A couple of words on the dynamic memory allocation

A pointer is declared. The pointer <u>points to nothing</u> (meaning the actual array doesn't exist yet)

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

  unsigned int size = atoi(argv[1]);

  double * v = nullptr ;

  // ....

  v = new double[size];

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

  // ...

  delete[] v;

}
```

Here the `new` operator allocates `size` places to host `double` : the actual array is created here !

The class `Vettore` in its full glory

```cpp
#ifndef __Vettore_h__
#define __Vettore_h__

#include <iostream>

using namespace std;

class Vettore {
 public:

  Vettore();                                  // constructor
  Vettore( int N );                           // constructor
  Vettore(const Vettore&);                    // copy constructor

  ~Vettore();                                 // destructor

  Vettore operator=(const Vettore&);  // assignment operator

  unsigned int GetN() const { return m_N;} ;
  void SetComponent( int , double) ;
  double GetComponent( int ) const ;

  double& operator[]( int ) const ;    // access operator

  void Scambia( int, int ) ;

 private:

  unsigned int m_N;
  double* m_v;

};

#endif // __Vettore_h__
```

1. Create `Vettore` objects or pointers

Include the header file of the `Vettore` class

Build a `Vettore` object using the constructor with no arguments (notice no parentheses!)

Build a `Vettore` object using the constructor with size as input

Build a `Vettore` object using the constructor with size as input (uniform initialization)

Build a `Vettore` pointer using the constructor with no arguments

Build a `Vettore` pointer using the constructor with size as input

Manipulate the `Vettore` both as object (.) or pointer (->)

```
#include <iostream>

#include "Vettore.h"

int main() {

  Vettore myvett_obj_1;
  Vettore myvett_obj_2(10) ;
  Vettore myvett_obj_3 {10} ;

  Vettore *myvett_poi_1 = new Vettore();
  Vettore *myvett_poi_2 = new Vettore(10);
  Vettore *myvett_poi_3 = new Vettore {10};

  myvett_obj_2.SetComponent(2,55) ;
  myvett_poi_2->SetComponent(2,55) ;

  cout << myvett_obj_2.GetComponent(2) << endl;
  cout << myvett_poi_2->GetComponent(2) << endl;

  cout << "Size of my vector is " << myvett_obj_1.GetN() << endl;
  cout << "Size of my vector is " << myvett_obj_3.GetN() << endl;
  cout << "Size of my vector is " << myvett_poi_2->GetN() << endl;

  return 0;

}
```

1. Be careful !

```
Vettore v (10 ) ;

Vettore * vp = new Vettore(10) ;

Vettore * vv = new Vettore[10] ;
```

- Creates <u>one object</u> of type `Vettore` using the constructor with one `unsigned int` as argument
  - one `Vettore` which holds an array of 10 places
- Creates a pointer to <u>one object</u> of type `Vettore` using the constructor with one `unsigned int` as argument
  - one `Vettore` which holds an array of 10 places)
- Creates <u>an array of 10 objects</u> of type `Vettore` using the constructor with no arguments for each object
  - ten `Vettore` each holding an array of size 0

Be careful (only for curious kids)

```
#include <iostream>
#include "Vettore.h"

int main() {

  Vettore * varray = new Vettore[5] ;
  cout << "Size = " << varray[2].GetN() << endl;

  unsigned int nv = 10;
  Vettore ** varrayp = new Vettore*[nv] ;
  for ( int k = 0 ; k < nv ; k++ ) { varrayp[k] = new Vettore(5) ;} ;

  for ( int k = 0 ; k < nv ; k++ ) {
    cout << "Size of array " << k << " = " << varrayp[k] -> GetN() << endl;
  } ;

}
```

Create an array of 5 `Vettore`. Each `Vettore` is built with the zero-argument constructor

Create an array of 10 pointers to a `Vettore`.

Each `Vettore` is built with the constructor accepting and unsigned int argument : 10 `Vettore`, each `Vettore` has a length of 5

2. On constant vectors and methods

```
class Vettore {

public :

  unsigned int GetN() const { return m_n; };
  ...

}

double CalcolaMedia ( const Vettore & v  ) {

  double accumulo = 0;

  if ( v.GetN() == 0 ) return accumulo;
  for ( int k = 0 ; k < v.GetN() ; k++ ) {
    accumulo += v.GetComponent(k) ;
  }
  return accumulo / double ( v.GetN() ) ;

};
```

This defines a constant method : the <u>method can't modify the content of the class</u>

- Here the `Vettore` v is passed <u>by reference</u>: most efficiency way (no copy !)
- a protection `const` is added so that the function `CalcolaMedia` can't modify the content of `v`

Notice that `GetN()` <u>must</u> be `const` if used in this way : you can call only constant methods on constant objects !

- Using proper `const` qualifiers might seem a bit overkilling but it's important to write robust code

3. Operator[]

```
int main ( int argc, char** argv ) {
  unsigned int size = atoi(argv[1]);
  double * v = nullptr ;
  // ....
  v = new double[size];
  for ( int k = 0; k < size ; k++ ) { v[k] = 0 } ;
  // ...
  delete[] v;
}
```

Access to the component k of the C-array

3. Operator[]

- With a C-array (`data`) one can access the i-th element by `double c=data[i]`
  - Can we do the same with an object of type `Vettore` ? Overload the `operator[]`

