Les bases de l'Informatique
Les langages de programmation

Thierry Vaira

BTS SN

v1.0 - 7 juillet 2017

Comment « parler » au processeur ?

- Un processeur « parle » le **langage machine** :
  - C'est une suite de bits interprétée par le processeur.
  - C'est le seul langage que le processeur puisse traiter.
  - Il est composé d'**instructions** et de **données** à traiter codées en binaire.
  - Exemple : l'instruction `10110000 01100001` met la valeur hexadécimale 61 dans le registre `AL`. Ici, `10110000` correspond à l'***opcode*** et `01100001` à la **donnée**.
  - Un processeur possède nativement un **jeu d'instructions** composé d'***opcode*** (code opération) et d'opérandes. Il existe des *opcodes* pour faire des opérations arithmétiques, logiques, etc ... Chaque instruction nécessite un certain temps (généralement un nombre de cycles d'horloge) pour s'exécuter.

Comment peut-on « parler » au processeur ?

- Le langage informatique le plus proche du langage machine est l'**assembleur** :
  - ➟ C'est un langage de bas niveau qui représente le langage machine sous une forme lisible par un humain.
  - ➟ Les combinaisons de bits du langage machine sont représentées par des **mnémoniques** (des symboles) faciles à retenir : `ADD` pour l'addition, `MOV` pour la copie de valeurs, etc ...
  - ➟ Exemple : l'instruction binaire `10110000 01100001` sera `movb $0x61,%al` en assembleur.

➟ *Il est en théorie possible de désassembler le code machine d'un programme. En pratique, c'est plus compliqué que cela car il est parfois difficile de savoir si une valeur en mémoire est un opcode ou une donnée. Dans tous les cas, on ne pourra jamais « remonter » plus haut que l'assembleur.*

Comment faire pour « parler » au processeur ?

- Les contraintes sont les suivantes :
  - ➟ L'ordinateur ne sait exécuter qu'un nombre limité d'opérations élémentaires représentées par des instructions codées en binaire
  - ➟ L'ordinateur ne comprend que le langage machine.
- On est donc obliger d'utiliser des « outils » pour traduire un langage simple vers le langage machine.
- On a ensuite créer des langages de programmation évolués (le plus proche possible de l'« humain ») et utilisables sur n'importe quel ordinateur.

➟ *L'arrivée des premiers langages évolués comme le Cobol et le Fortran datent des années 1950. On a depuis recensé plus de 600 langages de programmation.*

Les langages de programmation

- Quel que soit le langage évolué, il faut le **traduire en langage machine**. Il existe des approches différentes :
  - **Assemblage** (assembleur) : traduction du mnémonique (symbole) en *opcode*.
  - **Compilation** (Pascal, C, C++, C#, ...) : traduction de l'ensemble du programme écrit en langage évolué (**code source**) en langage machine (**code objet**). Le programme en langage machine sera ensuite exécuté.
  - **Interprétation** (Javascript, PHP, Python, Perl, ...) : traduction de chaque instruction avant de l'exécuter. On nomme souvent ces programmes des **scripts**. Ces langages utilisent maintenant une **compilation à la volée (*just-in-time compilation*)**.
  - **Compilation et Interprétation** (Java) : compilation en un code intermédiaire (*bytecode*) qui n'est pas du code machine mais un code pour une « machine virtuelle ». Ce code est ensuite interprété par un interpréteur (la machine virtuelle).

➠ *Les langages n'ont pas les mêmes performances. On utilise pas les mêmes langages pour faire un OS, un tableur, un site web ou une application pour smartphone.*

Historique (1/3)

- En 1970, **Ken Thompson**, créa un nouveau langage : Le B, descendant du BCPL (*Basic Combined Programming Language*, créé en 1967 par Martin Richards). Son but était de créer un langage simple, malheureusement, son langage fût trop simple et trop dépendant de l'architecture utilisée.
- En 1971, **Dennis Ritchie** commence à mettre au point le successeur du B, le **C**. Le résultat est convaincant : Le C est **totalement portable** (il peut fonctionner sur tous les types de machines et de systèmes), il est de **bas niveau** (il peut créer du code aussi rapide que de l'assembleur) et il permet de traiter des **problèmes de haut niveau**. Le C permet de quasiment tout faire, du *driver* au jeu. Le C devient très vite populaire.

