a.a. 2024-2025

Lezione 2 : Classes in C++

Prof. L. Carminati
Università degli Studi di Milano

Laboratorio Trattamento Numerico dei Dati Sperimentali

Passing information from outside

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

using namespace std;

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

  if ( argc < 4 ) {
    cout << "Uso del programma : " << argv[0] << " <ndati> <input_filename> <output_filename>" << endl;
    return -1;
  }

  cout << argc << endl;
  for ( int k = 0 ; k < argc ; k++ ) cout << argv[k] << endl;

  ifstream inputFile( argv[2] ) ;
  ofstream outputFile( argv[3]) ;
  unsigned int ndati = atoi( argv[1] );

  return 0 ;

}
```

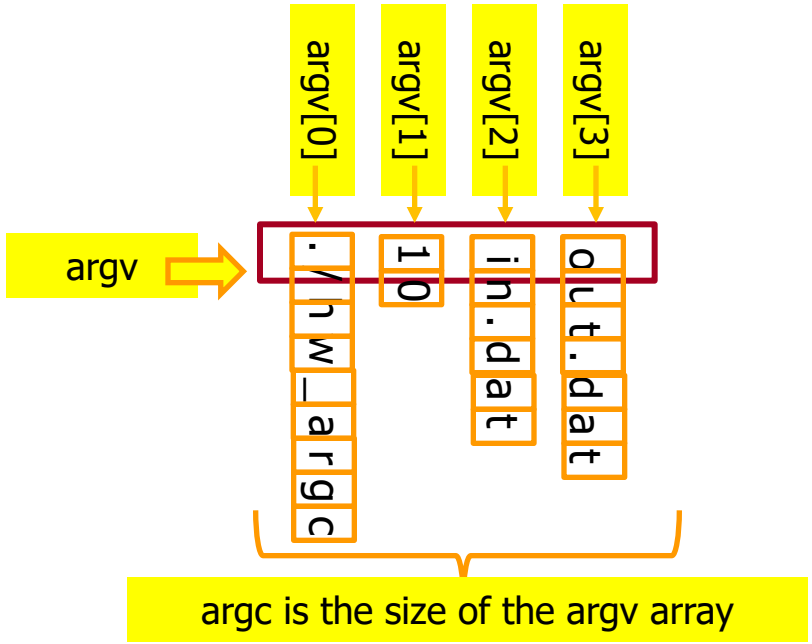
Accept inputs from the outside at execution time through the `argv` variable

Check that all required arguments are there

Number of elements to read (`argv[1]`) is passed as a `char*`, need to convert it into integer (use `atoi()` function from `cstdlib`)

Passing information from outside

- `argc` and `argv` are automatically filled by the executable and are available in the main




```
Leonardos-MBP-3:Lezione1 lcarmina$ g++ -o hw_argc hw_argc.cpp
Leonardos-MBP-3:Lezione1 lcarmina$ ./hw_argc
Uso del programma : ./hw_argc <ndati> <inputf> <outputf>
Leonardos-MBP-3:Lezione1 lcarmina$ ./hw_argc 10 in.dat out.dat
This is the value of argc 4
argv[0] = ./hw_argc
argv[1] = 10
argv[2] = in.dat
argv[3] = out.dat
Leonardos-MBP-3:Lezione1 lcarmina$
```

The problem of the data container size

1. If the number of elements to read is know at compilation time, will **never** change and it's "sufficiently small" (if too large might cause stack overflow)

```
int main() {

  double vdata[100];

}
```

"static" c-array (allocated in the stack memory)

2. If the number of elements to read is know at execution time (i.e. is passed by the user to the executable), can change from different executions
    1. Need to pass inside the main the number of elements to read
        1. using argc/argv
        2. cin (although might be annoying if you have to repeat several times, see later)
        3. Read from a configuration file (ifstream)
    2. Create a container with a suitable length
        1. Use a dynamic array (best option, allocate memory in the heap)

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

  int ndata = atoi(argv[1]) ;
  double * vdata = new double[ndata];

}
```

"dynamic" c-array (allocated in the heap memory)

The problem of the data container size

2. Use a VLA (Variable length array): not allowed in pure C++ (try to compile with the option `-pedantic-errors`), allowed in GNU compilers extensions. Not recommended for large arrays (might cause a stack overflow)

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

  int ndata = atoi(argv[1]) ;
  double vdata [ndata]

}
```

Variable Lenght Array (allocated in the stack memory)

If you don't know what you are doing always prefer dynamic arrays to static or VLA!

3. If the number of elements to read is unknown (at both compilation and execution time) and needs to be determined from the number of entries in the file). You might think to two different options:
   1. Open the file, count the element in the file, allocate the proper (dynamic) array, go back and fill the array (requires two loops on the input file)
   2. Is there a container that doesn't necessarily need to be declared with a size ? (yes, the `std::vector<>` (wait until next lesson)

Try to make the program a bit easier to read : split into functions

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

using namespace std;

double CalcolaMedia( double * , int ) ;
double * ReadDataFromFile ( const char*  , int ) ;
void Print ( const char* , double * , int ) ;

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

  if ( argc < 2 ) {
    cout << "Uso del programma : " << argv[0] << " <n_data> <filename> " << endl;
    return -1 ;
  }

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

  double * data = ReadDataFromFile ( filename, ndata ) ;
  cout << "Media = " << CalcolaMedia( data , ndata ) << endl;
  Print( "fileout.txt", data, ndata ) ;

}

double * ReadDataFromFile ( const char* Filename , int size  ) {
  double * data = new double[size];
  //   [ ... ]
  return data;
}

void Print ( const char* Filename, double * data, int size ) {
  //  [ ... ]
}

double CalcolaMedia( double * data , int size ) {
  //   [ ... ]
  return media ;
}
```

