The Nobel Prize in Physics 2024

Ill. Niklas Elmehed © Nobel Prize Outreach

John J. Hopfield

Prize share: 1/2

Ill. Niklas Elmehed © Nobe Outreach

Geoffrey E. Hinto

Prize share: 1/2

The Nobel Prize in Physics 2024 was awarded jointly to John J. Hopfield and Geoffrey E. Hinton "for foundational discoveries and inventions that enable machine learning with artificial neural networks"

The Nobel Prize in Physics 2024

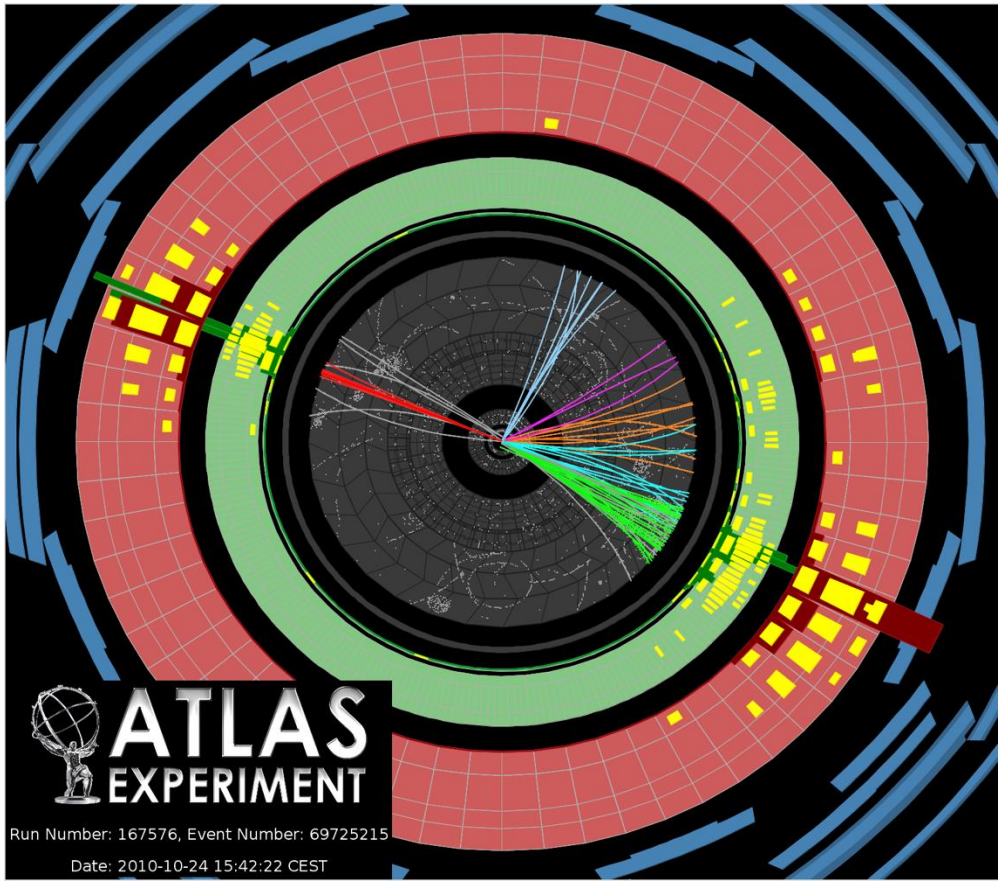
Because physics has contributed tools for the development of machine learning, it is interesting to see how physics, as a research field, is also benefitting from artificial neural networks. Machine learning has long been used in areas we may be familiar with from previous Nobel Prizes in Physics. These include the use of machine learning to sift through and process the vast amounts of data necessary to discover the Higgs particle. Other applications include reducing noise in measurements of the gravitational waves from colliding black holes, or the search for exoplanets.

The Nobel Prize in Physics 2024

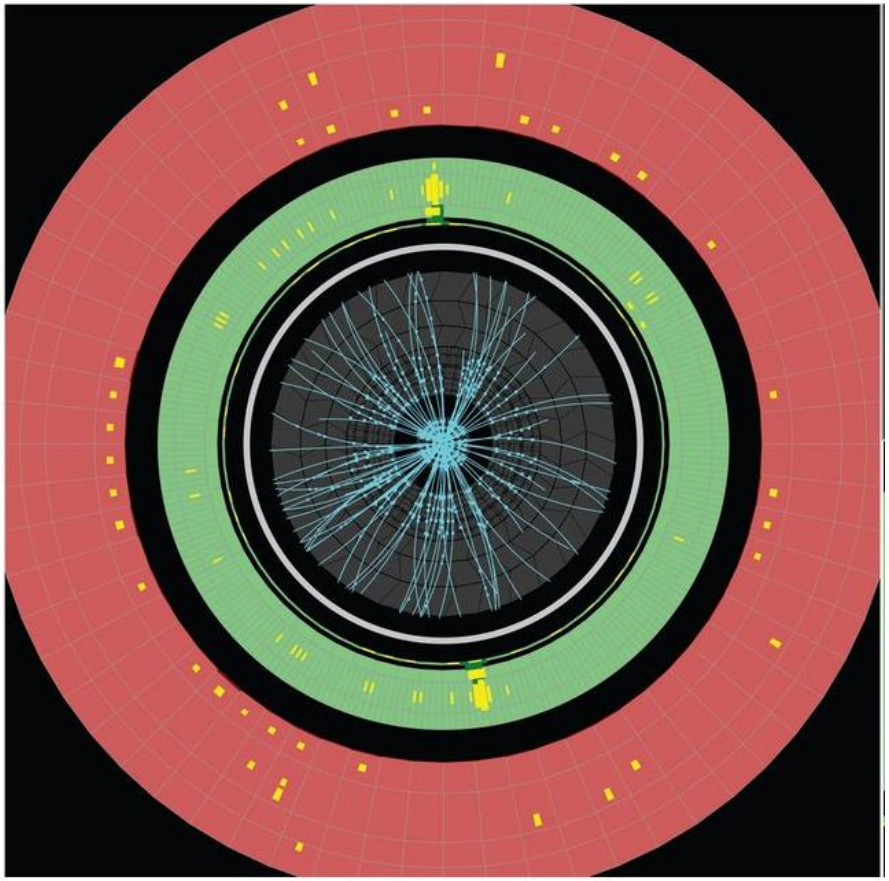
During the 1990s, ANNs became a standard data analysis tool within particle physics experiments of ever-increasing complexity. Highly sought-after fundamental particles, such as the Higgs boson, only exist for a fraction of a second after being created in high-energy collisions (e.g. ~10^{-22} s for the Higgs boson). Their presence needs to be inferred from tracking information and energy deposits in large electronic detectors. Often the anticipated detector signature is very rare and could be mimicked by more common background processes. To identify particle decays and increase the efficiency of analyses, ANNs were trained to pick out specific patterns in the large volumes of detector data being generated at a high rate.

