C++ - XLiFE++

Nicolas KIELBASIEWICZ*, Yvon LAFRANCHE**, Eric LUNÉVILLE*

ENSTA - Paristech*
IRMAR**

Roscoff - April 16, 2018

What does XLiFE++ look like ? What it is ? What it is not ?

1. Choice of the user interface:

Specific interpreter

- conceptor needs to define a language (syntax, keywords, etc.) and build the interpreter,
- one executable file created once during installation of the software,
- user learns this language and creates a script file to solve a problem.

What does XLiFE++ look like ? What it is ? What it is not ?

1. Choice of the user interface:

Specific interpreter

- conceptor needs to define a language (syntax, keywords, etc.) and build the interpreter,
- one executable file created once during installation of the software,
- user learns this language and creates a script file to solve a problem.

vs User's own program

- conceptor uses an existing programming language,
- user should use this programming language to create a main program corresponding to the problem to solve,
- one executable file created for each problem to solve,
- "open" solution: allows user's own contribution and interaction with other softwares under control of the user.

What does XLiFE++ look like ? What it is ? What it is not ?

1. Choice of the user interface:

Specific interpreter

- conceptor needs to define a language (syntax, keywords, etc.) and build the interpreter,
- one executable file created once during installation of the software,
- user learns this language and creates a script file to solve a problem.

vs User's own program

- conceptor uses an existing programming language,
- user should use this programming language to create a main program corresponding to the problem to solve,
- one executable file created for each problem to solve,
- "open" solution: allows user's own contribution and interaction with other softwares under control of the user.

XLiFE++ follow the second policy, but propose a set of objects designed to simplify user's task.

2. Problem oriented vs “set of tools”

XLiFE++ is a library, used for research and teaching: flexibility is thus a fundamental principle

2. Problem oriented vs “set of tools”

XLiFE++ is a library, used for research and teaching: flexibility is thus a fundamental principle

3. Self contained vs using external tools (interoperability)
   - mesh generation
   - linear algebra: matrix decomposition, linear system solver, eigenvalue problem solver

2. Problem oriented vs “set of tools”

XLiFE++ is a library, used for research and teaching: flexibility is thus a fundamental principle

3. Self contained vs using external tools (interoperability)
   - mesh generation
   - linear algebra: matrix decomposition, linear system solver, eigenvalue problem solver

XLiFE++ adopts both points of view:
   - mesh generation: internal tool for simple geometries, Gmsh for other geometries, and is able to read several standard mesh formats
   - linear algebra: internal tools have been developped, as long as interfaces to specialized libraries for performance purpose (UmfPack, Arpack, OpenMP)

2. Problem oriented vs “set of tools”

XLiFE++ is a library, used for research and teaching: flexibility is thus a fundamental principle

3. Self contained vs using external tools (interoperability)

- mesh generation
- linear algebra: matrix decomposition, linear system solver, eigenvalue problem solver

XLiFE++ adopts both points of view:

- mesh generation: internal tool for simple geometries, Gmsh for other geometries, and is able to read several standard mesh formats
- linear algebra: internal tools have been developped, as long as interfaces to specialized libraries for performance purpose (UmfPack, Arpack, OpenMP)

$\rightarrow$ XLiFE++ is a standalone software, opened to specialized tools

- post-processing: targeted free softwares are Paraview, Gmsh, Octave ($\sim$ Matlab)

For the presentation, we consider the following problem:

$$\textit{Given } f_\Omega \in L^2(\Omega), \textit{find } u \in H^1(\Omega) \textit{ such that}$$

$$\begin{cases} -\Delta u + u & = f_\Omega \quad \text{in } \Omega\,, \\ \dfrac{\partial u}{\partial \nu} & = 0 \quad \text{on } \partial\Omega = \Gamma_N\,, \quad (\textit{Neumann condition}) \end{cases}$$

whose variational formulation is:

$$\textit{Find } u \in V = H^1(\Omega) \;\; \textit{such that} \;\; a(u,v) = f(v)\,,\ \forall v \in V,$$

with

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u(x) \cdot \nabla v(x)\, dx + \int_\Omega u(x) v(x)\, dx\,, \\ f(v) &= \int_\Omega f_\Omega(x) v(x)\, dx\,. \end{aligned}$$

The definition of the problem in XLiFE++ is based on its variational formulation.

