Overloading Operators
friend functions
static data and methods

Shahram Rahatlou

SAPIENZA
UNIVERSITÀ DI ROMA

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

Corso di Programmazione++

Roma, 21 April 2009

Today's Lecture

- Overloading operators for built in types
- Friend methods
- Global functions as a way of operator overloading
- Static data members and methods

Overloading `bool Datum::operator<(const Datum& rhs)`

```
class Datum {
  public:

    bool operator<(const Datum& rhs) const;

   // …
}
```

return type is boolean

constant method since does not modify the object being applied to

```
bool Datum::operator<(const Datum& rhs) const {
  return ( value_ < rhs.value_ );
}
```

```
int main() {
  Datum d1( 1.2, 0.3 );
  Datum d3( -0.2, 1.1 );
  cout << "d1: " << endl;
  d1.print();
  cout << "d3: " << endl;
  d3.print();

  if( d1 < d3 ) {
    cout << "d1 < d3. d1 is:" << endl;
  } else {
    cout << "d3 < d1. d3 is:" << endl;
  }

  return 0;
}
```

Comparison based on the value_

error_ does not affect the comparison do you agree?

```
$ g++ -Wall -o app6 app6.cpp Datum.cc
$ ./app6
d1:
datum: 1.2 +/- 0.3
d3:
datum: -0.2 +/- 1.1
d3 < d1. d3 is:
```

Typical Error: Operators += and <=

```
int main() {
  Datum d1( 1.2, 0.3 );
  Datum d3( -0.2, 1.1 );

  d1 += d3;

  if( d1 <= d3 ) {
    cout << "d1 < d3. d1 is:" << endl;
  } else {
    cout << "d3 < d1. d3 is:" << endl;
  }

  return 0;
}
```

```
$ g++ -Wall -o app7 app7.cpp Datum.cc
app7.cpp: In function `int main()':
app7.cpp:12: error: no match for 'operator+=' in 'd1 += d3'
app7.cpp:14: error: no match for 'operator<=' in 'd1 <= d3'
```

Having defined =, +, and < separately does not provide automatically += and <=

These must be overloaded explicitly by the user

Tip:
Use < to quickly implement > and >= as well

Division and Multiplication of **Datum**

```
Datum operator*( const Datum& rhs ) const;
Datum operator/( const Datum& rhs ) const;
```

```
Datum Datum::operator*(const Datum& rhs) const {
  double val = value_*rhs.value_;

  // propagate correctly the error for x*y
  double err = sqrt( rhs.value_*rhs.value_*error_*error_ +
                     rhs.error_*rhs.error_*value_*value_ );
  return Datum(val,err);
}

Datum Datum::operator/(const Datum& rhs) const {
  double val = value_ / rhs.value_;

  // propagate correctly the error for x / y
  double err = fabs(val) * sqrt( (error_/value_)*(error_/value_) +
                           (rhs.error_/rhs.value_)*
                           (rhs.error_/rhs.value_) );

  return Datum(val,err);
}
```

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

#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -3.4, 0.7 );
  d1.print();
  d2.print();

  Datum d3 = d1 * d2;
  Datum d4 = d1.operator*(d2);

  d3.print();
  d4.print();

  Datum d5 = d1 / d2;
  Datum d6 = d2/d1 ;
  d5.print();
  d6.print();

  return 0;
}
```

```
$ g++ -Wall -o app8 app8.cpp Datum.cc
$ ./app8
datum: 1.2 +/- 0.3
datum: -3.4 +/- 0.7
datum: -4.08 +/- 1.32136
datum: -4.08 +/- 1.32136
datum: -0.352941 +/- 0.114305
datum: -2.83333 +/- 0.917613
```

To be meaningful you must compute correctly the error for the result as expected by the user

Otherwise your class is wrong and useless

Interactions between **Datum** and **double**

- It's intuitive to multiply a Datum by a double

- No problem... overload the * operator with necessary signature

```
class Datum {
  public:
    Datum operator*( const double& rhs ) const;
  // …

};

Datum Datum::operator*(const double& rhs) const {
  return Datum(value_*rhs,error_*rhs);
}
```

