Programmation 2
Lecture title: Introduction

F.Marche

IMAG

2021/2022

Références (les bases)

1. Programmer en langage C++, C.Delannoy, Eyrolles, 2014
2. Programmer en C++ moderne : De C++11 à C++20, C.Delannoy, Eyrolles, 2019
3. Le langage C++ : Initiez-vous à la programmation en C++ - Édition mise à jour avec la norme C++11, Jesse Liberty, Pearson, 2012

Operative System and compilers

The reference operative system for the course is Linux.

We have two reference compilers. The first is the gnu compiler (g++), at least version 4.6.1. It is normally provided with any Linux distribution. Check the version with `g++ -v`.

The second C++ compiler is clang, of the LLVM suite, downloadable from llvm.org. Use version 3.1 or higher. You may find it in most Linux distributions. With respect to the gnu compiler it gives better error messages (and it is faster).

We will stick to C++ standard, so in principle any compiler which complies to the standard should be able to compile the examples.

Development tools

The use of IDEs (Integrated Development environment) may help the development of a software. You may use the software eclipse (downloadable from www.eclipse.org and provided by several Linux distributions)

Another good IDE is Code::Blocks.

Of course, it is not compulsory to use an IDE. A good editor is sufficient ! Personally, I use Atom (which provides git workflows).

For small program, compilation can be done "by hand". For larger programs, the use of cmake may help to deal with the dependencies.

A note on the C++ language

C++ is constantly evolving. New standards called C++11, C++14, C++17 and C++20 have been produced.

The previous standard is indicated by C++98.

The new standard has introduced major addition and changes to the language (portability of C++98 code is however granted).

All major compilers implement at least part of the new standard fully.

We will make use of the most interesting features of the new standard in this course. We will use features already supported by gnu compiler g++ (since version 4.6.1) and clang++ (since version 3.1).

Introduction

A possible definition of scientific computing

Scientific computing is the discipline that allow to compute in an effective way a mathematical model described by a numerical algorithm.
The main objectives are

- To find the good compromise between efficiency (in terms of cpu time and memory usage), generality and re-usability of codes ;
- To control, as far as possible, errors introduced by computation.

To reach this objective we need good knowledge of numerical methods, computing languages and computer architectures.

Main programming language types

- Interpreted Instructions are processed sequentially and translated into actions. Examples are MATLAB, Python. Simplicity of programming and debugging. "Slow" code.
- Compiled The entire source code is translated to machine code, creating an executable. Examples are C, C++, FORTRAN. Faster code. Debugging more difficult. The executable is architecture dependent.
- Semi-compiled The source code is translated into an intermediate, architecture independent code (byte-code). The latter is interpreted "run-time" by a "run-time environment". Example : Java. Intermediate efficiency.

Alternatives for scientific computing

- FortranXX. Mainly procedural programming. Fortran90 has introduced modules and dynamic memory allocation, Fortran03 and Fortran08 some more advanced support of POO. Very good the intrinsic mathematical functions. Produces very efficient coding for mathematical operations.
- Java. There are quite a few numerical applications, yet Java codes are often not efficient enough for scientific computing. Only OO programming. Good for web computing.
- C. Much simpler than C++, it lacks the abstraction of the latter. Many commercial codes for engineering simulations are written in C.
- Python Very effective in building up user interfaces and connect to code written in other languages. Many modules for numerics, like PyNum. Its use in scientific computing is on the rise (see the FeNics project).
- Matlab There is support for OO programming and can be compiled (if you buy the compiler), but it is essentially an interpreted language. It's free "clones" Octave and *Scilab* are good alternatives.

The compilation process in C(++) - simplified-

Command `g++ -o prog prog.cc` would execute both compilation and linking steps.

MORE G++ OPTIONS (some are in fact preprocessor options)

-g	Produce debugging information	-O[0-3]	Optimization
-Wall	Activate warnings	-Idirname	Directory of header files
-DMACRO	Activate MACRO	-Ldirname	Directory of libraries
-std=c++0x	activate c++11 feature (g++ 4.6)	-std=c++11	activate c++11 feature (g++>=4.7/clang)

Compilation unit

A C++ program is normally formed by several source files (*.cpp) and related user defined or system header files (*.hpp).

The header files are normally kept either in the same directory of the source files or, more frequently, in a separate directory (typically called `include`).

Each source file contains the code that implements specific functionalities and, together with the set of header files it includes, is called a compilation unit.

The compilation process treats each compilation unit independently, producing object files (*.o).

The linker will gather the information of the object files to produce an executable program.

Some elements of C++

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 functionality is provided by the Standard Library and require to `include` the appropriate `header` file (also called `include` file).