Functions declarations: input and return type only are important at this stage

The main simply calls the functions

Functions implementation : code how the function should work

Try to make the program a bit easier to read : split into functions

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

using namespace std;

double CalcolaMedia( double * , int ) ;
double * ReadDataFromFile ( const char*  , int ) ;
void Print ( const char* , double * , int ) ;

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

  if ( argc < 2 ) {
    cout << "Uso del programma : " << argv[0] << " <n_data> <filename> " << endl;
    return -1 ;
  }

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

  double * data = ReadDataFromFile ( filename, ndata ) ;
  cout << "Media = " << CalcolaMedia( data , ndata ) << endl;
  Print( "fileout.txt", data, ndata ) ;

}

double * ReadDataFromFile ( const char* Filename , int size  ) {
  double * data = new double[size];
  //   [ ... ]
  return data;
}

void Print ( const char* Filename, double * data, int size ) {
  //  [ ... ]
}

double CalcolaMedia( double * data , int size ) {
  //   [ ... ]
  return media ;
}
```

Move the functions declarations into a dedicated file (header file, funzioni.h) and replace this with `#include "funzioni.h"`

The main doesn't change

Move the functions implementation into a dedicated file (`funzioni.cpp`)

Try to make the program a bit easier to read : split into functions

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

#include "funzioni.h"

using namespace std;

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

  if ( argc < 2 ) {
    cout << "Uso del programma : " << argv[0] << " <n_data> <filename> " << endl;
    return -1 ;
  }

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

  double * data = ReadDataFromFile ( filename, ndata ) ;
  cout << "Media = " << CalcolaMedia( data , ndata ) << endl;
  Print( "fileout.txt", data, ndata ) ;

}
```

Move the functions declarations into a dedicated file (header file, funzioni.h) and replace this with `#include "funzioni.h"`

then compile in this way

```
g++ esercizio1.2.cpp funzioni.cpp -o esercizio1.2
```

Compilation can be broken into three main parts

1. pre-processing
2. actual compilation
3. linking

It is the phase in which the directives to the compiler are managed:

- #Include expansion: the compiler loads the declarations of the functions defined in the included libraries in order to check that these functions are used correctly in the program
- Replacing constants defined with #define: the compiler searches for all occurrences of these constants in the program and replaces them with the corresponding values
- Management of other directives that we will not see ...

Try `gcc -E esercizio1.2.cpp`

Compilation can be broken into three main parts

1. pre-processing
2. actual compilation
3. linking

It is the phase in which the program is transformed into binary code (not yet in an executable). It consists of several sub-phases (not necessarily sequential):

- Type control check that the variables used have been declared, check that operations in expressions are used with arguments of an appropriate type, check that the functions are called with number and type parameters appropriate…
- Analysis and optimizations : elimination of dead code, optimisation of the code…
- Generation of the binary code

Try `g++ -c esercizio1.2.cpp`

Compilation can be broken into three main parts

1. pre-processing
2. actual compilation
3. linking

Multiple object modules are linked together to create an executable file

- file1.o file2.o ...: different modules object of the same program obtained through separate compilation (we will see shortly)
- standard and system libraries
- libfile1.a libfile2.a ...: several external libraries that provide functions used within the program The result of this phase is a single executable file

Try `g++ esercizio1.2.cpp -o esercizio1.2`

Makefile

The makefile helps you in organizing the compilation instructions: just create a new file called Makefile

```
esercizio1.2 : esercizio1.2.cpp funzioni.cpp funzioni.h
	g++ esercizio1.2.cpp funzioni.cpp -o esercizio1.2

clean:
	rm esercizio1.2
```

- Structure of the makefile

<target> : <dep1> <dep2> <dep3> …
[tab] <compilation instruction>

```
esercizio1.2 : esercizio1.2.o funzioni.o
	g++ esercizio1.2.o funzioni.o -o esercizio1.2

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

esercizio1.2.o : esercizio1.2.cpp funzioni.h
	g++ -c esercizio1.2.cpp -o esercizio1.2.o

clean:
	rm esercizio1.2
	rm *.o
```

- The name of the target must match the name of the output file
- Dependencies : list of files that needs to be checked to trigger a new compilation. The timestamps of the dependencies are compared to the timestamp of the output file.
- The option –c forces g++ to compile but not link
- To launch a compilation just type "make esercizio1.2" (if you simply write make it will execute the first target)

Only the parts of the programs which have been modified are recompiled !

Makefile

Some useful parameters of the gcc compiler are as follows:

- -o: allows you to specify the name of the executable file to be generated (as an alternative to a.out.
  - `Usage: g++ -o esercizio1.0 esercizio1.0.cpp`
  - Equivalente : `g++ esercizio1.0.cpp -o esercizio1.0`
- -Wall: enables the display of all warning messages generated during compilation
- -pedantic: check that the program exactly meets the rules of the C standard (ISO C). Report any violations of the standard with warning messages
- -c : compilation but no linking

It is always good to use gcc at least as follows (might need to add –c if needed):

```
g++ -Wall esercizio1.0.cpp -o esercizio1.0
```

exit/return examples : read elements from a file

```
#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 ;

}
```

Interrupt the program through `exit` or `return`: different error codes

Small note on exit/return

We have used exit/return in approximately the same way in the `main()` program :

- **`return`:** a return statement always returns the control of flow to the function which is calling. Return uses exit code which is int value, to return to the calling function. Using the return statement in the main function means exiting the program with a status code; for example, return 0 means returning status code 0 to the operating system. Let us look at a C++ program using the return statement.
- **`exit()`** : an exit statement terminates the program at the point it is used. When the exit keyword is used in the main function, it will exit the program without calling the destructor for locally scoped objects. Any created object will not be destroyed and will not release memory; it will just terminate the program. Also notice the `#include <cstdlib>` is necessary to call the library function `exit()`.