Event display of LHC collisions data with the ATLAS detector

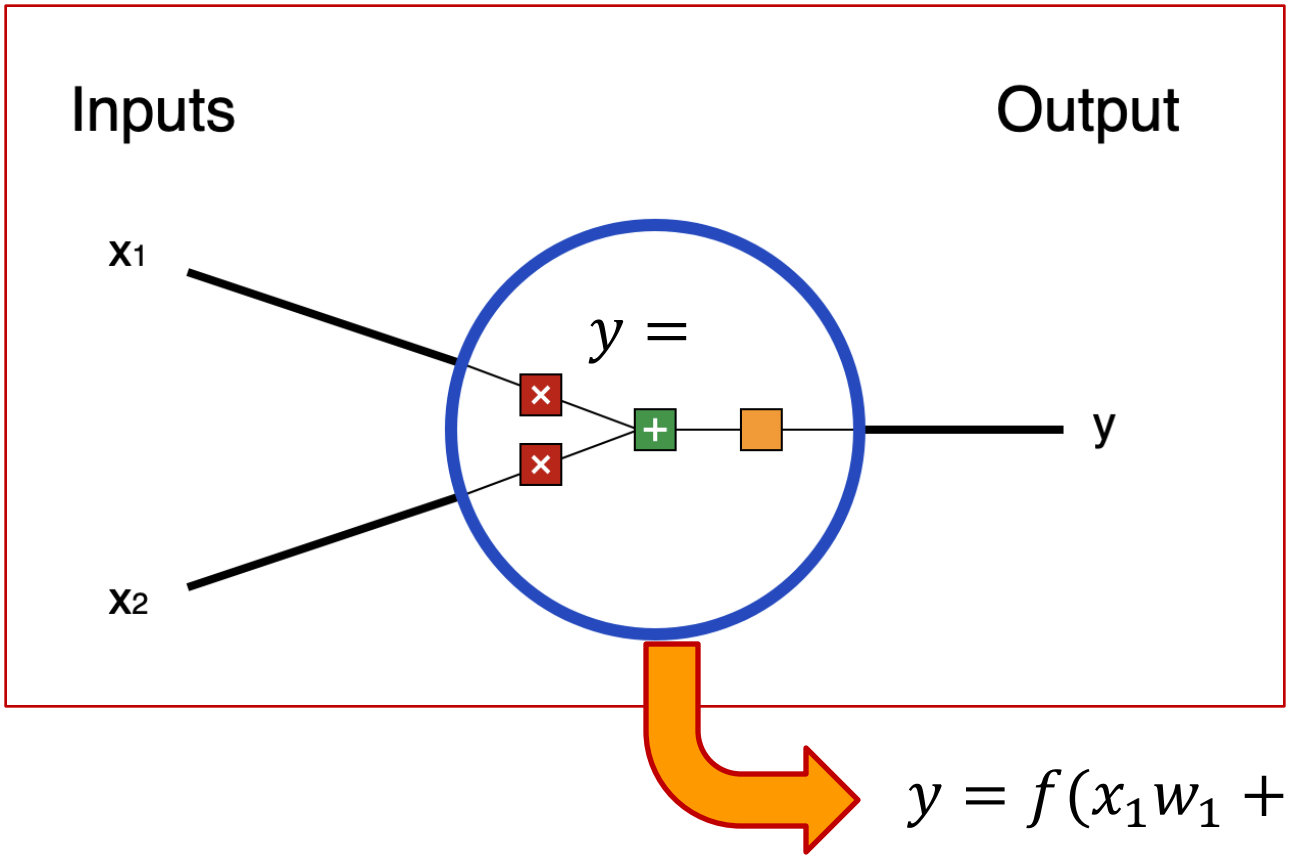
A di-jet event



A di-photon event

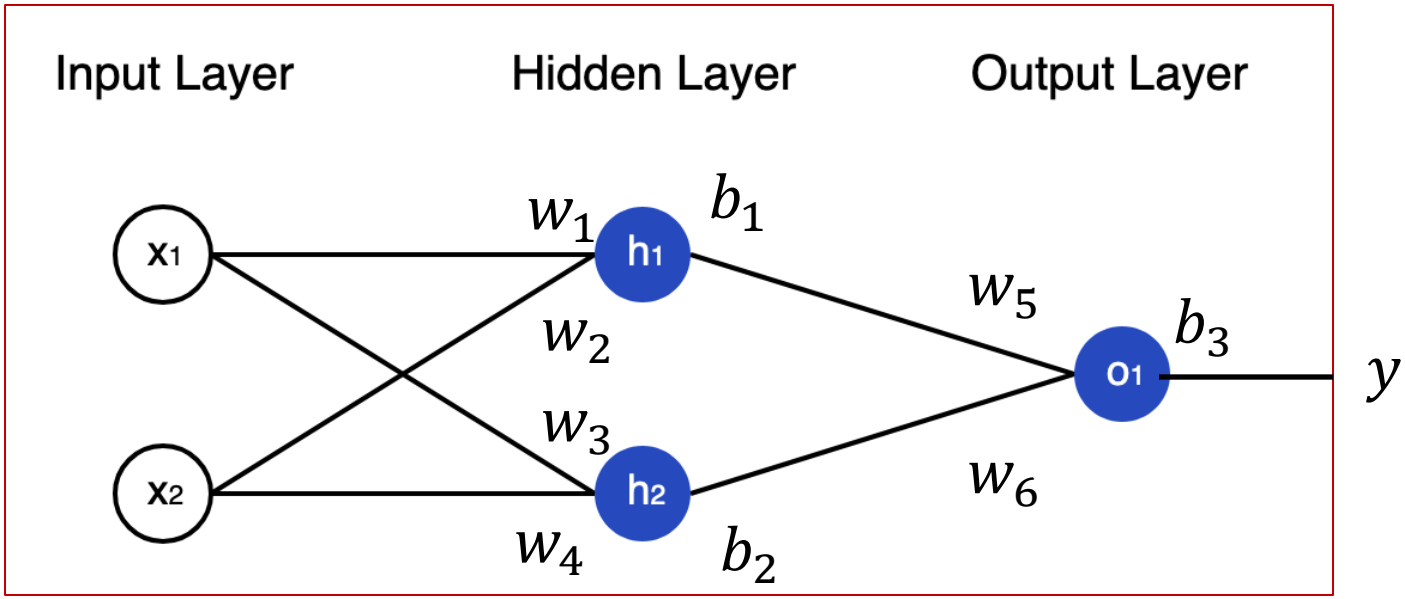


The basic building block of a Neural Network : the neuron



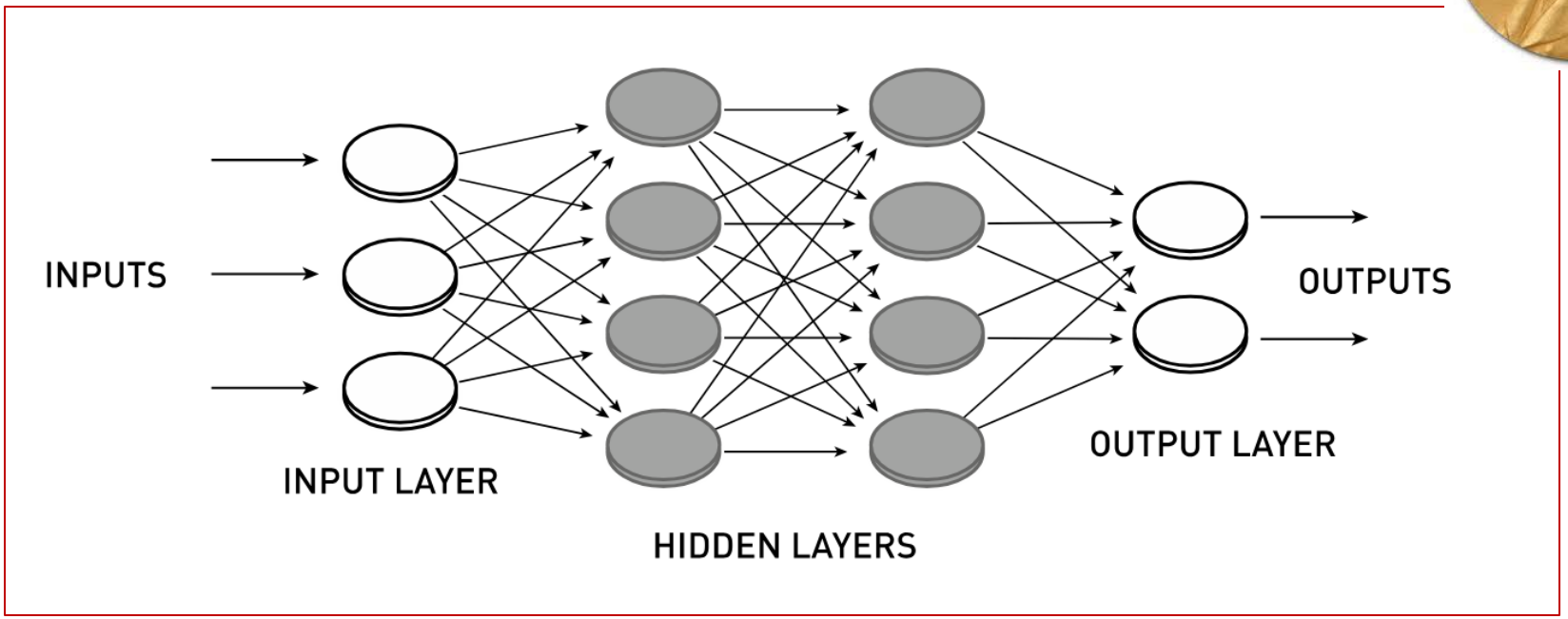
$$y = f(x_1 w_1 + x_2 w_2 + b)$$

A feedforward neural network



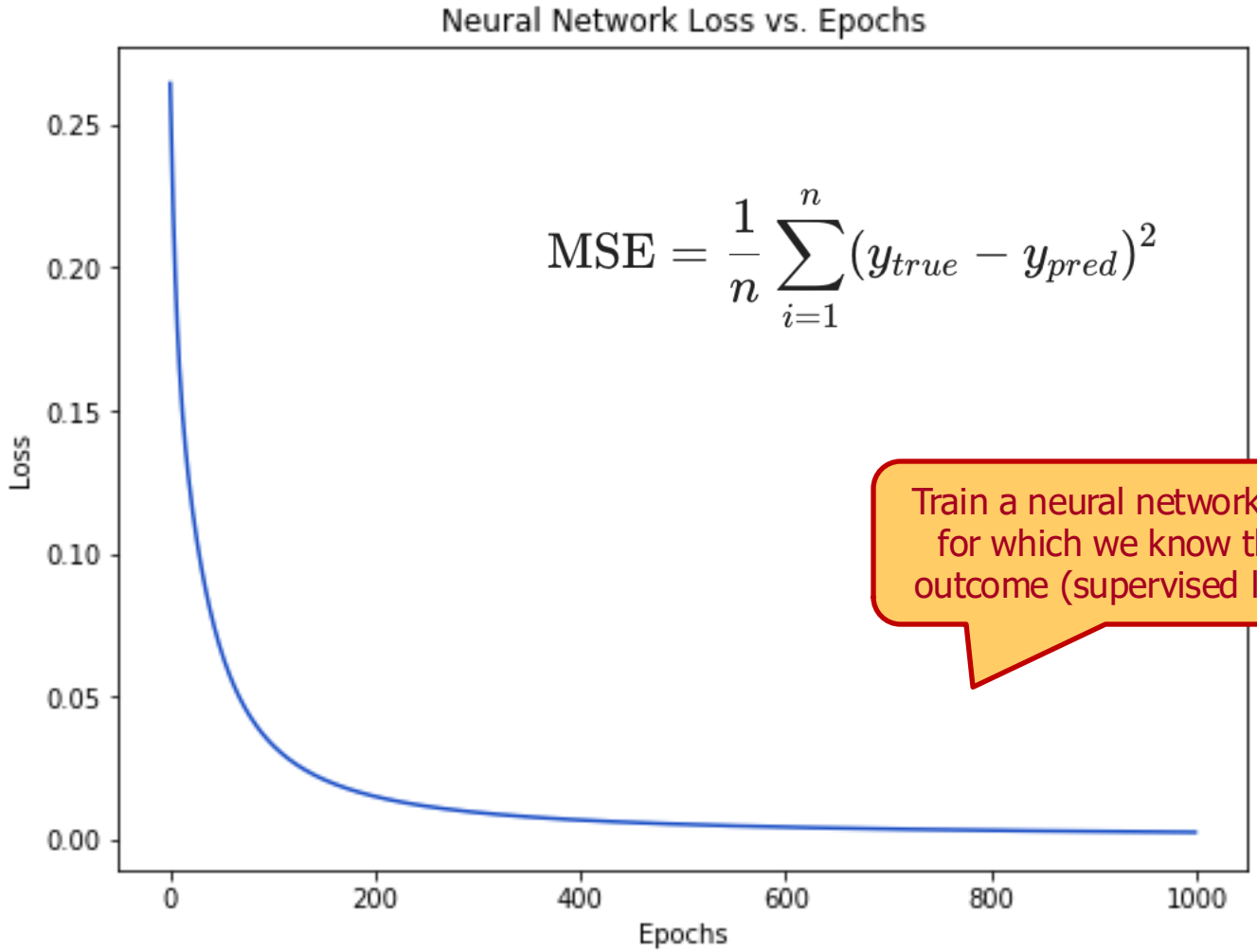
$$y = f(x_1, x_2, w_2, w_3, w_4, w_5, w_6, b_1, b_3, b_3)$$

A (deep) feedforward neural network



Nice to know but back to out original problem: can we use this structure for pattern recognition (separate photon from jets) ? Yes, this structure exhibits nice predictive behavior !

Training of a neural network

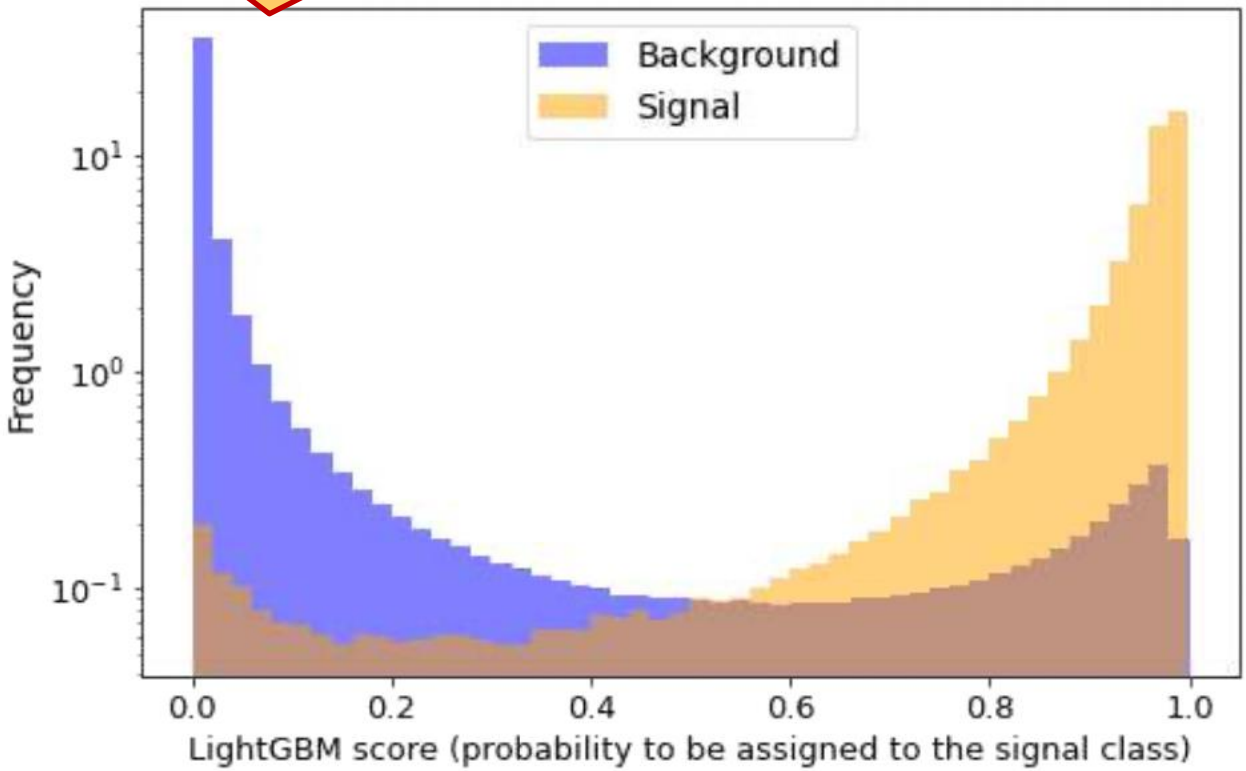


$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

Train a neural network on data for which we know the true outcome (supervised learning)