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

List of C++ standard header files (C++98)

<*algorithm*>	general algorithms
<*bitset*>	set of Booleans
<*cassert*>	diagnostics, defines *assert*() macro
<*cctype*>	C-style character handling
<*cerrno*>	C-style error handling
<*cfloat*>	C-style floating point limits
<*climits*>	C-style integer limits
<*clocale*>	C-style localization
<*cmath*>	mathematical functions (real numbers)
<*complex*>	complex numbers and functions
<*csetjmp*>	nonlocal jumps
<*csignal*>	C-style signal handling
<*cstdarg*>	variable arguments
<*cstddef*>	common definitions
<*cstdio*>	C-style standard input and output
<*cstdlib*>	general utilities
<*cstring*>	C-style string handling
<*ctime*>	C-style date and time
<*cwchar*>	C-style wide-character string functions
<*cwtype*>	wide-character classification
<*deque*>	double-ended queue
<*exception*>	exception handling
<*fstream*>	streams to and from files
<*functional*>	function objects
<*iomanip*>	input and output stream manipulators
<*ios*>	standard *iostream* bases
<*iosfwd*>	forward declarations of I/O facilities
<*iostream*>	standard *iostream* objects and operations
<*istream*>	input stream
<*iterator*>	iterators and iterator support
<*limits*>	numeric limits, different from <*climits*>
<*list*>	doubly linked list
<*locale*>	represent cultural differences
<*map*>	associative array
<*memory*>	allocators for containers
<*new*>	dynamic memory management
<*numeric*>	numeric algorithms
<*ostream*>	output stream
<*queue*>	queue
<*set*>	set
<*sstream*>	streams to and from strings

C and C++

The C language also provides many `header files`. Most of them have been inherited by C++, but the name has been changed using the following rule

C Header file	C++ header file
name.h	cname

For instance the C header file `assert.h` is available in C++ under the name `cassert`. Standard header file are normally included using **#include** <file> and not **#include** "file".

The latter format is usually reserved to user defined header files (the way the preprocessors searches directories to find the header file is different in the two cases).

In C++ the suffix `hpp` is preferred to `h` (but it is not compulsory).

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 namespace 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 namespace
...
double c=sqrt(2*a);// OK
```

With **using namespace** std you bring all names in the std namespace into the current scope.

The `main()` Program

The main program defines the entry point 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 execution.
Their processing is however a little cumbersome. In our course we will make use of the GetPot utility to make life easier (later).

What does `main()` return ?

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

The integer returned by the `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".

Another way to set the return status is by using the `exit()` statement (you must include the `cstdlib` header in this case).

Identifier and Name

- Identifier : an alphanumeric string that identifies a variable or a function uniquely. The identifier of a variable is its name, for a function it is its signature.
- Name : an alphanumeric string that identifies entities (variables, sets of overloaded functions).

  Names in C++ cannot begin with a numeric character and are case sensitive.

  Hiding applies to names.

  A name is fully qualified if it contains the name of its enclosing class or namespace using the scope resolution operator, i.e. `std::cout` is qualified, while `cout` is not.

  A name cannot be equal to a keyword of the language.

Scope

It is a part of a program made of statements enclosed by a pair of braces {} (with the exception of the global scope that is outside all braces).

Variables defined in the global scope are global variables (visible everywhere in the program and possibly qualified by the global scope identifier :: for disambiguation).

Variable defined in a local scope are local variables.

Scopes may be given a name, through the **namespace** statement, and a named scope is also called a namespace (later).

Another type of named scope is introduced by a class : the members of a class are in the scope of the class (later).

Scopes may be nested and names in named scopes can be addressed by qualifying them or brought to the local scope by the **using** statement.

Objects, Types and Variables

- Object : a memory location, which typically stores a variable.

  The process of creating an object is called construction or instantiation, while the process by which the object is erased from memory is called destruction.

- Variable : a named object in a scope, which represents a specific item of data that we keep track of in a program.

  All variables have a Type.
  A variable is a member of a class if it belongs to the definition of that class.
  A variable is constant is its content cannot be changed.

- Hiding : mechanism by which a name in an enclosed scope hides a name (function) declared in the enclosing scope. The object associated to the hidden name may still be accessed using its fully qualified name.

Operators and Expressions

- Operators : a particular function that combines one or two arguments (in one case three arguments) and returns a value.

  Example : + can be a unary operator (c=+5) or a binary operator (c=a+b)

  C++ allows overloading of most operators (later).

  An operator has a preferred signature and return type that should be complied with when overloading.

  Example : the signature and return type of the binary addition is T & **operator** + (T **const** &, T **const** &).
- Expression : combination of operators, function calls, and variables (or constants) that produce a value.

  Example : 5+a where a is a int, is an integer expression.

Declaration and definition

- Declaration : introduces an identifier for an object in a scope and specifies its type, allowing (compiler) to determine its size.

  Declaration may occur more than once in a program, but they must be identical.
- Definition : provides the information necessary to create an object in its entirety.

  Defining a function means providing a function body and defining a variable means specifying its value etc.

  A definition is also a declaration.
- Initialization : providing the initial value to an object during its construction.
- Assignment : providing a value to an already constructed object.

A first (and useless) example

```
// A simple (and useless) C++ program
#include<iostream> // for i/o
using namespace std; //loads the standard library in
    the global scope
int main()
{
  cout<<"The␣sum␣of␣5+2␣is␣"<< 5+2<< endl;
  /* if I not use namespace std:
     std::cout<<"The sum of 5+2 is "<< 5+2<< std::endl;
  */
}
```

Formatted i/o

C++ provides a sophisticated mechanism for i/o through the use of streams (more details later).

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 (default = cout)

A second example

