C++ standard template Library and advanced topics

andrea.rizzi@unipi.it

Standard Template library

- C++ comes with a default library of tools written with “templates”
  - I.e. tools that can be used on multiple classes
  - Some tools may require that the concrete class has some specific properties (e.g. it may need that the default constructor is available or that an operator< is available)
- Mostly used to implement
  - Containers
  - Algorithms acting on containers
- STL tools are defined in the “std” namespace
  - You already used “std::cout” and “std::endl” from the STL
- Include files do not end with “.h”

Vectors

- C arrays have fixed length… and are actually just pointers (how do you copy them?)
- STL introduces std::vector
  - Dynamic size
  - Can append to vector with push_back (or emplace_back)
  - Can initialize all elements to a given value

```
std::vector<float> v(100,-1372);
```

- operator[] is not protected against out of boundaries (like for C arrays)
- Function at() checks the boundary

```
#include <vector>

int main()
{
  std::vector<float> v(100);
  float a[100];

  v[1]=75.;
  a[1]=75;

  //vectors can be copied!
  std::vector<float> anotherv=v;

  //vectors can be resized!
  v.resize(120);
  v[105]=19;

  //we can append to vectors
  v.push_back(1273);
}
```

```
  v[100000]=1;   // operator[i] is faster
//Segmentation fault (core dumped)

  v.at(100000)=1; // at(i) is slower
//terminate called after throwing an instance of 'std::out_of_range'
//  what():  vector::_M_range_check: __n (which is 100000) >= this->size() (which is 121)
//Aborted (core dumped)
```

Iteration

- STL containers can be accessed via iterators
  - For vectors it does not really change much vs “for on indices”
  - Much more relevant for other containers (think about e.g. linked lists)
- Syntax improved a lot in C++11/14/17 !! (next week)

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
  vector<float> v(5);
  for(size_t i = 0; i < v.size() ; i++){
    v[i]=i*10;
  }

  for(vector<float>::iterator it = v.begin() ; it!=v.end() ; it++){
     cout << *it << " at iteration " << it-v.begin() << endl;
  }
  cout <<"backward"  << endl;
  for(vector<float>::const_reverse_iterator it = v.rbegin() ; it!=v.rend() ; it++){
     cout << *it << " at iteration " << it-v.rbegin() << endl;
  }
}
```

More on iterators

- Ranges for iterations are typically given as a [begin,end) pair [aka a “range”]
  - Where begin() is included, and end is “out of boundary”
  - operator * and -> are used to access the “pointed value”
- So iterators are like pointers
  - In fact, good old pointers are a simple form of iterator

http://cpp.sh/7iyy5

```
#include <iostream>

int main ()
{
  float a[] = {10,2,4,5};
  typedef float * floatiterator ;
  floatiterator begin= a;
  floatiterator end= a+4;

  for(floatiterator it=begin; it!=end;it++){
    std::cout << *it << " at position " << (it-begin) << std::endl;
  }

}
```

std::pair

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

int main () {
  std::pair <std::string,double> product1;                     // default constructor
  std::pair <std::string,double> product2 ("tomatoes",2.30);   // value init
  std::pair <std::string,double> product3 (product2);          // copy constructor

  product1 = std::make_pair(std::string("lightbulbs"),0.99);   // using make_pair (move)

  product2.first = "shoes";                  // the type of first is string
  product2.second = 39.90;                   // the type of second is double

  std::cout << "The price of " << product1.first << " is $" << product1.second << '\n';
  std::cout << "The price of " << product2.first << " is $" << product2.second << '\n';
  std::cout << "The price of " << product3.first << " is $" << product3.second << '\n';
  return 0;
}
```

Map

- Similar to python dict, but
  - It is ordered
  - The types of key and value are fixed
  - operator[] creates the element if it does not exist!

Python:

```
>>> a={}
>>> a["2"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '2'
```

```
a={"2":75}

for k,v in a.items():
  print(k,v,sep=" : ")
```

```cpp
#include <iostream>
#include <map>

using namespace std;
int main()
{
  map<int,float> dict;
  dict[10]=-1.;
  dict[7]=1.13;
  cout << "item 2 : " << dict[2] << endl;
  //hate this below? you will love C++17!
  for(map<int,float>::const_iterator it=dict.begin(); it!=dict.end() ; it++){
    cout << it->first << " => " << it->second << endl;
  }
}
```

http://cpp.sh/33dx5

Output

```
item 2 : 0
2 => 0
7 => 1.13
10 => -1
```

Output is sorted on the key!!

