Advanced Programming for Scientific Computing (PACS)
Lecture title: Some elements of C++

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2023/2024

A note on the C++ language

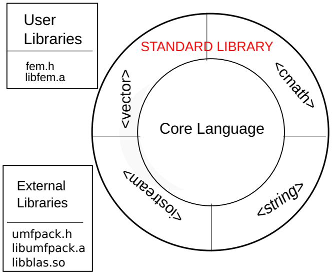
C++ has evolved enormously recently. The greatest changes happened with the C++11 standard (B. Stroustroup: "C++11 is like a new language"); C++20 introduced some other important additions. C++ after C++11 is nowadays termed modern c++.

All major compilers now implement C++20 standard. In this course, I will refer to C++20 features.

An important characteristics of C++ is backward compatibility. A code written using C++98 standard almost surely compiles if you use a C++20 compiler. You may indicate the standard you want with the option `-std=` of the compiler.

In the examples I have used some features of `-std=c++20`, even if most code follows C++11.

The structure of the C++ language



C++ is a highly modular language. The core language is very slim, being composed by the fundamental commands like `if`, `while`,.... The availability of other functionalities is provided by the *Standard Library* and require to `include`[1] the appropriate `header` files.

For instance, if we want to perform i/o we need to include `iostream` using **#include** <iostream>.

[1] C++20 has introduced modules, but are still not fully implemented

Headers inherited from the C language

Some C++ headers of the standard library represent the porting of headers of the C standard library.

You can recognise them since their name starts with the letter c. The pattern is

$$\boxed{\text{C library header; } \texttt{header.h} \rightarrow \text{C++ library name: } \texttt{header}}$$

For example, `math.h` is now `cmath`. You can still include the old C file, but it is highly discouraged. Always use the C++ standard header files.

The std namespace

Names of standard library objects are in the std namespace. Therefore, to use them you need either to use the full qualified name, for example std::vector<**double**> or bring the names to the current scope with the **using** directive:

```
#include <cmath> // introduces std::sqrt
...
double a=std::sqrt(5.0); // full qualified name
...
using std::sqrt; //sqrt in the current scope
...
double c=sqrt(2*a);// OK
```

With **using namespace** std you bring all names in the std namespace into the current scope. Beware: use it with care!

The `main()` Program

The main program defines the only entry point of an executable program.
Therefore, in the source files defining your code there must be one and only one `main`. In C++ the main program may be defined in two ways

```
int main (){ // Code
}
```

and

```
int main (int argc, char* argv[]){ // Code
}
```

The variables `argc` and `argv` allow to communicate to `main()` parameters passed at the moment of launching.

Their parsing is however a little cumbersome. The use of GetPot utility makes parsing easier.

What does main() return?

In C++ the main program returns an integer. This integer may be set using the return statement. If the return statement is missing, by default the program returns 0 if terminates correctly.

The integer returned by `main()` is called *return status* and may be interrogated by the operative system.

Therefore, you may decide to take a particular action depending on the return status. Remember that by convention status 0 means "executed correctly". If the main does not have a `return n` statement, and you do not call `std::exit(n)`, the returned value is 0.

Just to let you know: if you terminate a program with a call to `std::abort()` the program aborts and return status is undefined, but the system signal `SIGABRT` is sent instead. `abort()` is a brutal way of terminating a program. Don't do it.

Testing the return state of a program

In Unix, the variable `$?` contains the state of the last command

Here a small bash script that tests the state

```
#!/bin/bash
# Replace 'your_command' with the actual command you want to test
your_command
# Capture the exit status of the command
exit_status=$?
# Check if the exit status is not equal to zero
if [ $exit_status -ne 0 ]; then
    echo "The command exited with a non-zero status: $exit_status"
fi
```

The `#!/bin/bash` in the first line is the hashbang that tells the operative system that what follows is a bash (Bourne Again Shell) script.

Note: Do you know that with the hashbang `#!/usr/bin/python` you can create a python script that can be launched as any unix command?