```
#include <iostream>// include the standar module for
    input/output

int main() {
  using namespace std;// Bring std namespace in the
      local scope
  int n, m;  // declare n and m to be integers
  cout << "Enter two integers:" << endl; // output to
      screen
  cin >> n >> m; // assign  the input to n and m
  int sum=n+m; // initialise sum
  cout << "The sum of " << n << " and " << m
       << " is: " << sum << endl; // output to screen
}
```

Declarations and initialization

The statement **int** n,m; declares two integer variables.

C++ is a strongly typed language : all variables must be declared by specifying their type before they can be used.

This is in contrast with some interpreted languages like MATLAB® or Python, where the type of a variable is derived from the context.

This declaration also implies the instantiation in the computer memory and the initialization to the default value of zero.

Never assume that a variable is initialized automatically. Always initialize variable explicitly, it is safer !.

For instance no default initialization may be performed for member variables of a struct or a class type (which will be introduced later)

Initialization : old and new (C++11)

C++11 has introduced a new way of initializing variables extending the parameter list initialization that was available before only for fixed dimension arrays

```
int i(0); //i is initialized to 0
int j=0; // as before: equivalent to j(0)
int k{0}; // only c++11
int f={0}; // only c++11
// initializes an array of 3 elements
double c[]={0.,1, 3.} // OK
struct data{ int a; float b;}// Declaration
data d={3,4.0};// only c++11
// a vector<double> of 3 values (only c++11)
std::vector<double> v{4.,5.,6};
// a vector<int> of 2 values (only c++11)
std::vector<int> w={1,2};
```

Names and reserved keywords

Valid names in C++ are formed by sequence of alphanumeric ($[0, \ldots, 9]$ and $[a, \ldots, Z]$) and possibly the underscore (_) character.

A name cannot start with a digit (but it can start with the underscore) and is case sensitive : foo is a different that Foo.

There is another important restriction, a valid name cannot coincide with a reserved keyword used by the language. For instance we cannot use **if** nor **using**, since they are already reserved by the language.

There are 74 keywords, you may recognise the meaning of some of them, others will be introduced later on.

Names and reserved keywords

and	**and_eq**	**asm**	**auto**	**bitand**	**bitor**	**bool**
break	**case**	**catch**	**char**	**class**	**compl**	**const**
const_cast	**continue**	**default**	**delete**	**do**	**double**	**dynamic_cast**
else	**enum**	**explicit**	**export**	**extern**	**false**	**float**
for	**friend**	**goto**	**if**	**inline**	**int**	**long**
mutable	**namespace**	**new**	**not**	**not_eq**	**operator**	**or**
or_eq	**private**	**protected**	**public**	**register**	**reinterpret_cast**	**return**
short	**signed**	**sizeof**	**static**	**static_cast**	**struct**	**switch**
template	**this**	**throw**	**true**	**try**	**typedef**	**typeid**
typename	**union**	**unsigned**	**using**	**virtual**	**void**	**volatile**
wchar_t	**while**	**xor**	**xor_eq**			

An important remark

The issue of choosing relevant names for variables, functions and in general the various named items that compose a program is important. Yet in general it is better to use significant names (even if they may be long), because it makes the code more readable for a collaborator or a user. `triangleArea` or `degreeOfFreedom` is certainly more significant than `t` and `d`, for instance.

Moreover it is better to use a consistent style. A style we normally follow is to write variable and function names in lower case letter, with a upper case for the first letter of function names. When the name is composed by different words, we use the underscore to separate them. Names of user-defined types, also may begins with a upper case letter. Integer constant are sometimes written in all uppercase, like `NDIM`.

Scope and variable visibility

In C++ variables may be accessible only in parts of the program. This gives a great flexibility.

Any group of statements between two curly brackets ({ and }) defines a scope

Statements outside all curly brackets are in the global scope.

A namespace is a special scope with a name. We will discuss about them later.

A variable is visible in a given scope if it can be accessed without using the scope resolution operator ::.

Scopes can be nested : one can speak of an internal (or enclosed) and external (or enclosing) scope. The global scope is external to all scopes in the same compilation units.

A variable (or function) declared in a scope is local to that scope. It is visible inside it and in all enclosed scopes (unless hidden, see next).

Basic types

Basic types

Every variable in a C++ program has a well defined type, which has to be declared before the use of the variable.

A type basically identifies the memory storage requirement and the proper use of a variable.

Types may be built-in or user defined. Here, we deal with the former.

The basic unit for memory size is the byte, which is typically formed by 8 bits. The command **sizeof**(**typename**), where `typename` is the name of a type (or an object), returns the number of bytes occupied by a variable of that type (or by the object).

The standard only requires that
sizeof(short int) $\leq$ sizeof(int) $\leq$ sizeof(long int), and that the size of a short int is at least of 16 bits (i.e. 2 bytes) and that of a long int at least 32 bits (8 bytes).

Basic types

Type	*Description*
int	Integer type (usually 4 bytes). Used to store integer values.
long int	Integer type (maybe 8 bytes). Used to store integer values when large values are expected
short int	Integer type (maybe 2 bytes). Used to store small integers (seldom used)
char	Character type. The size is of just one byte. Used to store a single alphanumeric character. It is a integral type, that is it can be used also to store very small integers in the range $[-128, 127]$
float	Floating point type (usually 4 bytes). Used to store real values.
double	Floating point type (usually 8 bytes). Used to store large real values.
bool	Boolean (or logical) type. It may store either **true** (1) or **false** (0). Normally used in boolean expressions.
void	A special type with no type! Can be used only as return type for function which do not return a value, or for pointers to objects of unknown type (*void pointer***void** $*$).

Integral types (**int**, **long int**,**short int** and **char**) are also present in their unsigned version, used to store only positive numbers. This allows to have a larger maximum allowed value.

For instance if an **int** may be in the range $[-2147483647, 2147483648]$, the corresponding **unsigned int** may take values in $[0, 4294967295]$.

Constants

Often one needs to store values that are used only in read mode, that is never modified during the execution of the program.

Moreover one wants to make sure that the compiler does not allow us to modify them by mistake.

Constants can be used only in "reading mode", thus they are not lvalues.

A constant type is obtained by using the keywords **constexpr** (if its value is know at compile time, C++11 only) or **const** (if its value is only know at running time).