1. Exit is often handy when the control flow is complicated, and error codes must be propagated all way up. But be aware that this is bad coding practice ! actual error management should be preferred (or in C++ using exceptions).
2. Direct calls to exit() are especially bad if done in libraries as it will doom the library user, and it should be a library user's choice to implement some kind of error recovery or not.

https://stackoverflow.com/questions/3463551/what-is-the-difference-between-exit-and-return

Small note on exit/return : exceptions (only for curious kids)

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

using namespace std;

double * ReadDataFromFile ( const char*  , int ) ;

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 ;

  try {
    data = ReadDataFromFile( argv[2] , ndata );
  } catch ( int errorcode ) {
    cout << "Cannot open input file " << endl;
    exit(errorcode);
  }

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

}
```

With the `try{} catch {}` structure the decision on how to handle the problem is managed by the main not by the function

```
double * ReadDataFromFile ( const char* filename , int ndata  ) {

  double * data = new double[ndata];

  ifstream fin(filename);

  if ( !fin ) {
    //    cout << "Cannot open file " << filename << endl;
    //    exit(33);
    throw 33 ;
  } else {
    for ( int k = 0 ; k < ndata ; k++ ) {
      fin >> data[k] ;
      if ( fin.eof() ) {
        cout << "End of file reached exiting" << endl;
        exit(33) ;
      }
    }
  }

  return data;

}
```

Putting everything together (only for curious kids)

- Imagine you must run your program several times, each time on a different input dataset (eventually located on a different machine around the world): you may want to prepare a shell script (this is not C++, it's bash shell scripting, might also better be python)

```
#!/bin/bash

# script che esegue in cascata N volte lo stesso programma

if test $# -eq 0; then
  echo "Usage: submit <number of elements to read in each file>"
  exit
fi

for i in `seq 1 5`;
do

    # questo e' il comando di ese                              ametri di input )
    ./prova $1 data$i.dat

    # qui ripesco il valore di ritorno del programma ( return /exit )
    error_code=$?
    echo submit.sh : ====== execution ended with code $error_code

    # se l'esecuzione non e' andata a buon fine termino
    if [ $error_code -ne 0 ] ; then
        echo submit.sh : Crashing with code $error_code running on data$i.dat
        break
    fi

done
```

This is the main loop in the shell scripting language

Execute the program

Get the program return/exit value

Decide to continue or not based on the execution output

Putting everything together (only for curious kids) :