The NN prediction (score)

Check the performance of the NN on a different data set for which we know the true outcome



Working setup

- PC in the computing lab : native linux, all packages already installed (ROOT)
- Personal linux/mac laptop: two options to install ROOT
  - Download pre-compiled binaries
  - Use a package manager (snap, Homebrew, conda …)
- Personal windows laptop: three options :
  - Dual-boot (not easy)
  - WSL
  - VirtualBox

Working setups with some students, use the forum to get examples/help

https://myariel.unimi.it/mod/forum/view.php?id=79916

Creating plots

You can think to ROOT as a collection of classes designed for statistical analysis and visualization. We will mainly use three objects

- https://root.cern.ch/doc/master/classTH1F.html
- https://root.cern.ch/doc/master/classTGraph.html
- https://root.cern.ch/doc/master/classTF1.html

Simple examples on how to practically use the main objects can be found in https://labtnds.docs.cern.ch/Survival/root/

Histogramming with ROOT

```
#include <iostream>
#include <fstream>
#include "TH1F.h"
#include "TApplication.h"
#include "TCanvas.h"

#include "funzioni.h"

int main( int argc , char** argv ) {

  if ( argc < 2 ) { ... }

  // creo un processo "app" che lascia il programma attivo ( app.Run() ) in modo
  // da permettermi di vedere gli outputs grafici

  TApplication app("app",0,0);

  // leggo tutti i dati da file

  vector<double> v = ReadAll<double>( argv[1] );

  // Creo l'istogramma. L'opzione StatOvergflows permette di calcolare le informazioni
  // statistiche anche se il dato sta fuori dal range di definizione dell'istogramma

  TH1F histo ("histo","histo", 100, -10, 100000000) ;
  histo.StatOverflows( kTRUE );

  for ( int k = 0 ; k < v.size() ; k++ ) histo.Fill( v[k] );

  // accedo a informazioni statistiche

  cout << "Media dei valori caricati = " << histo.GetMean() << endl;

  // disegno

  TCanvas mycanvas ("Histo","Histo");
  histo.Draw();
  histo.GetXaxis()->SetTitle("measurement");

  app.Run();

}
```

#include header files for each ROOT class you plan to use

Leave the dummy application running so that I can see the plot

Call the constructor o the TH1F class

Fill the histogram

Create a support for the plot

Draw the histogram

TH1F constructor

- https://root.cern.ch/doc/master/classTH1F.html

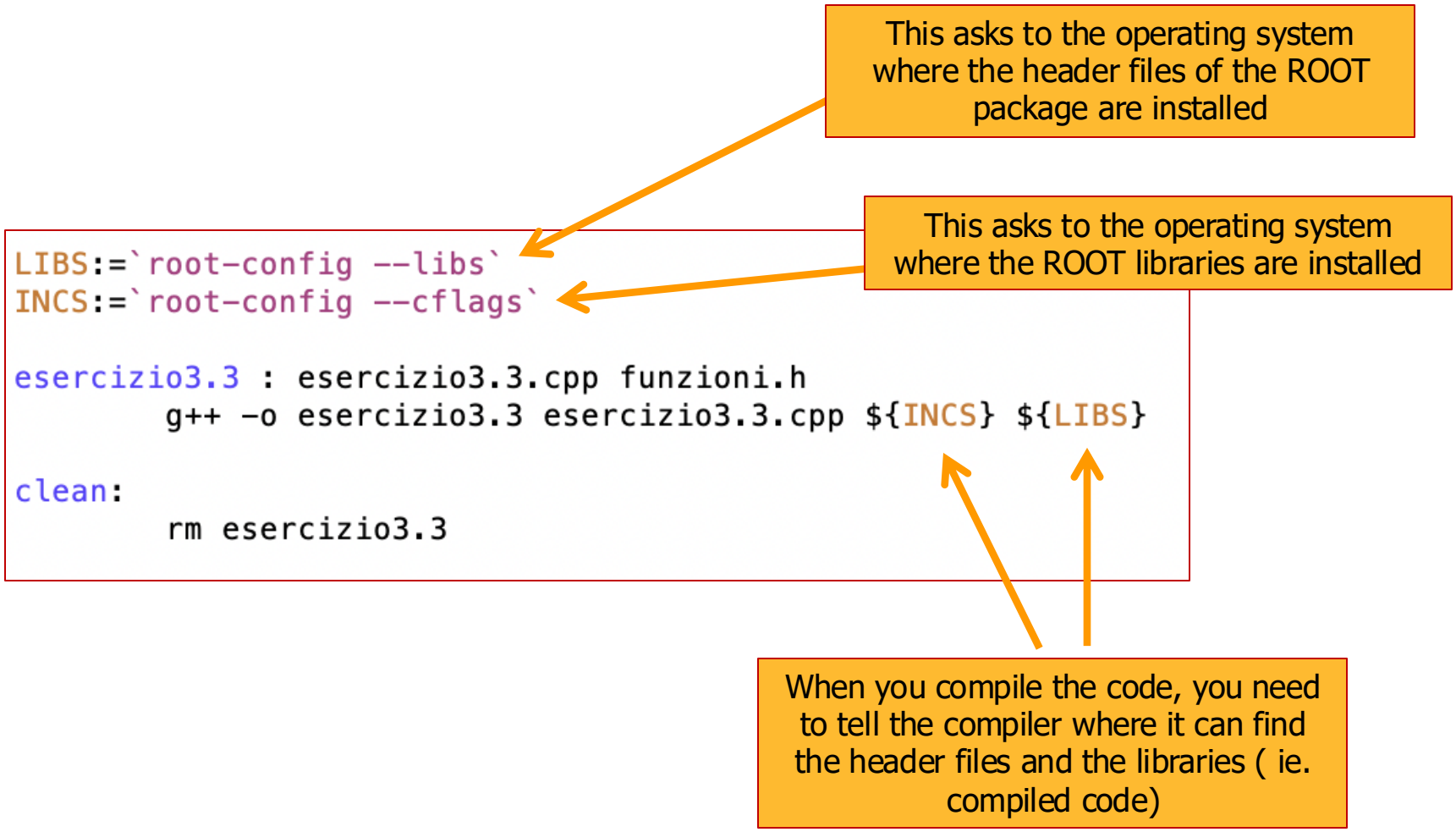
◆ TH1F() [2/6]

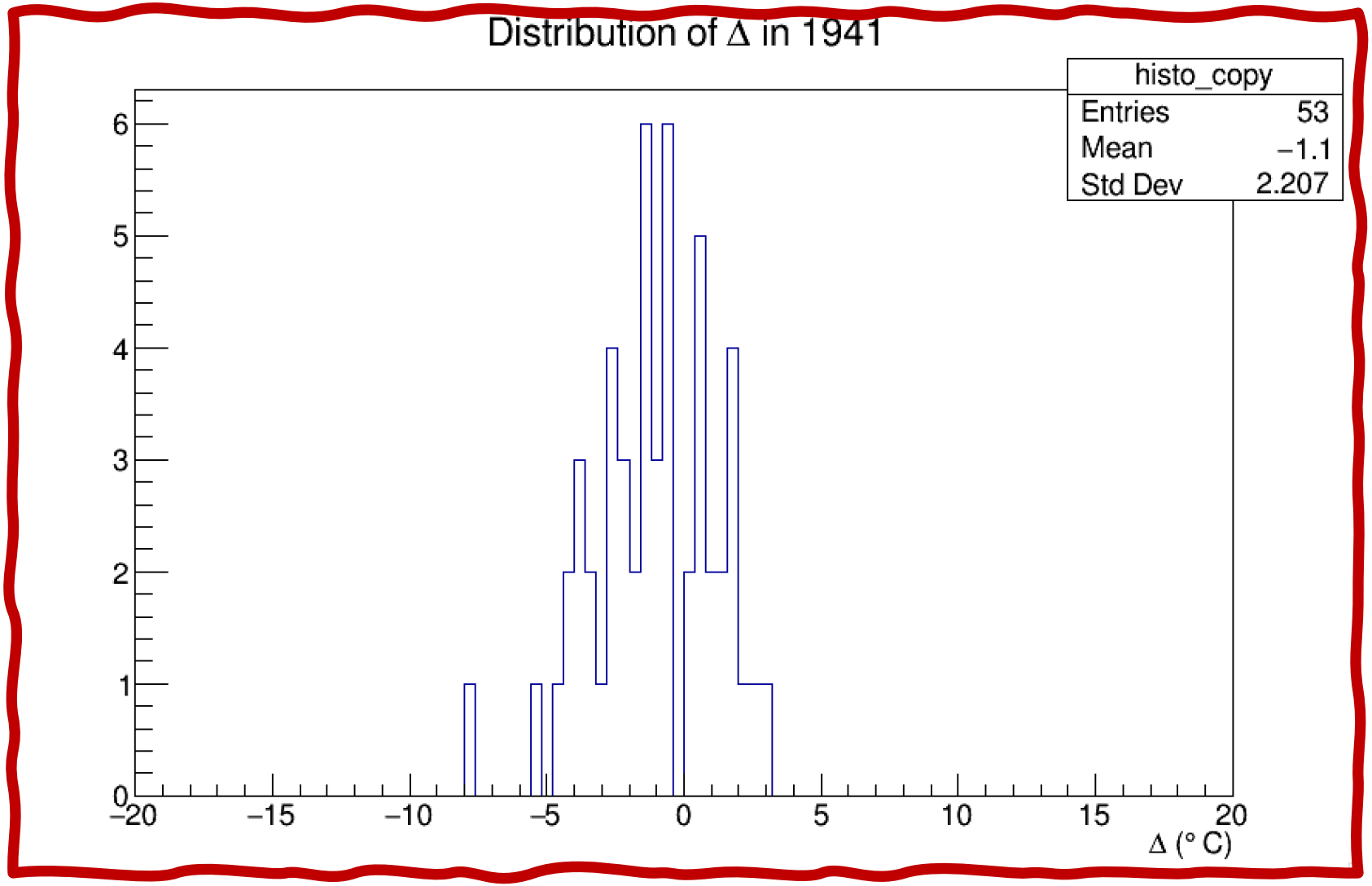
```
TH1F::TH1F ( const char *  name,
             const char *  title,
             Int_t         nbinsx,
             Double_t      xlow,
             Double_t      xup
           )
```

Create a 1-Dim histogram with fix bins of type float (see **TH1::TH1** for explanation of parameters)

Definition at line **9902** of file **TH1.cxx**.

