Static Data Members
Enumeration
std::pair, std::vector, std::map

Shahram Rahatlou

Computing Methods in Physics

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

Anno Accademico 2019/20

SAPIENZA
UNIVERSITÀ DI ROMA

Class Datum

- Use static data member to implement operator == for Datum
  - Implement also <= and >= with similar logic

```
class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    ~Datum() { };

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    Datum operator+( const Datum& rhs ) const;
    const Datum& operator+=( const Datum& rhs );

    Datum sum( const Datum& rhs ) const;

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


    bool operator==(const Datum& rhs) const;
    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 std::ostream& operator<<(std::ostream& os, const Datum& rhs);

    static void setTolerance(double val) { tolerance_ = val; };

  private:
    double value_;
    double error_;
    static double tolerance_;
};
```

```
#include "Datum.h"
#include <iostream>
#include <cmath>
using std::cout;
using std::endl;
using std::ostream;

double Datum::tolerance_ = 1e-4;

// functions …

bool Datum::operator==(const Datum& rhs) const {
  return (fabs(value_-rhs.value_)< tolerance_ &&
               fabs(error_-rhs.error_)< tolerance_    );
}
```

Using Datum::tolerance_

```
// app1.cc
#include "Datum.h"
#include <iostream>
using std::cout;
using std::endl;

int main() {

    Datum d1(-1.1,0.1);
    Datum d2(-1.0, 0.2);
    Datum d3(-1.11, 0.099);
    Datum d4(-1.10001, 0.09999999);

    cout << "d1: " << d1 << endl;
    cout << "d2: " << d2 << endl;
    cout << “d3: " << d3 << endl;
    cout << "d4: " << d4 << endl;


    for(double eps = 0.1; eps > 1e-8; eps /= 10) {
      Datum::setTolerance( eps );
      cout << "Datum tolerance  = " << eps << endl;

      if( d1 == d2 ) cout << "\t d1 same as d2" << endl;
      if( d1 == d3 ) cout << "\t d1 same as d3" << endl;
      if( d1 == d4 ) cout << "\t d1 same as d4" << endl;
    }
    return 0;
}
```

```
$ g++ -o /tmp/app app1.cc Datum.cc
$ /tmp/app
d1: -1.1 +/- 0.1
d2: -1 +/- 0.2
d3: -1.11 +/- 0.099
d4: -1.10001 +/- 0.1
Datum tolerance  = 0.1
          d1 same as d3
          d1 same as d4
Datum tolerance  = 0.01
          d1 same as d4
Datum tolerance  = 0.001
          d1 same as d4
Datum tolerance  = 0.0001
          d1 same as d4
Datum tolerance  = 1e-05
          d1 same as d4
Datum tolerance  = 1e-06
Datum tolerance  = 1e-07
Datum tolerance  = 1e-08
```

IO manipulators

```
//app2.cc
#include "Datum.h"
#include <iostream>
#include <iomanip>      // std::setprecision

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

int main() {

    Datum d1(-1.1,0.1);
    Datum d2(-1.0, 0.2);
    Datum d3(-1.101, 0.099);
    Datum d4(-1.10001, 0.09999999);

    cout << "d1: " << std::setprecision(9) << d1 << endl;
    cout << "d2: " << std::setprecision(9) << d2 << endl;
    cout << "d3: " << std::fixed << d3 << endl;
    cout << "d4: " << std::fixed << d4 << endl;


    for(double eps = 0.1; eps > 1e-8; eps /= 10) {
      Datum::setTolerance( eps );
      cout << "Datum tolerance  = " << std::scientific << eps << endl;

      if( d1 == d2 ) cout << "\t d1 same as d2" << endl;
      if( d1 == d3 ) cout << "\t d1 same as d3" << endl;
      if( d1 == d4 ) cout << "\t d1 same as d4" << endl;
    }
    return 0;
}
```

```
$ g++ -o /tmp/app app2.cc Datum.cc
$ /tmp/app
d1: -1.1 +/- 0.1
d2: -1 +/- 0.2
d3: -1.101000000 +/- 0.099000000
d4: -1.100010000 +/- 0.099999990
Datum tolerance  = 1.000000000e-01
          d1 same as d3
          d1 same as d4
Datum tolerance  = 1.000000000e-02
          d1 same as d3
          d1 same as d4
Datum tolerance  = 1.000000000e-03
          d1 same as d4
Datum tolerance  = 1.000000000e-04
          d1 same as d4
Datum tolerance  = 1.000000000e-05
          d1 same as d4
Datum tolerance  = 1.000000000e-06
Datum tolerance  = 1.000000000e-07
Datum tolerance  = 1.000000000e-08
```