Historique (2/3)

- En 1989, l'**ANSI** (*American National Standards Institute*) normalisa le C sous les dénominations **ANSI C** ou **C89**. Un programme écrit dans ce standard est compatible avec tous les compilateurs.
- En 1983, **Bjarne Stroustrup** des laboratoires Bell crée le **C++**. Il construit donc le C++ sur la base du C. Il garde une forte compatibilité avec le C.
- En 1999, l'**ISO** (*International Organization for Standardization*) proposa une nouvelle version de la norme, qui reprenait quelques bonnes idées du langage C++. Il ajouta aussi le type `long long` d'une taille minimale de 64 bits, les types complexes, l'initialisation des structures avec des champs nommés, parmi les modifications les plus visibles. Le nouveau document est celui ayant autorité aujourd'hui, est connu sous le sigle **C99**.

Historique (3/3)

- Certaines autres extensions du C ont elles aussi été standardisées, voire normalisées. Ainsi, par exemple, des fonctions spécifiques aux systèmes **UNIX**, sur lesquels ce langage est toujours très populaire, et qui n'ont pas été intégrées dans la norme du langage C, ont servi à définir une partie de la norme **POSIX**.
- Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (**C++98**), puis une seconde en 2003. Le standard actuel a été ratifié et publié par ISO en septembre 2011 (**C++11**). Mais certains compilateurs ne la supportent pas encore complétement.
- Les **langages C et C++ sont les langages les plus utilisés dans le monde de la programmation**.

Exemple 0a : premier programme en C

```c
#include <stdio.h>

int main (int argc, char **argv)
{
   int n;
   int i;

   printf("Donnez un entier : ");
   scanf("%d", &n);

   for(i = 0; i < n; i++)
      printf("Hello world !\n");

   return 0;
}
```

L'extension par défaut des fichiers écrits en langage C est .c

Exemple 0b : premier programme en C++

```
#include <iostream>

int main (int argc, char **argv)
{
   int n;

   std::cout << "Donnez un entier : " << std::endl;
   std::cin >> n;

   for(int i = 0; i < n; i++)
      std::cout << "Hello world !" << std::endl;

   return 0;
}
```

L'extension par défaut des fichiers écrits en langage C++ est .cpp ou .cc

GNU Compiler Collection

- *GNU Compiler Collection* (GCC) est un ensemble de compilateurs créés par le projet GNU.
- GCC est un logiciel libre capable de compiler divers langages de programmation, dont C, C++, Objective-C, Java, Ada et Fortran.
- GCC a été porté sur un nombre considérable de systèmes d'exploitation et de microprocesseurs.
- GCC signifiait à l'origine *GNU C Compiler*.

GCC est utilisé pour le développement de la plupart des logiciels libres dont le noyau Linux. MinGW (*Minimalist GNU for Windows*) est une adaptation de GCC pour ©Windows.

Fabrication

- Par défaut, gcc et g++ fabriquent un exécutable de nom a.out (ou a.exe). Sinon, on peut lui indiquer le nom du fichier exécutable en utilisant l'option -o.

Exemples 0a et 0b : fabrication de l'exécutable (sous Linux)

```
$ gcc <fichier.c>          OU      $ gcc <fichier.c> -o <executable>

$ g++ <fichier.cpp>        OU      $ g++ <fichier.cpp> -o <executable>
```

L'exécution de ces 2 programmes donne le même résultat :

```
$ ./a.out
Donnez un entier : 2
Hello world !
Hello world !
```

