Rappels et compléments : fonctionnalités de base du C++

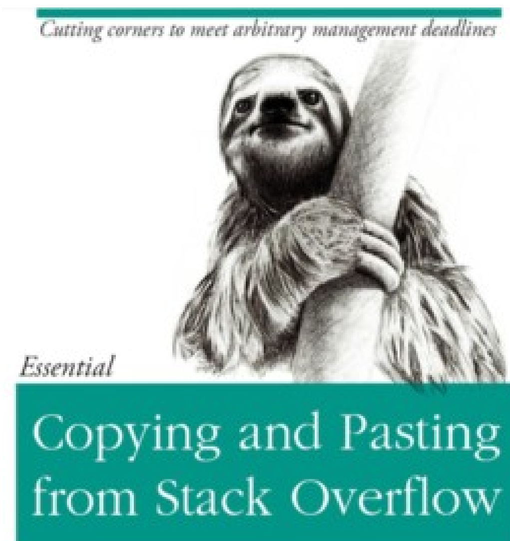
Généralités

- Écrire le code n'est qu'une étape :
  - Avant : conception, algorithmique (sur papier, au tableau, en réunion…)
  - Pendant : documentation, tests unitaires, débogage
  - Après : dépôt, diffusion, mises à jour, etc.
- On ne part qu'assez rarement de zéro ! Savoir comprendre un code est important.
- Les concepts que l'on verra seront parfois applicables à beaucoup de langages différents.

Généralités

- À propos du C++.
  - Créé dans les 70s. « C with classes », degré d'abstraction supplémentaire, poursuite de l'éloignement du langage machine.
  - Standardisé en 1998
  - En évolution : nouveaux standards c++11, c++14, c++17 et bientôt c++20.
- Avantages :
  - Très répandu
  - Efficace
  - Portable
  - Multi-paradigmes

- En difficulté ? Grande communauté de développeurs
  - cppreference.com : référence officielle, complète, assez absconse
  - cplusplus.com : référence également, un peu plus accessible, meilleure navigation
  - Stack Overflow : le classique. Forum d'entraide. Bien lire et comprendre les réponses avant de reproduire !



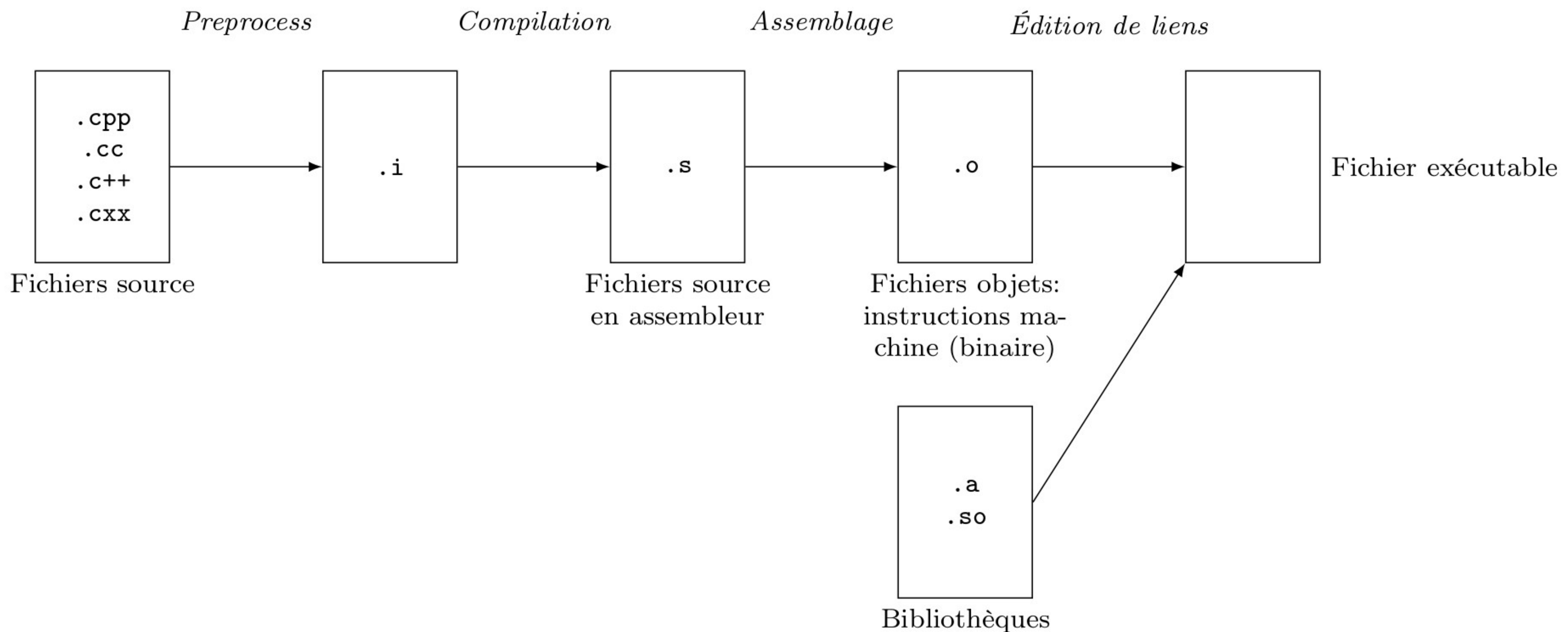
Programme d’illustration :

```
14 // "Loads" library headers for the declaration of cout, endl, fstream, sqrt, etc
15 #include <iostream>
16 #include <fstream>
17 #include <cmath>
18 #include <string>
19 #include <vector>
20
21 // we could "import" the whole namespace with this:
22 // using namespace std;
23
24 // New name for this type to avoid writing the long name
25 typedef unsigned int uint;
26
27 // Type for a 2D coordinate
28 struct point {
29   double x;
30   double y;
31 };
32
33 // Euclidian norm of a 2D coordinate
34 double norm2(const point M) {
35   return sqrt(pow(M.x,2)+pow(M.y,2));
36 }
37
```