Compile your program including ROOT classes

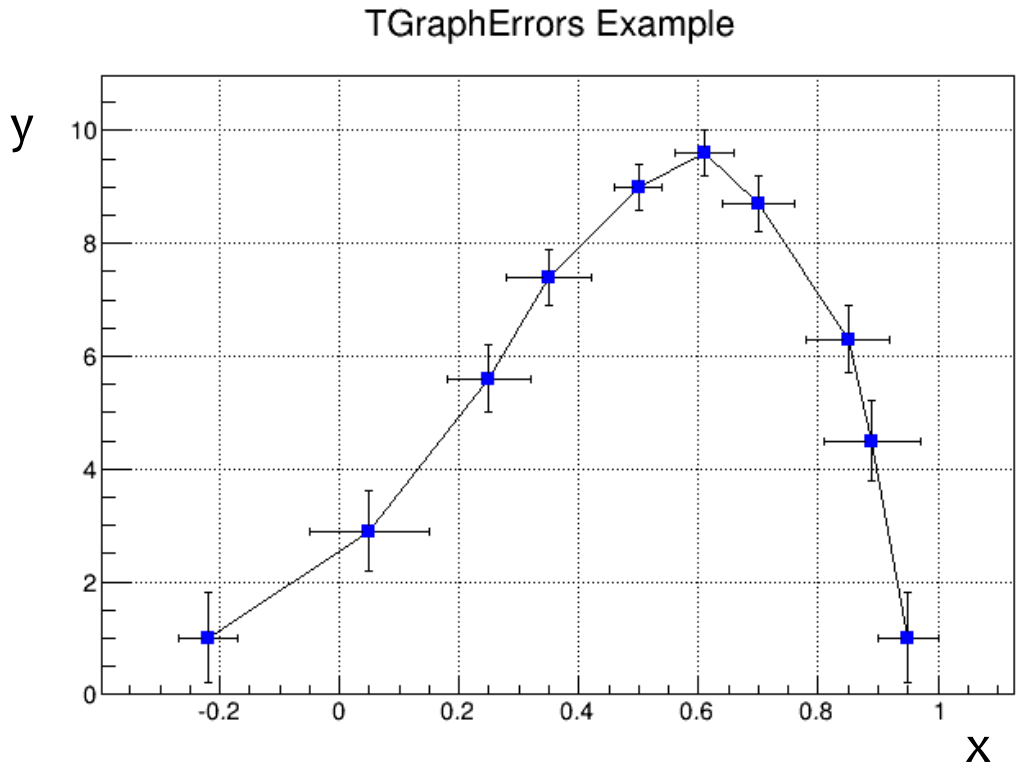



Distribution of Δ in 1941
histo_copy
Entries 53
Mean −1.1
Std Dev 2.207
0
1
2
3
4
5
6
−20
−15
−10
−5
0
5
10
15
20
Δ (° C)

TGraphErrors

Visualize the trend of average Δ as a function of the year (1941 to 2023) : `TGraphErrors`

- https://root.cern.ch/doc/v630/classTGraphErrors.html



Code snippet (I) : create and fill a `TGraphErrors`

```
// #includes

using namespace std;

int main(  ) {

  // oggetto per rappresentare un andamento x (anno) verso y (delta medio)

  TGraphErrors trend;

  // loop principale su tutti i files ( anni ) da analizzare

  for ( int i = 1941 ; i < 2024 ; i++) {

    string filename = to_string(i) + ".txt" ;

    vector<double> v = Read<double> ( filename.c_str() ) ;

    // inserire il calcolo di ave ed err

    cout << "  Anno " << to_string(i) << "  delta medio = " << ave << " +/- " << err <<  endl;

    // inserisco media e deviazione standard dalla media nel grafico

    trend.SetPoint(index, i, ave);
    trend.SetPointError( index , 0 , err);

    index++;

  }

  // grafica ....

}
```

See next slide

Code snippet (II) : customize and draw a `TGraphErrors`

```
TCanvas c;
c.cd();
c.SetGridx();
c.SetGridy();

trend.SetMarkerSize(1.0);
trend.SetMarkerStyle(20);
trend.SetFillColor(5);

trend.SetTitle("Andamento temperature");
trend.GetXaxis()->SetTitle("Anno");
trend.GetYaxis()->SetTitle("#Delta (#circ C)");
trend.Draw("apl3");
trend.Draw("pX");

c.SaveAs("trend.pdf");
```

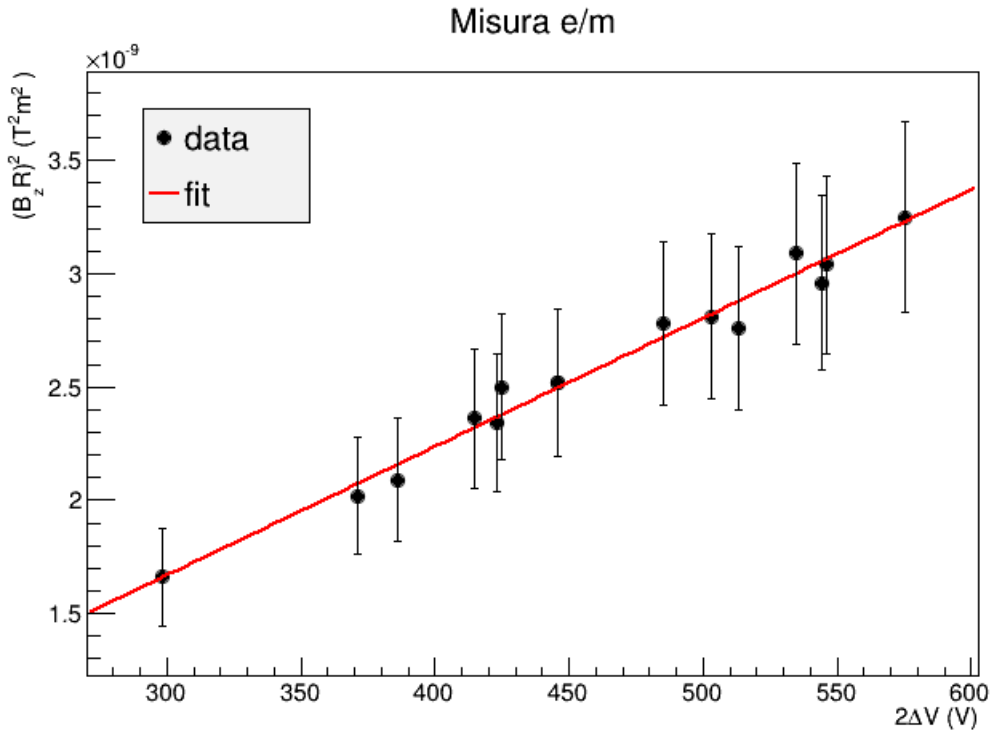
Climate data plot :

Do you notice anything suspicious ?

q/m measurement

A very interesting experiment that takes place in the physics laboratory concerns the measurement of the ratio between the charge and mass of an electron. An electron beam of known energy (determined by the 2ΔV potential applied across a capacitor) is deflected by a suitably generated magnetic field. From the measurement of the deflection radius r an estimate of the q/m ratio can be obtained. The measurement in the laboratory is rather complex (in particular the study of experimental systematics) but the q/m ratio can be determined by exploiting the relationship $(rB)^2 = (m/q)\ 2\Delta V$ as the angular coefficient of the relationship.

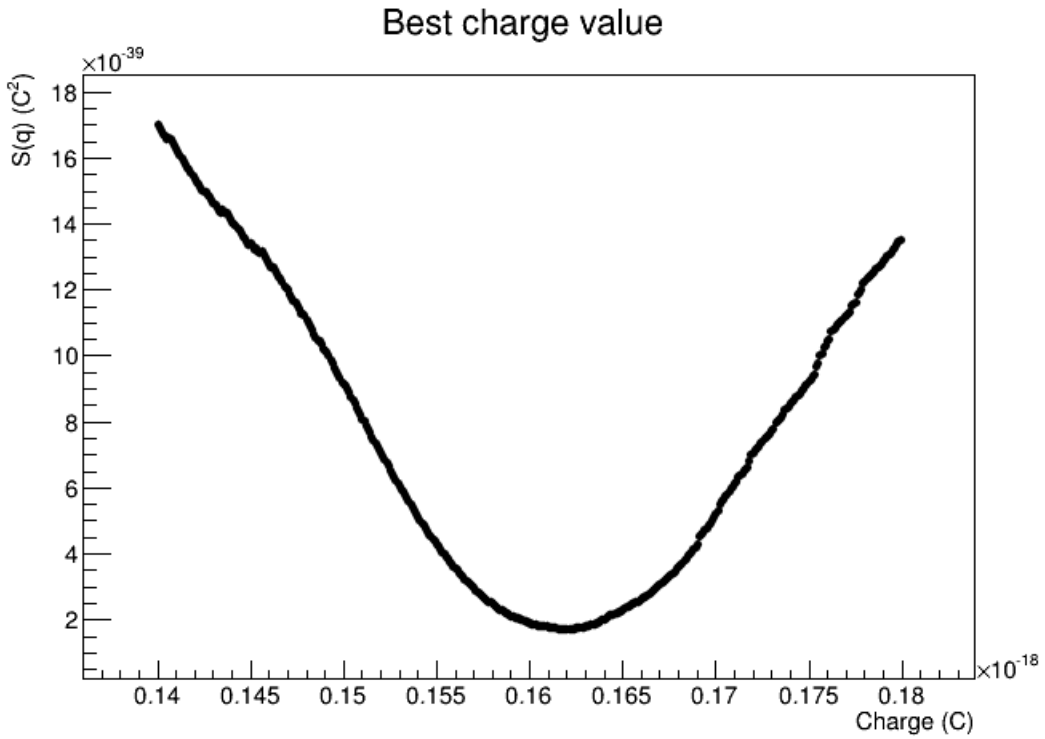
The data file shows the measurements taken in the laboratory: in the first column the value of 2ΔV, in the second column the value of $(rB)^2$ and in the third the error on the determination of $(rB)^2$. Try to write a code that allows you to view the collected data and determine the q/m ratio (in this case the most suitable ROOT object is the `TGraphErrors`.)