Enumerators

Enumerators

- Enumerators are set of integers referred to by identifiers
- There is natural need for enumerators in programming
  - Months: Jan, Feb, Mar, …, Dec
  - Fit Status: Successful, Failed, Problems, Converged
  - Shapes: Circle, Square, Rectangle, …
  - Colors: Red, Blue, Black, Green, …
  - Coordinate system: Cartesian, Polar, Cylindrical
- Enumerators make the code more user friendly
  - Easier to understand human identifiers instead of hardwired numbers in your code!
- You can redefine the value associated to an identifier w/o changing your code

Example of Enumeration

```
// enum1.cc
#include <iostream>
using namespace std;

int main() {
  enum FitStatus  { Succesful, Failed, Problems, Converged };

  FitStatus status;

  status = Succesful;
  cout << "Status: " << status << endl;

  status = Converged;
  cout << "Status: " << status << endl;

  return 0;
}
```

By default the first identifier is assigned value 0

Don't forget this one!

```
$ g++ -o /tmp/enum1 enum1.cc
$ /tmp/enum1
Status: 0
Status: 3
```

enums can be used as integers but not vice versa!

Another Example of Enumeration

- You can use arbitrary integer values for each of your identifiers
  - for example use RGB codes for main colours

```
// enum2.cc
#include <iostream>
using namespace std;

int main() {
  enum Color  { Red=1, Blue=45, Yellow=17, Black=342 };

 Color col;

  col = Red;
  cout << "Color: " << col << endl;

  col = Black;
  cout << "Color: " << col << endl;


  return 0;
}
```

```
$ g++ -o /tmp/app enum2.cc
$ /tmp/app
Color: 1
Color: 342
```

Common errors with enumeration

```
// enum3.cc
#include <iostream>
using namespace std;

int main() {
  enum Color  { Red=1, Blue=45, Yellow=17, Black=342 };

 Color col;

  col = Red;
  cout << "Color: " << col << endl;

  col = Black;
  cout << "Color: " << col << endl;

  col = 45; //assign int to enum

  int i = Red;

  return 0;
}
```

Can’t assign an int to an enum!

But you can assign an enum to an int

```
$ g++ -o /tmp/app enum3.cc
enum3.cc:16:9: error: assigning to 'Color' from incompatible type 'int'
  col = 45; //assign int to enum
        ^~
1 error generated.
```

Enumeration in Classes

- Use complete qualifier including namespace and class to use **public** enumerators

```
#ifndef Fitter_h_
#define Fitter_h_
// Fitter.h
namespace analysis {

  class Fitter {
    public:
      enum Status { Succesful=0,
                    Failed,
                    Problems };

      Fitter() { };

      Status fit() {
        return Succesful;
      }
    private:
  }; // class Fitter
} //namespace
#endif
```

```
//enum4.cc
#include "Fitter.h"
#include <iostream>
using namespace std;

int main() {

  analysis::Fitter myFitter;

  analysis::Fitter::Status stat =
                          myFitter.fit();

  if( stat == analysis::Fitter::Succesful ) {
    cout << "fit succesful!" << endl;
  } else {
    cout << "Fit had problems ... status = "
         << stat << endl;
  }

  return 0;
}
```

```
$ g++ -o /tmp/app enum4.cc
$ /tmp/app
fit succesful!
```