```
Leonardos-MacBook-Pro-4:Lezione2 lcarmina$ ./submit.sh
Usage: submit <number of elements to read in each file>
Leonardos-MacBook-Pro-4:Lezione2 lcarmina$ ./submit.sh 5
prova.cpp : Trying to open file data1.dat
prova.cpp : Data succesfully loaded
1 2 4 5 6
submit.sh : ====== execution ended with code 0
prova.cpp : Trying to open file data2.dat
prova.cpp : Data succesfully loaded
1 2 4 5 6
submit.sh : ====== execution ended with code 0
prova.cpp : Trying to open file data3.dat
prova.cpp : Cannot open file data3.dat
submit.sh : ====== execution ended with code 1
submit.sh : Crashing with code 1 running on data3.dat
Leonardos-MacBook-Pro-4:Lezione2 lcarmina$
```

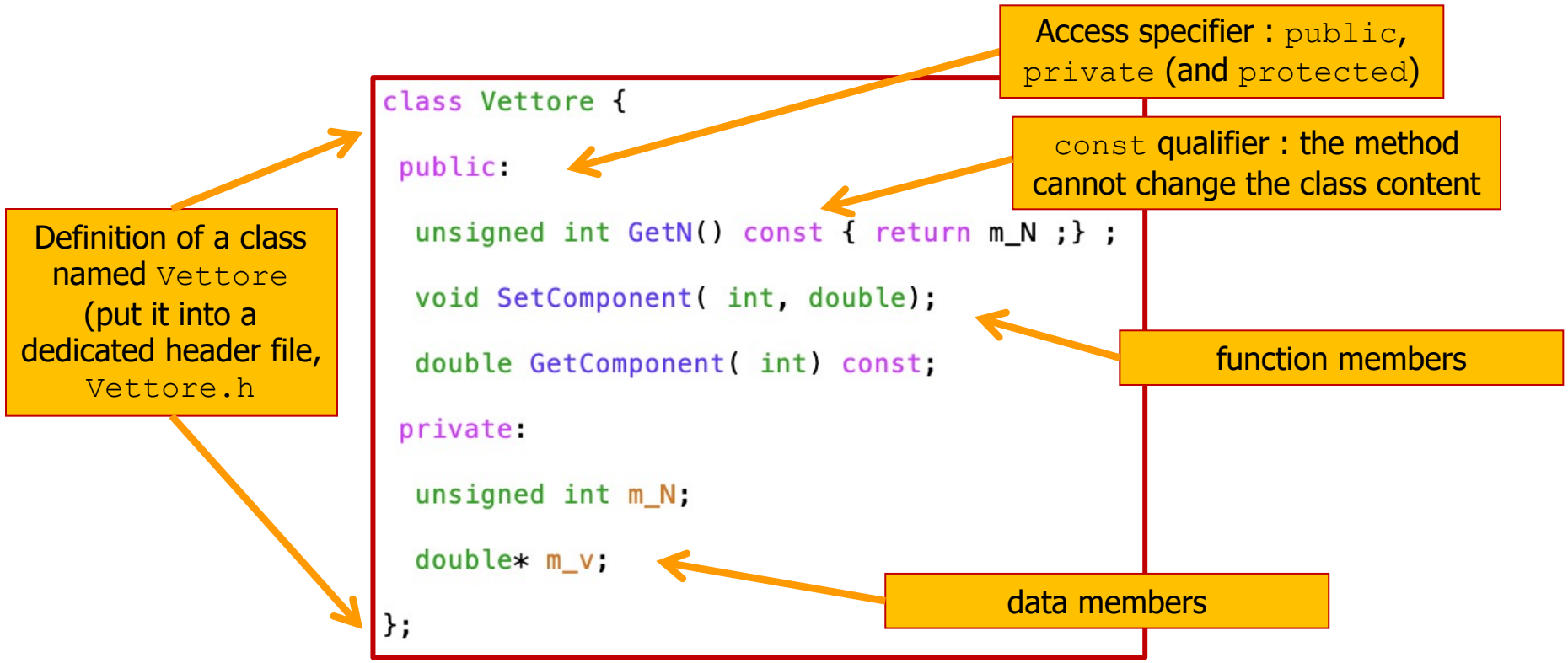
The prova executable crashes because data3.dat doesn't exist (return code is 1)

Classes in C++

Towards a smarter container: the `Vettore` class

The c-array is not a practical container: need to store its size in a separate variable, no control on the access to elements, can we cook up something better ? Let's build our own container!

- A *class* in C++ is a group of data elements and functions grouped together under one name. These data elements or functions, known as *members*, can have different types



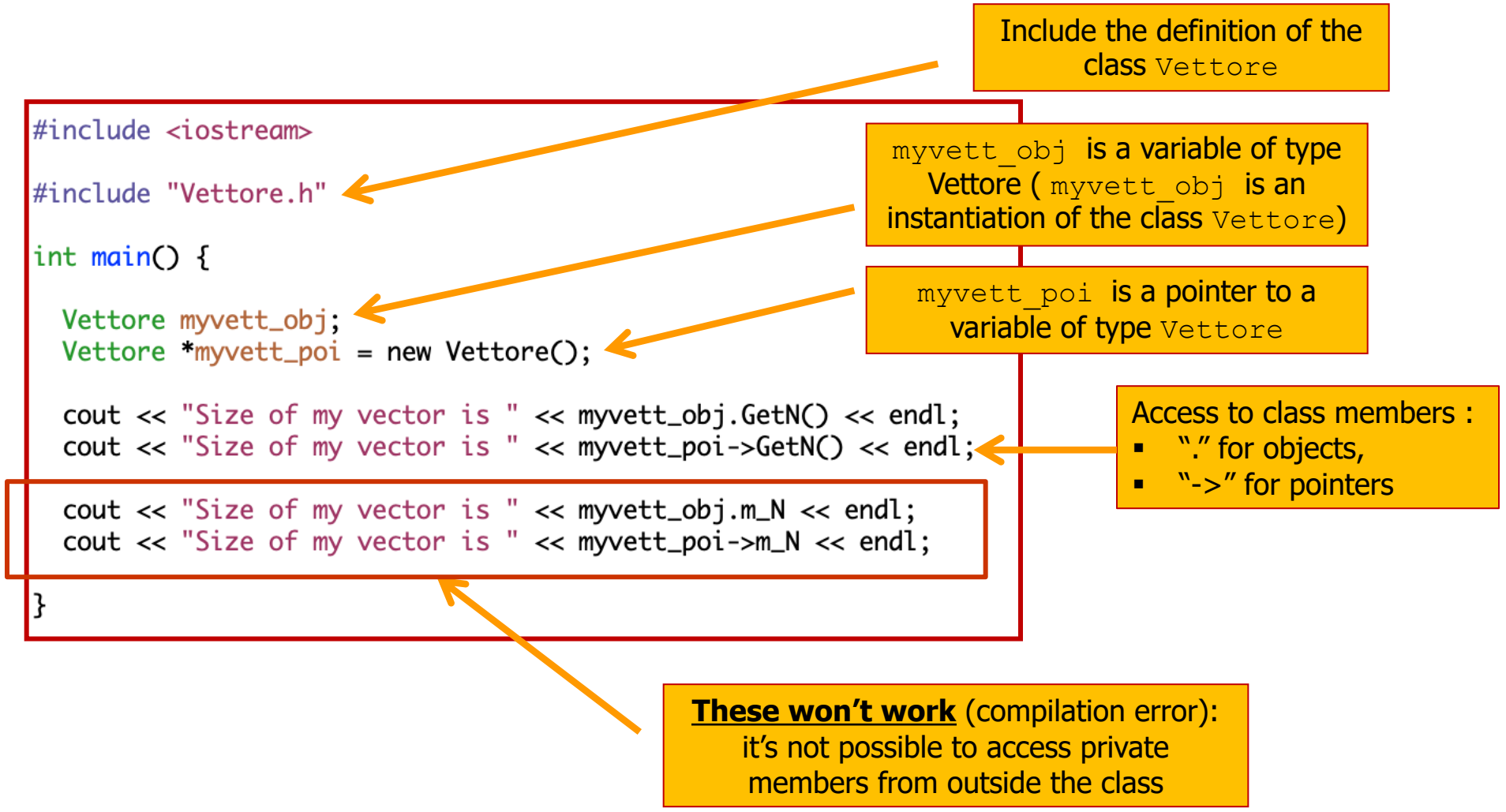
The vettore class : access specifiers

Access specifiers :

- private members of a class are accessible only from within other members of the same class (or from their *"friends"*).
- protected members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes.
- public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before any other *access specifier* has private access automatically.

The Vettore class : use our new container in the main program