More on maps

- Insertion in maps
  - map[key]=value
    - overwrites if exists
  - map::insert(std::pair(key,value))
    - does not insert if already exists
- map::find looks for a key
  - return map::end() if not found
  - If found, a valid iterator
- std::multimap
  - Keys are not unique
  - So cannot use operator[]
- From C++11 also
  - std::unordered_map
  - std::unordered_multimap

http://cpp.sh/9rx55

```
#include <iostream>
#include <map>
int main ()
{
  std::map<char,int> mymap;

  // first insert function version (single parameter):
  mymap.insert ( std::pair<char,int>('a',100) );
  mymap.insert ( std::pair<char,int>('z',200) );

  std::pair<std::map<char,int>::iterator,bool> ret;
  ret = mymap.insert ( std::pair<char,int>('z',500) );
  if (ret.second==false) {
    std::cout << "element 'z' already existed";
    std::cout << " with a value of " << ret.first->second << '\n';
  }

  std::map<char,int>::iterator found=mymap.find('a');
  if(found != mymap.end() )
  {
     std::cout << "Found with value " << found->second << std::endl;
  } else {
     std::cout << "Not found :-( " << std::endl;
  }
}
```

Set

http://cpp.sh/6sn5y

- Sets are unique container of elements
  - I.e. like a map, but keys only
- It is sorted (so checking if an element belong to the set is fast)
- The syntax is similar to map but rather than the pair<key,value> it just contains the keys

```
empty (1)   explicit set (const key_compare& comp = key_compare(),
                          const allocator_type& alloc = allocator_type());
            template <class InputIterator>
range (2)     set (InputIterator first, InputIterator last,
                   const key_compare& comp = key_compare(),
                   const allocator_type& alloc = allocator_type());
copy (3)    set (const set& x);
```

```cpp
#include <iostream>
#include <set>

bool fncomp (int lhs, int rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::set<int> set1;                              // empty set of ints

  int myints[]= {10,20,30,40,50};
  std::set<int> set2(myints,myints+5);            // range
  std::set<int> set3(set2);                       // a copy of second
  std::set<int> set4(set2.begin(), set2.end());   // iterator ctor.
  std::set<int,classcomp> set5;                   // class as Compare
  std::set<int,bool(*)(int,int)> set6 (fncomp);   // function pointer as Compare

  return 0;
}
```

Containers of custom objects

http://cpp.sh/4jw5n

```
#include <vector>
#include <map>
class Particle{
  public:
     Particle(float px, float py, float pz) : px_(px), py_(py), pz_(pz) {}
     float px_,py_,pz_, mass_, tau_;
     };
```

```
int main()
{
   std::vector<Particle> myParticles;
   myParticles.push_back(Particle(1.,2,100));

  //error: no match for 'operator<' (operand types are 'const Particle' and 'const Particle')
   std::map<Particle,std::vector<Particle>> decays;
   decays[Particle(3,4,5)]=myParticles;
}
```

Containers of custom objects

http://cpp.sh/4jw5n

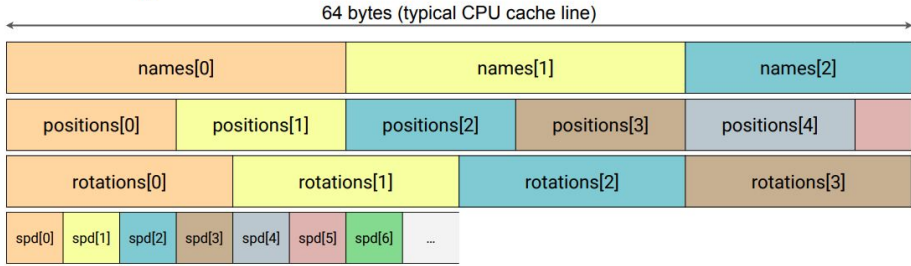
```
#include <vector>
#include <map>
class Particle{
  public:
     Particle(float px, float py, float pz) : px_(px), py_(py), pz_(pz) {}
     float px_,py_,pz_, mass_, tau_;
     };
bool operator<(const Particle & p1, const Particle &p2)
{
    if(p1.px_ < p2.px_) return true;
    if(p1.py_ < p2.py_) return true;
    if(p1.pz_ < p2.pz_) return true;
    if(p1.mass_ < p2.mass_) return true;
    return (p1.tau_ < p2.tau_);
}
int main()
{
   std::vector<Particle> myParticles;
   myParticles.push_back(Particle(1.,2,100));

  //without the operator< above it would say:
  //error: no match for 'operator<' (operand types are 'const Particle' and 'const Particle')
   std::map<Particle,std::vector<Particle>> decays;
   decays[Particle(3,4,5)]=myParticles;
}
```

