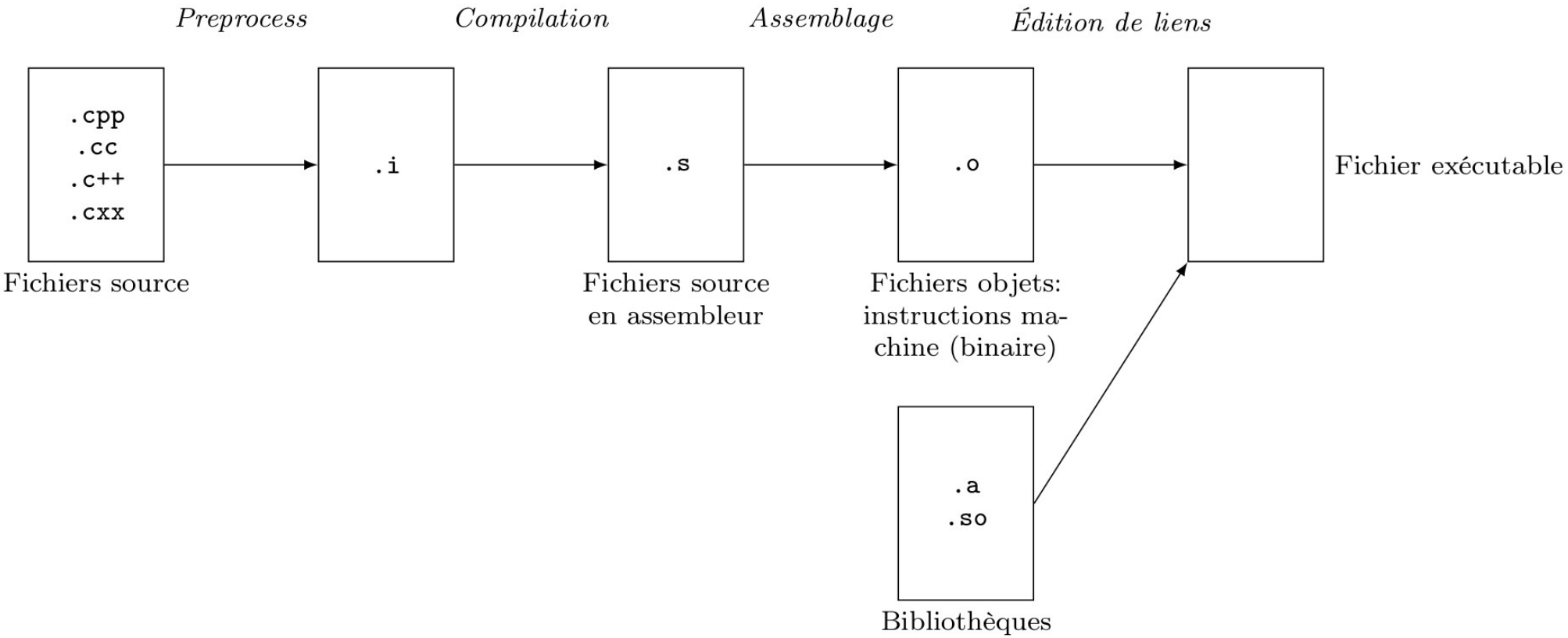
Structurer le code : compilation séparée, namespaces, GNU make

Structure du code

- Code de calcul scientifique : du simple script (centaines de lignes) à des solveurs génériques très complexes (centaines de milliers de lignes)
- Si tout est dans un seul fichier :
  - Difficultés de maintenance : il faut retrouver à quel endroit du fichier intervenir…
  - Pas d'organisation thématique ou alors nécessite des règles personnelles : code non-portable
  - Difficile de partager des morceaux de code.
  - Il faut TOUT recompiler à chaque fois (30 à 60 minutes de compilation pour certains codes)

Compilation séparée

› Étapes de compilation



› En règle générale, à chaque fichier source correspond un fichier objet.

Structure du code

› Placer les éléments ayant une thématique commune dans un fichier source séparé.

pointsNorm.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
typedef unsigned int uint;

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

  if (argc != 2) {
    std::cerr <<"Usage: " << argv[0] << " <inputFile>" << std::endl;
    return 1;
  }
  std::string inFileName(argv[1]);
  std::ifstream inFile(inFileName.c_str());
  if (not inFile.good()) {
    std::cerr << "Unable to open file " << inFileName << std::endl;
    abort();
  }

  uint size = 0;
  inFile >> size;
  std::vector<point> points(size);
  for (uint i=0u; i<points.size(); i++) {
    inFile >> points[i].x >> points[i].y;
  }

  for (const point& p: points)
    std::cout << norm2(p) << std::endl;

  return 0;