```
double& Vettore::operator[] (int i) const {
  if ( i >= 0 && i < m_N ) {
    return m_v[i];
  } else {
    cout << "Errore : indice non valido " << endl;
    exit(1);
  }
}
```

- `Operator[](int)` returns the i-th element by reference (i.e., a pointer to the element is returned): can be used to read and modify the i-th element
- Can have the same effect as both `GetComponent()` and `SetComponent()`.
- Must be declared `const` if used on const objects (e.g., in the `Media(const Vettore &)` method), so no more useful for writing

```
int main ( ) {

  Vettore v(10);

  v[3] = 4 ;
  cout << v[3] << endl;

  v.SetComponent(3,4);
  cout << v.GetComponent(3) << endl;

}
```

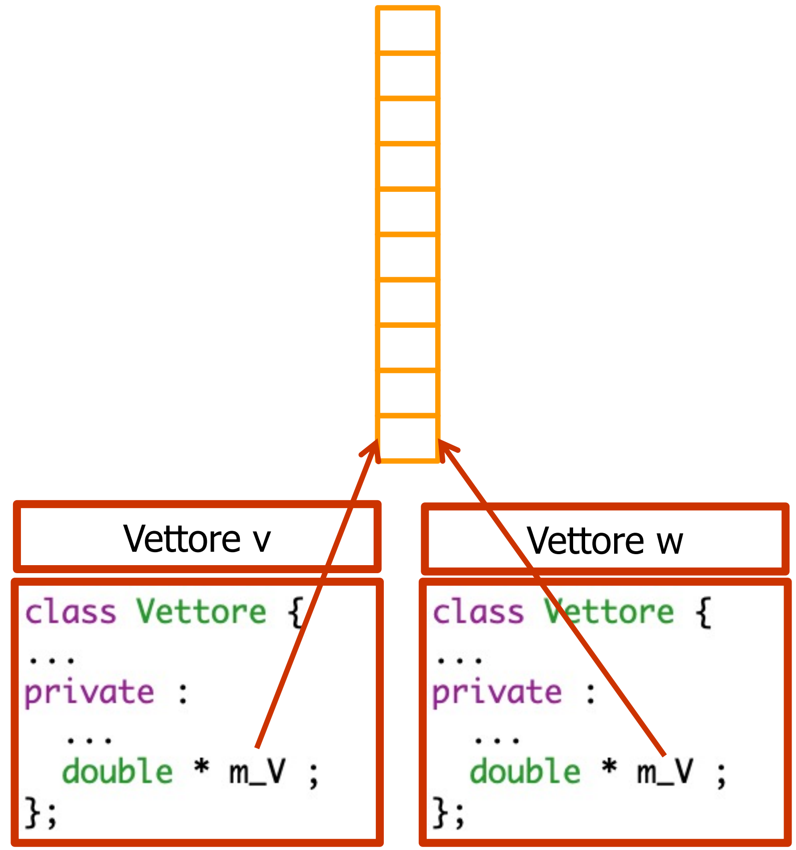
Modify component 3 of the `Vettore` v : can't be constant!

Access component 3 of the `Vettore` v

4. Copy constructor and assignment operator on a `Vettore`

- If no copy constructor is declared, a default (implicit) one will be used
- The default constructor will match `m_v` pointers of the `Vettore`: the two vectors will have the two internal pointers pointing to the same area in memory. <u>Any modification on v will be reflected on w</u>

```
int main() {

  Vettore v(10);
  cout << "Vettore v : = dimensione = " << v.GetN() << endl;
  for ( unsigned int k = 0 ; k < v.GetN() ; k++ )
    cout << v.GetComponent(k) << " " ;
  cout << endl;

  // copy constructor

  Vettore w=v;  //  o equivalentemente  Vettore w(v);

  // operatore di assegnazione

  Vettore z;
  z = v ;

  v.SetComponent(4,99);

  cout << "Vettore z : = dimensione = " << z.GetN() << endl;
  for ( unsigned int k = 0 ; k < z.GetN() ; k++ )
    cout << z.GetComponent(k) << " " ;

}
```



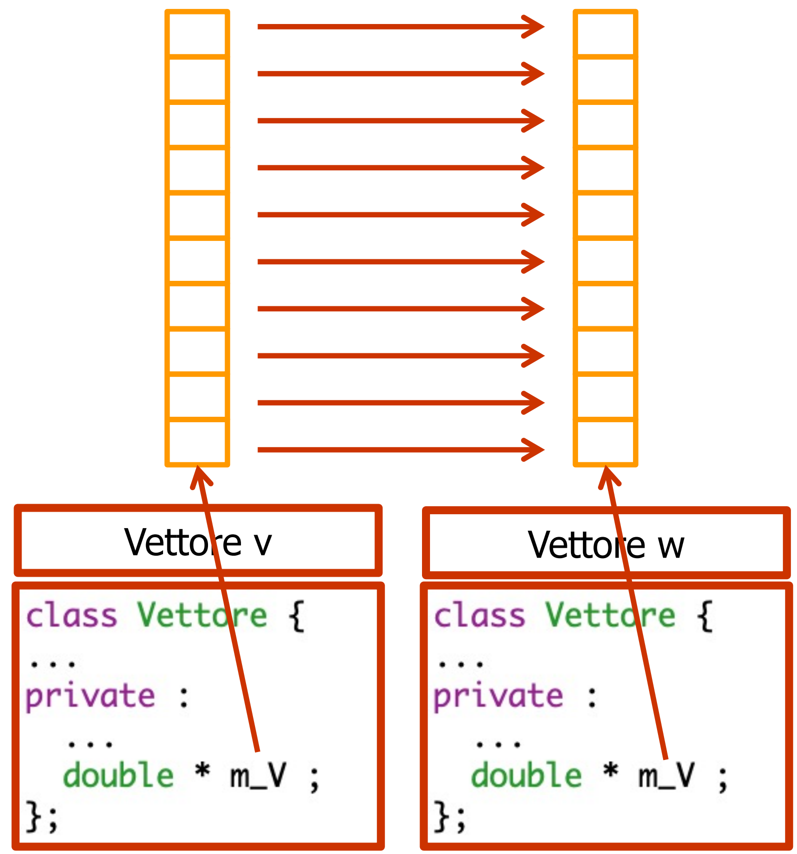
4. Copy constructor and assignment operator on a `Vettore`

- If no assignment operator is declared, a default (implicit) one will be used: same bad behavior as for the copy constructor !
- The correct assignment operator will allocate a new area in memory and copy all values from the input vector