Two simple examples to start with

A simple C++ program. In `Examples/src/SimpleProgram/main_ch1.cpp` A program that computes $\sum_{i=n}^{m} i$, reading m and n from file.

HeatExchange. A simple 1D finite element program, where we also give an example of the use of GetPot, the json file reader, and of gnuplot-iostream, to create the plot of the solution from within the code. (maybe it's not working if you are using a virtual machine).

Terminology

It is important to have a clear idea of some terms. On webeep I have put a file, `CppNomenclature.pdf`, tat interoduces the most important terms (in my opinion) used in programming and C++, with some details.
Here I will recall only the main ones.

Some nomenclature

- Name An alphanumeric string associated to variables or sets of overloaded functions. Names in C++ cannot begin with a numeric character and are case sensitive. A qualified name contains the name of its enclosing class or namespace, using the scope resolution operator `::`. A name cannot be equal to a `keyword` of the language (e.g. I cannot have a variable called **while**).
- Identifier An identifier is an *alphanumeric string* that identifies a variable or a function uniquely. The identifier for a function is its signature, which includes the type of the function parameters and, for function members, the possible `const` qualifier.
- Symbol The translation of an identifier in the compiled code.

Declaration and definition

- A (pure) declaration is a statement that informs the compiler about the existence and type of a variable or object, but does not allocate memory for it. A declaration can be done at any place in the code, as long as it's done before the variable or object is used. You can have multiple identical (pure) declarations.
- A definition is a declaration that also allocates memory for the variable or object. You can have only one definition of an object (one definition rule) (unless declared **inline** or **constexpr**).

Nomenclature: Scope

Every name in a C++ program is accessible only in some part of the source code called it's scope. A declaration introduces a name in a scope. We can distinguish

- Block scope A set of statements included between { and }.
- Local scope A name declared in a function is local. It extends from its point of declaration to the end of the block in which its declaration occurs.
- Namespace scope It is a scope with a name (even if we may have anonymous namespaces!). It extends from the point of declaration to the end of the namespace.
- Class scope. A name is called *member name* when defined in a class. Its scope extends from the opening { of its enclosing declaration to the matching }.

Facts about scope

- Scopes can be nested and an internal scope “inherits” the names of the external scope.
- A name declared in a scope *hides* the same name declared in the enclosing scope. If the enclosing scope is a namespace scope, name is still accessible using the scope resolution operator `::`.
- The most external *scope* is called global.
- The scope resolution operator `::` allows to access variables in the global scope.
- `for` and `while` define a scope that includes the portion with the test.
- A variable is destroyed when the program execution exits the scope where it has been defined.

In Scope/main_scope.cpp a small example about scope rules.

Nomenclature

- Type In C++, a type defines a set of possible values and a set of operations that can be performed on those values: **double** and **int** are examples of C++ types.
  In C++ you can extend the available types: when you define a class or an enumeration you are in fact increasing the set of types available in your code.
  C++ is a strongly and statically typed language, it means that any object has a well-defined type and the type is resolved at compile time. Static typing is one of the reasons of the higher efficiency of compiled languages like C++ compared with interpreters.

Nomenclature

- POD Plain Old Data: int, double, bool,... all basic types common with the C language. More precisely, compiling a POD in C++ produces the same memory layout as in C.
- Variable Qualifiers keyword that change the behaviour of a variable: **const**, **volatile**. A variable is of a cv-qualified type if it is either **const** or **volatile**.
- Adornments References are sometimes indicated as "adornments' of a type since they do not define a new type, but provide a different name to an existing object.
- Integral type. Types that can behave like an **int**: **int**, **unsigned int**, **long int**,**short int**, **bool**, **char**, **byte**, enumerators, and all pointers. They are the only values that can be set as template parameter.

Nomenclature

- Expressions Anything that produces a value, i.e. 5+a where a is a int, is an *integer expression*. The type of an expression is the type of the value provided by the expression. An expression may be composed by subexpressions, like in (a+b)(d+e).
- Constant expression. A constant expression is an expression that can be computed *at compile time*. Constant expressions may be defined with the `constexpr` specifier:

```
constexpr double pi=3.1415; //pi is a constant expression
```

- Literal. Literals are another case of constant expressions, they are written "literally". They are used to express given values within a program: in **double** a=2.3;, the expression `2.3` is a literal (of type **double**). In std::**string** a{‘‘Hello World’’}; the expression `Hello World` is a literal (of type **char**∗).