```
Datum Datum::operator*(const double& rhs) const {
  return Datum(value_*rhs,error_*rhs);
}
```

```
// app9.cpp

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d2 = d1 * 1.5;
  d2.print();

  return 0;
}
```

```
$ g++ -Wall -o app9 app9.cpp Datum.cc
$ ./app9
datum: 1.2 +/- 0.3
datum: 1.8 +/- 0.45
```

What about `double * Datum` ?

- Of course it is natural to do also
  - No reason to limit users to multiply always in a specific way
  - Not natural and certainly not intuitive

```
// app10.cpp

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();

  return 0;
}
```

- But this code does not compile
  - Do you understand why?

```
$ g++ -Wall -o app10 app10.cpp Datum.cc
app10.cpp: In function `int main()':
app10.cpp:10: error: no match for 'operator*' in '5.0e-1 * d1'
```

- Whose operator must be overloaded?
  - operator * of class Datum ?
  - operator * of type double ?

More on What about **double*Datum**

- The following statement

```
double x = 0.5
Datum d3 = x * d1;
```

  is equivalent to

```
double x = 0.5
Datum d3 = x.operator*( d1 );
```

- This means that we need operator * of type double to be overloaded, something like

```
class double {
  public:
   Datum operator*( const Datum& rhs );

};
```

- This is not allowed!
  - Remember: We can not overload operators for built in types!

- So? should we define a new double just for this? Seems crazy!
  - How many times we might need such functionality?

Interactions between **Datum** and **double**

- It's intuitive to multiply a Datum by a double

- No problem... overload the * operator with necessary signature

```
class Datum {
  public:
    Datum operator*( const double& rhs ) const;
  // …

};

Datum Datum::operator*(const double& rhs) const {
  return Datum(value_*rhs,error_*rhs);
}
```

```
Datum Datum::operator*(const double& rhs) const {
  return Datum(value_*rhs,error_*rhs);
}
```

```
// app9.cpp

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d2 = d1 * 1.5;
  d2.print();

  return 0;
}
```

```
$ g++ -Wall -o app9 app9.cpp Datum.cc
$ ./app9
datum: 1.2 +/- 0.3
datum: 1.8 +/- 0.45
```

What about `double * Datum` ?

- Of course it is natural to do also
  - No reason to limit users to multiply always in a specific way
  - Not natural and certainly not intuitive

```
// app10.cpp

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();

  return 0;
}
```

- But this code does not compile
  - Do you understand why?

```
$ g++ -Wall -o app10 app10.cpp Datum.cc
app10.cpp: In function `int main()':
app10.cpp:10: error: no match for 'operator*' in '5.0e-1 * d1'
```

- Whose operator must be overloaded?
  - operator * of class Datum ?
  - operator * of type double ?

More on **double*Datum**

- The following statement

```
double x = 0.5
Datum d3 = x * d1;
```

  is equivalent to

```
double x = 0.5
Datum d3 = x.operator*( d1 );
```

- This means that we need operator * of type double to be overloaded, something like

```
class double {
  public:
   Datum operator*( const Datum& rhs );

};
```

- This is not allowed!
  - Remember: We can not overload operators for built in types!

- So? should we define a new double just for this? Seems crazy!
  - How many times we might need such functionality?

Overloading Operators as Global Functions

- We can define a global operator to do exactly what we need
  - Declaration in header file OUTSIDE class scope
  - Implementation in source file. No scope operator needed
    - Not a member function

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
Datum operator*(const double& lhs, const Datum& rhs);
#endif
```

```
// Datum.cc
#include "Datum.h"
// implement all member functions

// global function!
Datum operator*(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value(), lhs*rhs.error() );
}
```

```
// app10.cpp

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();

  return 0;
}
```

```
$ g++ -Wall -o app10 app10.cpp Datum.cc
$ g++ -Wall -o app10 app10.cpp Datum.cc
$ ./app10
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
```

Another Example: Overloading `operator<<()`

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
ostream& operator<<(ostream& os, const Datum& rhs);
#endif
```

