Use case for `std::map`, `std::pair`, and `std::vector`

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

http://www.cplusplus.com/reference/utility/pair
https://en.cppreference.com/w/cpp/utility/pair

class template
std::pair

<utility>

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

Pair of values

This class couples together a pair of values, which may be of different types (`T1` and `T2`). The individual values can be accessed through its public members `first` and `second`.

Pairs are a particular case of `tuple`.

Template parameters

T1
Type of member `first`, aliased as `first_type`.

T2
Type of member `second`, aliased as `second_type`.

Application with `map`, `pair` and `vector`

Based on `examples/07/map1.cpp` and `Examinee.h`

```cpp
#ifndef Examinee_h
#define Examinee_h

#include<string>

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

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

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

  int id() const {
    return id_;
  }

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

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

#include "Examinee.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 an examinee 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) );

  Examinee gino("Gino", 110998);

  // databases of grades of all examinees
  //    key: examinee     value: grades
  std::map<Examinee, std::vector< std::pair<std::string, int> > > exams;
  exams[gino] = grades;

  Examinee 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; //CONTINUES...
```

Application with `map`, `pair` and `vector`

Based on `examples/07/map1.cpp` and `Examinee.h`

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

    // print out examinee data
    std::cout << "Examinee 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 examinees

  return 0;

}
```

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

`static` data members and methods

Shared Data Among 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 and modified via setters or dedicated member functions
- What if we wanted some data to be shared by ALL instances of a class?
  - Textbook 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 Among Objects

- High energy physics
  - Production vertex for particles in a collision
- A more fun example… Video Games!
  - WarCraft, StarCraft, Command and Conquer, Civilization, Halo, Clash of Clans, Fortnite, 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 increase personpower
- How can you keep track of all units and facilities present in all different parts of a complex game?
    - `static` might just do it!

Tolerance for Comparing `Datum`

```
Datum d1(1.01, 0.131);
Datum d2(0.99, 0.128);

if( d1 == d2 ) {
 // do something
 ...
}
```

- Comparison between two `Datum` objects
- When should == be true ?
- In a physics problem you often define a numerical tolerance, a detector precision, etc., when comparing quantities or measurements
  - All `Datum` objects could share a same tolerance for comparisons

`static` Data Members

Based on `examples/07/Unit.h` and `Unit.cc`

- `static` data member is common to ALL instances of a class
  - All objects 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

Based on `examples/07/static1.cpp` and `Unit.*`

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


int main() {
  Unit giorgio("Giorgio Parisi");
  cout << giorgio << endl;

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

  Unit* gan = new Unit("Gandalf");
  Unit neo("Neo");
  cout << "&neo.counter_: " << &neo.counter_ << endl;

  cout << "&(gandalf->counter_): " << &(gan->counter_) << endl;
  cout << neo << endl;

  delete gan;

  cout << neo << endl;

  return 0;
}
```

```
$ g++ -Wall -o static1 static1.cpp Unit.cc
$./static1
Giorgio Parisi Total Units: 1
&giorgio.counter_: 0x1040640f0
&neo.counter_: 0x1040640f0
&(gandalf->counter_): 0x1040640f0
Neo Total Units: 3
Neo Total Units: 2
```

All objects use the same variable!

Constructor and destructor in charge of bookkeeping

Using Member Functions with `static` Data

Based on `examples/07/Unit2.h` and `Unit2.cc`

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

```
#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() const { 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_--;
}

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

Does it Make Sense to Ask Objects for `static` Data?

Based on `examples/07/static2.cpp` and `Unit2.*`

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


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

  delete fra;

  return 0;
}
```

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

- `counter_` is not really an attribute of any object: it is 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
  - Some classes are mostly place holders for commonly used functionalities
    - We will see a number of such classes in `ROOT`

Example of `static` Member Function

Based on `examples/07/static3.cpp` and `Unit3.*`

```
#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;
}

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

```
#ifndef Unit3_h
#define Unit3_h

//...same Unit2.h

    static int getCount()

//...same Unit2.h
```

```
#include <iostream>
#include <string>
using namespace std;
#include "Unit3.h"

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

  //...same static2.cpp

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

  return 0;
}
```

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

Common Error with `static` Member Functions

Based on `examples/07/Unit3.cc` and `Unit3.h`

```
$ g++ -Wall -o static3 static3.cpp Unit3.cc
./Unit3.h:15:27: error: static member function cannot have 'const'
qualifier
    static int getCount() const { return counter_; }
                          ^~~~~~
1 error generated.
```

```
#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
```

`static` functions cannot be `const`!
They can be called without any object so there is no reason to make them constant

Features of `static` Methods

Based on `examples/07/Unit4.cc` and `Unit4.h`

- **They cannot be constant**
  - `static` functions operate independently from any object
  - They can be called before and after any object is created

```
class Unit{
  public:
    //Same as before...

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

    //Same as before...
};
```

- **They have no access to `this` pointer**
  - Recall: `this` is specific to individual objects
- **They cannot access non-static data members of the class**
  - Non-static data members characterize objects: how can data members be modified if no object was created yet?

```
$ g++ -Wall -o Unit4.cc
./Unit4.h:16:7: error: invalid use of member 'name_' in static member function
      name_ = "";
      ^~~~~
1 error generated.
```

`static` Methods in Utility Classes

Based on `examples/07/Calculator.h`

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

};

#endif
```

Example of Application

- Application to compute weighted average and error
  - Must accept arbitrary number of input data, each having a central value and an 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

Possible Implementation

Based on `examples/07/wgtavg.cpp`

```
#include <vector>
#include <iostream>

#include "Datum.h" // basic data object
#include "InputService.h" // class to handles input of data
#include "Calculator.h" // implements 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

Based on `examples/07/Datum.h`, `Calculator.h`, `InputService.h`

```
#ifndef Datum_h
#define Datum_h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    double value();
    double error();
    double significance();
    friend ostream& operator<<(ostream& os, const Datum& rhs);
  private:
    double value_;
    double error_;
};

#endif
```

```
#ifndef Calculator_h
#define Calculator_h

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

class Calculator {
 public:
  Calculator();

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

};

#endif
```

```
#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 do not know how the methods are implemented!

Application for Weighted Average

Based on `examples/07/wgtavg.cpp`

```
#include <vector>
#include <iostream>

#include "Datum.h" // basic data object
#include "InputService.h" // class to handles input of data
#include "Calculator.h" // implements 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 data file?
  - 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 it 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?

Back to Class `Datum`

Based on `examples/07/Datum.*`

- Use static data member to implement operator == for `Datum`

Implement <= and >= similarly

```
#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_);
}
```

```
#ifndef Datum_h
#define Datum_h
#include <iostream>

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_;
};

#endif
```

Using `Datum::tolerance_`

Based on `examples/07/DatumApp1.cpp`

```
#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++ -Wall -o DatumApp1 DatumApp1.cpp Datum.cc
$ ./DatumApp1
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
```

I/O manipulators

Based on examples/07/DatumApp2.cpp

```
#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++ -Wall -o DatumApp2 DatumApp2.cpp Datum.cc
$ ./DatumApp2
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
```