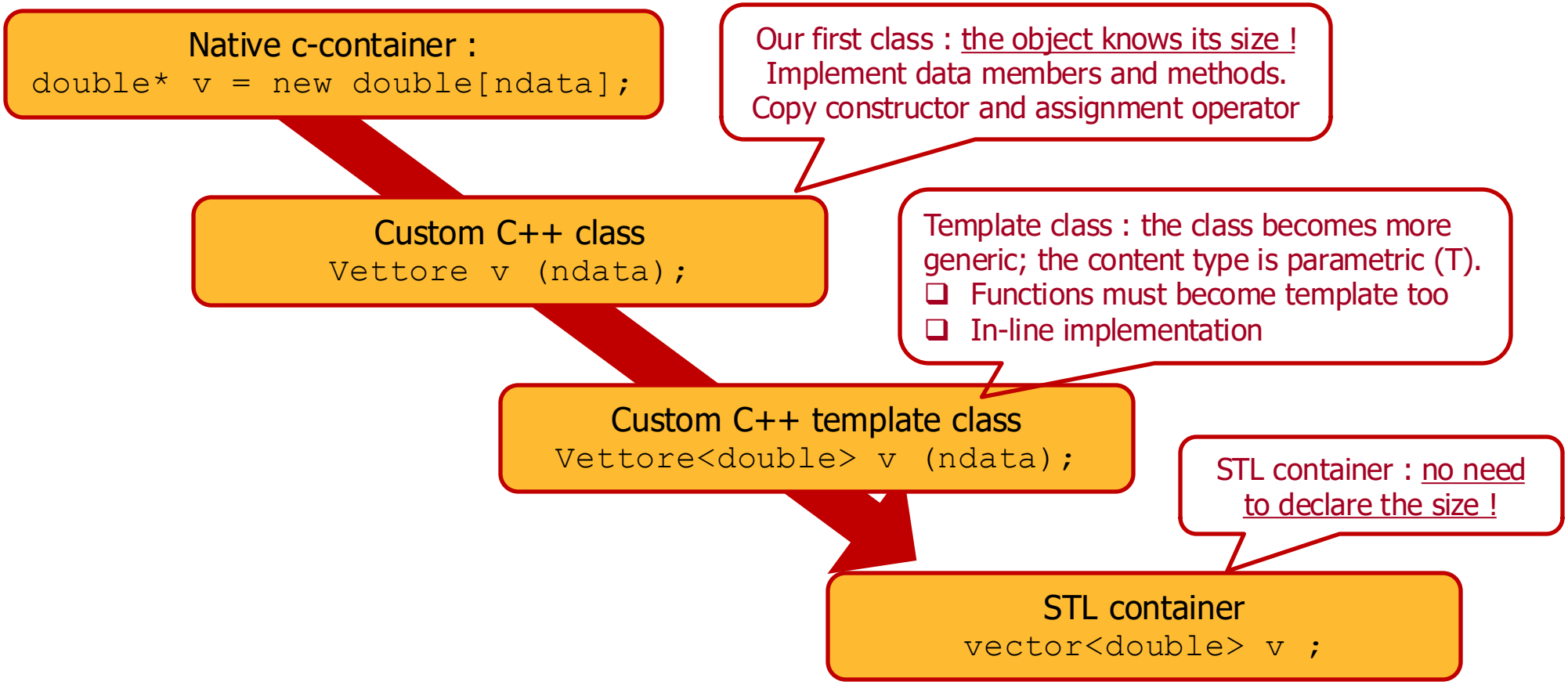
Measurement of the electron charge

The charge of the electron was measured for the first time in 1909 in a historic experiment by the American physicist Robert Millikan. This experiment is also carried out in the physics laboratory and consists of measuring the falling or rising speed of oil droplets electrified by friction in a region with an adjustable electric field. The data file reports the electrical charge (Q_i) measurements for a certain number of observed droplets.

The value of the charge can be determined as the minimum of the function $S(q)=\Sigma[(Q_i/k_i)-q]^2$ where k_i is the integer closest to the ratio Q_i/q. Try to write a code to represent the function S(q) and determine the value of the charge of the electron



What we should have learned so far :



Standard Template Library

- Containers: STL containers have been designed to obtain maximum efficiency accompanied by maximum genericity.
  - Different types of containers, each optimized for a specific set of operations
  - Sequential (vector, list, deque) or associative (map, set) containers

- Iterators: the iterator generalizes the concept of a pointer to a sequence of objects and can be implemented in many different ways (in the case of an array it will be a pointer, while in the case of a list it will be a link etc...). Iterators allow you to iterate over a container, accessing each element individually.
  - In reality, the particular implementation of an iterator is of no interest to the user, as the definitions concerning the iterators are identical, in name and meaning, in all containers.

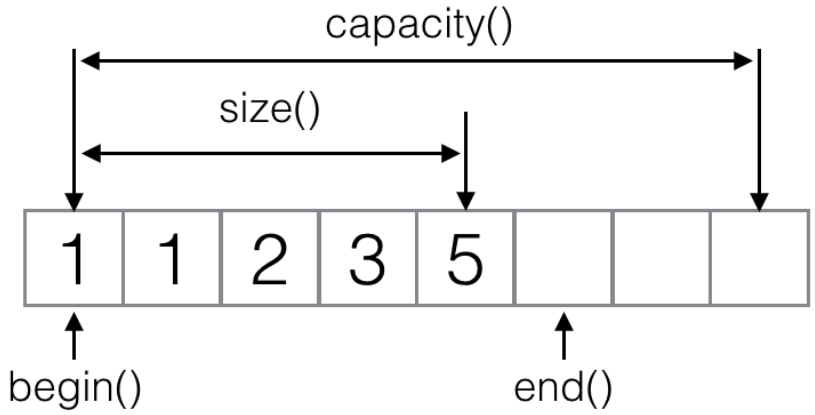
- Algorithms: from the user's point of view, both operations and iterators constitute a standard set, independent of the containers to which they are applied. In this way it is possible to write template functions with maximum genericity, without detracting from efficiency during execution.
  - The STL provides around sixty template functions, called "algorithms" and defined in the <algorithm> header file.

Containers

- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).
- Many containers have several member functions in common and share functionalities.
- The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.
- http://www.cplusplus.com/reference/stl/

The sequential container `vector`

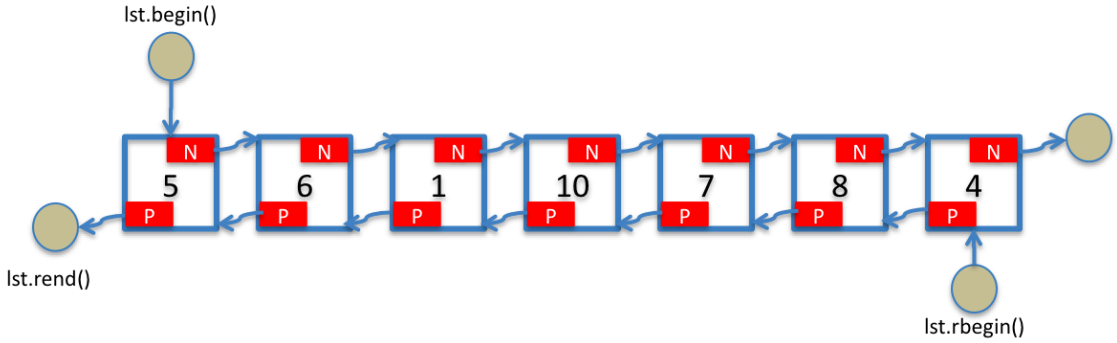
The `vector` class provides a data structure that occupies contiguous memory locations.



- efficient and direct access to any element of a `vector` via the indexing operator [] used in the same way as arrays in C and C++.
  - Just increment the pointer (iterator) by as many cells as I want (the cells are contiguous!)
  - Insertion at the tail of the vector is very efficient, insertion in the centre is not
- When a `vector` runs out of capacity, the vector automatically allocates a larger contiguous memory area, copies the original elements to the new area, and deallocates the old area.

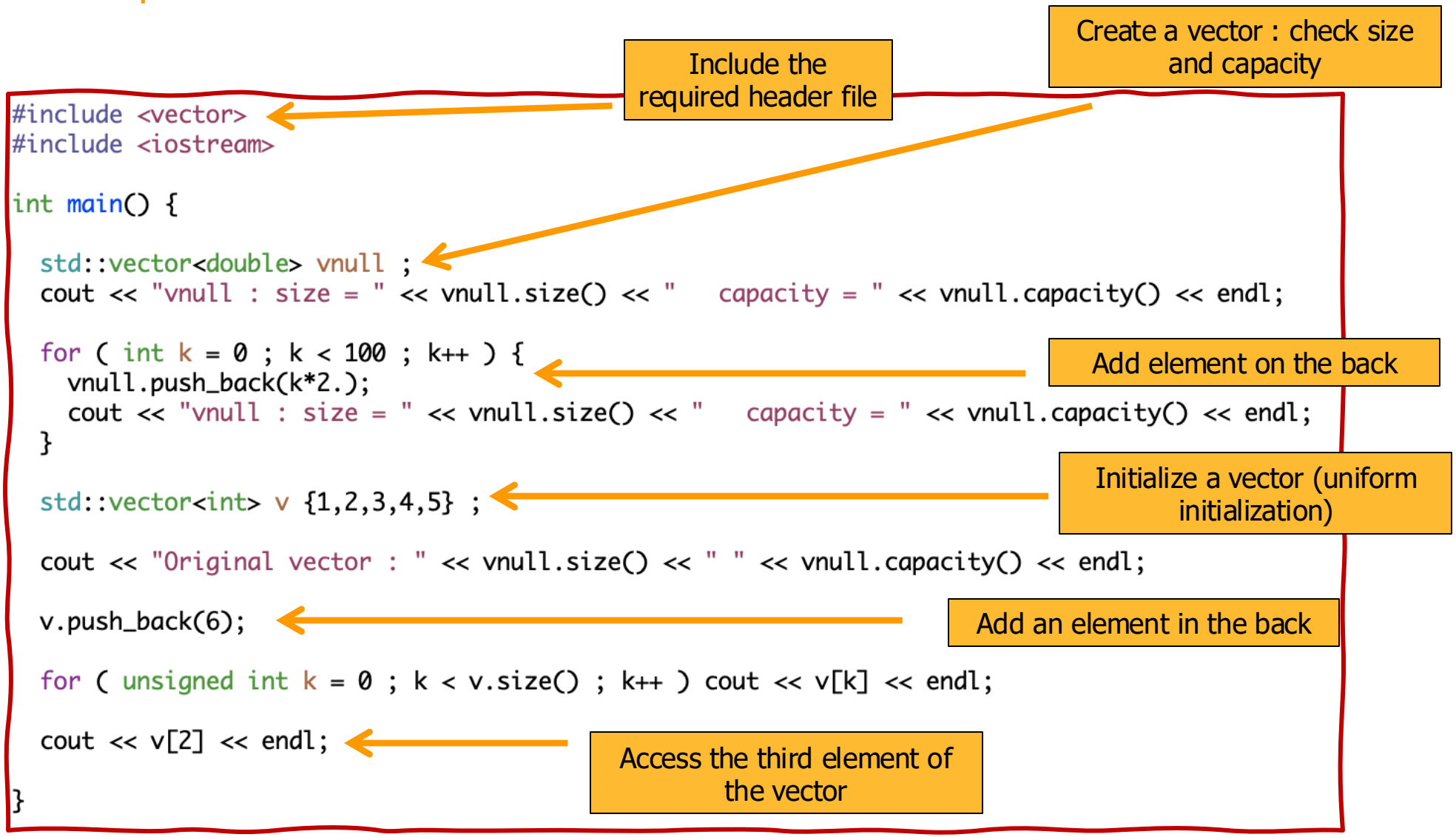
The sequential container `list`

The sequential container `list` is implemented as a double linked list: each node of the list contains a pointer to the previous node and one to the next node.