Containers of complex/large objects

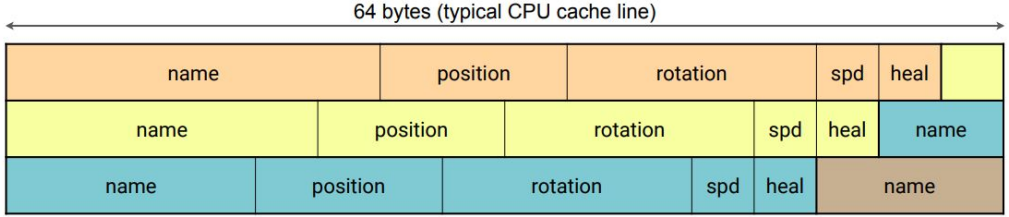
- Container of objs or obj of containers?
  - Most algorithms need a fraction of the object data
  - Containers of objects force to read all object data
- Think about your data access pattern!

```
struct Objects
{
    vector<string> names;          // 24 bytes each
    vector<Vector3> positions;     // 12 bytes each
    vector<Quaternion> rotations; // 16 bytes each
    vector<float> speeds;          // 4 bytes each
    vector<float> healths;         // 4 bytes each
};
```




```
struct Object                // 60 bytes:
{
    string name;             // 24 bytes
    Vector3 position;        // 12 bytes
    Quaternion rotation;     // 16 bytes
    float speed;             // 4 bytes
    float health;            // 4 bytes
};

// array of structures
vector<Object> allObjects;
```



Full talk on pitfalls in OO from “Unity” (game engine) lecture series: http://aras-p.info/texts/files/2018Academy%20-%20ECS-DoD.pdf

Sort (simple example)

- Sequence containers (e.g. vectors) can be sorted with
  - Std::sort
- If no “compare” function is specified, the “less” function is implied and the container is sorted ascending
  - Reverse sorting can be obtained passing reverse iterator (rbegin, rend) or by changing the sort function to be “greater” instead of “less”

```
#include <vector>
#include <algorithm>
#include <iostream>
int main()
{
   float a[] = {5.,3.1,1.0,7.32};
   std::vector<float> v(a,a+4);
   sort(v.begin(),v.end());
   for(size_t i=0;i<v.size();i++) std::cout << v[i] << std::endl;
   sort(v.rbegin(),v.rend());
   for(size_t i=0;i<v.size();i++) std::cout << v[i] << std::endl;
   sort(v.begin(),v.end(), std::greater<float>());
   for(size_t i=0;i<v.size();i++) std::cout << v[i] << std::endl;
}
```

http://cpp.sh/3luko

Sort complex(:-)) example

http://cpp.sh/7dbgw

```
#include <iostream>
#include <vector>
#include <algorithm>
class Complex{
    public:
    Complex(float real, float img) : r(real),i(img) {}
    float r; float i;
};
std::ostream& operator<<(std::ostream & o,const Complex &c) { return o << "Real " << c.r << " Img " << c.i;}

class CompareReal{
    public:
        bool operator() (const Complex & a, const Complex &b) {return a.r<b.r;}
};

struct CompareImg{
 bool operator()(const Complex & a, const Complex &b) {return a.i<b.i;}
};

int main()
{
   std::vector<Complex> v;
   v.push_back(Complex(1,2));
   v.push_back(Complex(2,1));
   v.push_back(Complex(3,3));
   sort(v.begin(),v.end(),CompareReal());
   for(size_t i=0;i<v.size();i++) std::cout << v[i] << std::endl;
   sort(v.begin(),v.end(),CompareImg());
   for(size_t i=0;i<v.size();i++) std::cout << v[i] << std::endl;
}
```