Enumerators and strings

- No automatic conversion from enumeration to strings

- You can use vectors of strings or std::map to assign string names to enumeration states

```
// color.cc

#include <iostream>
#include <map>
using std::cout;
using std::endl;

int main() {
  enum Color  { Red=1, Blue=45,
                Yellow=17, Black=342 };

  Color col;

  // using std::map
  std::map<int,std::string> colname;
  colname[Red] = std::string("Red");
  colname[Black] = std::string("Black");

  col = Red;
 cout << "Color: " << colname[col] << endl;

  return 0;
}
```

```
$ g++ -o /tmp/app color.cc
$ /tmp/app
Color: Red
```

std::map

http://www.cplusplus.com/reference/map/map/

https://en.cppreference.com/w/cpp/container/map

class template

std::map

<map>

```
template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`map` containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Maps are typically implemented as *binary search trees*.

std::pair

https://en.cppreference.com/w/cpp/utility/pair

http://www.cplusplus.com/reference/utility/pair/

cppreference.com   Create account   Search

Page Discussion   View Edit History

C++ / Utilities library / **std::pair**

std::pair

Defined in header <utility>

```
template<
    class T1,
    class T2
> struct pair;
```

`std::pair` is a struct template that provides a way to store two heterogeneous objects as a single unit. A pair is a specific case of a `std::tuple` with two elements.

If `std::is_trivially_destructible_v<T1> && std::is_trivially_destructible_v<T2>` is `true`, the destructor of `pair` is trivial. (since C++17)

Template parameters

T1, T2 - the types of the elements that the pair stores.

Application with **map**, **pair**, **vector**

```
// map1.cc
#include<iostream>
#include<vector>
#include<map>
#include <utility>      // std::pair, std::make_pair
#include<string>

#include "Student.h"

int main() {

  // pair object to associate two different types of data
  std::pair< std::string, int> grade = std::make_pair("MQR", 24);

  // grades of a student stored in a vector
  std::vector< std::pair< std::string, int> > grades; //
  grades.push_back( std::make_pair("MQR", 26) );
  grades.push_back( std::make_pair("Phys Lab", 27) );
  grades.push_back( std::make_pair("Cond Matt", 23) );

  Student gino("Gino", 110998);


  // databases of grades of all students
  //    key: student     value: grades
  std::map<Student,  std::vector< std::pair< std::string, int> >  > exams;
  exams[gino] = grades;

  Student tina("Tina", 121001);
  grades.clear(); // delete all previous values in the vector
  grades.push_back( std::make_pair("MQR", 29) );
  grades.push_back( std::make_pair("Phys Lab", 28) );
  grades.push_back( std::make_pair("Cond Matt", 25) );

  exams[tina] = grades;

  // loop over entries in the map
  for(std::map<Student,  std::vector< std::pair< std::string, int> >  >::iterator it = exams.begin(); it != exams.end(); it++ ) {

    // print out student data
    std::cout << "Student name: " << (it->first).name() << "\t id: " << (it->first).id() << std::endl;

    // loop over list of exams
    for(std::vector< std::pair< std::string, int> >::iterator vit = (it->second).begin(); vit != (it->second).end(); vit++) {
      // print name of each exams and relative grade
      std::cout << "\t Subject: " << vit->first << "\t grade: " << vit->second << std::endl;
    } // end: loop over grades

  } // end: loop over students

  return 0;

}
```

```
#ifndef Student_h
#define Student_h

#include<string>

class Student {
public:
  Student(const std::string& name, int id) {
    name_ = name;
    id_ = id;
  }

  bool operator<(const Student& rhs) const {
    return id_ < rhs.id_;
  }

  std::string name() const {
    return name_;
  }

  int id() const {
    return id_;
  }

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

```
$ g++ -o /tmp/app map1.cc
$ /tmp/app
Student name: Gino  id: 110998
       Subject: MQR       grade: 26
       Subject: Phys Lab  grade: 27
       Subject: Cond Matt       grade: 23
Student name: Tina  id: 121001
       Subject: MQR       grade: 29
       Subject: Phys Lab  grade: 28
       Subject: Cond Matt       grade: 25
```

Class Vector 3D

- How many and what type of data members?
- How can we handle different coordinate systems?
  - are the classes different?
  - do you need different attributes?
  - is it only a setup problem?
  - How do you distinguish polar vector from cartesian?
  - can you ask phi() and theta() to a cartesian vector?