```
#include <cmath>
...
constexpr double PI = 3.14159265359;
int n;
std::cin >> n
const int NDIM = n+2;
...
```

Enumerators

An enumeration is another in-built type that defines sets of named integer constants, called enumerators.

Let us suppose that we are writing a finite element code and we want to be able to identify if a node is internal or it belongs to the Dirichlet or Neumann.

A possibility is to store an integer for each node where by convention, 0 means internal, 1 Dirichlet and so on. This is fine but there is a drawback : one has to remember the meaning of the value, maybe by trusting that the code documentation is well written and kept updated. With am enumeration you are less likely to make errors.

Enumerators

```
enum BcType {Dirichlet, Neumann};
  ...
BcType nodeType = Dirichlet;
  ...

switch(nodeType){

case Dirichlet:
        // Do what required for a Dirichlet node
        ...
        break;
case Neumann:
        // Do what required for a Neumann node
        ...
        break;
default:
        cout<< "Error: Wrong condition for a node"<<endl;
}
```

Enumerators

The statement

```
BcType {Internal, Dirichlet, Neumann, Robin};
```

defines an enumeration called BcType which consists of the set of enumerators indicated inside the curly brackets. Those values are in fact integer constants and by default the first enumerator in the set (Internal) takes the value 0, the second (Dirichlet) 1 and so on.

The default may be changed by giving the values explicitly as in

```
enum BcType {Internal=-10, Dirichlet=10, Neumann=20, Robin=30};
```

Yet the most relevant issue is that an enumerator of type BcType can take only values in the corresponding set.

Implicit and explicit conversion

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

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

Note : It is safer to use explicit conversion and it is more efficient to use static casting.

C-style case, e.g. z= (**double**) a; , is allowed but discouraged in C++.

Special conversions

```
int a=5; char A='A'
bool c=a; c is true
a=A; a is the ASCII code of 'A'
```

Any non null value of POD is converted to **true** The null pointer is converted to `false`, a non-null pointer to `true`.

Note : This rule may be useful.

typedef

The command **typedef** creates an alias to a type and it is very useful to save typing or having to remember complex types.

```
typedef unsigned int Uint;
typedef vector<float> Vf;
typedef double Real;
typedef Real myvect[20];
...
Vf v;// v is a vector of floats
myvect z // z is an array of 20 doubles
```

A simple rule If you take out `typedef` you obtain the declaration of a variable. Typedefs are useful if you have long type names : a vector of a vector of pointers to double :

```
typedef vector<vector<double*> > Vvdb
```

the auto keyword (C++11)

The new C++11 standard has introduced a new keyword to simplify the handling of complicated types and also ease generic programming. 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 converts to a the type, omitting qualifiers : you should add & or const if needed.

Extracting the type of an expression (C++11)

You are probably aware of the command **sizeof**(), which returns the size of an expression or of a type (in bytes). For instance **sizeof**(**double**) returns 8 (in most systems). With C++11 we can finally 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 handy for generic programming.

Operators

Operators

An operator is a notation for built-in operations like additions (+), assignment (=) etc.

Operators can be unary or binary (there is only one ternary operator) depending whether we have one or two arguments.

Most operators are normally used in their symbolic form, like in a+b (a binary operator) or *i (a unary prefix operator), but they have also an extended form.

For the two operators just written the extended form is **operator**+(a,b) and **operator***(i), respectively.

The extended form is needed for operator overloading, a subject that we will treat later.

Assigment

This operator assigns a value to a variable and returns that value.

Therefore statements like **if**(a=b){...} are valid : the value contained in b is copied in a and returned, so it is at disposal for the test.

Beware that the meaning is completely different from the (apparently) similar statement **if**(a==b){...} which compares the content of the two variables and then operates the test.

The assignment is a right associative operator. i.e. a=b=c; is equivalent to a=(b=c); (as expected, since at the end of the statement we want both a and b be equal to c).

Arithmetic operators (+, −,∗, /, %)

They are binary operators a part from + and − which have a binary and unary form : −a changes the sign of a and +a does nothing but is present for completeness.

The operator % is the modulo of two integers : i %j returns the remainder of the integer division between i and j.

The modulo operator is often used for pretty printing : assume you want to print a long vector with 8 numbers on each line :

```
int const numbxline=8; // the number of numbers per line
for (unsigned int i=0;i<v.size();++i){
  cout<<v[i]; // no line feed
  if((i+1) % numbxline) cout<<" ";// print a space
  else cout<<endl; // go to next line
}
```

Increment and decrement operators (++,--)

These operators are present in both postfix and prefix form. Let us illustrate the difference.

++i	Prefix form	Increment i and fetch the value
i++	Postfix form	Fetch the value and increment i

We give a simple code snippet

```
int i=5;
cout<<++i<<endl; // prints 6 and now i=6
cout<<i++<<endl; // prints 5 and now i=6
```

Increment and decrement operators (++,--)

These operators are most often used in **for** loops. Which form is better in this case? Let us compare the two following loops