```
// Datum.cc
#include “Datum.h”
// implement all member functions

// global functions
ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << "Datum: " << rhs.value() << " +/- “
     << rhs.error() << endl;
  return os;
}
```

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

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();
  cout << d3 << endl;

  return 0;
}
```

```
$ g++ -Wall -o app4 app4.cpp Datum.cc
$ ./app4
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
Datum: 0.6 +/- 0.15
```

Overhead of operator overloading with global functions

```
// Datum.cc
#include “Datum.h”
// implement all member functions

// global functions
ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << "Datum: " << rhs.value() << " +/- “
     << rhs.error() << endl;
  return os;
}
```

- Global functions don’t have access to private data of objects
- Necessary to call public methods to access information
  - Two calls for each cout or even simple product
- Overhead of calling functions can become significant if a frequently used operator is overloaded via global functions

`friend` Methods

```
#ifndef DatumNew_h
#define DatumNew_h
// DatumNew.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
   // … other methods

    const Datum& operator=( const Datum& rhs );
    bool operator<(const Datum& rhs) const;

    Datum operator*( const Datum& rhs ) const;
    Datum operator/( const Datum& rhs ) const;

    Datum operator*( const double& rhs ) const;

    friend Datum operator*(const double& lhs, const Datum& rhs);
    friend ostream& operator<<(ostream& os, const Datum& rhs);

  private:
    double value_;
    double error_;
};
#endif
```

```
// DatumNew.cc
#include “DatumNew.h”
// implement all member functions

// global functions
Datum operator*(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value_, lhs*rhs.error_ );
}

ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << "Datum: " << rhs.value_ << " +/- "
     << rhs.error_; // NB: no endl!
  return os;
}
```

global methods declared **`friend`** within the class can access private members without being a member functions

```
$ g++ -o app5 app5.cpp DatumNew.cc
$ ./app5
Datum: 0.6 +/- 0.15
```

`static` data and methods

Shared data between Objects

- Objects are instances of a class
  - Each object has a copy of data members that define the attributes of that class
  - Attributes are initialized in the constructors or modified through setters or dedicated member functions

- What if we wanted some data to be shared by ALL instances of class ?
  - Example: keep track of how many instances of a class are created

- How can we do the book keeping?
  - External registry or counter.
    - Where should such a counter live?
    - how can it keep track of ANYBODY creating objects?
    - How to handle the scope problem?

Examples of Sharing Data between Objects

- High energy physics
  - Number of particles created in an interaction

- Perhaps more interesting example for you... Video Games!
  - Think about any of the flavors of WarCraft, StarCraft, Command and Conquer, Civilization, etc.

  - The humor and courage of your units depend on how many of them you have
    - If there are many soldiers you can easily conquer new territory
    - If you have enough resources you can build new facilities or many new manpower
  - How can you keep track of all units and facilities present in all different parts of a complex game?
    - **`static`** might just do it!

`static` Data Members

- static data member is common to ALL instances of a class
  - All object use EXACTLY the same data member
  - There is really only ONE copy of static data members accessed by all objects

```
#ifndef Unit_h
#define Unit_h

#include <string>
#include <iostream>

class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream&
     operator<<(std::ostream& os,
                const Unit& unit);

    static int counter_;

  private:
    std::string name_;
};
#endif
```

```
#include "Unit.h"
using namespace std;

// init. static data member.
// NB: No static keyword necessary.
// Otherwise... compilation error!
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
}

Unit::~Unit() {
  counter_--;
}

ostream&
operator<<(ostream& os, const Unit& unit) {
  os << unit.name_ << " Total Units: "
     << unit.counter_;
  return os;
}
```

Example of `static` data member

```
#include "Unit.h"
using namespace std;

// init. static data member.
// NB: No static keyword necessary.
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
}

Unit::~Unit() {
  counter_--;
}

ostream&
operator<<(ostream& os,
           const Unit& unit) {
  os << unit.name_ << " Total Units: "
     << unit.counter_;
  return os;
}
```

```
int main() {
  Unit john("John");
  cout << john << endl;

  cout << "&john.counter_: "
       << &john.counter_ << endl;

  Unit* fra = new Unit("Francesca");
  Unit pino("Pino");
  cout << "&pino.counter_: "
       << &pino.counter_ << endl;

  cout << "&(fra->counter_): “
       << &(fra->counter_) << endl;
  cout << pino << endl;

  delete fra;

  cout << pino << endl;
```