Mathematical objects involved (Ω, V, u, v, a, f) will correspond to objects in C++.

The C++ language has the following characteristics:

- compiled language (sources → (separate) compilation → binaries → linkage → executable)
- superset of C language
- one or several files (only one of them should contain the `main` function)
- strongly typed: basic types + allows the definition of user-defined types

The C++ language has the following characteristics:

- compiled language (sources → (separate) compilation → binaries → linkage → executable)
- superset of C language
- one or several files (only one of them should contain the `main` function)
- strongly typed: basic types + allows the definition of user-defined types

Definitions and syntactic elements:

- hierarchy: statement ∈ group ∈ function ∈ file,
- a statement (or instruction) is terminated by a semicolon **;**
- a group is delimited by curly braces **{ }**
- a user-defined type takes the form of a `class` in C++; a variable of this type is called an object
- a variable (of a basic type) or an object should be declared once (name + type), and exists only inside its scope (the group where it is declared for a local variable)

```
#include <string>
#include <vector>
using namespace std;

int main() {
  int nb; // no initialization
  float values[] = {1.5, 2.5};
  string today("monday");
  vector<float> V{1.5, 2.5};
}
```

Nota: in the following, the word “variable” is often used in a generic sense and may refer to an object as well

A function has the following characteristics:

- should be declared before use through its prototype (or signature):

```
type function_name(type1 arg1, type2 arg2, etc.  );
```

- an argument (or parameter) may be passed by value or by reference:
  - by value: a copy of the original variable (or object) is passed to the function
    this variable is protected against any modification by the function (allows recursion)
  - by reference: the function has direct access to the original variable (no copy)

A function has the following characteristics:

- should be declared before use through its prototype (or signature):

```
type function_name(type1 arg1, type2 arg2, etc.  );
```

- an argument (or parameter) may be passed by value or by reference:
  - by value: a copy of the original variable (or object) is passed to the function
    this variable is protected against any modification by the function (allows recursion)
  - by reference: the function has direct access to the original variable (no copy)

Syntax for a reference:

→ rv is not a variable: a reference is an “alias”
→ v and rv correspond to the same address in memory

```
type v;          // v variable of any type
type& rv = v;    // rv reference to v
```

A function has the following characteristics:

- should be declared before use through its prototype (or signature):

```
type function_name(type1 arg1, type2 arg2, etc.  );
```

- an argument (or parameter) may be passed by value or by reference:
  - by value: a copy of the original variable (or object) is passed to the function
    this variable is protected against any modification by the function (allows recursion)
  - by reference: the function has direct access to the original variable (no copy)

Syntax for a reference:

→ rv is not a variable: a reference is an “alias”
→ v and rv correspond to the same address in memory

```
type v;          // v variable of any type
type& rv = v;    // rv reference to v
```

Example with type = int:

```
void f(int v, int& r, const int& cr);

int main() {
  int i=1, j=2, k=3;
  f(i, j, k);
}
```

- This prototype indicates that i is passed by value, j and k by reference, but the function f cannot modify the original int corresponding to cr, which is k
- The references r and cr are initialized when the function f is called

Overloading:

A function may be overloaded: same name, but different arguments $\Longrightarrow$ different prototypes, so no ambiguity

```
void f(int v, int& r, const int& cr);
void f(int v, int& r);
void f(float v, double d = 1.2);
                // d has a default value
```

Overloading:

A function may be overloaded: same name, but different arguments $\Longrightarrow$ different prototypes, so no ambiguity

```
void f(int v, int& r, const int& cr);
void f(int v, int& r);
void f(float v, double d = 1.2);
                // d has a default value
```

Class = extension of struct (from C)

- A class is made of:
  - (not public) data members
  - member functions (or methods)
- constructor = function designed to build the object and initialize its data members
- generally several (overloaded) constructors
- Operator . (dot) to access a member:
  `object.member_name`

```
class ComplexNumber {
public:
  ComplexNumber() :re(1),im(0){} // constructor
  ComplexNumber(float x, float y)
                  :re(x),im(y){} // constructor
  float real(){return re;} // access functions
  float imag(){return im;}
private:
  float re, im; // data members
};

int main() {
  ComplexNumber za;
  float rp = za.real(); // rp equals 1
  ComplexNumber zb(1.5,2.5);
  float ip = zb.imag(); // ip equals 2.5
}
```