- Efficient implementation of insert and delete operations at any container location.
  - If most of these operations occur at the edges of the container, it is better to use the more efficient implementation of deque
- Direct access to an element in the list is not efficient (you have to go through each element, the cells are not contiguous)

The sequential container `vector`



Having some fun with `vector : at()` vs `operator[]()`

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

using namespace std;

int main() {

  double val = 42.42 ;

  // push_back, operator[] and at()

  vector<double> v1     ;
  cout << v1.size() << " " << v1.capacity() << endl;
  v1.push_back(val);
  cout << v1.size() << " " << v1.capacity() << endl;
  cout << v1[0] << endl;
  cout << v1[12] << endl;
  cout << v1.at(12) << endl;

  return 0;

}
```

Create a `vector` not setting explicitly the size

`operator[]` won't check the boundaries !

`at()` will check the boundaries (will give an error in this case)

Having some fun with `vector` : create a `vector` with a size

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

using namespace std;

int main() {

  double val = 42.42 ;

  // push_back

  vector<double> v2(10)     ;
  cout << v2.size() << " " << v2.capacity() << endl;
  v2.push_back(val);
  cout << v2.size() << " " << v2.capacity() << endl;

 return 0;

}
```

Create a `vector` with size=capacity=10

Size becomes 11, capacity probably 20

Having some fun with `vector`: insert an element

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

using namespace std;

int main() {

  // insert

  vector<double> v3 (10) ;
  cout << v3.size() << " " << v3.capacity() << endl;
  v3.insert(v3.begin()+2, 43.43) ;
  cout << v3.size() << " " << v3.capacity() << endl;

 return 0;

}
```

Create a `vector` with size=capacity=10

Insert 43.43 in the 3rd position (size will become 11, capacity probably 20)

Intermezzo : the range-based loop (from C++11)

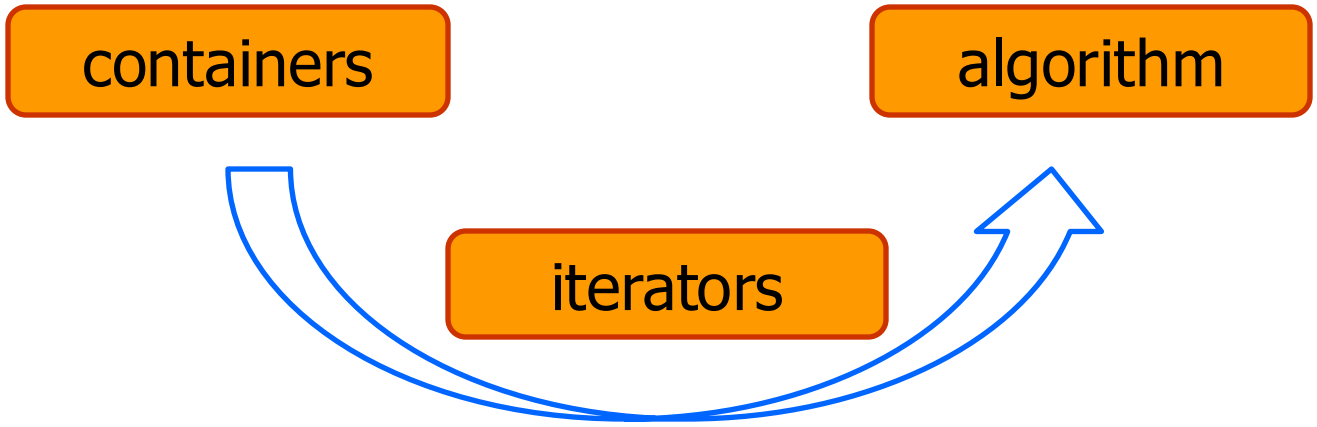
```cpp
#include <vector>
#include <iostream>

int main() {

  std::vector<int> v {1,2,3,4,5} ;

  // Range based loop

  for ( int x : v ) std::cout << "Vector element = " << x << std::endl;

  // Add a reference, you can modify the element

  for ( int &x : v ) x*=3;

  // Check the change in the vector content

  for ( auto x : v ) std::cout << x << std::endl;

  // works for usual C arrays

  int array[4] {1,2,3,4} ;

  for ( auto x : array ) std::cout << "Array element = " << x << std::endl;

}
```

The magic keyword `auto`

Iterators

The STL separates the algorithms from the containers. With the STL, you can access the elements of container via iterators.

- STL algorithms do not depend on the implementation details of the containers on which they operate.
- The algorithms are perfectly generic, in the sense that they can operate on any type of container (and on any type of elements), if it is equipped with iterators;



Iterators

- They are a generalization of pointers; they point to an object present in the container itself
- The advantage of iterators is that they offer a set of common operations with an interface that does not depend on the type of container being iterated
- As for pointers, operations are defined on each iterator:
  - De-referencing (accessing the pointed element) via the `*i` operator
  - Scroll to next/previous item `i++` or `i–`
  - Equality test: `i==j` or `i!=j`
- Two methods are defined on each container:
  - `c.begin()` returns an iterator that points to the first element of c
  - `c.end()` returns an iterator that points to the next element of the last
- An iterator can be (different containers support different iterators):
  - Forward: allow you to advance in a sequence `(i++)`
  - Bidirectional: allow you to scroll in both directions `(i++, i--)`
  - Random Access: allow you to jump from one element to another `(i++,i--,i+=5,i-=5)`

Iterators on a `vector`

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

using namespace std;

int main() {

  std::vector<int> v {3,2,6,4,5} ;

  cout << "Printing original vector : " << endl;
  vector<int>::iterator it ;
  for ( it = v.begin() ; it != v.end(); it++ ) std::cout << *it << std::endl;

  cout << "Printing original vector (again) : " << endl;
  for ( vector<int>::iterator it = v.begin() ; it != v.end(); it++ ) std::cout << *it << std::endl;

  for ( vector<int>::iterator it = v.begin()+2 ; it != v.end(); it++ ) {
    cout << "Replacing " << *it << " with 7 " << endl;
    *it = 7;
  }

  v.insert(v.begin(),-99);

  cout << "Printing modified vector : " << endl;
  for (auto it = v.begin() ; it != v.end() ; it++ )  std::cout << *it << std::endl;

}
```

Create an iterator and loop over the container : get begin and end from the container itself !

Same but in a more compact way

Iterate over a subrange

Insert an element in front

Hey look at this magic word : **auto** asks the compiler to deduce the type automatically !

Algorithms

- There are logically identical algorithms that do not depend on a particular implementation of the data structures on which they operate but only on some semantic properties
  - It is possible to abstract them from the particular implementation so as not to lose efficiency
- The STL provides algorithms that can be used in a general way on different types of containers
  - Algorithms operate on container elements only indirectly via iterators

    - http://www.cplusplus.com/reference/algorithm/
    - http://www.cplusplus.com/reference/numeric/

The `std::sort` algorithm

```
template <class RandomAccessIterator>  void sort (RandomAccessIterator first, RandomAccessIterator last);
```

```
#include <vector>
#include <algorithm>

int main( int argc , char** argv ) {

  // ...

  std::vector<double> v {4.,5.,7.,8.,2.} ;

  std::sort( v.begin() , v.end() ) ;

  // ...

}
```

The STL sort() function:

- The sort() function requires the start and end iterators as input
- The begin() and end() method is implemented in all STL containers.
- The container returns me a 'generic' object, the iterator. This way sort() works container-independently! (if the iterator is random access: i.e., it doesn't work on lists)

This function internally uses IntroSort. In more details it is implemented using hybrid of QuickSort, HeapSort and InsertionSort. By default, it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N*logN time, it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort.

Having fun with STL : compute average (I)

The usual way : a `vector` is passed by reference (with a const). Loop over the container through iterators

```
template <typename T> double CalcolaMedia( const vector<T>& v  ) {
  double accumulo = 0 ;
  if ( v.size() == 0 ) return accumulo;
  for ( auto it = v.begin() ; it != v.end(); it++ ) accumulo += *it ;
  return accumulo / double( v.size() )  ;
};
```

```
int main( int argc , char** argv ) {
  vector<double> v1 = ReadAll<double>( argv[1] );
  cout << "media    = " << CalcolaMedia( v1 ) << endl;
}
```

Having fun with STL : compute average (II)

Pass iterators ! (ideally working on any container)

```
template <typename T> double CalcolaMedia( T _start , T _end ) {

  double accumulo = 0;

  if ( (_end-_start)  == 0 ) return accumulo;

  for ( T it = _start; it != _end; it++ ) accumulo += *it ;

  return accumulo / double(_end-_start)  ;

};
```

Pass iterators ! (ideally working on any container) and use accumulators

```
template <typename T> double CalcolaMediaAcc( T _start , T _end ) {

  if ( (_end-_start)  == 0 ) return 0;
  return std::accumulate(_start, _end, 0.0) /  double(_end-_start)   ;

}
```

```
int main( int argc , char** argv ) {

  vector<double> v1 = ReadAll<double>( argv[1] );

  cout << "media    = " << CalcolaMedia( v1.begin(), v1.end() ) << endl;

}
```

The `Posizione` class

Suppose we need to represent the concept of space-position (in three dimensions in this concrete case, can be a vector of double in case of multi-dimensional space)

```
// ========================================================================
// This is the main class designed to hold a position ( three numbers)
// In principle I should also re-define the copy constructors and
// assigment operators but luckily the default ones are ok in this case
// ( no dynamic allocation of memory inside )
// ========================================================================

class Posizione {

public :

  Posizione() ;
  Posizione(double x , double y, double z ) ;

  double getX() const { return m_x ; } ;
  double getY() const { return m_y ; } ;
  double getZ() const { return m_z ; } ;

  void setX(double x) { m_x = x ;  } ;
  void setY(double y) { m_y = y ;  } ;
  void setZ(double z) { m_z = z ;  } ;

  double getDistance() const ;
  double getDistance( Posizione p ) const ;
  void printPosition() const ;

private :

  double m_x , m_y , m_z ;

};
```

Posizione.h

constructors

Access to elements

Compute distances

Real data members

The `Posizione` class

We can enrich our class providing additional useful operators :

- define some possible operations among `Posizione` objects and among `Posizione` and a double
- define the concept of "<" between `Posizione` objects

```
class Posizione {

public :

  Posizione() ;
  Posizione(double x , double y, double z ) ;

  double getX() const { return m_x ; } ;
  double getY() const { return m_y ; } ;
  double getZ() const { return m_z ; } ;

  void setX(double x) { m_x = x ;  } ;
  void setY(double y) { m_y = y ;  } ;
  void setZ(double z) { m_z = z ;  } ;

  double getDistance() const ;
  double getDistance( Posizione p ) const ;
  void printPosition() const ;

  bool operator< (const Posizione & b) const { return getDistance()<b.getDistance(); };

  Posizione operator+( Posizione p1 ) const ;
  Posizione operator/( double d ) const ;

private :

  double m_x , m_y , m_z ;

};
```

Posizione.h

Define the meaning of "less than" (defined inline fo leaziness)

Define the meaning of sum of positions and division of a position by a constant

Example : the `Posizione` class

```
Posizione::Posizione() {
  m_x = 0 ;
  m_y = 0 ;
  m_z = 0;
} ;

Posizione::Posizione(double x , double y, double z ) {
  m_x = x ;
  m_y = y ;
  m_z = z ;
} ;

double Posizione::getDistance() const {
  return sqrt( m_x * m_x + m_y*m_y + m_z * m_z ) ;
} ;

double Posizione::getDistance( Posizione p ) const {
  return sqrt( pow(m_x-p.getX(),2) + pow(m_y-p.getY(),2) + pow(m_z-p.getZ(),2) ) ;
};

void Posizione::printPosition() const {
  cout << "Posizione : x = " << m_x
       << "  y = " << m_y
       << "  z = " << m_z << endl;
} ;

Posizione Posizione::operator+( Posizione p1 ) const {
  Posizione sum ;
  sum.setX ( getX() + p1.getX() ) ;
  sum.setY ( getY() + p1.getY() ) ;
  sum.setZ ( getZ() + p1.getZ() ) ;
  return sum;
}

Posizione Posizione::operator/( double d ) const {
  Posizione frac ;
  frac.setX(  getX()/d ) ;
  frac.setY(  getY()/d ) ;
  frac.setZ(  getZ()/d ) ;
  return frac ;
}
```

Posizione.cpp

Class implementation (Posizione.cpp)

- constructors
- distance from the origin
- distance from another `Posizione`
- sum of two `Posizione` and division of a `Posizione` by a constant

Containers are very general : container of positions !

```
// ========================================================================
// In this main I create a vector of positions and play with it.
// Compute the average
// ========================================================================

int main() {

  Posizione p1(3,3,3) ;
  Posizione p2(2,2,2) ;
  Posizione p3(1,1,1) ;

  Posizione p4 = p1+p2 ;
  cout << "Summing now p1 and p2 " << endl;
  p4.printPosition() ;

  vector<Posizione> vp {p1,p2,p3} ;
  cout << vp[1].getX() << endl;
  for (auto it = vp.begin() ; it!=vp.end() ; it++ ) it->printPosition() ;

  cout << "Now compute average position from p1, p2, p3 " << endl;
  Posizione pmed = CalcolaMedia<Posizione>(vp) ;
  pmed.printPosition();

  return 0 ;

}
```

Create three Posizioni objects

Sum 2 Posizioni

Create a vector of Posizioni !

