Chapitre 2 : Les bases du C, partie 1

Notes de cours éditées par Alexandre Blondin Massé
modifié par Rachid Kadouche
Construction et maintenance de logiciels
INF3135

Département d'informatique
Université du Québec à Montréal

13 janvier 2018

Table des matières

Historique

- Années 70. Naissance du langage C, créé par Ritchie et Kernighan.
- Origine liée au système Unix (90% écrit en C).
- 1978 Publication du livre The C Programming Language, par Kernighan et Ritchie. On appelle cette première version le C K & R.
- 1983 ANSI forme un comité dont l'objectif est la normalisation du langage C.
- 1989 Apparition de la norme ANSI-C. Cette seconde version est appelée C ANSI.

Utilisation du langage C

- Langage d'implémentation de certains systèmes d'exploitation (Unix et dérivés) :
  - Près du langage machine ;
  - Pointeurs typés mais non contrôlés ;
- Adapté aux petits programmes et aux bibliothèques :
  - Efficacité du code généré ;
  - Compilation séparée ;
- Langage peu utilisé pour les applications de grande envergure :
  - Approche archaïque de la modularité ;
  - Typage laxiste.

Caractéristiques du langage C (1/2)

- Langage structuré, conçu pour traiter les tâches d'un programme en les mettant dans des blocs ;
- Il produit des programmes efficaces : il possède les mêmes possibilités de contrôle de la machine que le langage assembleur et il génère un code compact et rapide ;
- C'est un langage déclaratif. Normalement, tout objet C doit être déclaré avant d'être utilisé. S'il ne l'est pas, il est considéré comme étant du type entier ;
- La syntaxe est très flexible : la mise en page (indentation, espacement) est très libre, ce qui doit être exploité adéquatement pour rendre les programmes lisibles.

Caractéristiques du langage C (1/2)

- Le langage C est modulaire. On peut donc découper une application en modules qui peuvent être compilés séparément. Il est également possible de regrouper des programmes en librairie ;
- Il est flexible. Peu de vérifications et d'interdits, hormis la syntaxe. Malheureusement, dans certains cas, ceci entraîne des problèmes de lisibilités majeurs et de mauvaises habitudes de programmation ;
- C'est un langage transportable. Les entrées/sorties, fonctions mathématiques et fonctions de manipulation de chaînes de caractères sont réunies dans des librairies, parfois externes au langage (dans le cas des entrées/sorties par exemple).

Exemple de programme C

```c
// Directives au preprocesseur
#include <stdio.h>
#define DEBUT -2
#define FIN 10
#define MSG "Programme de demonstration\n"

// Prototypes de fonctions
int carre(int x);
int cube(int x);

// Fonction main
int main() {
    int i;
    printf(MSG);
    for (i = DEBUT; i <= FIN; i++) {
        printf("%d carre: %d cube: %d\n", i, carre(i), cube(i));
    }
    return 0;
}

// Implementation
int cube(int x) {
    return x * carre(x);
}

int carre(int x) {
    return x * x;
}
```

Compilation

1. Édition du programme source à l'aide d'un éditeur de texte ou d'un environnement de développement. L'extension du fichier est .c.
2. Compilation du programme source, traduction du langage C en langage machine. Le compilateur indique les erreurs de syntaxe, mais ignore les fonctions et les bibliothèques appelées par le programme. Le compilateur génère un fichier avec l'extension .o.
3. Édition de liens. Le code machine de différents fichiers .o est assemblé pour former un fichier binaire. Le résultat porte l'extension .out (sous Unix) ou .exe (sous Windows).
4. Exécution du programme. Soit en ligne de commande ou en double-cliquant sur l'icône du fichier binaire.

Exemple de compilation (1/2)

- Reprenons le fichier exemple.c
- On peut directement le compiler en exécutable :

```
$ gcc exemple.c
```

  ce qui produit le fichier a.out.
- Puis ensuite, on l'exécute :

```
$ ./a.out
Programme de démonstration
-2 carré: 4 cube: -8
-1 carré: 1 cube: -1
0 carré: 0 cube: 0
1 carré: 1 cube: 1
2 carré: 4 cube: 8
```

Exemple de compilation (2/2)

- En général, on compile en deux étapes ;
- D'abord, de .c vers .o :

```
$ gcc -c exemple.c
```

ce qui produit le fichier compilé (objet) exemple.o.

- Puis ensuite, la commande

```
$ gcc -o exemple exemple.o
```

produit un fichier exécutable nommé exemple.

