Overloading Operators
friend functions
static data and methods

Shahram Rahatlou

Computing Methods in Physics

`http://www.roma1.infn.it/people/rahatlou/cmp/`

Anno Accademico 2018/19

SAPIENZA
UNIVERSITÀ DI ROMA

Today's Lecture

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

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);
}
```

```
// app1.cc
#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 app1 app1.cc Datum.cc
$ ./app1
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 incorrect

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);
}
```

```
// app2.cc

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

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

  return 0;
}
```

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

```
$ g++ -Wall -o app2 app2.cc Datum.cc
$ ./app2
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

```
// app3.cc

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 app3 app3.cc Datum.cc
app3.cpp: In function `int main()':
app3.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?

A new Global Function

- We can define a global function to do what we need
  - Declaration in header file OUTSIDE class scope
  - Implementation in source file
- It works but not as natural to use

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

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

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

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

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

```
$ g++ -Wall -o app3 app3.cc Datum.cc
$ ./app3
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
```

```
// app3.cc

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

  Datum d3 = productDoubleDatum(0.5,d1);
  d3.print();

  return 0;
}
```

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() );
}
```

```
// app4.cc

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

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

  return 0;
}
```

```
$ g++ -Wall -o app4 app4.cc Datum.cc
$ ./app3
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
```

Another Example: Overloading `operator<<()`

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

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

```
#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
```

```
// app5.cc
#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 app5 app5.cc Datum.cc
$ ./app5
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
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 app6 app6.cc DatumNew.cc
$ ./app6
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
0.6 +/- 0.15
```

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: " << d1 << endl;
  cout << "d3: " << d3 << endl;

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

  return 0;
}
```

Comparison based on the value_

error_ does not affect the comparison do you agree?

```
$ g++ -Wall -o app7 app7.cc Datum.cc
$ ./app7
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 app8 app8.cc Datum.cc
app8.cc: In function `int main()':
app8.cc:12: error: no match for 'operator+=' in 'd1 += d3'
app8.cc: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

Datum::operator+=()

```
class Datum {
 //…
   Datum operator+( const Datum& rhs ) const;
   const Datum& operator+=( const Datum& rhs );

};
```

```
// app9.cc
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( 3.1, 0.4 );

  d1 += d2;
  d1.print();

  return 0;
}
```

```
// Datum.cc

const Datum& Datum::operator+=(const Datum& rhs) {
  value_ += rhs.value_;
  error_ = sqrt( rhs.error_*rhs.error_ + error_*error_ );
  return *this;
}
```

```
$ g++ -o app9 app9.cc Datum.cc
$ ./app9
d1: 1.2 +/- 0.3 d2: 3.1 +/- 0.4
d1+d2 = 4.3 +/- 0.5
```

Why `const& Datum operator+=()` ?

- Why not return by value?

```
Datum Datum::operator+=(const Datum& rhs) {
  double value = value_ + rhs.value_;
  double error = sqrt( rhs.error_*rhs.error_ + error_*error_ );
  return Datum(value,error);
}
```

- Why not return simple non-const reference?
  - non-const will also work almost always
  - use cases why const is needed not very common

```
Datum& Datum::operator+=(const Datum& rhs) {
  value_ += rhs.value_;
  error_ = sqrt( rhs.error_*rhs.error_ + error_*error_ );
  return *this;
}
```