Compute the average position

`CalcolaMedia()` is defined to return a T object : it works on Positions, now we know the meaning of sum of Posizioni and division of a Posizione by a constant

```
// ========================================================================
// This is the usual template function to compute the average of the
// elements in a vector
// ========================================================================

template <typename T> T CalcolaMedia( const vector<T> & v) {

    T accumulo ;
    if( v.size() ==0 ) return accumulo;

    for(int k=0 ; k< v.size() ; k++) { accumulo = accumulo + v[k]; }

    return accumulo / double( v.size() ) ;

};
```

How to order a set of `Posizione` objects ?

Let's take the usual pragmatic approach : I would like to sort a container of positions (?!?), what does this mean exactly ? I need to define a way to compare to positions

Two versions of the sort() algorithm in the STL : https://cplusplus.com/reference/algorithm/sort/

1. **Option 1**: use sort with no explicit comparator (requires the operator< to be defined inside the class of objects (`Posizione`) stored in the container)

```
template <class RandomAccessIterator> void sort (RandomAccessIterator first,
                                                 RandomAccessIterator last);
```

Notice RandomAccessIterator (doesn't work for lists)

2. **Option 2**: allows to define your own ordering criterium

```
template <class RandomAccessIterator, class Compare> void sort (RandomAccessIterator first,
                                                                RandomAccessIterator last,
                                                                Compare comp);
```

comp is a binary function that accepts two iterators and returns a value convertible to bool. We can use a function, functor or a lambda function (see next slides)

Option1: std::sort with no comparator

```
// Order using the simple sort

int main() {

  Posizione p1(3,3,3) ;
  Posizione p2(2,2,2) ;
  Posizione p3(1,1,1) ;

  vector<Posizione> vp {p1,p2,p3} ;

  Posizione p4 = p1+p2 ;
  vp.push_back(p4);

  for (auto it = vp.begin() ; it!=vp.end() ; it++ ) it->printPosition() ;

  // sort with no criterium, use the class operator<

  sort( vp.begin(), vp.end()  )

  for (auto it = vp.begin() ; it!=vp.end() ; it++ ) it->printPosition()

  return 0 ;

}
```

```
class Posizione {

public :

  Posizione() ;
  Posizione(double x , double y, double z ) ;

  double getX() const { return m_x ; } ;
  double getY() const { return m_y ; } ;
  double getZ() const { return m_z ; } ;

  void setX(double x) { m_x = x ;  } ;
  void setY(double y) { m_y = y ;  } ;
  void setZ(double z) { m_z = z ;  } ;

  double getDistance() const ;
  double getDistance( Posizione p ) const ;
  void printPosition() const ;

  bool operator< (const Posizione & b) const { return getDistance()<b.getDistance(); };

  Posizione operator+( Posizione p1 ) const ;
  Posizione operator/( double d ) const ;

private :

  double m_x , m_y , m_z ;

};
```

This is sorting option 1 : third input field is missing, relies on the `operator<` definition inside the class

Option2: std::sort using a dedicated comparator

For option 2 we need to create a comparator : the comparator can be a simple function returning a Boolean, a functor operator() or a *lambda function*

comparators.h

A simple bool function

A class called "functor" : we will use its operator() as a comparator

A lambda function

```
// this is a comparator function for the sorting

bool mycom ( Posizione p1 , Posizione p2 ) { return p1.getDistance() < p2.getDistance(); } ;

// this is a comparator functor ( useful when one need external parameters )

class comp_functor {

public :

  comp_functor( ) { m_ref.setX(0) ; m_ref.setY(0); m_ref.setZ(0); } ;
  comp_functor( Posizione p ) { m_ref = p ; } ;

  bool operator() (Posizione p1, Posizione p2 ) {return p1.getDistance(m_ref) < p2.getDistance(m_ref);} ;

private :

  Posizione m_ref;
};

// this is a lambda function : general, flexible and powerful

auto comp_lambda =  [] (Posizione i , Posizione j ) { return i.getDistance() < j.getDistance();} ;
```

Option2: std::sort using a dedicated comparator

```
// Order using comparators

#include "comparators.h"

int main() {

  Posizione p1(3,3,3) ;
  Posizione p2(2,2,2) ;
  Posizione p3(1,1,1) ;

  vector<Posizione> vp {p1,p2,p3} ;

  Posizione p4 = p1+p2 ;
  vp.push_back(p4);

  for (auto it = vp.begin() ; it!=vp.end() ; it++ ) it->printPosition() ;

  // sort with respect to a default Posizione ( 0,0,0 )

  sort ( vp.begin() , vp.end() , mycom );
  sort ( vp.begin() , vp.end() , comp_functor() );
  sort ( vp.begin() , vp.end() , comp_lambda);
  sort ( vp.begin() , vp.end() ,
         [] (Posizione i , Posizione j ) { return i.getDistance() < j.getDistance();});

   for (auto it = vp.begin() ; it!=vp.end() ; it++ ) it->printPosition() ;

  return 0 ;

}
```

This is sorting option 2 : third input field is a comparator: a function, a functor or a lambda function

The STL in its full power: lambda functions

C++ 11 introduced lambda expression to allow us write an inline function which can be used for short snippets of code that are not going to be reuse and not worth naming. In its simplest form lambda expression can be defined as follows:

```
[ capture clause ] (parameters) -> return-type
{
   function code
}
```

- Generally, return-type in lambda expression are evaluated by compiler itself and we don't need to specify that explicitly and -> return-type part can be ignored but in some complex case as in conditional statement, compiler can't make out the return type and we need to specify that.
- Syntax used for capturing variables :
  - [ ] can access only those variable which are local to it
  - [&] : capture all external variable by reference
  - [=] : capture all external variable by value
  - [a, &b] : capture a by value and b by reference

How to order a set of `Posizione` objects wrt to a custom `Posizione` ref ?

Let's go deeper in our problem: suppose that now I want to sort the positions depending on the distance from an external reference Posizione and not from the origin.

NOT POSSIBLE with a simple function !

```
// this is a comparator function for the sorting

bool mycom ( Posizione p1 , Posizione p2 ) { return p1.getDistance() < p2.getDistance(); } ;


// this is a comparator functor ( useful when one need external parameters

class comp_functor {

public :

  comp_functor( ) { m_ref.setX(0) ; m_ref.setY(0); m_ref.setZ(0) } ;
  comp_functor( Posizione p ) { m_ref = p ; } ;

  bool operator() (Posizione p1, Posizione p2 ) {return p1.getDistance(m_ref) < p2.getDistance(m_ref);} ;

private :

  Posizione m_ref;
};
```

An external parameter (ref) can be passed through constructor in a functor

The STL in its full power: functions, functors and lambdas

```cpp
// Order using comparators

#include "comparators.h"

int main() {

  Posizione p1(3,3,3) ;
  Posizione p2(2,2,2) ;
  Posizione p3(1,1,1) ;

  vector<Posizione> vp {p1,p2,p3} ;

  Posizione p4 = p1+p2 ;
  vp.push_back(p4);

  for (auto it = vp.begin() ; it!=vp.end() ; it++ ) it->printPosition() ;

  // sort with respect to an external referene

  Posizione ref(10,10,10) ;

  sort ( vp.begin() , vp.end() , comp_functor( ref) );
  sort ( vp.begin() , vp.end() ,
         [&] (Posizione i , Posizione j ) { return i.getDistance(ref) < j.getDistance(ref);});

  auto comp_lambda_ref =  [&] (Posizione i , Posizione j ) { return i.getDistance(ref) < j.getDistance(ref);} ;
  sort ( vp.begin() , vp.end() , comp_lambda_ref );

  for (auto it = vp.begin() ; it!=vp.end() ; it++ ) it->printPosition() ;

  return 0 ;

}
```

Sort using the functor (the reference is passed through the constructor) or directly a lambda function. The lambda captures the `Posizione` ref by reference ([&])

Why do we gain something with lambdas ?

```
// #include ....

int main() {

  Posizione p1(3,3,3) ;
  Posizione p2(2,2,2) ;
  Posizione p3(1,1,1) ;

  vector<Posizione> vp {p1,p2,p3} ;

  Posizione ref(10,10,10) ;

  sort (vp.begin(),
        vp.end(),
        [&] (Posizione i, Posizione j) { return (i.getDistance(ref)<j.getDistance(ref)) ;} );

  return 0 ;

}
```

Hey, look at this : reference is a position allocated in the main !

Capture values from the main by reference

One single line does the job : order a set of `Posizione` objects wrt the distance from a custom reference `Posizione`, cool !

Conclusions

- Template classes and functions allow us to do generic programming
- STL is the most refined implementation of this concept: data containers, iterators and algorithms. Extreme performance optimization
- The clear separation between containers and algorithms (mediated by iterators) is a key element of this approach
  - Why does media() have to be a `vector` method?

Backup

`string` in C++

- To represent "words" in C++ we can use an array of chars from C:
  - ```
    char myWord[10] ;
    myWord[0] = 'h';
    myWord[1] = 'e';
    myWord[2] = 'l';
    myWord[3] = 'l';
    myWord[4] = 'o';
    ```
- C++ has no real specific type for dealing with character sequences:
- The standard C++ libraries provide the `string` class, (`#include <string>`) which avoids all buffer allocation problems
  - the string is internally considered as a simple array of characters terminated by a NULL character (\0).
  - Series of useful methods to simplify operations
    ```

`string` in C++

A few features (all the rest in http://www.cplusplus.com/reference/string/string/):

- **`getline(cin, mystring)`**: this function is used to store a stream of characters as entered by the user in the object memory.

- **`mystring.push_back('s')`**: used to input a character at the end of the string.
- **`mystring.pop_back()`** : used to delete the last character from the string.
- **`mystring.capacity()`** : returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.
- **`mystring.length()`**: returns the length of the string
- **`mystring.begin()`** : returns an iterator to beginning of the string.
- **`mystring.end()`**: returns an iterator to end of the string.

Le string

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

using namespace std;

int main() {

  string nome = "Leonardo";
  cout << "Il mio nome = " << nome << endl;

  char c_cognome[30] = "Carminati";
  string cognome = c_cognome;

  cout << "Cognome = " << cognome << endl;
  cout << "Lunghezza cognome = " << cognome.size() << endl;

  string record = nome + " " + cognome;
  cout << record << endl;

  record.insert(nome.size()+1,"Carlo ");
  cout << record << endl;

  cout << record.find( "Carlo", 0) << endl;

  vector<string> cognomi;

  cout << "Inserisci cognome " << endl;
  cin >> cognome ;
  cognomi.push_back(cognome);

}
```