Types (aliases) to use in XLiFE++:

```
Dimen   minD = 0 ; // small positive integer
Number  minN = 0 ; //       positive integer
Int     nbI  = -1; //          signed integer

Real    twoPi = 2*pi_;// predefined constant
Complex z(1, 2);    // same precision as Real
String  day("monday");
Point   P(5.5,6.6);// points in 1D, 2D, 3D
```

```
// Vector<T> where T is any type
Vector<Real> V(10); // 10 values of type Real
Real elt2 = V(2);   // index > 0 (= V[1])

// Matrix<T> where T is any type
Matrix<Real> M(5,4); // matrix 5x4 of type Real
M(2,3) = 2.3;        // indices > 0
Real elt2_3 = M(2,3);
Vector<Real> C2 = M.column(2);//2nd column of M
Vector<Real> R1 = M.row(1);   //first row  of M
```

Types (aliases) to use in XLiFE++:

```
Dimen   minD = 0 ; // small positive integer
Number  minN = 0 ; //       positive integer
Int     nbI  = -1; //         signed integer

Real    twoPi = 2*pi_;// predefined constant
Complex z(1, 2);    // same precision as Real
String  day("monday");
Point   P(5.5,6.6);// points in 1D, 2D, 3D
```

```
// Vector<T> where T is any type
Vector<Real> V(10); // 10 values of type Real
Real elt2 = V(2);   // index > 0 (= V[1])

// Matrix<T> where T is any type
Matrix<Real> M(5,4); // matrix 5x4 of type Real
M(2,3) = 2.3;        // indices > 0
Real elt2_3 = M(2,3);
Vector<Real> C2 = M.column(2);//2nd column of M
Vector<Real> R1 = M.row(1);   //first row  of M
```

`Numbers, Ints, Reals, Complexes, Strings`

are aliases for

`Vector<Number>, Vector<Int>, Vector<Real>, Vector<Complex>, Vector<String>`

Other aliases for `Real` or `Complex` type:

`Reals`	$\Longleftrightarrow$	`RealVector`
`Complexes`	$\Longleftrightarrow$	`ComplexVector`
`Vector<Reals>`	$\Longleftrightarrow$	`RealVectors`
`Vector<Complexes>`	$\Longleftrightarrow$	`ComplexVectors`
`Matrix<Real>`	$\Longleftrightarrow$	`RealMatrix`
`Matrix<Complex>`	$\Longleftrightarrow$	`ComplexMatrix`
`Matrix<RealMatrix>`	$\Longleftrightarrow$	`RealMatrices`
`Matrix<ComplexMatrix>`	$\Longleftrightarrow$	`ComplexMatrices`

A variety of helpers are available:

- self explained constant names:
  `pi_, overpi_, over3_, sqrtOf2_, i_ ...`
- keywords (enumerations):
  - degree: `_P1, _P2,..., _Q1, _Q2,...`
  - shape: `_triangle, _quadrangle, _tetrahedron ...`
  - FE type: `_Lagrange, _CrouzeixRaviart, _Nedelec ...`
  - output format: `_vtk, _msh, _matlab ...`
  - `...`
- `key = value` system to make some function calls easier

A variety of helpers are available:

- self explained constant names:
  `pi_, overpi_, over3_, sqrtOf2_, i_ ...`
- keywords (enumerations):
  - degree: `_P1, _P2,..., _Q1, _Q2,...`
  - shape: `_triangle, _quadrangle, _tetrahedron ...`
  - FE type: `_Lagrange, _CrouzeixRaviart, _Nedelec ...`
  - output format: `_vtk, _msh, _matlab ...`
  - `...`
- `key = value` system to make some function calls easier

Typical usage:

```
Complex z = 1 + 2*i_; // equivalent to Complex z(1, 2);
Real k=2.345; Complex p = exp(-i_*k);

// Vector of string elements
Strings sideNames("y=ymin", "x=xmax", "y=ymax", "x=xmin");

// Rectangle [0,Pi]x[0,sqrt(3)] with 5 points on each side
Rectangle rect (_xmin=0, _xmax=pi_ ,_ymin=0, _ymax=sqrtOf3_, _nnodes=5,
      _side_names=sideNames);

// Mesh of the previous rectangle with quadrangles
Mesh mesh2d(rect, _quadrangle, 1, _structured);

Space V(Omega, _P2); // Lagrange P2 finite elements will be used
```

The linear form $f(v) = \int_\Omega f_\Omega(x) v(x)\, dx$ is translated in the program as:

```
LinearForm fv = intg(Omega, fOmega*v);
```

The linear form $f(v) = \int_\Omega f_\Omega(x) v(x)\, dx$ is translated in the program as:

```
LinearForm fv = intg(Omega, fOmega*v);
```

How to define such a function (non constant coefficients) to be used in an integral ?

