Introduction au C++

Premier programme

```cpp
#include <iostream>
#include <cmath>

int main()
{
    std::cout << "Hello world" << std::endl;
    return 0;
}
```

```
> Hello world
```

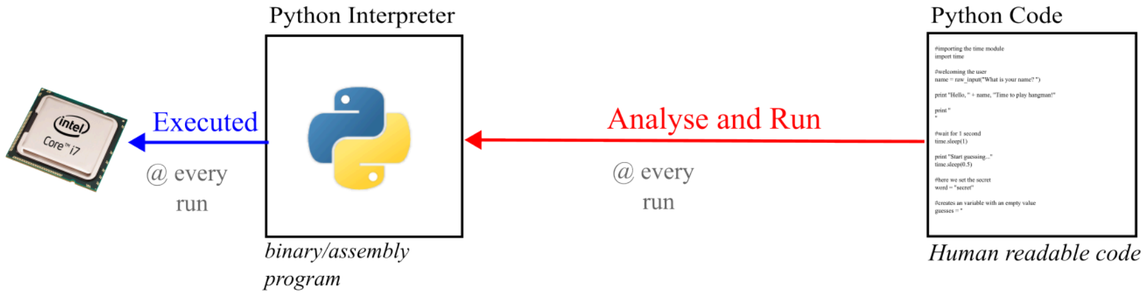
Commentaires: `//` ou `/* ... */`

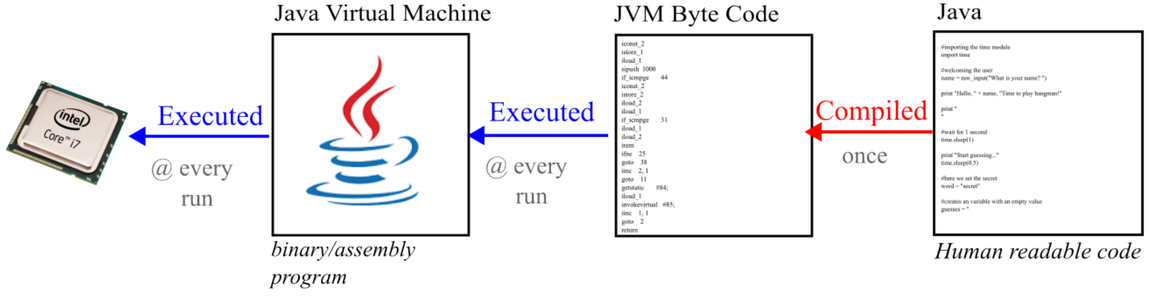
```cpp
// A comment in one line

/* A comment
   continuing on multiple
    lines */
```

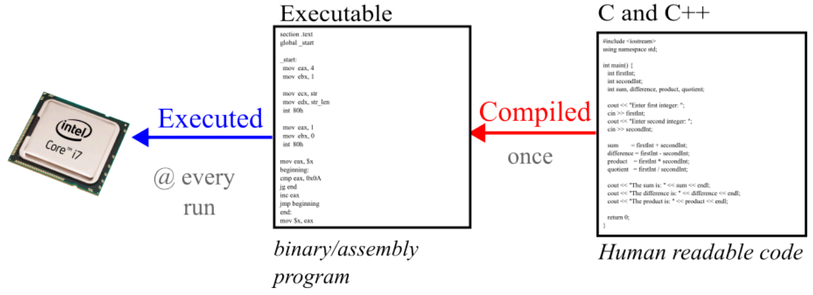
- `#include`: Inclusion de fichiers externes (fichiers de déclaration d'en-tête)
- `#include <iostream>`:
  Inclusion de la bibliothèque standard d'entrée/sortie
  Affichage en ligne de commande via std::cout
- `#include <cmath>`: Inclusion fonctions mathématiques classiques
- `int main()`: Fonction principale du programme -- point d'entrée
- `std::cout << "Hello world" << std::endl;`
  Affichage de "Hello world" suivi d'un retour à la ligne
  - `std` : **St**an**d**ard library
  - `::` : Opérateur de résolution de portée (scope resolution operator)
  - `cout` : **C**ommon **Out**put
  - `<<` : Opérateur C++ (ici de flux)
  - `endl` : **End l**ine
- `return 0;` : Fin du programme avec un code de retour 0

Toutes les instructions C++ se terminent par un point-virgule `;`

C++ Langage compilé







Compilation: Code source → Code machine

C++ → Executable en code assembleur

executé par le CPU

Re-compile uniquement si programme modifié

(+) Execution rapide, vérification de code statique

(-) Code bas niveau, moins dynamique qu'un langage interprété.