Mapping and reducing

Large data analysis can be often represented with two kind of operations

- Mapping (i.e. python “map”), i.e. apply a function to all elements of a container
- Reducing (i.e. python “reduce”), i.e. compute a scalar from elements of a container repeatedly applying a binary operation

How do you map&reduce in C++:

- std::transform
- std::accumulate

More algorithms

Non-modifying sequence operations

Defined in header <algorithm>

- all_of (C++11)
- any_of (C++11)
- none_of (C++11)
- for_each
- for_each_n (C++17)
- count
- count_if
- mismatch
- find
- find_if
- find_if_not (C++11)
- find_end
- find_first_of
- adjacent_find
- search
- search_n

Modifying sequence operations

Defined in header <algorithm>

- copy
- copy_if (C++11)
- copy_n (C++11)
- copy_backward
- move (C++11)
- move_backward (C++11)
- fill
- fill_n
- transform
- generate
- generate_n
- remove
- remove_if
- remove_copy
- remove_copy_if
- replace
- replace_if
- replace_copy
- replace_copy_if
- swap
- swap_ranges
- iter_swap
- reverse
- reverse_copy
- rotate
- rotate_copy

More..

And more..

Strings

- C++ has std::string class that allow to better handle strings compared to C “const char *”
  - Allow copy of strings
  - Allow composition of strings
  - Allow string comparison with operator ==
  - Support basic find and replace features
- Still nothing like python string handling!
  - Somehow a regression even compared to C “printf”
  - Boost::format (coming to std in C++20 !!) helps a bit
- string::c_str returns good old “const char *”

```cpp
#include <iostream>
#include <string>
int main()
{
 std::string s("Hello");
 s+=std::string(" ");
 s+="World";
 std::cout << s << std::endl;
 std::string replacement("Universe");
 std::string toreplace("World");
 if(replacement == toreplace){
    std::cout << "There is nothing to replace!" << std::endl;
 }else if(s.find(toreplace)==std::string::npos) {
    std::cout << "Cannot find "<< toreplace << std::endl;
 } else {
    s.replace(s.find(toreplace),toreplace.size(),replacement);
    std::cout << s << std::endl;
 }
}
```

http://cpp.sh/2vtb7

File read/write and parsing

- File can be treated as I/O streams (i.e. similar to std::cout)
- String writing is simple
- For string reading a “getline” function exists to read one line at time
  - More complex parsing is in general problematic, but in principle operator>> can be used
- Streams can also handle binary reading
  - See seek() and read() functions

http://cpp.sh/9tzns

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "One line in a  file.\n";
  myfile << "Another line in a  file.\n";
  myfile.close();

  string line;
  ifstream myfile_in ("example.txt");
  if (myfile_in.is_open())
  {
    while ( getline (myfile_in,line) )
    {
      cout << line << '\n';
    }
    myfile_in.close();
  }

  else cout << "Unable to open file";
  return 0;
}
```

File read/write and parsing

- File can be treated as I/O streams (i.e. similar to std::cout)
- String writing is simple
- For string reading a “getline” function exists to read one line at time
  - More complex parsing is in general problematic, but in principle operator>> can be used
- Streams can also handle binary reading
  - See seek() and read() functions

http://cpp.sh/45t5r

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << 1.3 << " " << 2.7 << std::endl
  myfile << 100 << " " << 20 << std::endl;
  myfile.close();

  string line;
  ifstream myfile_in ("example.txt");
  if (myfile_in.is_open())
  {
    float a,b;
    while ( myfile_in >> a >> b )
    {
      cout << a << " " << b << '\n';
    }
    myfile_in.close();
  }

  else cout << "Unable to open file";
  return 0;
}
```

File read/write and parsing

- File can be treated as I/O streams (i.e. similar to std::cout)
- String writing is simple
- For string reading a “getline” function exists to read one line at time
  - More complex parsing is in general problematic, but in principle operator>> can be used
- Streams can also handle binary reading
  - See seek() and read() functions

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos size;
  char * memblock;

  ifstream file ("a.out", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    //get the current position
    //that is file size because of ios::ate
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