```
// overloading costruttore di copia

Vettore::Vettore(const Vettore& V) {
  cout << "Calling copy constructor" << endl;
  m_N = V.m_N;
  m_v = new double[m_N];
  for (unsigned int i=0; i<m_N; i++) m_v[i]=V.m_v[i];
  cout << "Copy constructor called " << endl;
}
```

```
// overloading operatore di assegnazione

Vettore& Vettore::operator=( const Vettore& V) {
  cout << "Calling assignment operator" << endl;
  m_N = V.m_N;
  if ( m_v ) delete[] m_v;
  m_v = new double[m_N];
  for (unsigned int i=0; i<m_N; i++) m_v[i]=V.m_v[i];
  cout << "Assignment operator called"<< endl;
  return *this;
}
```

Free the memory already allocated first

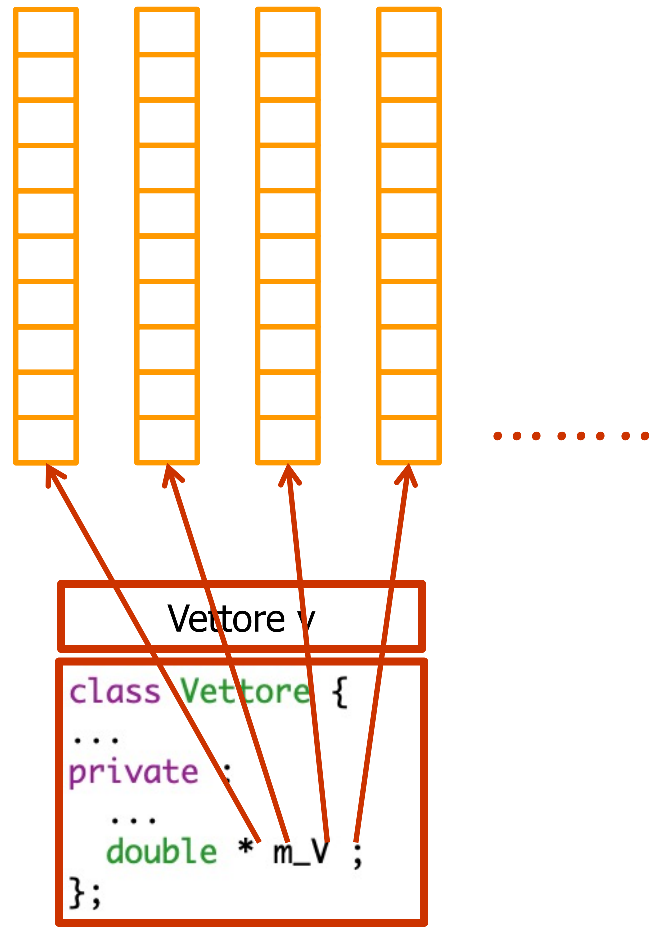


5. Default destructor

```
#include <iostream>
#include "Vettore.h"

int main() {

  for ( int k = 0 ; k < 100 ; k++ ) {

    Vettore v(10) ;

    // ... do something : read Vettore from a file,
    // compute something useful

  }

  return 0 ;