```
#include <iostream>

#include "Vettore.h"

int main() {

  Vettore myvett_obj;
  Vettore *myvett_poi = new Vettore();

  cout << "Size of my vector is " << myvett_obj.GetN() << endl;
  cout << "Size of my vector is " << myvett_poi->GetN() << endl;

  cout << "Size of my vector is " << myvett_obj.m_N << endl;
  cout << "Size of my vector is " << myvett_poi->m_N << endl;

}
```

Adding features to the new class : data members initialization (constructor)

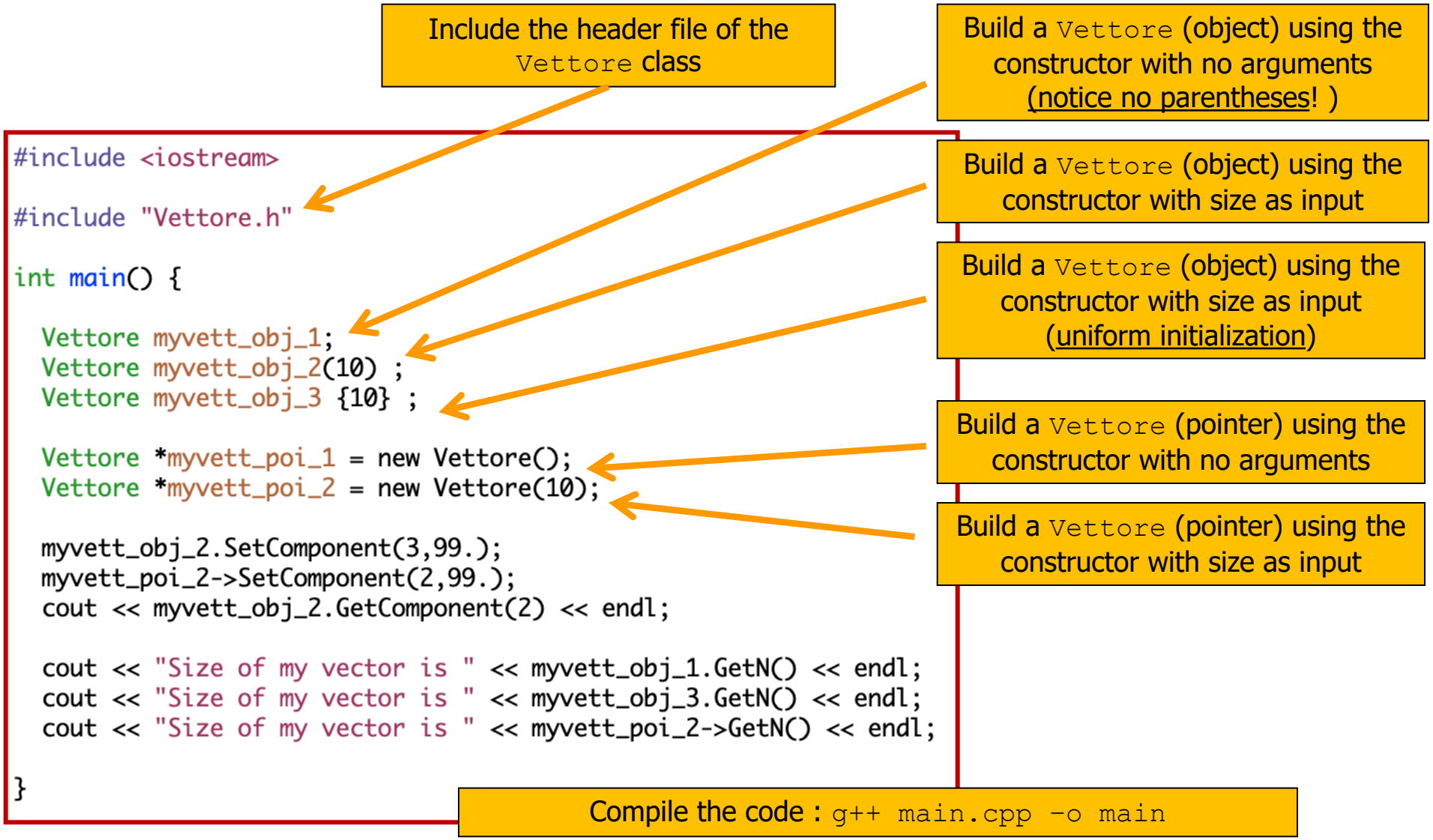
```
class Vettore {

 public:

  Vettore() {
    m_N = 0;
    m_v = NULL;
  };

  Vettore( int N ) {
    m_N = N;
    m_v = new double[N];
    for ( int k = 0 ; k < N ; k++ ) m_v[k] = 0;
  };

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

  unsigned int GetN() const { return m_N;} ;

  void SetComponent( int k , double val ) { m_v[k] = val; };

  double GetComponent( int k ) const { return m_v[k] ; };

 private:

  unsigned int m_N;
  double* m_v;

};
```

- The constructor is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.
- This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.
- Constructors can be overloaded.
- If not implemented a constructor with no arguments (default constructor) is provided implicitly (this is also true for destructors)

- The destructor fulfills the opposite functionality of the *constructor*. It is responsible for the necessary cleanup needed by a class when its lifetime ends.

In-line implementation of the methods

Create objects or pointers



Organizing the code : option 1 (one header file only)

Main.cpp

```
#include "Vettore.h"

int main() {

  Vettore v;
  return 0;

}
```

Vettore.h

```
class Vettore {

 public:

  Vettore() {
    m_N = 0;
    m_v = NULL;
  };

  Vettore( int N ) {
    m_N = N;
    m_v = new double[N];
    for ( int k = 0 ; k < N ; k++ ) m_v[k] = 0;
  };