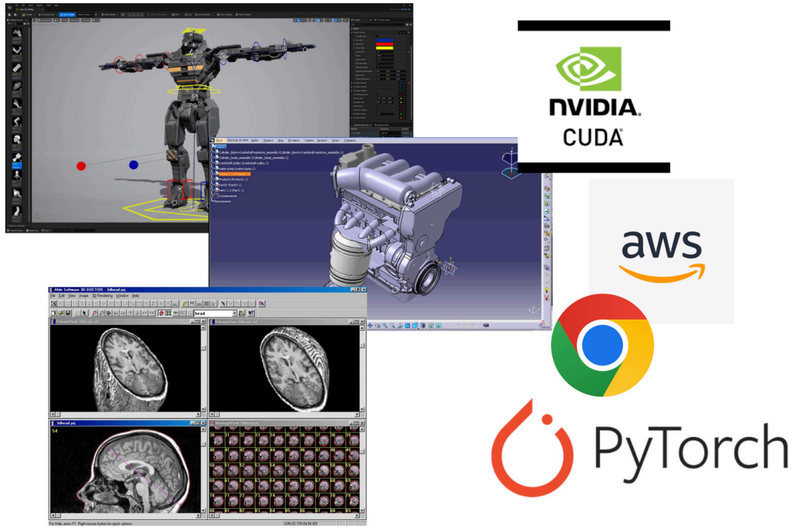
Sous Linux/MacOS, compilation d'un unique fichier: main.cpp

```
> g++ main.cpp -o main
```

Pourquoi C++

Langage le plus utilisé pour les applications scientifique + temps réel

- **Moteurs de jeu/Game engine**: Unity, Unreal, Godot + almost every AAA game
- **2D/3D**: Maya, Blender, Photoshop, Premiere, Catia, SolidWorks,
- **Calcul parallèle/GPU**: CUDA
- **DL frameworks/Vision**: PyTorch, TensorFlow, OpenCV
- **Système exploitation**: Windows
- **Web**: Chrome, Firefox, AWS, Facebook, etc.



[+] Pros

Performance

Robustesse

Haut et bas niveau

Unique au C++ avec Rust

Spécificités

Liberté de programmation

Compatibilité C

[-] Cons

Complexité

Chaine de compilation

Langage typé

Toute variable C++ doit avoir un type connu (à la compilation).

```cpp
int main() {
    int a = 1;
    int b = a+6;

    float c = 5.1f;
    float d = 2.0f * c;

    std::string s = "A word";
    s = s + " and another word";

    std::cout<<a<<","<<b<<std::endl;
    std::cout<<c<<","<<d<<std::endl;
    std::cout<<s<<std::endl;

}
```

```
> 1,7
> 5.1,10.2
> A word and another word
```

Inférence automatique du type possible par le compilateur.

```cpp
int main() {
    auto s = "My string";
    auto f = 3.14f;
}
```

Fonctions

```
typeRetour nomFonction(type nomArgument1, type nomArgument2, etc.)
{
    // ... code de la fonction
    return value;
}
```

```
int addition(int a, int b) {
    return a+b;
}

// no return: void
void display(float f) {
    std::cout << f << std::endl;
}
```

La signature (/en tête) de la fonction doit toujours être décrite avant son appel.

```
int addition(int a, int b)
{
    return a+b;
}

int main() {
    // OK
    // La signature (et le corps)
    //  est déclarée avant.
    int c = addition(5,3);
}
```

```
// Function signature (/header)
int addition(int a, int b);

int main() {
    // OK: La signature et déclarée avant.
    int c = addition(5,3);
}

int addition(int a, int b){
    return a+b;
}
```

```
int main() {
    // KO - ne compile pas,
    //  la fonction "addition"
    //  n'est pas déclarée.
    int c = addition(5,3);
}

int addition(int a, int b) {
    return a+b;
}
```

Fonctions - surcharge

Les fonctions peuvent être surchargées (/overloading).

Même nom, mais différents arguments.

```
// solve ax+b = 0
float solve(float a, float b) {
    return -b/a;
}
// solve ax^2+bx+c = 0
float solve(float a, float b, float c) {
    float delta = b*b - 4*a*c;
    return (-b + std::sqrt(delta))/(2*a);
}
```

```
// Usage
int main() {
    float x = solve(1.0f, 2.0f);
    float y = solve(1.0f, 2.0f, 1.0f);
}
```