}
```

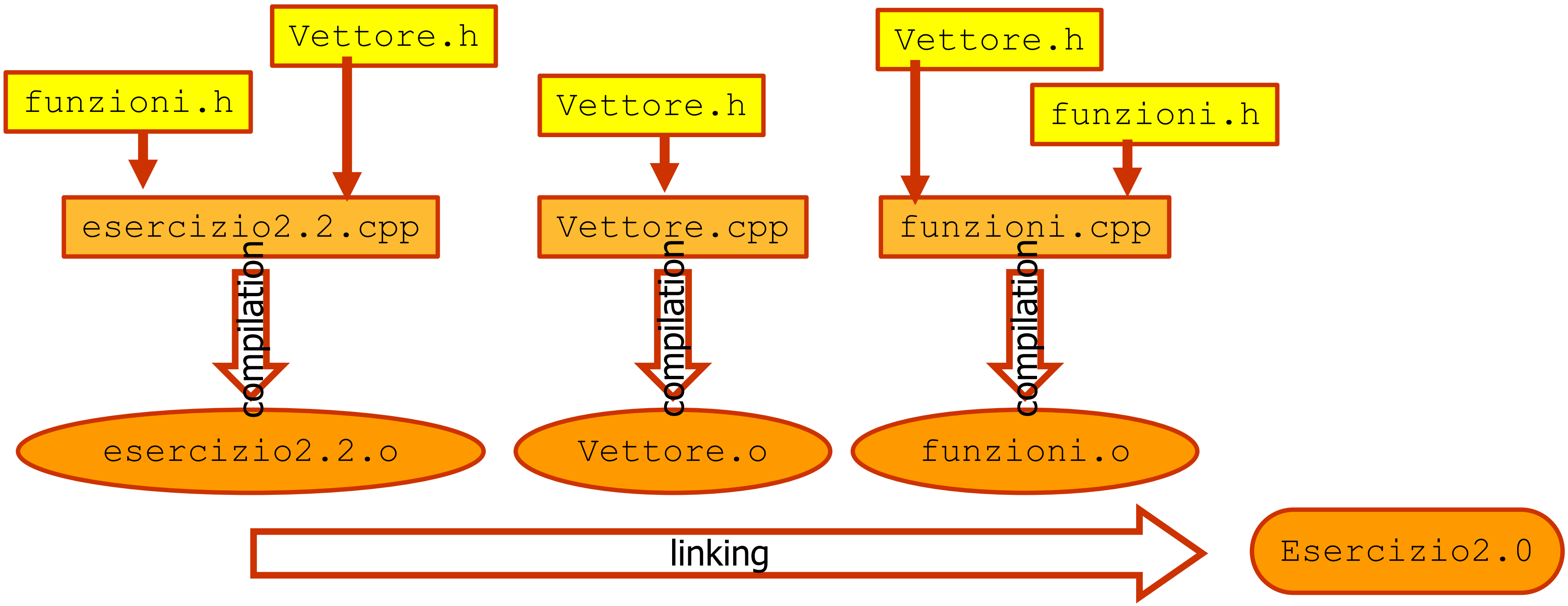
v goes out of scope

```
// distruttore

Vettore::~Vettore() {
  cout << "Calling destructor" << endl;
  delete[] m_v;
}
```



- The destructor is automatically called when the object goes out of scope (usually it doesn't need to be called explicitly).
- Consider the for loop above : an object v of type `Vettore` is created and destroyed 100 times
- If no destructor is declared, the default one will be used: it would only de-allocate `m_V` ! This could cause a memory leak

Compilation




```
esercizio2.2 : esercizio2.2.o Vettore.o funzioni.o
	g++ -o esercizio2.2 esercizio2.2.o Vettore.o funzioni.o
funzioni.o: funzioni.cpp funzioni.h Vettore.h
	g++ -c -o funzioni.o funzioni.cpp
esercizio2.2.o : esercizio2.2.cpp funzioni.h Vettore.h
	g++ -c -o esercizio2.2.o esercizio2.2.cpp
Vettore.o : Vettore.cpp Vettore.h
	g++ -c -o Vettore.o Vettore.cpp
```

Functions and template classes in C++ (and a quick and dirty introduction to STL)

Laboratorio Trattamento Numerico dei Dati Sperimentali

Prof. L. Carminati
Università degli studi di Milano

Introduction (I)

- In C++ it is possible to define templates of classes and functions by parameterizing the used types :
  - in classes, you can parameterize the types of data-members
  - in functions (and in class member functions) the types of the arguments and the return values can be parameterized.

- Templates allow the creation of "generic" functions and classes for which the type of data on which they operate is specified as a parameter: it is the foundation of generic programming
  - Key benefits: you can use functions and/or classes with different data without explicitly re-coding a different version for each different data type. They make code reuse possible.
  - In this way, the maximum independence of the algorithms from the data to which they are applied is reached:
  - for example, a sorting algorithm can be written only once independently on the type of data to be sorted (as long as an order relation is defined = ,<, >)

Introduction (II)

- Templates are resolved statically (i.e., at the compilation level) and therefore do not entail any additional costs at runtime;

- The C++ Standard Library provides pre-built structures of template classes and algorithms via template functions (Standard Template Library). Three main ingredients :
  - Container classes (linked lists, maps, vectors, etc.) that can be used by specifying, when creating objects, the specific type to replace the parameterized one.
  - Generic algorithms that work on container classes .
  - Complete independence of containers/algorithms from the data type: access elements through generic iterators

Let's look again at the `Vettore` class header file

```
#ifndef __Vettore_h__
#define __Vettore_h__

#include <iostream>

using namespace std;

class Vettore {

public:

  Vettore();
  Vettore( int N);
  Vettore(const Vettore& V) ;

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

  Vettore& operator=(const Vettore& V) ;

  double& operator[]( int ) const ;

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

  void Scambia( int primo , int secondo ) ;

private:

  int m_N;
  double* m_v;

};

#endif // __Vettore_h__
```

`Vettore` holds a bunch of `double` numbers: can we make it parametric so that it could hold any type of objects ?

Examples of `template` classes

```
#ifndef __Vettore_h__
#define __Vettore_h__

#include <iostream>

using namespace std;

class Vettore {

public:

  Vettore();
  Vettore( int N);
  Vettore(const Vettore& V) ;

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

  Vettore& operator=(const Vettore& V) ;

  double& operator[]( int ) const ;

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

  void Scambia( int primo , int secondo ) ;

private:

  int m_N;
  double* m_v;

};

#endif // __Vettore_h__
```

```
#ifndef __Vettore_h__
#define __Vettore_h__

#include <iostream>

using namespace std;

template <typename T> class Vettore {

public:

  Vettore();
  Vettore( int N);
  Vettore(const Vettore& V) ;

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

  Vettore& operator=(const Vettore& V) ;

  double& operator[]( int ) const ;

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

  void Scambia( int primo , int secondo ) ;

private:

  int m_N;
  T* m_v;

};

#endif // __Vettore_h__
```

`T` will be replaced with the true data type when creating a specific version of the function.

Examples of `template` functions

Functions working with template classes must be `template` themselves :

- The typical structure of a `template` function is the following :