All objects use the same variable!

constructor and destructor in charge of bookkeeping

```
$ g++ -Wall -o static1 static1.cpp Unit.cc
$ ./static1
John Total Units: 1
&john.counter_: 0x449020
&pino.counter_: 0x449020
&(fra->counter_): 0x449020
Pino Total Units: 3
Pino Total Units: 2
```

Using member functions with static data

```
#ifndef Unit2_h
#define Unit2_h

#include <string>
#include <iostream>

class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream&
    operator<<(std::ostream& os,
               const Unit& unit);

    int getCount() { return counter_; }

  private:
    static int counter_;
    std::string name_;
};
#endif
```

```
#include "Unit2.h"
using namespace std;

// init. static data member
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
}

Unit::~Unit() {
  counter_--;
}

ostream&
operator<<(ostream& os, const Unit& unit) {
  os << "My name is " << unit.name_
     << "! Total Units: " << unit.counter_;
  return os;
}
```

- All usual rules for functions, arguments etc. apply
- Nothing special about public or private static members or functions returning static members

Does it make sense to ask objects for static data?

```
// static2.cpp

#include <iostream>
#include <string>
using namespace std;
#include "Unit2.h"


int main() {
  Unit john("John");
  Unit* fra = new Unit("Francesca");
  cout << "john.getCount(): " << john.getCount() << endl;
  cout << "fra->getCount(): " << fra->getCount() << endl;

  delete fra;

  return 0;
}
```

```
$ g++ -Wall -o static2 static2.cpp Unit2.cc
$ ./static2
john.getCount(): 2
fra->getCount(): 2
```

- counter_ is not really an attribute of any objects
  - It is mostly a general feature of all objects of type Unit
- In principle we would like to know how many Units we have regardless of a specific Unit object
- But how can we use a function if no object has been created?

`static` member functions

- static member functions of a class can be called without having any object of the class!

- Mostly (but not only) used to access static data members
  - static data members exist before and after and regardless of objects
  - static functions play the same role

- Common use of static functions is in utility classes which have no data member
  - Recall InputService in our lab session
  - Some classes are mostly place holders for commonly used functionalities

Example of `static` Member Function

```
#ifndef Unit3_h
#define Unit3_h
#include <string>
#include <iostream>
class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream&
    operator<<(std::ostream& os,
               const Unit& unit);

    static int getCount() { return counter_; }

  private:
    static int counter_;
    std::string name_;
};
#endif
```

```
#include "Unit3.h"
using namespace std;

// init. static data member
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
  cout << "Unit(" << name
       <<") called. Total Units: "
       << counter_ << endl;
}

Unit::~Unit() {
  counter_--;
  cout << "~Unit() called for "
       <<  name_ << ". Total Units: "
       << counter_ << endl;
}

ostream&
operator<<(ostream& os, const Unit& unit) {
  os << "My name is " << unit.name_
     << "! Total Units: " << unit.counter_;
  return os;
}
```

```
int main() {
  cout << "units: " << Unit::getCount() << endl;

  Unit john("John");
  Unit* fra = new Unit("Francesca");

  cout << "john.getCount(): " << john.getCount() << endl;
  cout << "fra->getCount(): " << fra->getCount() << endl;
  delete fra;

  cout << "units: " << Unit::getCount() << endl;

  return 0;
}
```

```
$ g++ -Wall -o static3 static3.cpp Unit3.cc
$ ./static3
units: 0
Unit(John) called. Total Units: 1
Unit(Francesca) called. Total Units: 2
john.getCount(): 2
fra->getCount(): 2
~Unit() called for Francesca. Total Units: 1
units: 1
~Unit() called for John. Total Units: 0
```