```
for(int i=0;i<1000;++i){....} \\prefix form
```

```
for(int i=0;i<1000;i++){....} \\postfix form
```

They both give the same final result, yet the first is preferable since the postfix form of the operator has to create a temporary to store to value before the increment.

So in for loops use always the prefix form.

The in-built increment and decrement operators can be applied only to integral types, to pointers and (partially) to iterators. Yet you can always define a version for other types as we will see later on.

Compound assignment (+=, −=, ∗=, ...)

All arithmetic operators (and other operators as well...) have also a compound assignment counterpart of the form op=, where op is the original operator.

The statement a op= b is equivalent to a= a op b, but in general is more efficient. Suppose we want to compute the arithmetic mean of the values stored in the STL vector v of doubles :

```
double sum(0.0);
 for (unsigned int i=0;i!=v.size();++i) {
     sum += v[i]; // accumulate
     sum /= v.size(); // equivalent to sum =sum/v.size()
  }
```

Comparison operators (<, <=, >, ==, >=, != , etc.)

These are logical operators, that is they return a boolean value.

The meaning is straightforward, so we will not take too much time in their description.

We only warn again about the common mistake of writing an assignment operator = instead of the equality operator ==.

Logical operators (&&, ||, !)

&&	Logical and (binary op.)
||	Logical or (binary op.)
!	Logical not (unary op.)

The || operator is not exclusive, i.e. the value of **true** || **false** is **true**.

Conditional expression operator (?:)

This is the only trinary operator and its meaning is the following.

The expression cond?a:b tests the boolean expression cond and if it is **true** it returns a, otherwise it returns expression b.

Let us define a home-made max() function using this statement :

```
double max (double const & a, double const & b){
 return a>b?a:b; // if a>b return a; otherwise b
 }
...
double c=max(3.0,4.0); // c is 4.0
...
```

Control structures

Control structures : if−else

its general form is

```
if (expr 1){
... // block 1
}
else if (expr 2){
  ... //block 2
}
    ... //possible other blocks
else{
  ... //else block
}
```

The number of blocks is arbitrary and you can have also just a single block (an **if**-statement).
If a block is formed by one line only the braces can be omitted.
In all cases a block defines a scope, so variables declared inside it are local.

Control structures : **while**

The while statement is a loop structure where the loop depends on a condition. It is present in two forms,

```
while (control expr){
    //while block
}
```

and

```
do{
    //do–while block
} while (control expr)
```

In the first form the control expression is checked and the block execution repeated as long as its value is **true**. In the second form the block is executed and then repeated as long as the value of the expression is **true**.

An example

```
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;}
```

Control structures : for loop

The **for** loop is the classical control structure for loops. The form is

```
for (initialisation; condition; increase){
     ... for block
}
```

The initialisation is an expression which is executed only once when we enter the loop. It is normally used to initialise variables used in the statement block.

Variables declared here are local. condition is a boolean expression that is evaluated at the beginning of each cycle.

```
int i(0);
... // I use the ''global'' variable i as a counter
    // I have an empty initialisation
for(i; i<10; ++i){
  ... //do something
  }
a=b[i]; // i is here equal to 10!
...
```

Control structures : **switch** conditional structure

It is often used in conjunction with enumerations.

```
enum FEMOrder{Linear, Quadratic, Cubic};
...
FeMOrder eOrder=getFemOrder(element);
switch (eOrder){
  case Linear:
     ... // operations for linear elements
     break;
  case Quadratic:
     ... // operations for quadratic elements
     break;
  case Cubic:
     ... // operations for cubic elements
     break;
  default:
    cout<<" Element of order not recognised";
    exit(1);
}
```

Pointers and references

Introduction to pointers

An area in memory able to store a value of a given type is called an object. A variable is then an object with an identifier (name) by which we can access its value. However, there is another way to access an object : through pointers.

A pointer is itself an object, but it may contain the memory address of another object.

A pointer able to address an object of type T is of type T* (a pointer to T). The address of an object is obtained by the address of operator & (also called addressing operator).

The value of the object pointed by a given pointer may be accessed by using the dereferencing operator *.

A pointer may be given the value 0, in this case it is a null pointer.

Introduction to pointers

A pointer that does not contain a valid address of an existing object is invalid, or dangling. Dangling pointers are very dangerous, since trying to dereference them causes unpredictable results.

In the best case the interruption of the program with a `Segmentation fault` error. Segmentation faults occurs when your program is trying to access an area of memory which is forbidden.

In the worst case your program will keep running but it will give the wrong results ! It may be very difficult to debug.
The following statement

```
int * a;
```

which declares a pointer to integer named a is dangerous because a is now a dangling pointer. It is better to write instead

```
int * a = nullptr;
```

which initializes a to the "null pointer".

An example

Here a code snippet to remind pointer syntax :

```
int a = 5;
float f = 3.14;
int * pa=0; // Pointer to int, initialises to null
pa = &a;          // Now pa points to a
float* pf = &f;// Pointer to float initialises with the address of
*pa = 7; // Now a contains 7
cout << a <<" " << *pf << endl; //prints 7  3.14
cout << pa << endl;// prints the address of a (not so useful)
if (...){   // a conditional structure
    int c; // a local automatic variable!
    pa = &c; // pa points to c
    ...
}
/* Beware: now pa is a dangling pointer
    since c has been destroyed. Better set it to null: */
pa = nullptr;
```

References

Another way to access an object is by reference. References are alias to an existing object.

A reference type is declared by adding the ampersand & to the name of the type.

We stress that, being an alias, a reference must always refer to an existing object, so it must always be initialises.