```
template <typename T> ZZZZ nomefunzione (input1,input2…)
{

  // corpo della funzione

  return … ;
};
```

- When the compiler creates a specific version of a generic function, it is said to have created a “generated function”
- The act of generation is called *instantiation*: a generated function is a specific instance of a template function
- T will be replaced with the true data type when creating a specific version of the function. The compiler instantiates the template function based on the current parameters specified at the time of the call
- ZZZZ is the return type of the function

Definition of a `template` function

```cpp
Vettore Read ( unsigned int N , char* filename ) {

  Vettore v(N);

  ifstream in(filename);

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

```cpp
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;
}
```

Using `template` classes and functions

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of T by the type passed as the actual template parameter (`double` in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

```
#include <iostream>
#include <fstream>
#include <cstdlib>

#include "Vettore.h"
#include "funzioni.h"

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

  if ( argc < 3 ) {
    cout << "Uso del programma : " << argv[0] << " <n_data> <filename> " << endl;
    return -1 ;
  }

  int ndata = atoi(argv[1]);
  double* data = new double[ndata];
  char * filename = argv[2];

  // usiamo il contenitore Vector per immagazzinare double !

  Vettore <double> v = Read<double>( ndata , filename );

  Print( v );

  cout << "media    = " << CalcolaMedia<double>( v  ) << endl;
  cout << "varianza = " << CalcolaVarianza<double>( v  ) << endl;
  cout << "mediana  = " << CalcolaMediana<double>( v ) << endl;

  Print( v );

}
```

Compilation with `template` functions

Note : when we deal with template classes, we must be careful with the compilation. We use an inline implementation (="all in .h, no .cpp") for the class methods.

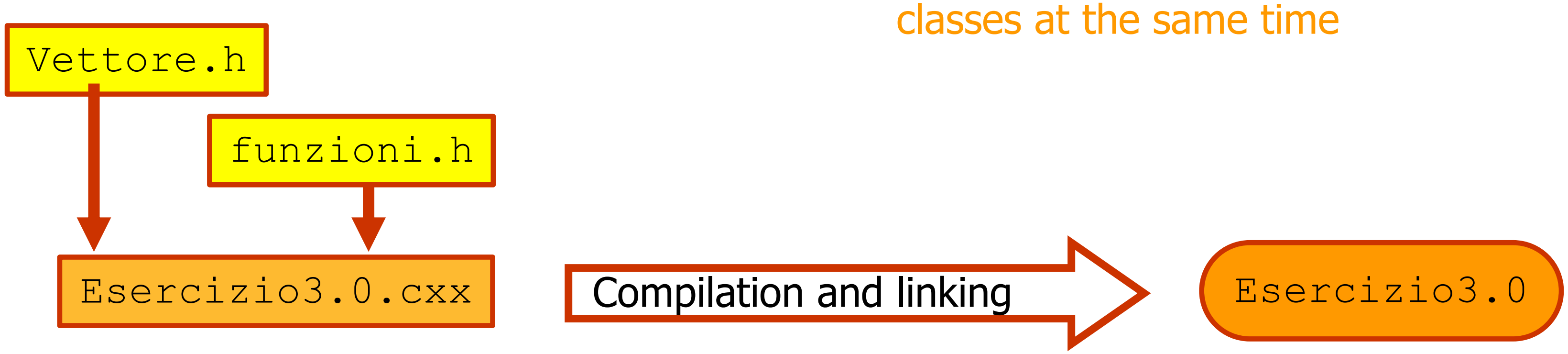
- In fact, a standalone compilation of a template class would not be possible: how could the compiler allocate the correct amount of memory if the type that will be used has not yet been defined?

```
esercizio2.1: esercizio2.1.o Vettore.o funzioni.o
        g++ esercizio2.1.o Vettore.o funzioni.o -o esercizio2.1

esercizio2.1.o: esercizio2.1.cpp funzioni.h
        g++ -c eserciio2.1.cpp -o esercizio2.1.o

funzioni.o: funzioni.cpp funzioni.h
        g++ -c funzioni.cpp -o funzioni.o

vettore.o: Vettore.cpp Vettore.h
        g++ -c Vettore.cpp -o Vettore.o
```

```
esercizio3.0 : esercizio3.0.cpp Vettore.h funzioni.h
        g++ esercizio3.0.cpp -o esercizio3.0
```

Compilation

- Through `#include` the `main()` incorporates both the declaration and the implementation of functions and classes at the same time




```
esercizio3.0 : esercizio3.0.cpp Vettore.h funzioni.h
	g++ esercizio3.0.cpp -o esercizio3.0
```

- Advantages: only one compilation instruction
- Disadvantages: a change to any of the classes or functions causes a total recompilation

An alternative way to implement the methods of the template `Vettore` class

Vettore.h

```
// ============================================================
// Dichiarazione
// ============================================================

template <typename T> class Vettore {

public :

  Vettore() ;
  Vettore( int N) ;

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

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

  //  ...

private :
  int m_N;
  T* m_v;
};

// ====================================================================
// Implementazione
// ====================================================================

template <typename T> Vettore<T>::Vettore() {

  m_N = 0;
  m_v = NULL;

} ;

template <typename T> Vettore<T>::Vettore( int N) {

  // ...

};

template <typename T>  void Vettore<T>::SetComponent( int i , T a ) {

  // ...

};

#endif // __Vettore_h__
```

To improve the code readability the methods can be implemented outside the class declaration