}
```

› Pb : **`main`** ne sait pas ce qu'est **`point`** ou **`norm2`**.

geometry.cpp

```cpp
#include <cmath>
// Type for a 2D coordinate
struct point {
  double x;
  double y;
};

// Euclidian norm of a 2D coordinate
double norm2(const point& M) {
  return sqrt(pow(M.x,2)+pow(M.y,2));
}
```

Structure du code

- Rajouter le prototype des fonctions avant **`main`**.

pointsNorm.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
typedef unsigned int uint;

struct point {
  double x,y;
}
double norm2(const point& M);

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

  if (argc != 2) {
    std::cerr <<"Usage: " << argv[0] << " <inputFile>" << std::endl;
    return 1;
  }
  std::string inFileName(argv[1]);
  std::ifstream inFile(inFileName.c_str());
  if (not inFile.good()) {
    std::cerr << "Unable to open file " << inFileName << std::endl;
    abort();
  }

  uint size = 0;
  inFile >> size;
  std::vector<point> points(size);
  for (uint i=0u; i<points.size(); i++) {
    inFile >> points[i].x >> points[i].y;
  }

  for (const point& p: points)
    std::cout << norm2(p) << std::endl;

  return 0;

}
```

- Pb : pénible et assez inutile si on a des centaines de fonctions...

geometry.cpp

```cpp
#include <cmath>
// Type for a 2D coordinate
struct point {
  double x;
  double y;
};

// Euclidian norm of a 2D coordinate
double norm2(const point& M) {
  return sqrt(pow(M.x,2)+pow(M.y,2));
}
```

Structure du code

› Créer un fichier en-tête (header : .h, .hpp , .h++)

pointsNorm.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "geometry.hpp"

typedef unsigned int uint;

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

  if (argc != 2) {
    std::cerr <<"Usage: " << argv[0] << " <inputFile>"
    return 1;
  }
  std::string inFileName(argv[1]);
  std::ifstream inFile(inFileName.c_str());
  if (not inFile.good()) {
    std::cerr << "Unable to open file " << inFileName << std::endl;
    abort();
  }

  uint size = 0;
  inFile >> size;
  std::vector<point> points(size);
  for (uint i=0u; i<points.size(); i++) {
    inFile >> points[i].x >> points[i].y;
  }

  for (const point& p: points)
    std::cout << norm2(p) << std::endl;

  return 0;

}
```

geometry.hpp

```cpp
#ifndef _GEOMETRY_HPP_
#define _GEOMETRY_HPP_

#include <cmath>
// Type for a 2D coordinate
struct point {
  double x;
  double y;
};

// Euclidian norm of a 2D coordinate
double norm2(const point& M);

#endif // _GEOMETRY_HPP_
```

geometry.cpp

```cpp
#include "geometry.hpp"

double norm2(const point& M) {
  return sqrt(pow(M.x,2)+pow(M.y,2));
}
```

Structure du code

- Dans un fichier d'en-tête, on met :
  - Des prototypes de fonctions
  - Des définitions de types structurés et de classes (cf partie POO), des renommages de types.
  - Des déclarations de variables globales (il faut alors utiliser le mot clé **`extern`**)
  - Des templates (semetres suivant ! (?))
- On ne met généralement pas :
  - Des définitions de variables globales (sans le mot clé extern) ou de macros (#define), sauf s'il s'agit d'un fichier qu'on sait inclus par **tous** les fichiers sources.
  - Des définitions de fonctions (avec le corps de la fonction)
  - Des définitions de fonctions membres définies en dehors de leur classe (partie POO)
- Un élément ne peut être déclaré et défini qu'une seule fois.
- Attention aux dépendances circulaires.

Structure du code

- Bonnes pratiques :
  - Séparer le **`main`** du reste. Ne mettre dans le même fichier que ce qui ne sera pas réutilisé ailleurs.
  - Un seul fichier par type de fonctionnalités apparentées.
  - Noms explicites. Pas de **`machin.hpp`**.
  - Arborescence. Exemple :

```
\_o< ls
Authors  build  CepsConfig.h.in  cmake  CMakeLists.txt  contrib  data  doc
     KnownBugs  NotForRelease  README.md  src
\_o< ls src
applications  cardiac  CMakeLists.txt  common  geometry  linearAlgebra  ode  pde
\_o< ls src/geometry/
AbstractElement.cpp  JunctionElement.cpp  partitioning
AbstractElement.hpp  JunctionElement.hpp  readers
CMakeLists.txt       Mesh.cpp             Simplex.cpp
examples             Mesh.hpp             Simplex.hpp
Geometry.cpp         Node.cpp             tests
Geometry.hpp         Node.hpp             writers
```

Espaces de nommage (namespace)

- Créer des régions de codes pour limiter la portée des noms.
- Un namespace peut contenir tous les éléments du langage C++.
- Namespace anonyme : portée des variables limitée au fichier (remplace les variables déclarées **`static`**)

