INFO-F105 – Langages de programmation 1
Programmes C++

1 Compilation

1.1 Compilation vs interprétation

Un langage *interprété* comme Python nécessite l'installation d'un *interpréteur*, qui permet de comprendre le code source et de l'exécuter (commande `python`).

Un langage *compilé* comme C++ nécessite l'installation d'un *compilateur*, qui permet de comprendre le code source et de le traduire en instructions compréhensibles par le processeur.

Performances & portabilité

Un avantage d'un programme interprété est qu'il peut s'exécuter sur n'importe quelle machine indépendamment de son architecture tant que l'interpréteur y est installé.

Un avantage d'un programme compilé est qu'il est potentiellement bien plus performant car il communique directement avec la machine plutôt que de passer par un programme intermédiaire (interpréteur, machine virtuelle, ...).

1.2 Processus de compilation

Comme vu au cours, la compilation en C++ se fait en deux étapes majeures :

1. Compilation des fichiers sources `.cpp`/`.hpp` en *fichiers objets* `.o`
2. Édition des liens des fichiers objets `.o` pour former un exécutable

Compile time

La compilation transforme le code source (du texte) en langage machine (du binaire).

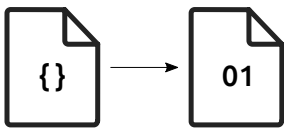

Figure 1: Étape 1 de la compilation

Link time

L'édition des liens (*link time*) assemble les différents fichiers compilés (fichiers objets) en un seul fichier exécutable. Cette étape est souvent indépendante de la précédente pour autoriser chaque fichier objet à être compilé séparément. Cela permet d'accélérer le temps de compilation lorsque seuls certains fichiers ont été modifiés.

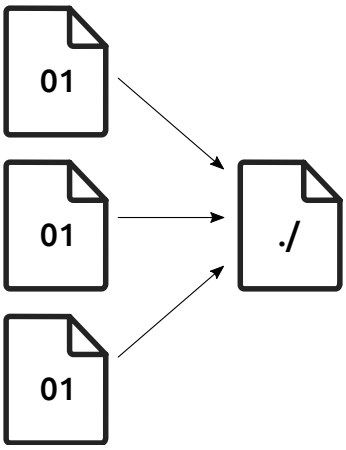

Figure 2: Étape 2 de la compilation

1.3 Compilateur

Le compilateur de référence pour ce cours est `gcc`. Vous trouverez sur l'UV un document qui explique en détail comment l'utiliser.

1.4 Langage machine

Les fichiers objets `.o` et fichiers exécutables contiennent des instructions d'assembleur encodées en binaire qu'on appelle le *langage machine*.

```
mov eax, 0x12345678
```

Par exemple, voici comment se traduit l'instruction ci-dessus sur une architecture `amd64` :

```
b8  78 56 34 12
--  -----------
^-> Opcode pour l'instruction MOV sur le registre eax
    ^-> Opérande 0x12345678 en little-endian
```

On remarque que l'opérande est encodée en *little-endian* et ses *bytes* (8 bits) sont donc inversés. C'est un choix d'architecture commun à tous les processeurs Intel et AMD, mais aussi aux puces Apple et de nombreux autres processeurs ARM.

2 Structure

Un code C++ commence par une fonction `main` écrite dans un fichier `main.cpp`.

```cpp
int main() {
  return 0;
}
```

Cette fonction retourne une valeur entière appelée le *code de retour*, dont la valeur `0` signifie que le programme s'est exécuté sans erreur.

Les autres fichiers viennent par paires :

- `*.cpp` : contient l'*implémentation* du code
- `*.hpp` : contient l'*interface* du code, c'est-à-dire uniquement les *déclarations*

Implémentation

```cpp
// sum.cpp
int sum(int a, int b) {
  return a + b;
}
```

Déclarations

```cpp
// sum.hpp
#ifndef _SUM_HPP_
#define _SUM_HPP_

int sum(int a, int b);

#endif
```

Les lignes commençant par des `#` ne sont pas des commentaires, mais des *directives*. Elles permettent de donner des indications particulières au compilateur :

- `#define NOM` : définit le symbole `NOM` (≈ une variable)
- `#ifndef NOM` et `#endif` : le code entre ces directives n'est compilé que si `NOM` est défini.

Les fichiers `.hpp` doivent toujours utiliser ces directives pour s'assurer que les fonctions ne soient définies qu'une seule fois.

Import

Pour utiliser la fonction `sum` dans un autre fichier que `sum.cpp`, il est nécessaire d'importer le fichier `.hpp` avec la directive `#include` :

```cpp
#include "sum.hpp"
```