```
double a; // a is a double
  ...
double & ra = a; //ra is a reference to a
ra = 5; //Now a is also equal to 5
   ...
a = 10;
cout << ra; // Will print 10
```

References

We have a symbol with two different meaning :

- The ampersand in front of an object returns the address of that object,
- after the name of a type it defines a reference to that type.

The use of references will become important with functions and with polymorphic classes, and we will discuss more about them later.

References cannot be reassigned to another variable. The reference and the referenced object cannot be dissociated.
If the referenced object is destroyed, the reference becomes dangling and this is a very dangerous situation, even worse than dangling pointers, since references cannot be reassigned.

Arrays

C style arrays

An array for 100 doubles is declared with the statement

```
double a[100];
```

The array elements are numbered from 0 to 99 and we can access the 10th element in the array by simply writing a[9].

Arrays can be initialized in the following way

```
double a[]={2.,3.,5.,9.4};
```

We can have multidimensional arrays as well, like

```
int p[2][3]={{1,2,3},{4,5,6}};
```

where p is an array of integers with 2 rows and 3 columns initialised as

$$p = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

C style arrays

Remind that the layout of a multidimensional array in memory is contiguous and linear, with row wise organisation.

A similar layout is present for arrays with more than two dimension, likely

```
double a[3][5][10],
```

whose elements may be addresses by a[i][j][k].

Initialise arrays with more than two dimensions :

```
int p[2][2][3]={{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}};
```

yet normally arrays are read from a file or computed.

C style arrays

There is no bound check on C-style array, then the code containing the statements

```
double a[20];
  ...
double c=a[30];
```

will compile and run, but with unexpected (and wrong) results. It must be care of the programmer to verify that arrays are properly addressed in the program.

C++ STL vectors

The C++ standard library has introduced a new type of arrays which has dynamic memory allocation and whose use is much simpler than traditional C-style arrays. It is one of the container classes of the STL (later ...).

To use STL vectors we need to include the <vector> header. Being part of the standard library, all functionality are provided under the namespace stl. However, for the sake of brevity, we will omit to indicate std:: in the rest of this paragraph.

To declare an empty vector of int you write

```
vector<int> v;
```

while if you want a vector with 10 elements you write

```
vector<int> v(10);
```

C++ STL vectors

In this case a contiguous storage area sufficient for 10 **int**s is requested to the operative system and the first 10 elements are initialised using the default constructor for the **int**s, which just set them to zero.

If we want to initialise the vector with another value for all elements, we add another entry to the declaration :

```
vector<int> v(10,5);// a vector of 10 elements initialised with 5
```

You can build a vector from another vector,

```
vector<int> p(v);/ ps is now a copy of v
```

To add an element at the end of the vector we can use the method push_back.

```
p.push_back(5); // now pi has 11 elements
        // the 11th element is equal to 5.
```

Dynamic memory management

Operator **new** returns a pointer to a chunk of contiguous memory in the free storage, the memory area dedicated to dynamically allocated objects, while the stack is used to store objects whose size is known at compile time.

The main difference is that while the dimension of the stack can computed at compilation time, requests to the free storage are at run time, so the operator **new** may fail if not enough memory is available on the system.

Dynamic memory management

The main ways to use the **new** operator are listed here (T indicates a generic type (in-built or user defined)). The operator always returns a pointer to the given type.

new T	Allocates storage for an element of type T, which is build by calling the default constructor
new T(z)	Allocates storage for an element of type T and passes z to the constructor
new T[m]	Allocates storage for m elements of type T, here we have in fact called operator **new**[]

The two operators share the same name but they are different.

Memory allocated with **new** must be deleted using the **delete** operator,

If we used **new**[] we must use **delete**[]

Examples

```
double * a = new double(3);// a points to a double
   // initialized with 3
cout << *a; // 3 is written on the terminal
int n;
cin >> n; // n is read from the terminal
int * ia= new int[n] // ia points to a storage area for n integers
delete a; // Release memory. Now a is dangling.
a = 0;// It is a good practise to set to null unused pointers
delete[] ia; // Release 100 integers from memory
ia = 0;
```

Dynamic arrays and pointers

Pointer and arrays are linked concepts : pointers are the basic way to have dynamic arrays in C++

```
double * p = new double[3];
..
for (int i=0; i<3; ++i) p[i] = i*i;
...
delete[] p; // remember to cleanup when p not needed anymore
```

p[i] is perfectly equivalent to *(p+i). Adding the integer i to pointer p return the pointer obtained by offsetting p of $i * sizeof(double)$ bytes (since p is a pointer to **double**).

Note that **no bound checking is performed**, it is your full responsibility.

Multidimensional dynamic arrays

They may be implemented as dynamic arrays of pointers : (n and m are here given integers)