Conditions, boucles

```
if(conditionIsTrue) {
    // code
}
else if(anotherCondition) { //optional
    // code
}
else { // optional
    // code
}
// Braces are optionnal for one-line instruction.
```

```
void compare(int x) {
    if(x>5)
        std::cout << "x is greater than 5" << std::endl;
    else if(x==5)
        std::cout << "x is equal to 5" << std::endl;
    else
        std::cout << "x is less than 5" << std::endl;
}
```

```
// Special case: ternary operator
// (condition) ? true-case : false-case;
int a = (x>5)? 1 : 0;
```

```
for(initialization; conditionContinue; increment) {
    // code
}
// Braces are optionnal for one-line instruction.
```

```
for(int k=0; k<5; k++)
    std::cout << k << std::endl;
// 0 1 2 3 4
```

```
while(conditionContinue) {
    // code
    // increment
}
```

```
do{
    // code
    // increment
} while(conditionContinue);
```

```
int k=0;
while(k<5) {
    std::cout<<k<<std::endl;
    k = k+1; }
```

Classes

```
// Declaration of a class
//  (signature/header)
struct vec3{
    // Attributs
    float x,y,z;
    // Methods
    float norm();
}
// struct or class keyword
```

```
// Implementation of the class methods
float vec3::norm() {
    return std::sqrt(x*x + y*y + z*z);
}
```

```
// Use of the class
int main() {
    vec3 v = {1.0f, 2.0f, 3.0f};
    v.y = 3.0f;
    float n = v.norm();
}
```

Implementation: similaire aux fonctions,
avec opérateur de résolution de portée `className::`

Mot-clé `struct` and `class` quasi identique
`struct` visibilité par défaut: `public`
`class` visibilité par défaut: `private`

Signature de la classe doit être avant son utilisation.

Constructeurs / destructeurs

```
struct vec3{
    float x,y,z;

    // Empty constructor
    vec3();
    // Custom constructor with 1 argument
    vec3(float x);

    // Destructor
    ~vec3();
}
// struct or class keyword
```

```
// Initialize all values to 0
vec3::vec3()
    :x(0.0f), y(0.0f), z(0.0f)
{ }
// Initialize all values to v
vec3::vec3(float v)
    :x(v), y(v), z(v)
{ }
// Say goodbye
vec3::~vec3() {
    std::cout << "Goodbye vec3" << std::endl;
}
```

Tableaux de valeurs (std::vector)

Conteneur de valeurs de taille adaptable dynamiquement: std::vector<Type>

```cpp
#include <vector>

int main() {
    std::vector<float> T = {1.0f, 2.0f, 3.0f};
    T.push_back(4.0f);
    T[0] = -1.0f;
    // T = {-1.0f, 2.0f, 3.0f, 4.0f}

    T.resize(6);
    T[4] = T[0] + T[1];
    T[5] = 0.0f;

    for(int k=0; k<T.size(); k++)
        std::cout << T[k] << ", ";

    return 0;
}
```

```cpp
// Template (/generic) type in brackets
//  must be specified at compile time
std::vector<float>
std::vector<int>
std::vector<vec3>
std::vector< std::vector<int> >
...
```

```cpp
// std::vector<Type> has a size 0 by default
std::vector<int> T;

T[5] = 12;  // Error: exceeding vector size

// Use push_back, or resize
T.push_back(3); // now T has size 1
T.resize(10); // now T has size 10
// T[0] ... T[9]
```

Références

En C++, les arguments des fonctions sont passés *par copies*

- Modifications sur la variable copiée restent locales
- Copie potentiellement couteuse pour de gros objets (ex. tableaux)

Possibilité de passer des *références* en argument

Syntaxe: `Type& variable`

```cpp
void increment(int a) {
    a = a+1;
}
int main() {
    int x = 3;
    increment(x);
    std::cout<<x<<std::endl; // 3
}
```

Copie

```cpp
void increment(int& a) {
    a = a+1;
}
int main() {
    int x = 3;
    increment(x);
    std::cout<<x<<std::endl; // 4
}
```

Référence (int& a)

Pour passer des structures de grandes tailles non modifiable

références constantes

Syntaxe: `Type const& variable`

```cpp
float sum(std::vector<float> const& T) {
    float value = 0.0f;
    for(int k=0; k<T.size(); k++)
        value += T[k];
    return value;
}
```

Organisation des fichiers

Organization typique avec des classes

- Un fichier d'en-tête par classe (.hpp/.h)
- Un fichier d'implémentation de la classe (.cpp)
- Un fichier utilisant la classe, incluant le fichier d'en-tête.