The linear form $f(v) = \int_\Omega f_\Omega(x) v(x)\, dx$ is translated in the program as:

```
LinearForm fv = intg(Omega, fOmega*v);
```

How to define such a function (non constant coefficients) to be used in an integral ?

The user has to define a function whose prototype is imposed:

- the first argument is the point on which the function will be evaluated (it can be a `Vector<Point>` for vector case);
- the second (optional) argument contains additional parameters:

```
Real fOmega(const Point& p, Parameters& pa = defaultParameters) {
  Real x=p(1), y=p(2), a=1., b=1;
  return x*(a-x)*y*(b-y);
}
int main() {
...
}
```

The linear form $f(v) = \int_\Omega f_\Omega(x) v(x)\, dx$ is translated in the program as:

```
LinearForm fv = intg(Omega, fOmega*v);
```

How to define such a function (non constant coefficients) to be used in an integral ?

The user has to define a function whose prototype is imposed:

- the first argument is the point on which the function will be evaluated (it can be a `Vector<Point>` for vector case);
- the second (optional) argument contains additional parameters:

```
Real fOmega(const Point& p, Parameters& pa = defaultParameters) {
  Real x=p(1), y=p(2), a=1., b=1;
  return x*(a-x)*y*(b-y);
}
int main() {
...
}
```

Return values can be Real/Complex, Reals/Complexes, RealMatrices/ComplexMatrices

The linear form $f(v) = \int_\Omega f_\Omega(x) v(x)\, dx$ is translated in the program as:

```
LinearForm fv = intg(Omega, fOmega*v);
```

How to define such a function (non constant coefficients) to be used in an integral ?

The user has to define a function whose prototype is imposed:

- the first argument is the point on which the function will be evaluated (it can be a `Vector<Point>` for vector case);
- the second (optional) argument contains additional parameters:

```
Real fOmega(const Point& p, Parameters& pa = defaultParameters) {
  Real x=p(1), y=p(2), a=1., b=1;
  return x*(a-x)*y*(b-y);
}
int main() {
...
}
```

Return values can be Real/Complex, Reals/Complexes, RealMatrices/ComplexMatrices

Here, *a* and *b* are constants defined inside the function.
To make them vary according to a particular context, two solutions:

1. make them external variables (not always suitable, error prone, initialization and use should be in the same scope)
2. use the second argument designed for that purpose

1. Both parameters are available through the argument *pa*
   → provided that each of them have been assigned a label:

```
Real fOmega(const Point& p, Parameters& pa) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

1. Both parameters are available through the argument *pa*
   → provided that each of them have been assigned a label:

```
Real fOmega(const Point& p, Parameters& pa) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. Define a `Parameters` object in the main function containing both values:

```
int main() {
  ...
  Parameters params("a",2.);     // each element is a couple (label, value)
  params << Parameter("b",3.); // insert the second element
```

1. Both parameters are available through the argument *pa*
   → provided that each of them have been assigned a label:

```
Real fOmega(const Point& p, Parameters& pa) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. Define a `Parameters` object in the main function containing both values:

```
int main() {
  ...
  Parameters params("a",2.);    // each element is a couple (label, value)
  params << Parameter("b",3.); // insert the second element
```

3. Associate the parameters with the function tanks to a `Function` object:

```
  Function myf(fOmega, params);
```

1. Both parameters are available through the argument *pa*
   → provided that each of them have been assigned a label:

```
Real fOmega(const Point& p, Parameters& pa) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. Define a `Parameters` object in the main function containing both values:

```
int main() {
  ...
  Parameters params("a",2.);    // each element is a couple (label, value)
  params << Parameter("b",3.); // insert the second element
```

3. Associate the parameters with the function tanks to a `Function` object:

```
  Function myf(fOmega, params);
```

4. Use *myf* in the linear form definition:
   ⟹ this ensures that the parameters will be taken into account during the computation

```
  LinearForm fv = intg(Omega, myf*v); // instead of intg(Omega, fOmega*v)
```