Problem with Returning by-value

```
class Foo {

public:
  Foo() { name_ = ""; x_= 0; }
  Foo(const std::string& name, const double x) { name_ = name; x_
= x; }
  double value() const { return x_; }
  std::string name() const { return name_; }

  Foo operator=(const Foo& rhs) {
    Foo aFoo(rhs.name_,rhs.x_);
    cout << "In Foo::operator=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

  Foo operator+=(const Foo& rhs) {
    Foo aFoo(std::string(name_+"+"+rhs.name_), x_ + rhs.x_);
    cout << "In Foo::operator+=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

  void reset() {
    x_ = 0.;
    name_ = "";
  }

private:
  double x_;
  std::string name_;
};

// global function
ostream&  operator<<(ostream& os, const Foo& foo) {
  os << "Foo name: "  << foo.name() << " value: " << foo.value()
     << " address: " << &foo;
  return os;
}
```

```
// fooapp3.cc
int main() {
  Foo f1("f1",1.);
  Foo f2("f2",2.);
  Foo f3("f3",3.);

  cout << " before f1+=f2 " << endl;
  f1 += f2;
  cout << "after f1+=f2\n" << f1 << endl;

  cout << " before f1 = f3 " << endl;
  f1 = f3;
  cout << "after f1 = f3\n" << f1 << endl;

  return 0;
}
```

```
$ g++ -o fooapp3 fooapp3.cc
$ ./fooapp3
 before f1+=f2
In Foo::operator+=: value: 3, name: f1+f2, &aFoo: 0x7ffeec53e6e8
after f1+=f2
Foo name: f1 value: 1 address: 0x7ffeec53e7a0
 before f1 = f3
In Foo::operator=: value: 3, name: f3, &aFoo: 0x7ffeec53e6c8
after f1 = f3
Foo name: f1 value: 1 address: 0x7ffeec53e7a0
```

Assignment never happens! the left-hand-side is never modified by the operators

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;

  return 0;
}
```

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

- 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
  - Some classes are mostly place holders for commonly used functionalities
    - we will see a number of such classes in ROOT for mathematical

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
```

```
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;
}
```

```
#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;
}
```

```
$ 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
```

Common 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 cannot 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
  - Recall: **`this`** is specific to individual objects

`static` Methods in Utility Classes

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

```
#ifndef Calculator_h
#define Calculator_h

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

class Calculator {
 public:
  Calculator();

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

};
#endif
```

Example of Application

- Application to compute weighted average and error
  - Application must accept an arbitrary number of input data
  - Each data has a central value x and uncertainty
  - Compute weighted average of input data and uncertainty on the average
- Possible extensions
  - Provide different averaging methods
  - Uncertainties could be also asymmetric ($x\ ^{+\sigma 1}_{-\sigma 2}$)
  - Consider also systematic errors
  - Compute correlation coefficient and take it into account when computing the average and its uncertainty
  - Use ROOT to make histogram of data points and plot a coloured band to indicate the average and its uncertainty overlaid on the histogram

Possible implementation

```
// wgtavg.cc
#include <vector>
#include <iostream>

#include “Datum.h"// basic data object
#include “InputService.h" // class dedicated to handle input of data
#include “Calculator.h” // impelments various algorithms

using std::cout;
using std::endl;

int main() {

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

  Datum r1 = Calculator::weightedAverage(dati);
  cout << “weighted average: “ << r1 << endl;

  Datum r2 = Calculator::arithmeticAverage(dati);

  Datum r3 = Calculator::geometricAverage(dati);


  return 0;
}
```

Interface of Classes

```
#ifndef Calculator_h
#define Calculator_h

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

class Calculator {
 public:
  Calculator();

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

};
#endif
```

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    double value() ;
    double error() ;
    double significance();
  private:
    double value_;
    double error_;
};
#endi
```

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

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

You see the interface but don’t know how the methods are implemented!

Application for Weighted Average

```
// wgtavg.cc
#include <vector>
#include <iostream>

#include “Datum" // basic data object
#include "InputService" // class dedicated to handle input of data
#include "Calculator" // impelments various algorithms

using std::cout;
using std::endl;

int main() {

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

  Datum r1 = Calculator::weightedAverage(dati);
  cout << “weighted average: “ << r1 << endl;

  Datum r2 = Calculator::arithmeticAverage(dati);

  Datum r3 = Calculator::geometricAverage(dati);


  return 0;
}
```

```
$ g++ -c InputService.cc
$ g++ -c Datum.cc
$ g++ -c Calculator.cc
$ g++ -o wgtavg wgtavg.cpp InputService.o Datum.o Calculator.o
```

Questions

- What about reading a file of data?
  - how to communicate the file name and where?
    - in main or in InputService?
- Do you need any arguments for these functions?
- Who should compute correlation?
  - should be stored?
    - if yes, where?
  - should the data become an attribute of some object?
    - If yes, in which class?
- what about generating pseudo-data to test our algorithms?
  - where would this generation happen?
  - in the main() method or in some class?