```
double ** a = new double*[n]; //dynamic array of n pointers
for (int i=0; i<n; ++i) a[i]=new double[m]; //space for the columns
...
a[2][3]=5;
...
double c=a[0][1];
...
// memory cleanup when a no more needed
for (int i=0; i<n;++i) delete a[i];
delete[] a;
```

This implementation is highly inefficient, because the different blocks are in general not stored in a contiguous area of memory.

Functions

Functions basics

Functions are a basic feature of the C++ language and we have already introduced them in several examples.

Distinguish between function declarations which are statements by which we give the necessary information to the compiler to perform all what necessary to use the function, and definitions, where we actually specify how a function works

Remind the general rule that a definition is also a declaration.

In C++ there are two types of functions : free functions (also called just functions) and methods of a class. In this section we deal only with the first.

Functions : declaration

```
return_type fname(type_arg1 arg1, type_arg2 arg2,...);
```

The return_type can be equal to **void** the function does not return any object (we cannot omit the return type of a function).

fname is the name of the function, which is used for the scoping mechanism.

The function can have zero, one or more arguments, of type type_arg1, type_arg1, etc. The name of the arguments, arg1, arg2, etc, can be omitted in a declaration.

```
double f(double x);
void g(double const &a, vector<double> & v);
double h(int, double, double);
```

are all valid declarations.

Functions : signature

A very important feature of C++ function is that they are not unequivocally identified just by their name, but by their signature, which is formed by the name and the type of all arguments.

The return type is not part of the signature. Thus, the function g of the example has signature

```
g(double const&, vector<double>&);
```

and it is a different function than, say

```
g(double const&, double *);
```

even if the two functions share the same name.

However, for what concerns scoping mechanism what matters is the name.

Functions : definition

The definition of a function differs from the declaration from the fact that the name of the argument cannot be omitted and that the body of the function is present.

```
 return_type fname(type_arg1 arg1, type_arg2 arg2,...){
...  function body
 return expr;
}
```

The function body is a block of statements which implements the functionality of the function, the **return** statement is compulsory if the function has a return_type different from **void**, and must be followed by an expression of type return_type, otherwise it is omitted.

The function arguments are local variables.

Functions : arguments transmission

The arguments in the function definition are normally called formal arguments, to distinguish them from the actual arguments which are the ones passed to the function at the moment of the call.

```
double f(double a, double & b, double const & c){
double y;
.... // some statements
return y;
}


int main(){
double x,z;
.....
double k=f(x,z,3.0)
}
```

Functions : arguments transmission

When the program reaches the function call :

1. the formal arguments are initialized with the corresponding actual arguments. So it is like we were writing

```
double a=x;
double& b=z;
double const & c=3.0;
```

and the formal arguments are local to the function.

Having declared some of the formal arguments as references allows us to address directly the object of the calling program.

If the reference is not constant, like in the case of the variable b we can modify the content of the corresponding actual arguments.

Therefore, a function may return values to the calling program not only with the **return** statement, but also by having non constant references as formal argument.

Functions : arguments transmission

1. With basic types, we make a local copy, like for the variable a. Any modification to a is only local to the function and does not affect the corresponding actual argument x.
2. The body of the function is executed.
3. When the program hits a return statement the value of the expression at his right, the value of the local variable y in this case is evaluated and returned.
4. The assignment operator in the calling program assign the returned value to the variable k.

If we declare a formal argument as constant reference we cannot modify its content inside the function, so the corresponding actual argument is effectively passed in "read-only" mode.

Functions : arguments transmission

Another way of having an interaction between the calling program and the function is by pointers. For instance in the code,

```
void LUDec(double** M){
}
int main(){
double ** A;
// fill in matrix A with some values
LUDec(A);
}
```

what is passed to the function is a pointer and when the formal argument M is initialized with A it points to the same memory area as A. Consequently, changes in the values of the array M reflect directly on A.

Passing arguments : some rules

- Prefer calling-by-reference when dealing with big objects. You may avoid an unnecessary copy and it is in general more efficient. The only case where call-by-value could be more efficient is with integers ;
- Always declare **const** formal arguments that the function does not modify.
  1. it makes the program more readable, particularly when calling-by-reference or passing a pointer, since the user can immediately recognize that the function is not going to modify the content of the passed argument
  2. it may allow the compiler to do optimisations that will be precluded with non constants.
- A literal constant can be passed to the function only by value or by constant reference ;

Functions : the returned value

A function with a return value different from **void** should contain at least one **return** statement.

The standard mechanism for returning a value is to create a temporary object in memory (or in a CPU register if it fits) with value evaluated from the expression at the right of the **return** statement.

- Never return a value dereferenced from a local pointer :

```
Point calculateBaricenter(Triangle const & t){
 Point* p = new Point; // create a point
...
return *p; // return the point
}
```

Functions : the returned value

A local Point is created on the free store, its value calculated in the expression *p and returned. The memory for storing the local Point is never returned to the operative system, and there is no way of doing it.

In the previous example there is apparently no need of using dynamic allocation, The code

```
Point calculateBaricenter(Triangle const & t){
 Point p; // a Local point
...
return p; // return the point
}
```

is fine. A temporary Point is created to hold the returned value.

Functions : the returned value

Another example :

```
vector<Point> calculateBaricenter(vector<Triangle> const & t){
  vector<Point> p(t.size()) ; // create a vector of points
  ...
  return p; // return the vector
}
```

This example is fine. However, because of the way the return mechanism works a temporary vector of points will be created to hold the value of p. If the vector is big this is inefficient.

Functions : the returned value

The solution is to pass the vector to be computed as argument, by reference :