```
namespace {
  int a,b;
  void func1() { // ...
  };
}
```

- Namespaces non-anonymes : permettent de différencier des éléments ayant le même nom

```
namespace linalgLib1 {
  struct Matrix { ...} ;
}
namespace linalgLib2 {
  struct Matrix {... } ;
}

linalgLib1::Matrix A;
linalgLib2::Matrix B;
```

- « Raccourcis »

```
using namespace linalgLib1; // Use ALL names from namespace
                            // (potentially dangerous)
using std::cout; // Import only this name from the namespace to be able to
                 // use "cout" only
```

Compilation séparée

- Une belle organisation mais…

```
g++ -std=c++11 -Wall -g -std=c++11 file1.cpp -c file1.o
g++ -std=c++11 -Wall -g -std=c++11 file2.cpp -c file2.o
...
g++ -std=c++11 -Wall -g -std=c++11 file537.cpp -c file537.o
g++ file1.o file2.o ... file537.o -o finally
```

- On veut automatiser la compilation !
- GNU **`make`** : donner les instructions de compilation via un fichier, appelé **`Makefile`**. Ne compiler que ce qui est nécessaire : ce qui a été mis à jour.

Compilation séparée

› Un premier Makefile pour un seul fichier source

```
myExecutable: mySource.cpp
	g++ -Wall -g -std=c++11 myProgram.cc -o myExe
```

› Pour compiler, taper

```
$> make
```

› Cible : fichier à créer, qui peut dépendre d'autres fichiers et cibles, dont on donne les règles pour le créer.

› Syntaxe des cibles, attentions aux tabulations devant les instructions !

```
targetName : dependency1 dependency2 dependency3 ...
	rule1
	rule2
	rule3
	...
```

› Pour compiler une cible particulière (sinon, première cible du fichier)

```
$> make targetName
```

Compilation séparée

› Plusieurs fichiers :

Makefile

```
myExe : pointsNorm.o geometry.o
        g++ pointsNorm.o geometry.o -o myExe

pointsNorm.o : pointsNorm.cpp geometry.hpp
        g++ -Wall -g -std=c++11 -c pointsNorm.cpp

geometry.o : geometry.cpp geometry.hpp
        g++ -Wall -g -std=c++11 -c geometry.cc

clean :
        rm -rf *.o *~
```

Compilation séparée

- Variables :
  - Prédéfinies dans le langage make :
    - $@ : cible courante
    - $^ : les dépendances de la cible
    - $< : la première dépendance
    - $? : les dépendances qui ne sont plus à jour
    - Plein d'autres : doc
  - Définies par l'utilisateur

Makefile

```
CXX = g++
CXXFLAGS = -Wall -g -std=c++11

myExe : pointsNorm.o geometry.o
        $(CXX) $^ -o $@

pointsNorm.o : pointsNorm.cpp geometry.hpp
        $(CXX) $(CXXFLAGS) -c $<

geometry.o : geometry.cpp geometry.hpp
        $(CXX) $(CXXFLAGS) -c $<

clean :
        rm -rf *.o *~
```

Compilation séparée

› Règles par type de fichier :

Makefile

```
CXX = g++
CXXFLAGS = -Wall -g -std=c++11
OBJS = pointsNorm.o geometry.o

myExe : $(OBJS)
        $(CXX) $^ -o $@

%.o : %.cc
        $(CXX) $(CXXFLAGS) -c $<
%.o : %.cpp
        $(CXX) $(CXXFLAGS) -c $<

clean :
        rm -rf *.o *~
```

› Problème...

Compilation séparée

› Headers : programme ou fonctionnalité du compilateur pour faire la liste des en-têtes dont dépendent un fichier source.

Makefile

```
CXX = g++
CXXFLAGS = -Wall -g -std=c++11
SRCS = pointsNorm.cpp geometry.cpp
# Automatically replaces .cpp by .o in file names
OBJS = $(SRCS:%.cpp=%.o)
DEPS = $(SRCS:%.cpp=%.d)

all : myExe

myExe : $(OBJS)
        $(CXX) $^ -o $@

# The dependency on header files is described by files in $(DEPS)
-include $(DEPS)

# The -MMD option makes gcc parse the source file
# and write the dependency as a makefile target in a .d file
%.o : %.cpp
        $(CXX) $(CXXFLAGS) -MMD -c $< -o $@

clean :
        rm -rf *.o *.d *~
```