1. Both parameters are available through the argument *pa*
   → provided that each of them have been assigned a label:

```
Real fOmega(const Point& p, Parameters& pa) {
  Real x=p(1), y=p(2), a=pa("a"), b=pa("b");
  return x*(a-x)*y*(b-y);
}
```

2. Define a `Parameters` object in the main function containing both values:

```
int main() {
  ...
  Parameters params("a",2.);    // each element is a couple (label, value)
  params << Parameter("b",3.); // insert the second element
```

3. Associate the parameters with the function tanks to a `Function` object:

```
  Function myf(fOmega, params);
```

4. Use *myf* in the linear form definition:
   ⟹ this ensures that the parameters will be taken into account during the computation

```
  LinearForm fv = intg(Omega, myf*v); // instead of intg(Omega, fOmega*v)
```

A `Parameter` object can handle any type the user needs in this context

How to get a normal vector inside a function ?

1. Use the member function **getVector**:

```
Real f(const Point& P, Parameters& pa)
{
  Reals vn = pa.getVector(_n);
  return vn(1);
}
```

How to get a normal vector inside a function ?

1. Use the member function **getVector**:

```
Real f(const Point& P, Parameters& pa)
{
  Reals vn = pa.getVector(_n);
  return vn(1);
}
```

2. Define a `Parameter` whose label is "_n" using the **associateVector** function:

```
Parameters pars;
pars.associateVector(_n);
Function myf(f,pars);
```

How to get a normal vector inside a function ?

1. Use the member function **getVector**:

```
Real f(const Point& P, Parameters& pa)
{
  Reals vn = pa.getVector(_n);
  return vn(1);
}
```

2. Define a `Parameter` whose label is "_n" using the **associateVector** function:

```
Parameters pars;
pars.associateVector(_n);
Function myf(f,pars);
```

The `Parameters` will contain the normal vector on condition that the computation engine detects there is one `Parameter` labeled "_n" passed to function. In this case only, the true normal vector is computed, stored “inside” the `Parameter` "_n" and can then be retrieved by the **getVector** function.

Recall the variational formulation of the considered problem:

$$\textit{Find } u \in V = H^1(\Omega) \textit{ such that } a(u,v) = f(v)\,,\ \forall v \in V,$$

with

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u(x) \cdot \nabla v(x)\, dx + \int_\Omega u(x)v(x)\, dx\,, \\ f(v) &= \int_\Omega f_\Omega(x)v(x)\, dx\,. \end{aligned}$$

Let Ω be the unit square, and choose $f_\Omega(x,y) = cos(\pi x) * cos(\pi y)$.

This problem can be translated in XLiFE++'s framework as shown in next slide.

Practically:

Record this file in a new directory, open a terminal, change to this directory and type in the commands:

`xlifepp.sh ; make` (or `xlmake` without `cmake`)

Then launch the executable file just created.

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real coscos(const Point& P, Parameters& pa = defaultParameters) {
  Real x=P(1), y=P(2);
  return cos(pi_ * x) * cos(pi_ * y);
}
int main() {
  init(); // mandatory initialization of xlifepp
   // Domain = unit square -> (exact) approximation by a mesh
  Square sq(_origin=Point(0.,0.), _length=1, _nnodes=11, _domain_name="Omega");
  Mesh mesh2d(sq, _triangle, 1, _structured);
   // Get a handle (Omega) to the domain from the mesh via its name
  Domain Omega = mesh2d.domain("Omega");
   // Mathematical objects
  Space V(Omega, P1, "V");  // Lagrange P1
  Unknown u(V, "u");
  TestFunction v(u, "v");
  BilinearForm auv = intg(Omega, grad(u) | grad(v)) + intg(Omega, u * v);
  LinearForm fv = intg(Omega, coscos * v);
   // Linear algebra
  TermMatrix A(auv, "a(u,v)");
  TermVector B(fv, "f(v)");
  TermVector U = directSolve(A, B);// Matlab equivalent: U = A\B
   // Export the result
  saveToFile("U", U, vtu);    // produces the file U_Omega.vtu
  saveToFile("U", U, msh);    // produces the file U_Omega.msh
  saveToFile("U", U, matlab);// produces the file U_Omega.m
}
```