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

  unsigned int GetN() const { return m_N;} ;

  void SetComponent( int k , double val ) { m_v[k] = val; };

  double GetComponent( int k ) const { return m_v[k] ; };

 private:

  unsigned int m_N;
  double* m_v;

};
```

Makefile

```
main : main.cpp Vettore.h
        g++ main.cpp -o main
```

Organizing the code : option 2 (split declaration and implementation)

main.cpp

```
#include "Vettore.h"

int main() {

  Vettore v;
  return 0;

}
```

Vettore.h

```
class Vettore {

 public:

  Vettore();
  Vettore( int N);

  ~Vettore();

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

 private:

  unsigned int m_N;
  double* m_v;

};
```

Vettore.cpp

```
#include "Vettore.h"

Vettor::Vettore() {
  m_N = 0;
  m_v = nullptr;
};

Vettore::Vettore( int N ) {
  m_N = N;
  m_v = new double[N];
  for ( int k = 0 ; k < N ; k++ ) m_v[k] = 0;
};

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

void Vettore::SetComponent( int k ,double val) {
  m_v[k] = val;
};

double Vettore::GetComponent( int k) const {
  return m_v[k] ;
};
```

Each member declared using the scope operator

Makefile (1)

```
main : main.cxx Vettore.cpp Vettore.h
        g++ main.cpp Vettore.cpp -o main
```

Makefile (2)

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

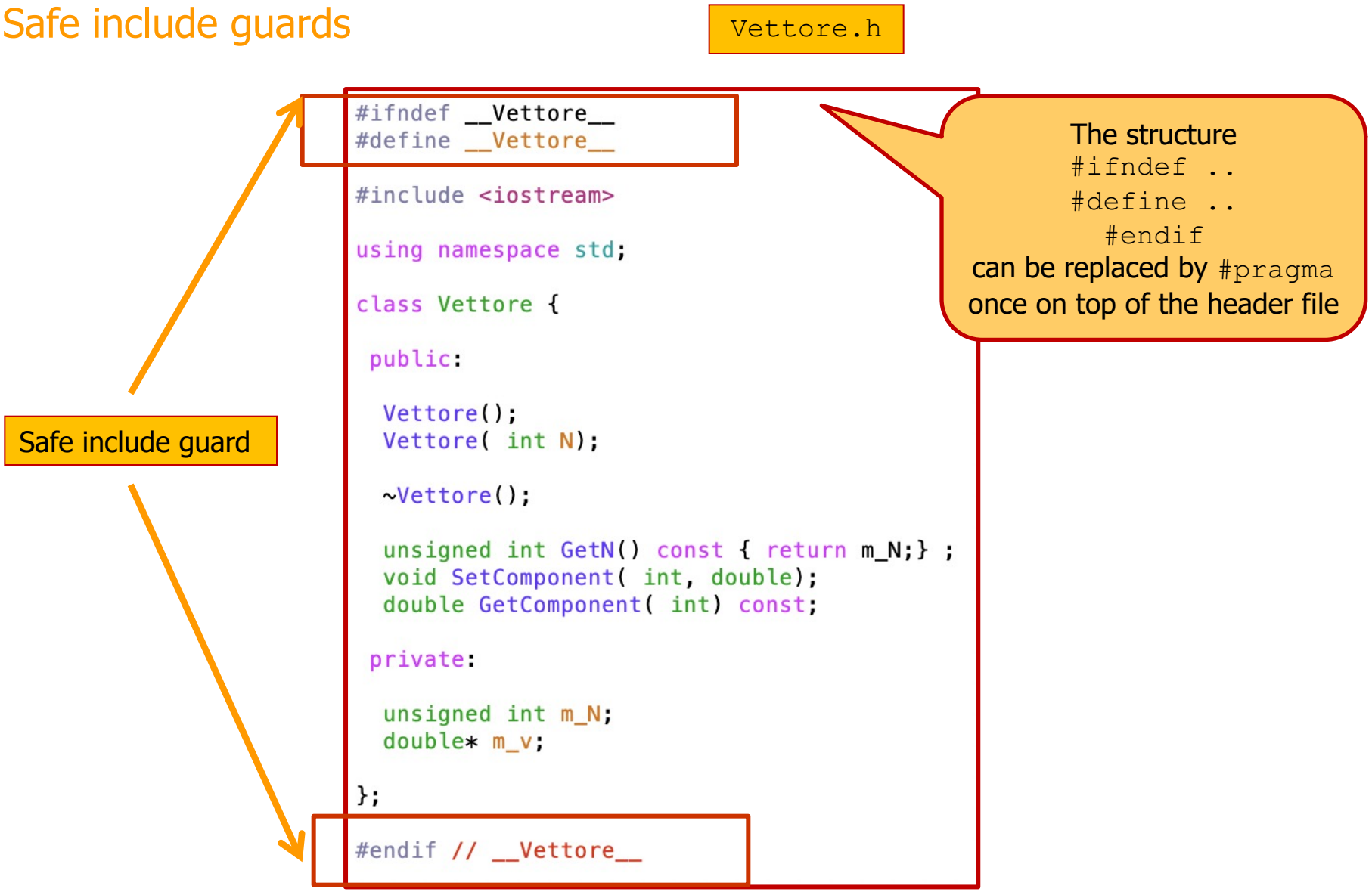
Organising the code

At least two options to organize our class code:

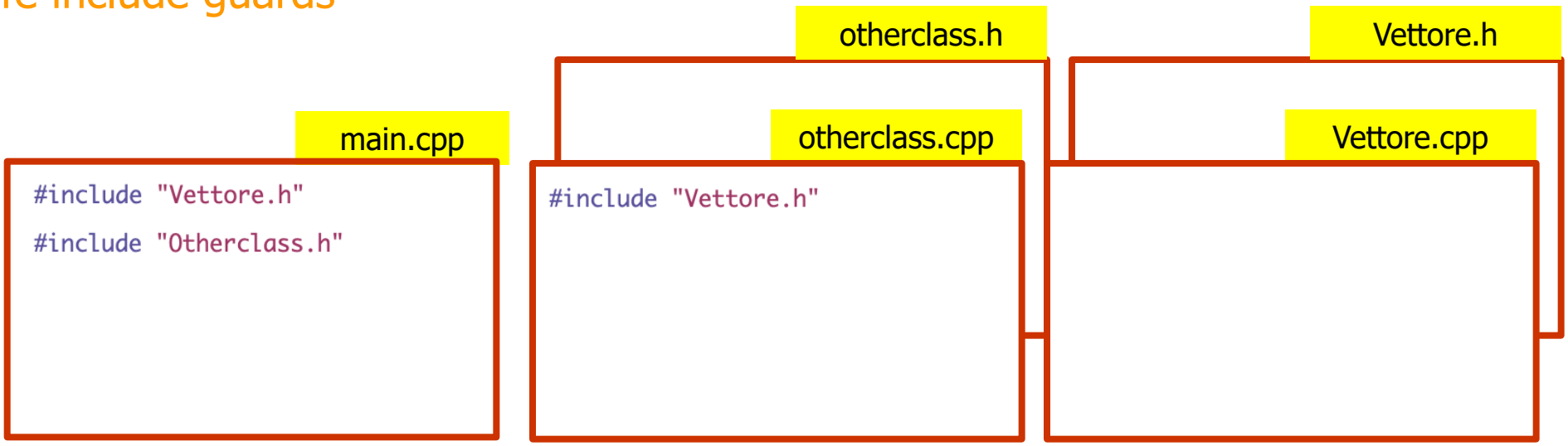
- Just put everything into Vettore.h (both declaration and implementation): easy, fast
- Separate into Vettore.h (declaration) and Vettore.cpp (implementation) : a bit more complex to handle but better from the compilation point of view (allows to compile only parts of code which are really necessary). At least two options to compile our code :
  - Makefile (1) : compile all code together. One shot. Recompile everything each time
  - Makefile (2) : compile only what has been modified

Safe include guards

Vettore.h



Safe include guards