Object persistency (lack of)

- Python has nice interface to dump objects into files
  - For dictionaries using “json” files also make the files human readable!
  - For other objects “pickle” can be used
- C++ has no nice interface to dump objects into files and read them back
  - Several external libraries exists
  - In HEP we mostly use ROOT toolkit

```
[1]  import pickle

     favorite_color = { "lion": "yellow", "kitty": "red" }
     pickle.dump( favorite_color, open( "save.p", "wb" ) )
```

```
[2]  # Load the dictionary back from the pickle file.
     favorite_color_read = pickle.load( open( "save.p", "rb" ) )
     print(favorite_color_read)
```

```
{'lion': 'yellow', 'kitty': 'red'}
```

```
[4]  import json

     favorite_color = { "lion": "yellow", "kitty": "red" }
     json.dump( favorite_color, open( "save.json", "w" ) )
```

```
[5]  # Load the dictionary back from the pickle file.
     favorite_color_read = json.load( open( "save.json", "r" ) )
     print(favorite_color_read)
```

```
{'lion': 'yellow', 'kitty': 'red'}
```

```
!cat save.json
```

```
{"lion": "yellow", "kitty": "red"}
```

Exceptions

- Sometimes programs produces “errors”, e.g.
  - Unexpected or invalid data
  - Unavailability of some resource
  - Domain problems
- The whole program being in error is typically handled with an “exit code”
- How about individual functions or pieces of code being “in error”
  - E.g. sqrt(-1.) ?
- C++, like python, has “exceptions”
- Exceptions can be catched and the error handled by the calling code
- Exceptions are “expensive”
  - Do not use them as “if/else” !!!
- Exceptions may leave a mess behind
  - E.g. undeleted pointers
- If not catched they terminate the program

```
#include <iostream>
using namespace std;
class Test{
    public:
        Test() {std::cout << "Creating " << this << std::endl;}
        ~Test() {std::cout << "Deleting " << this << std::endl;}
};
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 1.;
   double z;
   try {
      Test * t = new Test();;
      z = division(x, y);
      delete t;
   } catch (const char* msg) {
      cerr << msg << endl;
      cerr << "I will set z to zero in this case" << endl;
      z=0;
   }
   cout << z << endl;
   return 0;
}
```

`int y = 1.;`

```
Creating 0x55d6ad744e70
Deleting 0x55d6ad744e70
50
```

`int y = 0.;`

```
Creating 0x559fb1b1de70
Division by zero condition!
I will set z to zero in this case
0
```

Example usage of static: Singleton

- Singleton are classes for which a single object can exist in the program
  - Useful to pass around global shared data (e.g. experiment setup) without percolation

```
#include <iostream>
class Singleton {
public:
   static Singleton* instance();
private:
   Singleton(){};  // Private so that it can  not be called
   Singleton(Singleton const&){};             // copy constructor is private
   Singleton& operator=(Singleton const&){};  // assignment operator is private
   static Singleton* m_instance;
};

Singleton* Singleton::m_instance = 0;

Singleton* Singleton::instance(){
 if(!m_instance) m_instance=new Singleton();
 return m_instance;
}
int main(){
  std::cout << Singleton::instance()  << std::endl;
  std::cout << Singleton::instance()  << std::endl;
}
```