Literal types

The type of a numeric literal may be specified by a suffix. By default a numeric literal containing a dot (.) is a `double`, otherwise it is an `int`. Other literal suffixes:

```
3 // int
3u // unsigned int
-123456789l // long int
12345678912345ul // unsigned long int
3.4f; // float
3.4 // double
3.4L // long double
```

Important Note: 3/4 is equal to 0: you are making an integer division! And if you do **auto** a =10;, variable a is an **int**.

Complex literals

If you include <complex> and set

```
using namespace std::complex_literals;
```

you have the complex literals

`i`, `if` and `il`

for the imaginary unit, expressed as double, float and long double respectively:

```
auto c = 4.0 + 3.0i // c is a complex<double>
auto d = 5.0L + 4.5il // d is a complex<long double>
```

String literals

String literal should deserve a lenghty discussion since strings are special in several aspects. One of which is that they are interpreted. For instance, the sequence "\\n" when printed means "line feed". If you want to not interpret special characters like \n in a string, you must use special raw string literals.

```
std::string S=R"foo(Hello\n World\n)foo";
std::cout<<S;
```

will be printed verbatim as `Hello\n World\n`.

In C++ you have std::**string** (very nice!), but also the old fashioned C-style null-terminated string (basically **char**∗).

C-strings and C++ std::string

A C-style null-terminated string is implicitely convertible to a std::**string**, that's why

```
std::string s="this is a string";
```

is perfectly fine, even if "this is a string" is in fact a **char** *!.
If you want an alphanumeric literal to by interpreted as a std::**string** you may use the "s" suffix:

```
using namespace std::string_literals; // you need this!
auto s="this is a string"; // s is a char*
auto S="this is a string"s; // S is a std::string
```

We will not enter into too many details. You may have a look to some string literals in the folder StringLiterals.

Fixed width integer types

With the proliferation of different architecture sometimes it may be necessary to write safe code to specify exactly what we mean with **int**: is it a 16 bit or a 32 bit integer? The header <cstdint> defines integer types guaranteed to have a specific width.

```
#include <cstdint>
int_8_t a; // 8 bits integer
int_32_t b; // 32 bit integer
uint_64_t c;// 64 bits unsigned integers..
```

And there are many others: take a look here.

If lives, or lots of money depend on your code... it is better to know which type of integer you are dealing with! Remember that integer overflow is difficult to detect.
Look at the small code in IntegerOverflow.

Initialization and assignment

```
vector<double> v={2,3,4};// v is initialized to a vector of 3 eleme
double a{3.5};// a is initialized
double b=0.5;// b is initialized
b=a; // the value of a is assigned to b
vector<double>z{v};// z is initialised with the value of v
double k; // k is not initialized
```

Beware that **double** a=b is an initialization: a is created by copying the value contained in b (copy-construction). While b=a is an assignment: the value of the existing variable a is changed by copying the value in b (copy-assignment).

Initializations of the form **double** a{3.5}; or (old style) **double** a(3.5); are called direct initializations. They are almost equivalent to **double** a=3.5; (copy initialization), the difference concerns conversion rules (we will discuss this issue later).

An important note

Don't assume that a variable is initialized automatically. Initializing variables explicitly is safer!.

Always initialize pointers to a value. If the object is not available at the moment, initialize pointers to the null pointer. Never have dangling pointers in a program.

```
double * p=0; // OK. A null pointer. But prefere nullptr
double * x=nullptr;// a null pointer (much better!)
int * ip=new int[20]; //Ok pointer to a C-array of int
double * pp;// NO!NO!NO!
```

A reference must always be initialized (bound ia a more precise term) to an existing object!