Typical Error with `static` Member Functions

```
#ifndef Unit3_h
#define Unit3_h

#include <string>
#include <iostream>

class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream& operator<<(std::ostream& os, const Unit& unit);

    static int getCount() const { return counter_; }

  private:
    static int counter_;
    std::string name_;
};
#endif
```

```
$ g++ -Wall -c Unit3.cc
In file included from Unit3.cc:1:
Unit3.h:15: error: static member function `static int Unit::getCount()'
 cannot have `const' method qualifier
```

Typical error! static functions can not be const! Since they can be called without any object no reason to make them constant

Features of static methods

- They can't be constant
  - static functions operate independently from any object
  - They can be called before and after any object is created

- They can not access non-static data members of the class
  - non-static data members characterize objects
  - how can data members be modified if no object created yet?

```
class Unit {
  public:

    static int getCount() {
      name_ = "";
      return counter_;
    }

};
```

```
$ g++ -c  Unit4.cc
In file included from Unit4.cc:1:
Unit4.h: In static member function `static int Unit::getCount()‘:
Unit4.h:21: error: invalid use of member `Unit::name_' in
static member function
Unit4.h:15: error: from this location
```

- No access to this pointer in static functions
  - As usual, this is specific to individual objects

Revisiting Our Application

```
#ifndef Calculator_h
#define Calculator_h

#include <vector>
#include "Datum.h"
#include "Result.h"

class Calculator {
 public:
  Calculator();
  void setData(std::vector<Datum>& data);

  Result weightedAverage();
  Result arithmeticAverage();
  Result geometricAverage();
  Result fancyAverage();

private:
  std::vector<Datum> data_;
};
#endif
```

```
#ifndef InputService_h
#define InputService_h
#include <vector>
#include "Datum.h"

class InputService {
 public:
   InputService();
   std::vector<Datum> readDataFromUser();
 private:
};
#endif
```

- Do you think we could use static in these classes?
- Should we use static data members or static member functions?
- Are there any benefits?

`static` Methods in Utility Classes

```
#ifndef Calculator_h
#define Calculator_h

#include <vector>
#include "Datum.h"
#include "Result.h"

class Calculator {
 public:
  Calculator();
  // void
  //setData(std::vector<Datum>& data);

  static Result
    weightedAverage(const std::vector<Datum>& dati);
  static Result
    arithmeticAverage(const std::vector<Datum>& dati);
  static Result
    geometricAverage(const std::vector<Datum>& dati);
  static Result
    fancyAverage(const std::vector<Datum>& dati);

private:
  //std::vector<Datum> data_; no data needed!
};
#endif
```

```
#ifndef InputService_h
#define InputService_h
#include <vector>
#include "Datum.h"

class InputService {
 public:
   InputService();
   static std::vector<Datum> readDataFromUser();
 private:
};
#endif
```

- Classes with no data member and (only) static methods are often called utility classes

Current Implementation of Our Application

```
// app1.cc
#include <vector>

class Datum; // basic data object
class InputService; // class dedicated to handle input of data
class Calculator; // impelments various algorithms
class Result; // how is Result different from Datum ?

int main() {

  InputService input;
  std::vector<Datum> dati = input.readDataFromUser();

  Calculator calc;
  calc.setData( dati );

  Result r1 = calc.weightedAverage();
  Result r2 = calc.arithmeticAverage();
  Result r3 = calc.geometricAverage();
  Result r3 = calc.fancyAverage();

  r1.display();

  return 0;
}
```

Application Revisited

```
// app1new.cc
#include <vector>

class Datum; // basic data object
class InputService; // class dedicated to handle input of data
class Calculator; // impelments various algorithms
class Result; // how is Result different from Datum ?

int main() {

  //InputService input;    // unnecessary!
  std::vector<Datum> dati = InputService::readDataFromUser();

  //Calculator calc;    // not necessary anymore!
  //calc.setData( dati );

  Result r1 = Calculator::weightedAverage(dati);
  Result r2 = Calculator:: arithmeticAverage(dati);
  Result r3 = Calculator:: geometricAverage(dati);
  Result r3 = Calculator:: fancyAverage(dati);

  r1.display();

  return 0;
}
```