```
main : main.cpp Vettore.cpp Vettore.h Otherclass.h Otherclass.cpp
        g++ main.cpp Vettore.cpp Otherclass.cpp -o main
```

- If no safe inclusions guards (`#ifndef ... #endif`) are declared this will cause a compilation error due to multiple declaration of the `class Vettore`
- Can be replaced with `#pragma once` directive

Copy constructor and assignment operator

Vettore.h

```
#ifndef __Vettore__
#define __Vettore__

#include <iostream>

using namespace std;

class Vettore {

 public:

  Vettore();
  Vettore( int N );

  ~Vettore();

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

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

 private:

  unsigned int m_N;
  double* m_v;

};

#endif // end of __Vettore__
```

Copy constructor : create a new `Vettore` from an existing one.

- If not explicitly defined a default one is available, but its behavior might not be the correct one
- can be overloaded
- Normally invoked when in the main you do

```
Vettore b(10)
Vettore a = b
```

Assignment operator: an existing `Vettore` is initialized as a copy of an existing one

- If not explicitly defined a default one is available, but its behavior might not be the correct one
- can be overloaded
- Normally invoked when in the main you do

```
Vettore b(10)
Vettore a;
a = b
```

Copy constructor and assignment operator

Vettore.cpp

```
// 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 << "Assigned operator called" << endl;
  return *this;
}
```

- The default constructor (provided by C++ if you don't declare it) would copy the content of Vettore a into Vettore b
- If you rely on the default copy constructor (and assignment) you would have a new Vettore with the same size and the same pointer !
- This would mean two Vettore actually sharing the same array (one modification on Vettore a will have the same effect on Vettore b)
- Need to specify that the new Vettore has a pointer to a different memory area which will contain the same values

"this" identifies a special pointer that contains the address of the instance of the class that invoked the method.

Improving the Vettore class

Vettore.cpp

```
void Vettore::SetComponent( int i, double a) {

  if ( i>= 0 && i<m_N ) {
    m_v[i]=a;
  } else {
    cout << "Errore: indice non valido " << endl;
    exit (1);
  }

}

double Vettore::GetComponent( int i) const {

  if ( i >= 0 && i<m_N ) {
    return m_v[i];
  } else {
    cout << "Errore: indice non valido " << endl;
    exit(2);
  }

}

double& Vettore::operator[] ( int i) {

  if ( i>=0 && i < m_N ) {
    return m_v[i];
  } else {
    cout << "Errore: indice non valido" << endl;
    exit(3);
  }

}
```

Notice the bad practice of putting exit in a method

- Encapsulation and data hiding paradigm : modify important data only through dedicated members
- You might want to make your code safer and more robust : for example, check that the component you are tying to access or to write is not outside the allocated memory

Working with a Vettore

We can now create functions to work with an object of class Vettore : read a Vettore from file, print the content of a Vettore, compute the mean/median/variance of the elements in a Vettore etc.

funzioni.h

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

using namespace std;

Vettore Read( int, const char* ) ;

double CalcolaMedia( const Vettore & ) ;
double CalcolaVarianza( const Vettore & ) ;
double CalcolaMediana( Vettore ) ;
//double CalcolaMediana (const Vettore &);

void Print( const Vettore & ) ;
void Print( const Vettore & , const char* ) ;

void selection_sort( Vettore & );
```