- Il s'exécute simplement en entrant

```
$ ./exemple
```

Gestion de la compilation

- On a vu un peu plus tôt les deux étapes pour créer un exécutable en C :
  - On compile le fichier .c en un fichier .o ;

```
$ gcc -c exemple.c
```

  - On lie les fichiers .o en un seul fichier exécutable.

```
$ gcc -o exemple exemple.o
```

- Problème : Il est long de relancer la compilation chaque fois qu'on apporte une modification au fichier source.
- Solution : Utilisation d'un Makefile.

Makefiles

- Existent depuis la fin des années ’70.
- Gèrent les dépendances entre les différentes composantes d’un programme ;
- Automatisent la compilation en minimisant le nombre d’étapes ;
- Malgré qu’ils soient archaïques, ils sont encore très utilisés (et le seront sans doute pour très longtemps encore) ;
- Certaines limitations des Makefiles sont corrigées par des outils comme Autoconf et CMake.

Exemple

- Reprenons notre fichier exemple.c
- Un Makefile minimal pour ce fichier serait le suivant :

```
exemple: exemple.o
    gcc -o exemple exemple.o

exemple.o: exemple.c
    gcc -c exemple.c
```

- La syntaxe est de la forme

```
<cible>: <dépendances>
<tab><commande>
```

(le caractère <tab> est très important !)

Exécution d'un Makefile

- Pour utiliser un Makefile, il suffit de taper

```
make
```

- On obtient alors

```
gcc -c exemple.c
gcc -o exemple exemple.o
```

et les fichiers .o et l'exécutable sont produits.

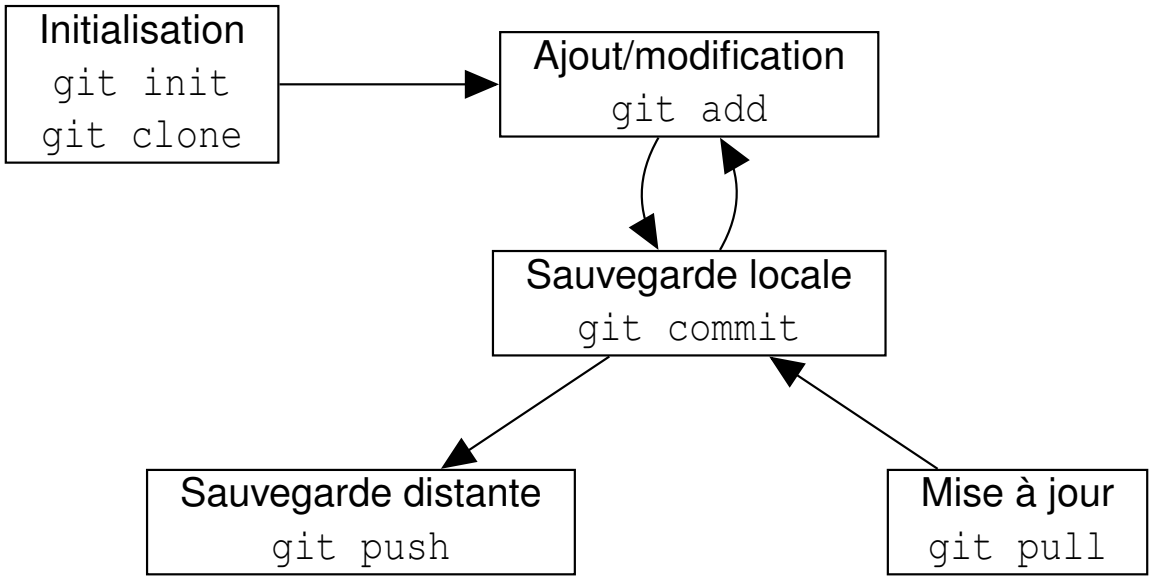
Développer seul

- On entend parfois dire à tort que le logiciel Git n'est utile que lorsqu'on travaille en équipe.
- C'est pourtant très utile :
  - Permet de récupérer une ancienne version ;
  - Fournit naturellement une copie de sauvegarde ;
  - Permet de structurer le développement ;
  - Constitue un aide-mémoire de tout ce qui s'est passé, etc.

Dépôt Git

- Un dépôt Git est simplement un répertoire muni d'un historique Git ;
- Tout l'historique se trouve dans un répertoire caché nommé `.git`, disponible à la racine du projet ;
- Ainsi, si vous supprimez ce dossier caché, vous supprimez par la même occasion tout l'historique ;
- Il existe des plateformes qui permettent d'héberger des dépôts : Github, Bitbucket, GitLab, etc.
- On communique avec ces plateformes via des connexions SSH ou HTTPS ;
- Dans le cadre de ce cours, vous utiliserez GitLab.