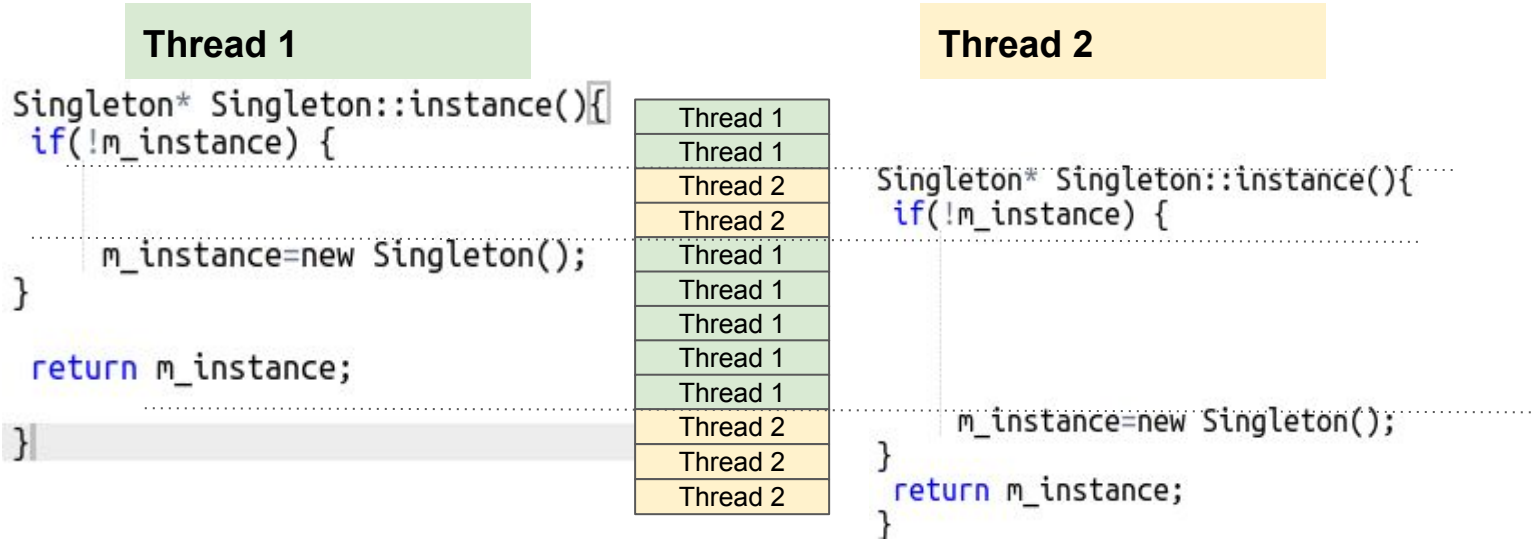
This means **m_instance** is shared among all object instances and is available even without an instance

Smart pointers

- What happens when your class/function returns a pointer?
  - How is the calling code supposed to know if it should delete the pointer when done or not?
    - Risk to leak or double delete
  - “Ownership” of pointers is never clear (e.g. in ROOT toolikt, often pointers are returned but the memory is handled by the root toolkit)
- What are smart pointers?
  - “Wrappers” around standard pointers that allow to auto delete pointers properly
- How does it work? What “properly” means?
  - Unique pointer: a unique pointer is a wrapper that cannot be “copied” but only “assigned” to another pointer, i.e. you ensure that when you pass the pointer to something else you forget the info about the pointer value so that you cannot delete
  - Reference counted / shared pointers: each time a copy of the pointer is made, some counter is incremented. The counter is decreased when a copy is deleted. The actual pointer deletion happens when the last copy is gone and the counter is set to zero.
- In C++11 and beyond smart pointers are implemented in the language
  - So I will not show an example on how to make your own smart pointer

Thread safety

- The usage “static” keyword helps sharing data across different objects of the same class
- What happens if multiple copies of your functions are run simultaneously by different threads?



Exercise

- Create a phonebook class
  1. Each entry of the phonebook should be another class with some properties (e.g. name, phonenumber, email) and a function to pretty print the information
     - Possibly extend to support for multiple phone numbers (e.g. a map ”phonetype” -> “num”)
  2. Add a function that sorts the entries (operator< on a string does lexicographical comparison)
  3. The class should have a find function on each of the name field
     - Should return both the found element and a bool saying if it was found or not
  4. Try to implement a function that using “transform” adds a fixed (+39) international prefix to all numbers that do not have one (i.e. not starting with “+”)
- Store the phonebook data in a file
  1. One entry per line
     - How would you handle spaces in the name?
       - Hint: getline function can also help splitting string (strings can be seen as streams with stringstream)
     - How would you handle multiple phone numbers?

Monday assignment 2

Second assignment

- Template the 4-vector definition (e.g. to use with float vs double)
  - Be sure that sum can be done between FourVector<double> and FourVector<float>
- Implement a PtEtaPhiMassVector
- Create member functions to convert from one class to the other
  - You can overload the cast operator!:

```
operator AClassName() const { return ...something... }
```

- Create operators to sum from different types

https://github.com/lucabaldini/cmepda/blob/master/HEPCPPmodule/HEPCppLesson1/Assignment2/fourvector.h