Brace ({}) initialization

We will give more details in the uniform (brace) initialization. Here I just recall that in general we can initialize variable with braces or parenthesis

```
double x0; // ok but value undetermined!
double x1(20.); //Ok x value is 20.
double x2{20.}; //Ok x value is 20.
float x3(x1); // ok double converted to float
float x4{x1};// ERROR: narrowing not allowed
float x4{static_cast<float>(x1)};// Ok
double x5{};// Ok initialized by 0
double x6();// NO! it's a function decl.
double x6={};// Ok, initialized by zero
auto x = double{6.0};// it works a is a double in. with 6
double x7=();// NO! It does not make sense
```

In modern c++ brace initialization is preferred.

Note: **double** a{}; initializes a by zero, while **double** a; leaves it uninitialized.

Formatted i/o

C++ provides a sophisticated mechanism for i/o through the use of streams. We will see more details later on. For the time being, we recall that streams may be accessed by the iostream header file.

```
#include <iostream>
..
 std::cout <<''Give me two numbers'' << std::endl;
 std::cin >> n >> m;
```

The standard library provides 4 specialized streams for i/o to/from the terminal.

std::cin	Standard Input (buffered)
std::cout	Standard Output (buffered)
std::cerr	Standard Error (unbuffered)
std::clog	Standard Logging (unbuffered)

Redirecting i/o

`cin`, `cerr` and `clog` are by default addressed to the keyboard and the terminal. But, you can redirect them at operative system level:

```
./myprog <input.dat # read cin from a file
./myprog 2>err.txt #redirecting stderr to a file
./myprog &>allout.txt #redirect both stderr and stdout
./myprog 1>out.txt 2>err.txt #to different files
```

or internally in the program (we will see it a a dedicated lecture).

Unbuffered means that the stream is immediately sent to the output, with no internal buffer. The internal buffer is used to make i/o more efficient, but it may be deleterious for error messaging, since if the program fails the message may not be printed out.

Implicit and explicit conversion

```
int a=5; float b=3.14;double c,z;
c=a+b(implicit conversion)
c=double(a)+double(b) (conv. by construction)
z=static_cast<double>(a) (conv. by casting)
```

C++ has a set of (reasonable) rules for the implicit conversion of POD. The conversion may also be indicated explicitly, as in the previous example.

Note: It is safer to use explicit conversion, but implicit conversions are very handy! Yet, if you want to make your intentions clear use explicit conversions.

C-style cast, e.g. `z= (double) a;`, is allowed but discouraged in C++. Don't use it. **static_cast** is safer.

Casts

Casting is an expression used when a value of a certain type is explicitly "converted" to a value of a different type. In C++ we have three types of cast operator (well, 4 if we count also C-style cast)

1. **static_cast**<T>(a) is a safe cast: it converts a to a value of type T only if the conversion is possible (in the lecture on classes we will understand better what it means). For instance **static_cast**<**double**>(5) is possible (but unnecessary since conversion is here implicit) but **static_cast**<**double** *>(d), when d is a double, gives an error because there is no way to convert a double value to a pointer to double value.
2. **const_cast**<T>(a) removes constness. It is a safe cast as well.
3. **reinterpret_cast**<T>(a). This is an unsafe cast, to be used only when necessary and with extreme care. It takes the bit-wise representation of a and interprets it as if it were of type T. It is up to you ensuring it makes sense: no checks are made. There are limitations on its use, but not much relevant.

the `using` keyword and template alias

```
using Real=double; // equivalent to typedef double Real
// func is a pointer to function double->double
 using func = double (*) (double);
// a function taking a function as argument
double integrate(funct f, double a, double b);
//usage
Real a; // Defining a double
// integrate the sin();
auto result=integrate(std::sin,0.,10.);
```

Prefer **using** to the old style **typedef**.

For functions, we will see that we have a much better option than pointers to function!

A suggestion

Use **using** to define aliases to types that are either complicated, or that you may change in the future, or just to give a more significant name, easier to recall

```
// maybe in the future I may want to use float instead
using Real=double;
// This is my choice for Vectors
using Vector=std::vector<Real>;
// To simplify life
using MapIter=std::map<std::string,std::string>::iterator;
...
```

What is a type?