```
  void calculateBaricenter(vector<Triangle> const & t,
     vector<Point> & p){
  p.resize(t.size()) ; // make the vector of the right si
  ...
  // No return statement
}
```

In this version we operate directly on the actual vector<Point> passed to the function, since we have a reference, no temporary is created.

Functions : the returned value

One may think that the situation could have been resolved by passing a reference as return type :

```
vector<Point> & calculateBaricenter(vector<Triangle> const & t){
vector<Point> p(t.size()) ; // create a vector of points
...
return p; // return the vector
}
```

This is a **big mistake**. The variable p is local to the function, so it is destroyed after function call. There is no problem if in the main program we store the returned value in an object, for instance with the instruction

```
vector<Point> baric=calculateBaricenter(t);
```

but this implies again creating two vector of Points twice, the temporary p and baric to hold the result.

One then may tempted to do something of the type

```
vector<Point> & baric=calculateBaricenter(t);
```

After all the function is returning a reference. This is simply a disaster. When the local variable p is destroyed (and it happens immediately after the call of the function calculateBaricenter) the reference baric is a dangling reference. If you are lucky the program aborts. If you are unlucky you get false results.

Therefore, since in the best case we are not gaining anything a good rule is

Never return references to local objects. If you have to return big objects it is better tu use a non-constant reference as argument

Overloading of functions

The inclusion of the argument types in the function identifier is the mechanism by which we have function overloading in C++.

We can have functions with the same name but different argument types (and number).The compiler then selects the one that fits best.

Let see an example. We want to write a function to compute the Euclidean norm of vectors. We want to treat vectors of real numbers. Moreover, we want to account for the fact that we may store the vector in a dynamic array or, alternatively, on STL vectors of doubles.

Overloading of functions

We put all the declarations in a header file "norm.hpp" :

```
#include<vector>
namespace LinearAlgebra{
  using namespace std;
  // Computes euclidian norm. Vector stored in STL vector
  double norm(vector<double>const & v);
  // Computes euclidean norm. Vector stored in a dynamic array
  double norm(double const* v, unsigned int n);
}
```

We have set our declarations in the namespace LinearAlgebra and included the required standard headers. Note that we have passed the STL vectors as constant references to avoid local copies.

Overloading of functions

Now here is the implementation

```
#include "norms.hpp"
#include <cmath>
namespace LinearAlgebra{
  using namespace std;
  double norm(vector<double>const & v)
  {
    return norm(&(v[0]),v.size());
  }
  double norm(double const* v, unsigned int n)
  {
    double res(0);
    for(unsigned int i=0;i<n;++i)res+=v[i]*v[i];
    return sqrt(res);
  }
}
```

Overloading of functions

We have implemented the version accepting STL vectors in function of that for standard dynamic arrays. It implies a small overhead (an extra function call) but it simplifies code maintenance. If there are no particular efficiency issues it is always better to reuse code instead of replicating it.

Overloading of functions

Now the main

```
#include<iostream>
#include<vector>
#include "norms.hpp"
int main()
{
  using namespace std;
  using namespace LinearAlgebra;
  vector<double>v1;
  double * v2;
  v2= new double[10];
  // just to fill some values
  for (int i=0;i<10;++i)v2[i]=sqrt(double(i));
  v1.reserve(10);
  for (int i=0;i<10;++i)v1.push_back(v2[i]/2);

  cout<< "norm of v1="<< norm(v1)<<endl;
  cout<< "norm of v2="<< norm(v2,10)<<endl;
  delete[] v2;
}
```

Overloading of functions

The compiler is able to resolve all calls to the different norm() functions by selecting the one that best match the requested signature.

It follows some complex matching rules that we are not reporting here. The general idea is that if there is a function with a signature that match perfectly it is chosen (it is the case in our example).

Yet, it may be possible that a function is selected because the signature matches after some allowed conversion of the arguments.

If there are no matching functions of functions that matches equally well an error message is issued and the compilation stops.

Class string

For the manipulation of characters strings. See the code main_strings.cpp.

Exercice 1

Write a function with the following prototype

```
std::pair<double,double>
quadraticRoot(const double & a, const double & b, const double &
```

which returns the root(s) of the quadratic equation $ax^2 + bx + c = 0$. Make the program abort with a suitable error message if the discriminant is negative.

Test your function with a main() program.

Exercice 2

Let us define **typedef double** AD[3]; *//an alias to double[3]*

- write a function with the following prototype

  ```
  double normRow(AD* v, int const &j)
  ```

  and which computes the 2-norm of the element in the j^{th} row of the array v.

- write a function with the following prototype

  ```
  double maxRow(double v[][3], int const &j)
  ```

  and which computes the ∞-norm element in the j^{th} of the array v

Test your functions in a main() program.

Exercice 3

You may find in the file data_Gauss_Legendre.cpp the tabulated values of the Gauss-Legendre quadrature rules on the reference segment $[-1, 1]$. The quadrature rule associated with the array Quad1D_2[4] has 2 points (and two associated weights) and is exact for polynomials of degree < 4. Same rule applies to the other arrays. The data is tabulated up to the rule with 10 nodes, which is exact for polynomials of degree < 20.

1. write a program which reads the data from this file,
2. check the exactness (up to machine accuracy) of these rules through the computations of several integrals of polynomials on the segment $[-1, 1]$
3. compute approximations of the integrals of the function $x \mapsto \sin(x)$ on $[-1, 1]$ with increasing order rules and compare the results.