Create a string from an array of charachters

Concatenate strings

Insert a substring in a given position

Fill a string from standard input

The `Posizione` class example :

```
class Posizione {

public :

  Posizione() ;
  Posizione(double x , double y, double z ) ;

  double getX() const { return m_x ; } ;
  double getY() const { return m_y ; } ;
  double getZ() const { return m_z ; } ;

  double getR()     const { ... } ;
  double getPhi()   const { ... } ;
  double getTheta() const { ... } ;
  double getRho()   const { ... } ;

  void setX(double x) { m_x = x ;  } ;
  void setY(double y) { m_y = y ;  } ;
  void setZ(double z) { m_z = z ;  } ;

  double getDistance() const ;
  double getDistance( Posizione p ) const ;
  void printPosition() const ;

  bool operator< (const Posizione & b) const { return getDistance()<b.getDistance(); };

  Posizione operator+( Posizione p1 ) const ;
  Posizione operator/( double d ) const ;

private :

  double m_x , m_y , m_z ;

};
```

Let's add other features to the class: for example the conversion to spherical or polar coordinates

Encapsulation and data hiding

```
Int main() {

  foo test(99);
  double myVar = test.getX();
  cout << test.compute() << endl;

  test.setX(88);
  cout << test.compute() << endl;

}
```

- The main can't access directly data in the class
- The access is regulated by interfaces (constructors, methods)
- **Why ?**

- Normally the main and the classes are written by different persons
- Encapsulation guarantees that some rules are placed in the communication between main and classes
- A change inside the class doesn't have any impact on the main (in terms of coding)

```
class foo{
public:
        foo(double x);
        double setX(double x);
        double getX();
        double compute()
private:
        m_x;
}
```

Encapsulation and data hiding

```cpp
#include "Posizione.h"
#include <cmath>

// costruttore di default
Posizione::Posizione() {
  m_x=0.;
  m_y=0.;
  m_z=0.;
}
// costruttore a partire da una terna cartesiana
Posizione::Posizione(double x, double y, double z) {
  m_x=x;
  m_y=y;
  m_z=z;
}
// distruttore (puo' essere vuoto)
Posizione::~Posizione() {
}

// Coordinate cartesiane
double Posizione::getX() const {
  return m_x;
}
double Posizione::getY() const {
  return m_y;
}
double Posizione::getZ() const {
  return m_z;
}

// Coordinate sferiche
double Posizione::getR() const {
  return sqrt(m_x*m_x+m_y*m_y+m_z*m_z);
}
double Posizione::getPhi() const {
  return atan2(m_y,m_x);
}
double Posizione::getTheta() const {
  return acos(m_z/R());
}

// raggio delle coordinate cilindriche
double Posizione::getRho() const {
  return sqrt(m_x*m_x+m_y*m_y);
}
```

```cpp
#include "Posizione.h"
#include <cmath>

// costruttore di default
Posizione::Posizione() {
  m_x=0.;
  m_y=0.;
  m_z=0.;
}
// costruttore a partire da una terna cartesiana
Posizione::Posizione(double x, double y, double z) {
  m_x=sqrt(x*x+y*y+z*z);        // R
  m_y=atan2(y,x);               // phi
  if ( m_x>0. ) m_z=acos(z/m_x);  // theta
  else m_z=0.;
}
// distruttore (puo' essere vuoto)
Posizione::~Posizione() {
}

// Coordinate cartesiane
double Posizione::getX() const {
  return m_x*cos(m_y)*sin(m_z);
}
double Posizione::getY() const {
  return m_x*sin(m_y)*sin(m_z);
}
double Posizione::getZ() const {
  return m_x*cos(m_z);
}

// Coordinate sferiche
double Posizione::getR() const {
  return m_x;
}
double Posizione::getPhi() const {
  return m_y;
}
double Posizione::getTheta() const {
  return m_z;
}

// raggio delle coordinate cilindriche
double Posizione::getRho() const {
  return m_x*sin(m_z);
}
```

For some reason we decide that the three internal variables of the class are no longer the Cartesian coordinates but the spherical ones (complete redesign of the class). We can adapt the accessor methods and constructors

No changes in main! (if the main had had direct access to m_x, m_y and m_z I would have had to modify the main too, causing enormous damage to the users of my class!)

Template classes and functions

```
#ifndef __Vettore_h__
#define __Vettore_h__

#include <iostream>

using namespace std;

template <typename T> class Vettore {

 public:

  Vettore() ;
  Vettore(  unsigned int N) ;

  ~Vettore() { delete [] m_v ;} ;

   unsigned int GetN() const { return m_N;}
  void SetComponent(  unsigned int i , T a );
  T GetComponent(  unsigned int i ) const ;

  void Scambia(  unsigned int primo ,  unsigned int secondo );

 private:

   unsigned int m_N;

   T* m_v;

};

#endif // __Vettore_h__
```

```
template <typename T> Vettore<T> Read ( unsigned int N , char* filename ) {

  Vettore<T> v(N);

  ifstream in(filename);

  if ( !in ) {
    cout << "Cannot open file " << filename << endl;
    exit(11);
  } else {
    for (unsigned int i=0; i<N; i++) {
      T val ;
      in >> val ;
      v.SetComponent( i, val ) ;
      if ( in.eof() ) {
        cout << "End of file reached exiting" << endl;
        exit(11);
      }
    }
  }
  return v;
}
```

More on `list` and `vector`

`list` and `vector` containers have pros and cons

```
#include <list>
#include <vector>
#include <iostream>
using namespace std;

int main() {

  std::vector<double> v {4,5,7,8} ;
  cout << "vector : size = " << v.size() << "   capacity = " << v.capacity() << endl;

  std::list<double> l {3,4,5}; ;
  cout << "list : size = " << l.size() << endl;

  v.push_back(6);
  l.push_back(6);

  v.insert(v.begin()+2 , -99) ;
  l.insert(++l.begin() , -99) ;

  cout << "Accessing component of a vector " << v[2] << endl;
  // cout << "Accessing component of a list " << l[2] << endl;

  cout << "Print the list content" << endl;
  for ( auto x : l ) cout << x << endl;
  cout << "Print the vector content" << endl;
  for ( auto x : v ) cout << x << endl;

}
```

No capacity() method for a list

Queuing ok for both containers

Insertion (non-queuing) optimal for list, suboptimal for vector

it is not possible to access the i-th element of a list with [] (no random access iterator!)

Having fun with STL : read all elements from file (I)

```
// ========================================================================
// Questa funzione legge N elementi da file e li carica in un vettore
// ========================================================================

template <typename T> vector<T> ReadAll1( const char* filename ) {

  vector<T> v;

  ifstream in(filename);

  if ( !in ) {
    cerr << "File " << filename << " doesn't exist" << endl ;
    exit(11);
  }

  T appo;
  while ( in >> appo ) v.push_back(appo) ;

  in.close();

  cout << "New ReadAll1 : vector size = " << v.size() << endl;

  return v;

};
```

The usual way : loop over the file and push_back elements

Having fun with STL : read all elements from file (II)

```
// ============================================================================
// Questa funzione legge N elementi da file e li carica in un vettore
// ============================================================================

template <typename T> vector<T> ReadAll2( const char* filename ) {

  ifstream in(filename);

  if ( !in ) {
    cerr << "File " << filename << " doesn't exist" << endl ;
    exit(11);
  }

  std::vector<T> v;
  std::istream_iterator<T> start(in) ;
  std::istream_iterator<T> end;

  while ( start != end ) v.push_back(*start++);

  in.close();

  cout << "ReadAll2 : vector size = " << v.size() << endl;

  return v;

};
```

Create an iterator over the ifstream (input file). Get pointer to the first element (start). End is a pointer to end-of-stream

Loop over the stream through iterators : access to the element and increment pointer until the end of the stream

Having fun with STL: read all elements from file (III)

```
// ========================================================================
// Questa funzione legge tutti gli elementi di un file e li carica in un vettore
// ========================================================================

template <typename T> vector<T> ReadAll3( const char* filename ) {

  ifstream in(filename);
  if ( !in ) {
    cerr << "File " << filename << " doesn't exist" << endl ;
    exit(11);
  }

  std::vector<T> v ((std::istream_iterator<T>(in)) ,std::istream_iterator<T>()) ;

  in.close();

  cout << "ReadAll3 : vector size = " << v.size() << endl;

  return v;

};
```

Use a specific vector constructor that accept the iterators to the starting point and end points of the stream

Having fun with STL : read all elements from file (IV)

```
// ========================================================================
// Questa funzione legge tutti gli elementi di un file e li carica in un vettore
// ========================================================================

template <typename T> vector<T> ReadAll4( const char* filename ) {

  ifstream in(filename);

  if ( !in ) {
    cerr << "File " << filename << " doesn't exist" << endl ;
    exit(11);
  }

  std::vector<T> v ;
  std::copy( std::istream_iterator<T>(in) ,std::istream_iterator<T>(),
             back_inserter(v) ) ;

  in.close();

  cout << "ReadAll4 : vector size = " << v.size() << endl;

  return v;

};
```

Use std::copy algorithm

back_inserter makes sure that the the size of v is sufficient (elements are push_back(ed))

Having some fun with `vector`

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

using namespace std;

int main() {

  // reserve

  vector<double> v4(5) ;
  v4.push_back(99);
  cout << v4.size() << "
  v4.reserve(10);
  cout << v4.size() << " " << v4.capacity() << endl;
  cout << v4.at(8) << endl;

  return 0;

}
```

Create a `vector` with size=capacity=5

push_back an element, size becomes 6

Increase capacity to 10

Will give an error : element 8 doesn't exist
(size is still 6)

Having some fun with `vector`

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

using namespace std;

int main() {

  // resize

  vector<double> v5(5) ;
  v5.push_back(99);
  cout << v5.size() << " " << v5.capacity() << endl;
  v5.resize(10,0);
  cout << v5.size() << " " << v5.capacity() << endl;
  cout << v5.at(8) << endl;

 return 0;

}
```

Create a `vector` with size=capacity=5

push_back an element, size becomes 6

Fill the vector with zeros up to size 10

Ok, element 8th exists

Why should I care about iterators ? Algorithms !

```cpp
#include <vector>
#include <list>
#include <iostream>
#include <numeric>
#include <algorithm>

using namespace std;

int main() {

  std::vector<double> v {3,2,6,4,5,5} ;

  //  vector<double>::iterator start = v.begin();
  //  vector<double>::iterator end = v.end();

  auto start = v.begin();
  auto end = v.end();

  double media =  std::accumulate( start , end , 0.0 ) /  (double)(end - start) ;
  //  double media =  std::accumulate( start , end , 0.0 ) /  (double)(v.size()) ;
  cout << "Compute average : " << media << endl;

  cout << "Printing original vector : " << endl;
  for (auto it = v.begin() ; it != v.end() ; it++ )  std::cout << *it << std::endl;

  std::sort( v.begin(), v.end() ) ;
  // std::sort( start , end ) ;

  cout << "Printing sorted vector : " << endl;
  for (auto it = v.begin() ; it != v.end() ; it++ )  std::cout << *it << std::endl;

  std::list<double> l {3,2,6,4,5,5} ;

  double medial =  std::accumulate( l.begin() , l.end() , 0.0 ) /  (double)(l.size()) ;
  cout << "Compute average (list)" << medial << endl;

  l.sort();
  cout << "Printing sorted list :" << endl;
  for (auto it = l.begin() ; it != l.end() ; it++ )  std::cout << *it << std::endl;

}
```

The STL (and Boost libraries) provides generic algorithms that work with iterators, regardless of the container on which they are applied!

`std::accumulate` allows you to add all the elements of the container from start to end (accumulated starting from 0)

`sort()`: reorders the elements from least to greatest in the range specified by the iterators

The `list` is the only case of a dedicated `sort()` method (it has no random access iterators)