Before dwelling into **auto** and decltype(**auto**) it is important to understand that a type in C++ is in fact formed by different components:

- The "basic type", like **int** or std:;vector<**double**>, which gives information on how the object should be interpreted (and the size it uses up in stack memory). This is indeed the actual type of an object;
- The possible qualifiers: **const** or **volatile** that define the type of access: **const double** a indicates that a is "read only";
- "Adornments" which indicate that the variable is an "alias" to an existing object. They are the l-value and r-value references: **double** & b=a, **const double**&& c=a, here b and c are alternative secondary names for a. (if you don't know what a r-value reference is, don't worry, we will see them later on).

In **const double** & z=a I am defining a reference to a constant double. The "basic" type is **double**. The type is qualified as **const**, so I cannot change a through b.

the auto keyword

In the case where the compiler may determine automatically the type of a variable (typically the return value of a function or a method of a class) you may avoid to indicate the type and use **auto** instead.

```
vector<double>
  solve(Matrix const &,vector<double> const &b);
vector<int> a;
...
auto solution=solve(A,b);
// solution is a vector<double>
auto & b=a[0];
// b is a reference to the first element of a
```

auto returns the base type, omitting qualifiers and adornments, i.e. you must add & or/and `const` yourself if you need them.

Some warnings

auto is a very nice feature. However it may sometimes make the program less understandable, so declare the type explicitly when you think it helps readability.

Beware, moreover, that not always things are what they seem....

```
std::vector<bool> a; // a vector of boolean
...
auto p = a[3]; // p IS NOT A bool!
```

A vector<**bool**> is treated specially: the boolean values are packed inside the vector (one bit per boolean). The returned value of [] operator is a special proxy to a `bool`, convertible `bool`. But in fact you would like to have a full-fledged bool for `p`, so you should avoid auto in this case:

```
bool p = a[3]; // Avoid auto in this case
```

Other uses of auto

We will see other uses of auto in the lecture about functions and lambda expressions.

Extracting the type of an expression

You are probably aware of the command **sizeof**, which returns the size of a type (in bytes). For instance **sizeof**(**double**) returns 8 (in most systems). It can be applied also to expressions: **sizeof** a+b; returns the size of the type of the expression value. We can interrogate also the type of an expression using decltype()

```
const int& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1; // const int& X1
decltype(i) x2; // int x2
decltype(a->x) x3; // double x3
```

This new feature is very useful in generic programming.

decltype(**auto**)

decltype(**auto**) is a different form of **auto** that has different type deduction rule.
We will see its usage in the lecture on functions.

Pointers

Pointers are variables that store the address of an object, and enable us to get the object they point to by dereferencing the pointer (with the exception of pointer to **void** whose discussion is postponed). This code

```
double x =5.0;
double * px= &x; // get the address of x
std::cout<< *px; // dereferencing px
```

It prints 5.

A special pointer called null pointer, and indicated by nullptr (or simply 0), specifies that the pointer does not contain any valid address. A null pointer converts to **false**, while a non-null pointer converts to **true**. So the statement **if** (px !=nullptr) is often written just as simply **if**(px) (but we suggest you to be explicit!).

Pointers, smart and not

In modern C++ we can use pointers for different purposes: to handle resources dynamically or to implement polymorphism, for instance.

For handling resources dynamically, in modern C++ it is much better (almost mandatory!) to use smart pointers.

We have a specific lecture on smart pointers.

Constant variables

The type qualifier **const** indicates that a variable cannot be changed, thus a constant variable must be initialized with a value (with an exception that we will see later).
Since C++11 we have the keyword **constexpr** to indicate constant expressions whose value is known at compile time.

```
double const pi=3.14159265358979;
double const Pi=std::atan(1.0)*4.0;
const unsigned ndim=3u;
double constexpr Pi=3.14159265358979;
float constexpr E(2.7182818f);
double constexpr PI=std::atan(1.0)*4.0;// NOT POSSIBLE
constexpr unsigned Ndim=3u;
```

It is better to understand the difference.