Étapes de fabrication

1. Le préprocesseur (pré-compilation)
   - Traitement de toutes les directives `#xxxxx`
   - Inclusion de fichiers (`.h`)
   - Substitutions lexicales : les "macros"
2. La compilation
   - Vérification de la syntaxe
   - Traduction dans le langage d'assemblage de la machine cible
3. L'assemblage
   - Traduction finale en code machine
   - Production d'un fichier objet (`.o` ou `.obj`)
4. L'édition de liens
   - Unification des symboles internes
   - Étude et vérification des symboles externes (bibliothèques `.so` ou `.DLL`)
   - Production du programme exécutable

Étapes de fabrication avec gcc (1/2)

Pré-compilation

```
$ gcc -E <fichier.c> -o <fichier_precompile.c>
```

Compilation

```
$ gcc -S <fichier_precompile.c> -o <fichier.s>
// ou
$ gcc -c <fichier.c> -o <fichier.o>
```

Assemblage

```
$ as <fichier.s> -o <fichier.o>
```

Étapes de fabrication avec gcc (2/2)

Édition des liens

```
$ gcc <fichier.o> -o <executable>
// ou
$ gcc <fichier1.o> <fichier2.o> <fichiern.o> -o <executable>
// voir aussi : $ ld -dynamic-linker <fichier.o> -o <executable>
```

Décomposition des étapes

```
$ gcc -v -save-temps <fichier.c> -o <executable>

// Sinon combiner en une seule ligne de commande toutes les étapes (
    préprocesseur, compilation, assemblage et édition des liens)
$ gcc <fichier.c> -o <executable>
```

Fabrication avec gcc

Syntaxe de base

```
$ gcc <fichier.c> -o <executable>
```

Utilisation de bibliothèques

```
$ gcc <fichier.c> -o <executable> -l<bibliotheque>
```

GCC possède de très nombreuses options (cf. `man gcc`). Le processus de fabrication d'un exécutable est habituellement automatisé et `gcc` est souvent utilisé dans les `Makefile` par le programme `make`.

C vs C++

- On peut passer progressivement de C à C++.
- Il suffit en premier lieu de **changer de compilateur (par exemple** `gcc` $\rightarrow$ `g++`**)**, sans nécessairement utiliser les nombreuses possibilités supplémentaires qu'offre C++.
- Le principal intérêt du passage de C à C++ est de profiter pleinement de la puissance de la **programmation orientée objets (POO)**.

L'option `-std` des compilateurs `gcc`/`g++` permet de choisir la norme à appliquer au moment de la compilation. Par exemple : `-std=c89` ou `c99` pour le C et `-std=c++98` pour le C++.

Environnement de développement

Les premiers éléments essentiels de ce qui constitue la **boîte à outils** du programmeur que l'on appelle aussi **environnement de développement** sont :

- un programme pour saisir le texte du code source (un **éditeur de texte** à ne pas confondre avec un traitement de texte) et le modifier si nécessaire.
- un **compilateur** pour traduire le code source en code objet et un éditeur de liens pour lier les différentes portions de code objet et en faire un programme exécutable.

Les éditeurs de texte historiquement utilisés par les développeurs sont : `vi`, `vim` et `emacs`. Il existe maintenant des éditeurs GUI (*Graphical User Interface*) comme `Notepad++` (©Windows), `Kate`, `gedit` ou `geany` (Linux).

Environnement de développement intégré

Un EDI (ou IDE pour *Integrated Development Environment*) peut contenir de nombreux outils supplémentaires comme :

- la **documentation**
- un **débogueur** (*debugger*) qui permet de trouver des erreurs et de les éléminer
- un **gestionnaire de version**
- ...

➠ Il existe de nombreux EDI (ou IDE) pour le langage C/C++ et on en utilisera certains notamment en projet. On peut citer : Visual C++, Builder, Qt Creator, Code::Blocks, devcpp, eclipse, netbeans, etc ... Ils peuvent être très pratique mais ce ne sont que des outils et l'apprentissage du C/C++ ne nécessite pas forcément d'utiliser un EDI. Ils améliorent surtout la productivité dans un cadre professionnel.

Conclusion

Les EDIs : ce ne sont que des outils !

« Posséder un marteau ne fait pas de vous un architecte »