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

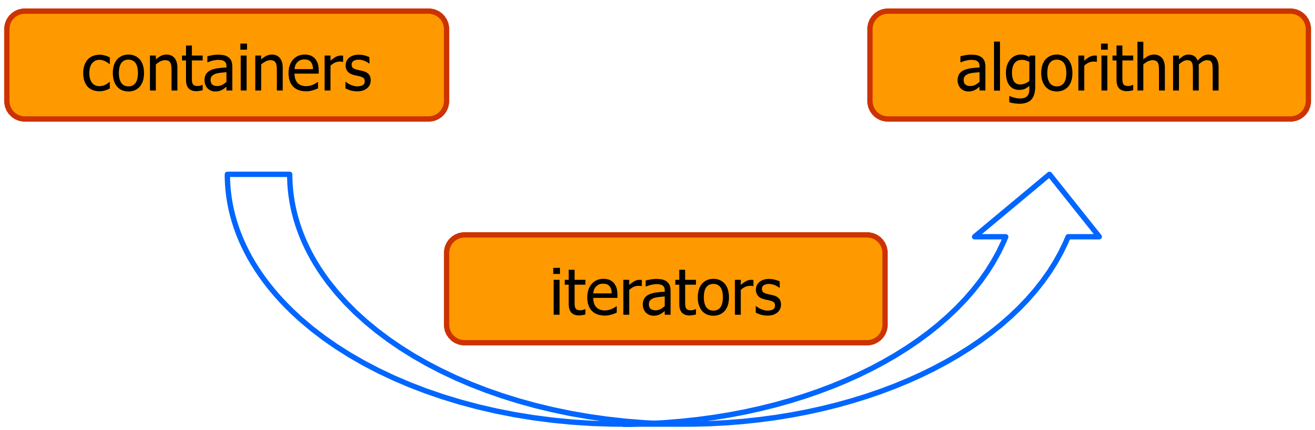
- Iterators: the iterator generalizes the concept of a pointer to a sequence of objects and can be implemented in many ways (in the case of an array it will be a pointer, while in the case of a list it will be a link etc...). Iterators allow you to iterate over a container, accessing each element individually.
  - 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.

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;



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).

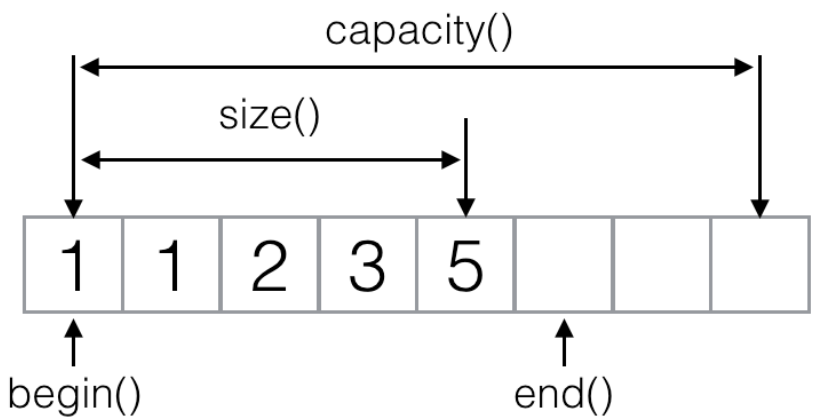
- Many containers have several member functions in common and share functionalities.

- 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`

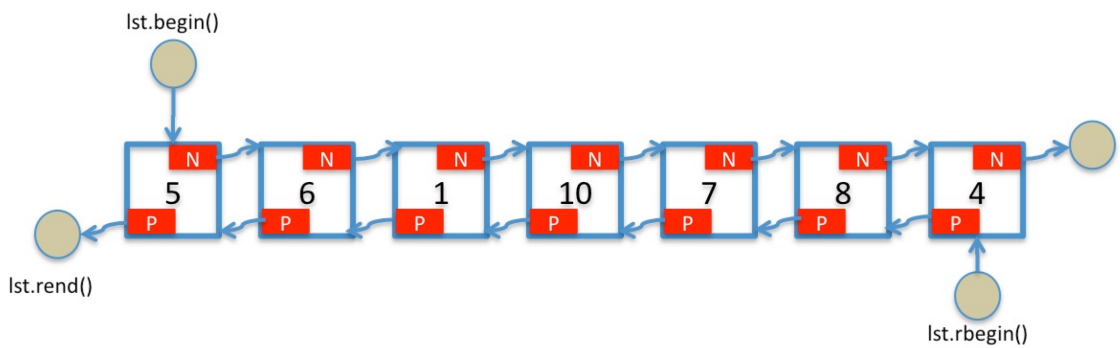
The `vector` class provides a data structure that occupies contiguous memory locations.



- 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`

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

int main() {

  std::vector<double> vnull ;
  cout << "vnull : size = " << vnull.size() << "   capacity = " << vnull.capacity() << endl;

  for ( int k = 0 ; k < 100 ; k++ ) {
    vnull.push_back(k*2.);
    cout << "vnull : size = " << vnull.size() << "   capacity = " << vnull.capacity() << endl;
  }

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

  cout << "Original vector : " << vnull.size() << " " << vnull.capacity() << endl;

  v.push_back(6);

  for ( unsigned int k = 0 ; k < v.size() ; k++ ) cout << v[k] << endl;

  cout << v[2] << endl;

}
```

Include the required header file

Create a vector : check size and capacity

Add element on the back

Initialize a vector (uniform initialization)

Add an element in the back

Access the third element of the vector

Adapt the functions to use the new data container

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

  vector<T> v;

  ifstream in(filename);

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

Adapt the functions to use the new data container

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

  vector<T> v;

  ifstream in(filename);

  if ( !in ) {
    cout << "Cannot open file " << filename << endl;
    exit(11);
  }

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

  return v;
};
```