Setting away **const**

Let's recall that in fact you may still change the value of a constant variable using the special cast operator called **const_cast**.

However, if you are obliged to use **const_cast** it means that your code IS POORLY DESIGNED!.

const_cast is only for emergency situations, typically if you have to interface with poorly written or C code, see the next slide.

A case for const_cast<T>

Real world in imperfect. We may have the necessity of stripping the **const** attribute, for instance to be able to call legacy functions where the author forgot to use **const** to indicate arguments that are not changed.
In this case we may use **const_cast**<T>().

```
// this function does not change a and b but by mistake
// they are  taken as double & and not const double &
double minmod(double & a, double & b);
...
// A function that uses minmod
double fluxlimit(double const & ul, double const& ur){
...
minmod(const_cast<double>(ul),const_cast<double>(ur));
```

const vs. constexpr

While a const variable, i.e. an object with a name, is indeed a variable a **constexpr** is not really a variable.
The compiler is free to not allocated memory for it and use its value directly at compile time:

```
constexpr  double a=5.0;
double const b=a*a; // The compiler may compute 25
```

Constant expressions are immutable. There is no way to change their value.
Integral constexpr may be used as tempalte parameter values.

```
constexpr int N=10;
std::array<double,N> arr;// I can use it as template argument
```

is exactly like

```
std::array<double,10> arr;
```

const vs constexpr, concluding remarks

In conclusion,

- Use **const** to indicate that a variable is not meant to change. You have more efficient and safer code!
- Always declare **const** methods that do not change the state (i.e any variable member) of a class;
- Use **constexpr** to indicate constants (like π or e) that are immutable, or integers that you may use as template argument. The compiler may use them directly at compilation time, producing a more efficient code.

Note: The C++20 header <numbers> contains a lot of useful numeric constants, given as **constexpr**.

const and constexpr are your friend!

Use them!. If function parameters are references to something that is not changed inside the body of the function they must be declared const, it is not an option!.

```
void LU(Matrix const & A, Matrix & l,Matrix & u);
```

Here A is not changed by the function, so we must declare it const.

Remember that a non-const references doesn't bind to a temporary object:

```
LU(CreateBigMatrix(),L,U);
```

won't work if A had been declared a non-const reference! While a const reference can bind to a temporary.

A Note: Here, when use the term reference alone I mean l-value references (we will discuss the && stuff later on.)

constance rules

`const` (and `constexpr`) are associated to the item on their left, unless they are the first keyword in the type definition, in which case they apply to the item on the right. The statement

```
double const * const p;
```

means *p is a constant pointer to a constant double*. Neither the value of the pointer nor that of the pointed object can be changed!. While,

```
double const * p;
```

is a pointer to a constant double. You can change the pointer, but not the value!

Enumerators

I assume that you already now about enumerators:

```
enum bctype {Dirichlet,Neumann,Robin};\\definition
...
bctype bc;
..
switch(bc){
 case Dirichlet:
...
 case Neumann:
...
 Default:
...
}
```

They are just "integers with a name", implicitly convertible to integers.

Scoped enumerators

The implicit conversion to integers may be unsafe. In C++ we also have scoped enumerators that behave as a user defined type and are not implicitly convertible to int (you need esplicit casting).

```
enum class Color {RED, GREEN, BLUE};
...

Color color = Color::GREEN;
...
if (color==Color::RED){
    // the color is red
}
```

Now the test **if** (color==0) would fail, but you can still do explicit conversions, if needed, **int** ic=**static_cast**<**int**>(color);

Move semantic

We will have a lecture on move semantic, where I will describe other ways of saving memory, an important issue if you are dealing with “big data”.

Code documentation

Documenting a code is important. I use Doxygen for generating reference manuals automatically from the code. To this aim, `Doxygen` requires to write specially formatted comments. We will show examples during the course and during the exercise sessions. The web site contains an extensive manual and examples.