C++ « core » , fonctionnalités standard basiques

```
38 int main(int argc, char* argv[]) {
39
40   // Read the name of the input file given in command line
41   // argc is the number of words in command line
42   // argv are the words themselves
43   if (argc != 2) {
44     std::cerr <<"Usage: " << argv[0] << " <inputFile>" << std::endl;
45     return 1;
46   }
47   std::string inFileName(argv[1]);
48
49   // Open input file. Tell if it fails and abort
50   std::ifstream inFile(inFileName.c_str());
51   if (not inFile.good()) {
52     std::cerr << "Unable to open file " << inFileName << std::endl;
53     abort();
54   }
55
56   // Read the size of the array to be read
57   uint size = 0;
58   inFile >> size;
59   std::vector<point> points(size);
60   for (uint i=0u; i<points.size(); i++) {
61     inFile >> points[i].x >> points[i].y;
62   }
63
64   // C++11 syntax for loops on STL elements
65   for (const point& p: points)
66     std::cout << norm2(p) << std::endl;
67
68   // All good
69   return 0;
70
71 }
```

› Un langage compilé



C++ « core » , fonctionnalités standard basiques

- Un langage compilé
  - Compilateurs : GNU gcc, g++ (~ gcc avec inclusion automatique des bibliothèques standard), compilateurs intel, compilateurs avec MPI (pour les calculs parallèles), etc.
  - Créer les fichiers objets puis lier les fichiers

```
g++ myFile.cpp -c myFile.o
g++ myFile2.cpp -c myFile2.o
g++ myFile1.o myFile2.o -o myExecutable
```

  - Compiler directement

```
g++ myFile.cpp myFile2.cpp  -o myExe
```

  - Options de compilation :
    - -std=c++11 : inclure les fonctionnalités du standard c++11 (aussi pour 14, 17)
    - -Wall : all warnings
    - -g : symboles de débogage
    - -On : optimisation de degré n (souvent n=3)
    - Bien bien d'autres… : man gcc

C++ « core » , fonctionnalités standard basiques

› Directives de preprocessing :

```
14 // "Loads" library headers for the declaration of cout, endl, fstream, sqrt, etc
15 #include <iostream>
16 #include <fstream>
17 #include <cmath>
18 #include <string>
19 #include <vector>
```

› Lues avant même le que le code ne soit traité par le compilateur. Ici « import » de bibliothèques.