`std::vector` is the ideal container to fill when reading elements from a file and you don't know in advance how many elements to read

How our analysis code would look like using `std::vector`

```
#include <iostream>
#include <fstream>
#include <cstdlib>

#include <vector>

#include "funzioni.h"

using namespace std;

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

  if ( argc < 3 ) {
    cout << "Uso del programma : " << argv[0] << " <n_data> <filename> " << endl;
    return -1 ;
  }

  vector<double> v = Read<double>( atoi(argv[1]) , argv[2] );

  cout << "media    = " << CalcolaMedia<double>( v ) << endl;
  cout << "varianza = " << CalcolaVarianza<double>( v ) << endl;
  cout << "mediana  = " << CalcolaMediana<double>( v ) << endl;

  vector<double> vall = = ReadAll<double>( argv[2] );

}
```

Include the vector header file

Include the funzioni header file

Read N elements

Read all elements

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

This asks to the operating system where the header files of the ROOT package are installed

This asks to the operating system where the ROOT libraries are installed

```
LIBS:=`root-config --libs`
INCS:=`root-config --cflags`

esercizio3.3 : esercizio3.3.cpp funzioni.h
	g++ -o esercizio3.3 esercizio3.3.cpp ${INCS} ${LIBS}

clean:
	rm esercizio3.3
```

When you compile the code, you need to tell the compiler where it can find the header files and the libraries (ie. compiled code)

Backup

5. Only for curious kids : the move semantic (rvalue references)

- An *lvalue* (*locator value*) represents an object that occupies some identifiable location in memory (i.e. has an address).
- *rvalues* are defined by exclusion: an *rvalue* is an expression that *does not* represent an object occupying some identifiable location in memory.

```
int main() {

  int a = 4 ;
  int b ;
  b = 4 ;
  4 = b ;

  double a  = media ( .... ) ;

  media( ...  ) = 4 ;

  Vettore v = ReadFromFile( ... ) ;

}
```

a and b are lvalues : they have an address, can stay on the left side of an assignment operator

The literal constant "4" is a rvalue, doesn't have an identifiable memory location

The value returned by `media()` is a rvalue, it's a temporary value that disappear once the calculation is done

1. A `Vettore` is created inside `ReadFromFile`
2. The `Vettore` is copied in a temporary `Vettore` through copy constructor when return is called
3. The `Vettore` is passed to the `Vettore v` using copy constructor

5. Only for curious kids : the move semantic (rvalue references)

```
int main() {

  int a = 4 ;
  int b ;
  b = 4 ;
  4 = b ;

  double a  = media ( .... ) ;

  media( ...  ) = 4 ;

  Vettore v = ReadFromFile( ... ) ;

}
```

3

```
Vettore Read ( unsigned int N , char* filename ) {
  Vettore v(N);
  ifstream in(filename);

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

1

2

1. A Vettore is created inside ReadFromFile
2. The Vettore is copied in a temporary Vettore through copy constructor when return is called
3. The Vettore is passed to the Vettore v using copy constructor

5. Only for curious kids : the move semantic (rvalue references)

- In C++ 11 the notion of reference to rvalues is introduced (&&) : can steal information from a temporary object !!

```
// move constructor

Vettore::Vettore( Vettore&& V) {
  cout << "Calling move constructor" << endl;
  m_N = V.m_N;
  m_v = V.m_v;
  V.m_N = 0;
  V.m_v = nullptr;
  cout << "Move constructor called" << endl;
}

// move assigment operator

Vettore& Vettore::operator=( Vettore&& V) {
  cout << "Calling move assignment operator " << endl;
  delete [] m_v ;

  m_N = V.m_N;
  m_v = V.m_v;

  V.m_N = 0;
  V.m_v = nullptr;
  cout << "Move assignment operator called" << endl;
  return *this;
}
```

The move constructor and the move assignment operators accept a && as input (reference to a rvalue)

They steal the content of the input object and reset the input

5. Only for curious kids : the move semantic (rvalue references)

```
int main() {

  int a = 4 ;
  int b ;
  b = 4 ;
  4 = b ;

  double a  = media ( .... ) ;

  media( ...  ) = 4 ;

  Vettore v = ReadFromFile( ... ) ;