Beside providing "*doxygenated*" comments to introduce classes and methods, it is important to comment the source code as well, in particular the critical parts of it.

Do not spare comments, but avoid meaningless ones and ... maintain the comments while maintaining your code: a wrong comment is worse than no comment.

How to generate doxygen documentation of the examples

- edit the `DoxyfileCommon` in the Examples root directory, replacing the last line in

  ```
  INCLUDE_PATH = ./include \
                 . \
  /home/forma/Work/pacs/PACSCourse/Material/Examples/include
  ```

  with `MyRootDirectory/include` (full path!).
- Copy the file `DoxyfileCommon` in the folder where you want to generate documentation, renaming it `Doxyfile`
- run `doxygen`
- in `./doc/html` you find the documentation in html, just run your favourite browser and load `index.html`.

Some nice utilities: GetPot and Json++

To be able to input parameters and simple data from files, or from the command line, we provide two utilities in the course repository: GetPot and JSON for Modern C++ The former is in src/Utilities, the second is a submodule in Extras/json (look at the `README_PACS.md` file to install it for the other examples).

For simple visualization of results, the program gnuplot is a valid tool. It allows to plot the results on the screen or produce graphic files in a huge variety of formats. It is driven by commands that can be given interactively or through a script file.

An interesting add-on that allows `gnuplot` to be called from within a program is gnuplot-iostream, and is provided in src/Utilities.

An example of command parsing with GetPot

```
int main (int argc, char** argv)
GetPot    cl(argc, argv);
// Search if we are giving -h or --help option
if( cl.search(2, "-h", "--help") ) printHelp();
// Search if we are giving -v version
bool verbose=cl.search("-v");
// Get file with parameter values
// with option -p filename
string filename = cl.follow("parameters.pot","-p");
```

This enable to parse options passes on the command line: if you rune

```
main -p file.dat
```

filename in the code will contain the string `file.dat`.

An example of GetPot file

GetPot allows also reading parameters from a file:

```
# You can insert comments everywhere in the file.
#
a=10.0  # Assigning a number.
vel=45.6
index=7
[parameters] # A section.
sigma=9.0
mu=7.0
[./other]  # A subsection.
p=56
```

Reading from a getpot file

```
std::ifstream file(``parameters.dat'');
GetPot gp(file);// read the file in a getpot object
double a=gp(``a'',0.0);// 0.0 is the default if a not there
// If you use automatic deduction of type, the type is that
// of the defualt value. In this case, float
auto sigma=gp(``parameters/sigma'',10.0);
auto p=gp(``parameters/other/p'',10.0);
  ...
```

Many other tools and goodies may be found in the online manual at GetPot.

Examples of Json++: a json file

```
{
    "answer": {
        "everything": 42
    },
    "happy": true,
    "list": [1,0,2],
    "stringlist":["first","second","third"],
    "name": "Niels",
    "nothing": null,
    "object": {
         "currency": "USD",
          "value": 42.99
            },
     "pi": 3.141
 }
```

Unfortunately json files do not allow for comments (a real pity).
But json format is a standard.

Reading the json file using the provided utility

See the code in Extras/json/MyExamples/test.cpp

Another nice utility: gnuplot

Gnuplot, see www.gnuplot.info, is a portable command-line driven graphing utility originally created to allow scientists and students to visualize mathematical functions and data interactively.

It is simple and portable to various architecture. We will see example of its use. More details on the indicated web site.

With gnuplot-iostream, provided in `stc/Utilities`, you can even create the plot of the solution from within the code. See instructions in the indicated web site (it is used in some examples).

The Eigen library

The Eigen library (Version 3) is a library of high performance matrices and vectors that we will use often during the course. I will give the details in another lecture. Yet, I am mentioned it here since you need to have it installed to run some of the examples.

If you use the module architecture that will be explained during exercise session you have nothing to do, Eigen library is available. If not you can install it from a package for your distribution or simply download it from the web and follow the instruction (quite simple, is a template only library, no compilation needed to install it!).

If not provided with the modules, you should modify the `Makefile.inc` file to specify where the directory with the header files are contained.