Passing the Vettore by reference (faster) protected by a const

funzioni.cpp

```
#include "funzioni.h"

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() ) ;

}

Vettore Read ( int N, const char* filename) {

  Vettore v(N);

  ifstream in(filename);

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

No need to pass the size!

Working with a Vettore

We can now create functions to work with an object of class Vettore : read a Vettore from file, print the content of a Vettore, compute the mean/median/variance of the elements in a Vettore etc.

funzioni.h

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

using namespace std;

Vettore Read( int, const char* ) ;

double CalcolaMedia( const Vettore & ) ;
double CalcolaVarianza( const Vettore & ) ;
double CalcolaMediana( Vettore ) ;
//double CalcolaMediana (const Vettore &);

void Print( const Vettore & ) ;
void Print( const Vettore & , const char* ) ;

void selection_sort( Vettore & );
```

`Vettore` passed by reference (no copy !) , protected by the `const` qualification: the function can't modify the input `Vettore`

`Vettore` passed by value: a temporary internal copy is created which can be manipulated (re-ordered), no modifications to the original `Vettore`

`Vettore` is passed by reference (no copy) and no `const` is used : the function can manipulate (re-order in this case) the `Vettore` in the main

Ready to enjoy the `Vettore` class in its full power !

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: a *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, don'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

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

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 an rvalue)

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

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

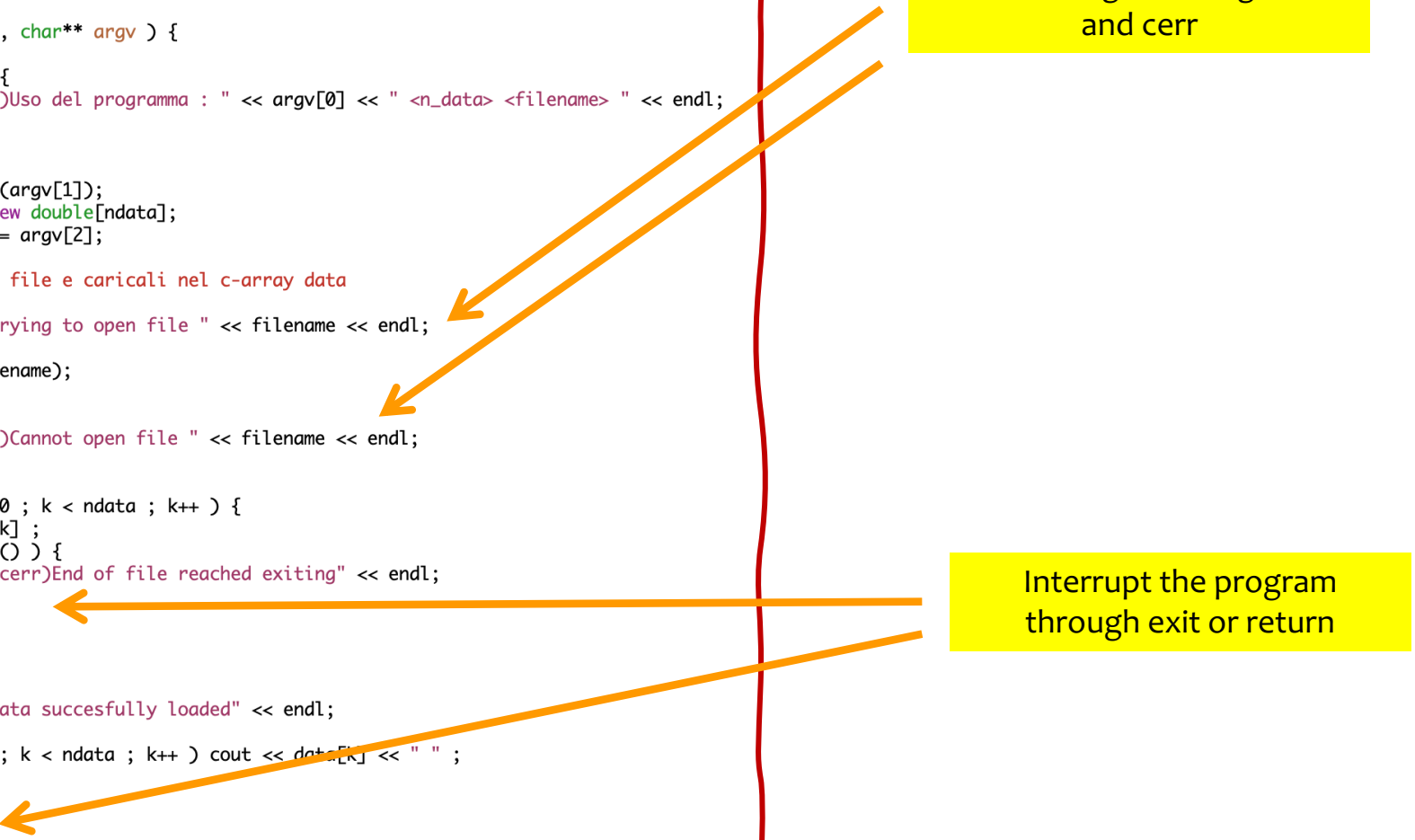
```
int main() {

  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 :

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

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

```
#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 ;

}
```



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

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)

Be careful !

```
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];

  // leggi dati da file e caricali nel c-a

  ifstream fin("filename");

  if ( !fin ) {
    cout << "Cannot open file " << filename << endl;
    exit(33);
  } else {
    for ( int k = 0 ; k < ndata ; k++ ) {
      fin >> data[k] ;
      if ( fin.eof() ) {
        cout << "End of file reached exiting" << endl;
        exit(33) ;
      }
    }
  }
}
```

- "`filename`" means that you are trying to open a file called exactly `filename`
- `filename` means that you are trying to open a file whose name is stored in the variable filename