› Aussi : macros **`#define`** (structures de contrôle et/ou remplacement litteral)

```
#define NDIM 2
#define ND two

std::vector<double> x(NDIM,0); // replaced by x(2,0); at preprocessing
std::vector<double> y(ND,0); // replaced by y(two,0);
```

```
#ifdef FOO
  // Instructions here will be compiled if
  // FOO is previously defined using #define
#else
  // This will be discarded if FOO is not previously defined
#endif

// Also exists: #ifndef
```

C++ « core » , fonctionnalités standard basiques

- Langage statiquement typé :
  - Les variables ont un type qui ne change pas au cours de l'exécution
  - Toute variable doit être déclarée avant son utilisation.

```
int a;
double x;
a = 10;
unsigned int i = 1;
```

  - Type structuré pour regrouper des variables de types de base

```
struct point {
  double x;
  double y;
}
```

  - Renommer des types qui sont compliqués :

```
typedef unsigned int uint;
uint i = 10u;

typedef VtkSmartPointer<VtkXMLPUnstructuredGridReader> vtkReader;

using evolFunc = double (*) (double,double); // C++ 11
```

  - C++11 : type **`auto`** (dangereux quand on débute)

```
auto M = Eigen::Matrix<double,3,3>::Zero();
```

- Structures de contrôle : **`if/else, switch`**

```
if (expr1) {
  // do things if expr1 is true
}
else if (expr2) {
  // if not, do things if expr2 is true
}
else {
  // if not, do this
}
```

  - Tout ce qui n'est pas 0 est assimilable à **`true`**
  - Syntaxe compacte **`if/else`** si les blocs n'ont qu'une affectation de la même variable :

```
res = expr ? resIfTrue : resIfFalse;

// Equiv
// if (expr)
//   res = resIfTrue;
// else
//   res = resIfFalse;

// example:
double a,b;
// ...
double max = a>b ? a : b;
```

- Structures de contrôle : **`if/else, switch`**

  - **`switch`** :

```
switch (var) {
  case 0:
    // do things if var==0
    break;
    // do not forget the "break", otherwise, next case is done even if var!=1
  case 1:
  {
    // You need extra-brackets if new variables are defined:
    int a;
    break;
  }
  case 2:
  case 3:
    // This will be done if var==2 or var==3
    break;
  default:
    // This will be done in all other cases
    std::cerr << "Error: wrong value for var" << std::endl;
}
```

  - **`switch`** réservé aux types assimilables aux entiers (entiers, caractères, pointeurs), pas les réels ou les chaînes de caractères !

Structures de contrôle : boucles

- **`while`** : ne pas oublier de mettre à jour la condition

```
while (condition) {
  // do things as long as condition is true.
  // If condition is false at start, nothing is done.
}

do {
  // do things once then repeat until condition is false
} while (condition);
```

```
for (<init>; <condition>; <increment steps>) {
  // ...
}

// is equivalent to

{
  <init>
  while (<condition>) {
    // ...
    <increment steps>
  }
}
```

- C++11 : pour des boucles sur les conteneurs STL (incl. vector)

```
for (int n : {0, 1, 2, 3, 4, 5})
  std::cout << n << ' ';

std::vector<int> values = {0, 1, 2, 3, 4, 5};
for (int& v : values) {
  v++;
  std::cout << v << ' ';
}
```

C++ « core » , fonctionnalités standard basiques

› Fonctions :

- Partie du code séparée, dédiée à une tâche spécifique, qui sera appelée plusieurs fois.

```
33 // Euclidian norm of a 2D coordinate
34 double norm2(const point M) {
35   return sqrt(pow(M.x,2)+pow(M.y,2));
36 }
```

- Une fonction doit être déclarée (!= définie) avant son utilisation.
  - Déclaration : énoncé du prototype de la fonction

```
<returnType> <funcName> (argType1, argType2 , ...);
```

  - Définition : prototype + instructions

- Plusieurs fonctions peuvent avoir le même nom : résolution avec les arguments
- Arguments par défaut (déclaration)

```
// p-norm of vector, default is euclidian
double normP (std::vector<double>& v, double p=2.);
```

› Fonctions, passage des arguments :

- Les variables passées en argument à l'appel d'une fonction sont **copiées**.

```
void addOne(int a) {
  a = a+1;
}

int main() {
  int a = 2;
  addOne(a);
  std::cout << a << std::endl; // Prints 2!
  return 0;
}
```

› Pointeurs, références :

- Un pointeur est une variable contenant une adresse mémoire.

```
int a = 10;
int* p; // Pointer on an int
int *q; // This syntax can also be found at declaration

p = &a; // Operator & "to designate adress of a"
q = p;  // q also points on p

// Prints an address, then the value of p.
// Note the dereference operator '*' to access the value of pointed variable
std::cout << p << " " << *p << std::endl;
(*q)++;
std::cout << *p << std::endl; // Print previous value+1!

// Pointer on function. Don't forget parentheses
double (*pf) (double); // pf is a pointer on a function that takes a double as
     argument and returns a double
int (*pf2) (double,double);

double func1(double arg1) {
  // ...
}
int func2(double arg1, double arg2) {
  // ...
}
pf  = &func1; // OK
pf2 = &func2; // OK
```