En tête (header)

vec2.hpp

```
#pragma once
struct vec2 {
    float x,y;
    float norm();
};
vec2 operator+(vec2 const& a, vec2 const& b);
```

Implémentation (body)

vec2.cpp

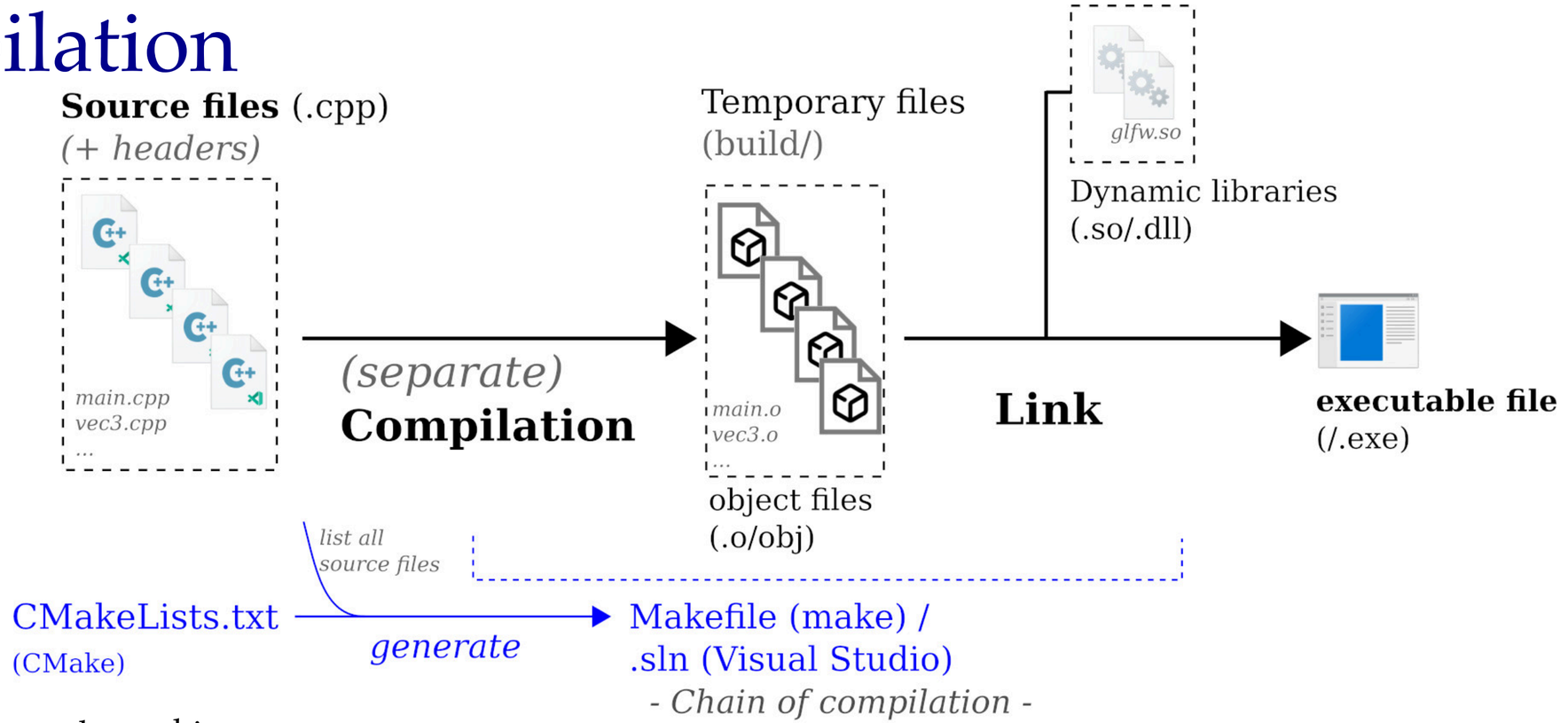
```
float vec2::vec2() {
    return std::sqrt(x*x + y*y);
}
vec2 vec2::operator+(vec2 const& a, vec2 const& b) {
    return {a.x + b.x, a.y + b.y};
}
```

Utilisation

main.cpp

```
#include "vec2.hpp"
int main() {
    vec2 v1 = {1.0f, 2.0f};
    float n = v1.norm();
    vec2 v2 = v1 + vec2{3.0f, 4.0f};
    return 0;
}
```

Chaine de compilation



Compilateur:
Logiciel qui traduit le code source en code machine
Sous linux: g++, clang++
Sous Windows: Visual Studio (ou MinGW).

Compilation séparée:
Conversion des fichiers d'implémentations (.cpp) en fichiers objets (.o) indépendamment les uns des autres.

Edition de lien: Fusion des fichiers objets et des bibliothèques externes pour créer un unique fichier executable.

Makefile: Fichier de configuration pour compiler un projet sous Linux.

CMake: Logiciel permettant de générer un fichier de configuration pour compiler un projet
La configuration de CMake se fait via un fichier `CMakeLists.txt`
Sous Linux: génère un Makefile par défault
Sous Windows: génère un projet Visual Studio