cpp
#include <iostream>
using namespace std;

#include "Vector4.h"

int main() {

  Vector v2(3475);
  v2[45] = 3.4;
  cout << "v2[45]: " << v2[45] << endl;

  return 0;
}
```

```
$ g++ -o app4 app4.cpp Vector4.cc
$ ./app4
Vector::Vector(3475) called
v2[45]: 3.4
Vector::~Vector() called
```

Why not return by value?

- Now that we have full access to each element why return a constant reference at all?

- No reason! Return by-value for read-only access
  - Remember no real gain between constant reference and value for double or other simple types
  - constant reference still appropriate when with vectors of huge objects
    - can gain in speed and memory usage by returning a constant reference for read-only usage

```
// read-only access
double operator[](int index) const;

// allow modification by client
double& operator[](int index);
```

- Multiple signatures of same operator allow transparent use of Vector for all const and non-const use cases

Vector Interface after All Changes

```
#ifndef Vector_h
#define Vector_h
class Vector {
  public:
    // constructors
    Vector( int size  = 0);
    ~Vector();

    int size() const { return size_; }

    // read-only access
    double operator[](int index) const;

    // allow modifcation by client
    double& operator[](int index);

  private:
    int size_; // size of array
    double*  data_; // pointer to actual data!
};
#endif
```

Missing Feature: No Protection Against Bad Index

```
// app6.cpp
#include <iostream>
using namespace std;
#include "Vector5.h"

int main() {

  Vector v2(13);
  v2[2312] = 3.4;
  cout << "v2[15]: " << v2[15] << endl;

  return 0;
}
```

```
$ ./app6
Vector::Vector(13) called
Segmentation fault (core dumped)
```

- No compilation error
  - Our Vector class is only a wrapper around built-in C array
  - All functionalities are directly delegated to arrays
- Runtime problem
  - Program crashes because we try to access bad memory location

So why using this class instead of bare C array?

Smart Overload of `operator[]`

- Remember: **`operator[]`** is a member function
  - You can do much more than returning a value
  - For example: check validity of index and generate error

```
#include <cstdlib> // prototype for std::exit

double
Vector::operator[](int index) const {
  if( index < 0 || index >= size_ ) {
     cout << "bad index " << index^
          << " not in range [0:" << size_
          << "]" << endl;
     std::exit( -1 ); // exit program
  } else { // good index
    return data_[index];
  }
}

double&
Vector::operator[](int index) {
  if( index < 0 || index >= size_ ) {
     cout << "bad index " << index^
          << " not in range [0:" << size_
          << "]" << endl;
     std::exit( -1 ); // exit program
  } else { // good index
    return data_[index];
  }
}
```

```
// app7.cpp
#include <iostream>
using namespace std;
#include "Vector5.h"

int main()  {

  Vector v2(13);
  const double x = v2[7884];

  return 0;
}
```

```
$ ./app7
Vector::Vector(13) called
bad index 7884 not in range [0:13]
```

Quick and dirty solution:
- Exit from the main program when error occurs
- Not so elegant nor practical
- We will learn about C++ exceptions in a few weeks for error handling

private function `Vector::validIndex(int index)`

```
class Vector {
  private:
    bool validIndex(int index) const;
};

bool
Vector::validIndex(int index) const {
  if( index < 0 || index >= size_ ) {
     cout << "bad index " << index^
          << " not in range [0:" << size_
          << "]" << endl;
     return false;
  } else {
    return true;
  }
}
```

```
double
Vector::operator[](int index) const {
  if( !validIndex(index) ) {
     std::exit( -1 ); // exit program
  } else { // good index
     return data_[index];
  }
}

double&
Vector::operator[](int index) {
  if( !validIndex(index) ) {
     std::exit( -1 ); // exit program
  } else { // good index
     return data_[index];
  }
}
```

- Avoid duplication of code in two member fucntions
- Implement ONE method do check validity of index provided by client
  - Can be used in any method of the Vector using indices
- Make function private
  - Functionality needed for internal use in the class
  - No reason to make this function public

Overloading of `operator=()`

- Few considerations before implementing this method

- What do we do for vectors of different length?