}
```

In this case a move constructor is called, the code execution is much mode efficient, no unnecessary copies !

1. A Vettore is created inside ReadFromFile
2. The Vettore is copied in a temporary Vettore through move copy constructor (no copy of elements !)
3. The temporary Vettore is passed to the Vettore v using move copy constructor (again no copy of elements!)

Assert : `void assert ( int expression);`

https://www.cplusplus.com/reference/cassert/assert/

- If the argument *expression* of this macro with functional form compares equal to zero (i.e., the expression is *false*), a message is written to the standard error device and abort is called, terminating the program execution.

- The specifics of the message shown depend on the particular library implementation, but it shall at least include: the *expression* whose assertion failed, the name of the source file, and the line number where it happened. A usual expression format is:

  Assertion failed: *expression*, file *filename*, line *line number*

- This macro is disabled if, at the moment of including <assert.h>, a macro with the name NDEBUG has already been defined. This allows for a coder to include as many assert calls as needed in a code while debugging the program and then disable all of them for the production version by including a line like: #DEFINE NDEBUG at the beginning of the code, before the inclusion of <assert.h>.

- Therefore, this macro is designed to capture programming errors, not user or run-time errors, since it is generally disabled after a program exits its debugging phase.

Best practice

Use assertions to document cases that should be logically impossible.

Assert : `void assert ( int expression);`

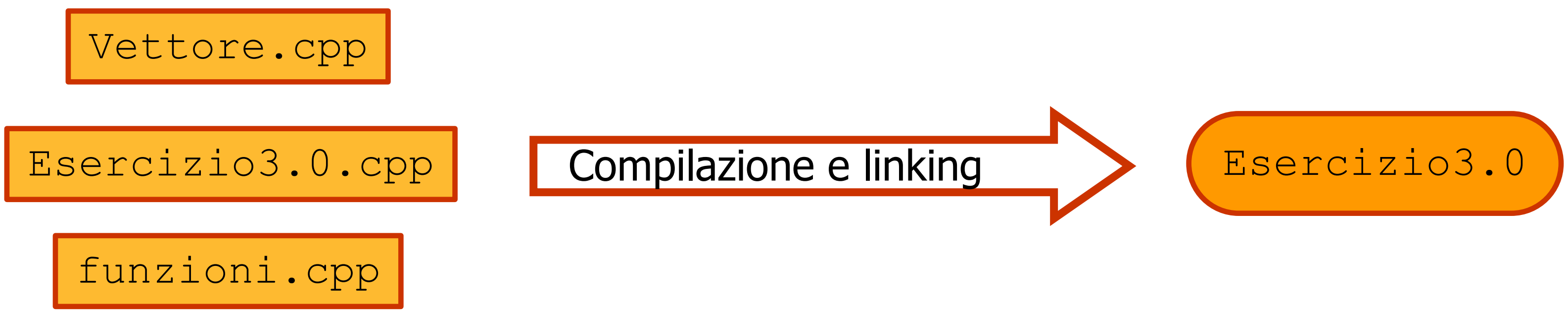
https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/

- Sometimes assert expressions aren't very descriptive Fortunately, there's a little trick you can use to make your assert statements more descriptive. Simply add a string literal with the message you want to see, joined by a logical AND :

```
assert( ( m_N > i ) && "Errore : l'indice e' troppo grande");
```

- Here how it works :
  - A string literal ("Errore … grande") always evaluates to `true`
  - If the condition `(m_N > i)` is `true` then `(true && true)` is `true` : no message is given by the assert and no abort is called
  - If the condition `(m_N > i)` is `false` then `(false && true)` is `false` : assert will give the message in the string literal and the abort will be called

Compilazione (I)




```
esercizio3.0 : esercizio3.0.cpp Vettore.h funzioni.h Vettore.cpp funzioni.cpp
        g++ esercizio3.0.cpp funzioni.cpp vettore.cpp -o esercizio3.0
```

- Vantaggi : una sola istruzione di compilazione
- Svantaggi : una modifica ad una qualsiasi delle classi o funzioni causa una ricompilazione totale

Putting everything together :

```
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ ./submit.sh
Usage: submit <number of elements to read in each file>
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ ./submit.sh 10
(cout)Trying to open file data1.dat
(cerr)End of file reached exiting
execution ended with code 2
Crashing with code 2 running on data1.dat
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ ./submit.sh 4
(cout)Trying to open file data1.dat
(cout)Data succesfully loaded
1 2 4 5
execution ended with code 0
(cout)Trying to open file data2.dat
(cout)Data succesfully loaded
1 2 4 5
execution ended with code 0
(cout)Trying to open file data3.dat
(cerr)Cannot open file data3.dat
execution ended with code 1
Crashing with code 1 running on data3.dat
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$
```

The prova executable crashes on the first input file because tries to read too many elements (return code is 2)

The prova executable crashes because data3.dat doesn't exist (return code is 1)

cout/cerr examples

```
#include <fstream>
#include <iostream>
#include <stdlib.h>

using namespace std;

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

  if ( argc < 3 ) {
    cout << "(cout)Uso del programma : " << argv[0] << " <n_data> <filename> " << endl;
    return 99 ;
  }

  int ndata = atoi(argv[1]);
  double* data = new double[ndata];
  char * filename = argv[2];

  // leggi dati da file e caricali nel c-array data

  cout << "(cout)Trying to open file " << filename << endl;

  ifstream fin(filename);

  if ( !fin ) {
    cerr << "(cerr)Cannot open file " << filename << endl;
    exit(1);
  } else {
    for ( int k = 0 ; k < ndata ; k++ ) {
      fin >> data[k] ;
      if ( fin.eof() ) {
        cerr << "(cerr)End of file reached exiting" << endl;
        exit(2) ;
      }
    }
  }

  cout << "(cout)Data succesfully loaded" << endl;

  for ( int k = 0 ; k < ndata ; k++ ) cout << data[k] << " " ;
  cout << endl;

  return 0 ;

}
```

Send messages through cout and cerr

Interrupt the program through exit or return

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

```
#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`

cout/cerr and exit/return examples : read elements from a file

Re-direct the output stream (cout) to a file

```
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ ./prova
(cout)Uso del programma : ./prova <n_data> <filename>
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ ./prova 10 data.dat
(cout)Trying to open file data.dat
(cerr)End of file reached exiting
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ ./prova 10 data.dat > log.log
(cerr)End of file reached exiting
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ ./prova 10 data.dat > log.log 2> err.log
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ more log.log
(cout)Trying to open file data.dat
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$ more err.log
(cerr)End of file reached exiting
Leonardos-MacBook-Pro-3:LezioneTeoria2 lcarmina$
```

Re-direct the output stream (cerr) to a file

In practice the possibility to redirect cout and cerr into different output files allows to create a log-file (cout messages from program execution) and an err-log (cerr messages from errors)