C++ « core » , fonctionnalités standard basiques

- Pointeurs, références :
  - Une référence peut être vue comme un alias d'un autre variable.

```
int a = 1;
int& ra(a); // References must be initialized to a given "target"
```

- Passage d'arguments (suite) : par pointeur, par référence

```
void addOne(int a) {
  a = a+1;
}
void addTwo(int* a) {
  *a = *a+2;
}
void addThree(int& a) {
  a = a+3;
}
void addFour(const int& a) {
  a = a+4; // Compilation error! Cannot modify a.
}

int main() {
  int a = 2;
  addOne(a);
  std::cout << a << std::endl; // Prints 2!
  addTwo(&a);
  std::cout << a << std::endl; // Prints 4.
  addThree(a);
  std::cout << a << std::endl; // Prints 7.
  return 0;
}
```

Portée des variables

- Une variable locale est connue dans le bloc qui la contient

```
{
  int i = 10;
  {
     int j = 5;
     int i = 5; // Previous i is not known
  }
  // here j is unknown and i==10
}
// here i and j are unknown
```

- Variables globales : déclarées en dehors de tout bloc, connues dans tout le fichier. Usage à limiter ! (haut potentiel de bugs)
- Variables locales statiques

```
void f() {
  static int i = 0;
  std::cout << i << std::endl;
  i++;
}

int main() {
  f(); f(); f(); f(); // Prints 0 1 2 3
  return 0;
}
```

C++ « core » , fonctionnalités standard basiques

› Variables dynamiques

- Emplacement mémoire nécessaire pour une ou plusieurs variables réservé à l'exécution (et non à la compilation)
- C++ : operateurs **`new`** et **`delete`** (C : **`malloc, free`**)

```
double* p, *q;
{
  double x;
  p = new double; // Allocates memory for a single double
                                // p now points on that memory region
                                // Mem is allocated until delete p is called
  q = &x;
}
*q = 1.; // Won't work ! q points on something that does not exist anymore
*p = 1.;  // OK
delete p; // Mem freed
*p = 1.; // Error !
```

```
double* p = new double[100];
p[0] = 1.; // Change value of first term in array. Equiv: *p = 1.;
p[5] = 2.; // Change value of sixth term in array. Equiv: *(p+5) = 2.;
p[200] = 1.; // Undefined, probably segfault
delete[] p; // Frees the memory that was allocated
```

à utiliser pour de la programmation bas niveau. Privilégier les types **`std::vector`** et **`std::string`** pour les tableaux et chaînes de caractères.

- C++11 : **`std::unique_ptr`** (gère les libérations mémoire automatiquement)

La fonction `main`

```
38 int main(int argc, char* argv[]) {
39
40   // Read the name of the input file given in command line
41   // argc is the number of words in command line
42   // argv are the words themselves
43   if (argc != 2) {
44     std::cerr <<"Usage: " << argv[0] << " <inputFile>" << std::endl;
45     return 1;
46   }
```

- Retourne toujours un **`int`** : c'est le signal envoyé à l'OS pour indiquer comment s'est achevée l'exécution (0 : OK, aussi **`EXIT_SUCESS`**, **`EXIT_FAILURE`**)
- A pour argument **`(void)`** (équiv. **`()`**) ou **`(int , char**)`** (équiv. **`(int,char*[])`**)
  - **`argc`** : nombre de mots dans la ligne de commande qui a lancé l'exécution
  - **`argv`** : les mots eux mêmes, chaînes de caractères « à la C »
  - « **`argc`** », « **`argv`** » : convention partagée entre plusieurs langages

- Les fonctions **`inline`**

```
inline double posPart(double a) {
  return 0.5*(abs(a)+a);
}
```

  - Le compilateur remplace l’appel à ces fonctions par les instructions elles-mêmes.
  - Économie des étapes de copie des arguments et du résultats.
  - **`inline`** est de préférence à réserver aux fonctions courtes appelées très souvent.

› Entrées sorties, fichiers

```cpp
#include <iostream>

int a = 10;
std::cout << "This is written on standard output "  << a << std::endl;
std::cerr << "This is written on standard error" << std::endl;

std::cin >> a; // Value of a is read from standard input

// I/O in files
#include <fstream> // File stream

// Read in file
std::ifstream f("myFile.txt"); // "input file stream"
f >> a;

// Write in file (file is overwritten)
std::ofstream f("anotherFile.txt"); // "output file stream"
f << "This is written in file " << a << std::endl;
```