```
int main() {

  Vector v1(217);
  Vector v2(13);

  v2 = v1;

  return 0;
}
```

- We have few options
  - Generate error: only assignment for vector of same size
  - Re-size the left-hand-side vector to match the right-hand-size
  - Decision is up to you based on your use case
    - Ask yourself: is Vector an appropriate name for my class? ☺

Implementation of `operator=()`

```
// assigment operator
   const Vector& operator=(const Vector& rhs);
```

```
const Vector&
Vector::operator=(const Vector& rhs) {
  if(size_ != rhs.size_) {
    cout
  << "vectors of different size. changing from "
         << size_ << " to " << rhs.size_
         << " to match rhs.size()"
         << endl;
  }

  // delete old array of data
  delete[] data_;

  // now modify self to match the rhs
  size_ = rhs.size_;
  data_ = new double[rhs.size_];

  // copy values from rhs to self
  for(int i=0; i<size_;++i) {
    data_[i] = rhs.data_[i];
  }

  // return modified self
  return *this;
}
```

```
// app8.cpp
#include <iostream>
using namespace std;
#include "Vector5.h"

int main() {

  Vector v1(57);
  cout << "v1[47]: " << v1[47] << endl;

  Vector v2(3);
  for(int i=0; i<3;++i) {
    v2[i] = i;
  }

  v1 = v2;

  cout << "v1[2]: " << v1[2] << endl;
  cout << "v1[47]: " << v1[47] << endl;

  return 0;
}
```

```
$ g++ -o app8 app8.cpp Vector5.cc
$ ./app8
Vector::Vector(57) called
v1[47]: 0
Vector::Vector(3) called
vectors of different size. changing from 57 to 3 to match rhs.size()
v1[2]: 2
bad index 47 not in range [0:3]
```

Considerations on `operator=()`

```
const Vector&
Vector::operator=(const Vector& rhs) {
  if(size_ != rhs.size_) {
    cout
  << "vectors of different size. changing from "
           << size_ << " to " << rhs.size_
           << " to match rhs.size()"
           << endl;
  }

  // delete old array of data
  delete[] data_;

  // now modify self to match the rhs
  size_ = rhs.size_;
  data_ = new double[rhs.size_];

  // copy values from rhs to self
  for(int i=0; i<size_;++i) {
    data_[i] = rhs.data_[i];
  }

  // return modified self
  return *this;
}
```

- new and delete are expensive operations
- We should use them only when necessary
- Always remember that new without appropriate delete will cause memory leak in your program

Improved Implementation of `operator=()`

```
const Vector&
Vector::operator=(const Vector& rhs) {
  if( &rhs == this ) {
    cout << "avoiding self assignment" << endl;
    return *this;
  }

  if(size_ != rhs.size_) {
    cout
 << "vectors of different size. changing from "
     << size_ << " to " << rhs.size_
     << " to match rhs.size()"
         << endl;

    // delete old array of data
    delete[] data_;

    // now modify self to match the rhs
    size_ = rhs.size_;
    data_ = new double[rhs.size_];
 }

  // copy values from rhs to self
  for(int i=0; i<size_;++i) {
    data_[i] = rhs.data_[i];
  }

  // return modified self
  return *this;
}
```

```
// app9.cpp
#include <iostream>
using namespace std;
#include "Vector5.h"

int main() {

  Vector v1(3);
  for(int i=0; i<3;++i) {
    v1[i] = i;
  }

  v1 = v1;

  return 0;
}
```

```
$ ./app9
Vector::Vector(3) called
avoiding self assignment
Vector::~Vector() called
```

new and delete are called only
if Vectors of different size are used

No need to delete and make new
if assigning an object to itself

An Even Better Implementation of `operator=()`?

```
const Vector&
Vector::operator=(const Vector& rhs) {
  if( &rhs == this ) {
    cout << "avoiding self assignment" << endl;
    return *this;
  }

  if(size_ != rhs.size_) {
    cout
 << "vectors of different size. changing from "
     << size_ << " to " << rhs.size_
     << " to match rhs.size()"
         << endl;

    // delete old array of data
    delete[] data_;

    // now modify self to match the rhs
    size_ = rhs.size_;
    data_ = new double[rhs.size_];
 }

  // copy values from rhs to self
  for(int i=0; i<size_;++i) {
    data_[i] = rhs.data_[i];
  }

  // return modified self
  return *this;
}
```

- Could we further reduce use of new and delete?
- Do we really have to re-allocate a new array if the lhs.size_ > rhs.size_ ?
- Provide possible solutions for next lecture

Exercise: Missing Features to Implement

- Resize an existing Vector object
- Copy constructor
- operators to do arithmetics
- comparison operators
- operator overloading via global functions
  - input/output via iostream