Flux opérationnel (*workflow*)



Dans le cadre du TP1 :

- Machine personnelle : `git pull` ;
- Serveur Malt : `git push` ;

Initialisation

- Il y a deux façons possibles d'initialiser un répertoire versionné par Git :
  - Avec la commande `git init`, lorsqu'on démarre un nouveau projet ;
  - Avec la commande `git clone`, lorsqu'on souhaite récupérer une copie d'un projet existant.
- TPs, TP2 : vous devrez cloner des projets ; initialiserez vous-mêmes le projet.
- Si vous travaillez sur votre machine personnelle et que vous souhaitez vérifier le comportement de votre projet sur Malt ou Java, il suffira de cloner votre projet.

Commit (1/2)

- Un *commit* est une sauvegarde de l'état de votre projet ;
- `git add` pour ajouter un fichier ou une modification ;
- Avant de faire un *commit* :
  - Vérifier l'état de votre projet avec `git status` ;
  - Au besoin, vérifiez les modifications avec `git diff` ;
- Après avoir fait un *commit* :
- Le projet devient en état propre (*clean*) ;
- Il est ensuite possible de faire `git push`, mais pas nécessaire.

Commit (2/2)

- Le message de *commit* est très important :
  - La première ligne doit être courte (50 caractères ou moins) ;
  - Si nécessaire, on ajoute des paragraphes ;
  - Il faut laisser une ligne vide entre chaque paragraphe.
  - Éviter les messages bilingues : choisir une langue et s'y tenir.
- Il faut faire des *commits* souvent, mais sans exagérer
- `git log` devrait bien résumer l'histoire du projet ;

Types de base

- Types numériques :

Type	Taille
`char` (signé ou pas)	1 octet
`short` (signé ou pas)	2 octets
`int`(signé ou pas)	2 ou 4 octets
`long` (signé ou pas)	4 octets
`float`	4 octets
`double`	8 octets
`long double`	16 octets

- Type vide : `void`. Définit le type d'une fonction sans valeur de retour ou la valeur nulle pour les pointeurs.

Booléens

- Pas de type booléen natif.
- En C, la valeur 0 est considérée comme `faux` alors que toutes les autres valeurs entières sont considérées comme `vrai`.
- Depuis le standard C99, il existe la librairie `stdbool.h` qui définit les constantes `true` et `false` ainsi que le type `bool`.

```
#include <stdbool.h>
...
bool valide = true;
if (valide) {
    printf("OK\n");
} else {
    printf("ERREUR\n");
}
valide = !valide;
```

Déclaration des variables

Une variable

- doit être déclarée avant son utilisation, en début de bloc ;
- est visible seulement dans le bloc où elle est déclarée ;
- peut être initialisée lors de la déclaration ;
- non initialisée a un comportement imprévisible, puisque la valeur qu'elle contient peut être quelconque ;

```
char c = 'e';
int a, b = 4;
float x, y;
unsigned int d = fact(10);
```

Constantes

- À l'aide de l'instruction `#define` :

```
#define PI 3.141592654
```

- Avec le mot réservé `const` :

```
const float PI = 3.141592654; // Ne fonctionne pas pour les dimensions des tableaux
```

- À l'aide d'un type énumératif :

```
enum {
    MAX_SIZE = 34
};// Seulement pour les constantes entières
```

- Il est essentiel de déclarer des constantes plutôt que des valeurs (magiques) directement dans les programmes.

Notation

- le suffixe `u` ou `U` pour indiquer une valeur non signée ;
- le suffixe `l` ou `L` pour indiquer une valeur longue.
- le préfixe `0` indique une valeur octale ; Par exemple, `064` dénote le nombre décimal $\underline{6} \times 8^1 + \underline{4} \times 8^0 = 52$.
- le préfixe `0x` indique une valeur hexadécimale ; Par exemple, `0X34` dénote ce même décimal $\underline{3} \times 16^1 + \underline{4} \times 16^0 = 52$.
- Un caractère, entre apostrophes `'` , est un nombre ; Par exemple, `'4'` correspond au décimal 52 (code ASCII).

```
char i = 52, j = 064, k = 0X34, l = '4';
printf("%d %d %d %d\n", i, j, k, l);
// affiche : 52 52 52 52
```

Caractères spéciaux

Quelques caractères utiles :

- `\n`, le caractère de fin de ligne ;
- `\t`, le caractère de tabulation ;
- `\\`, le caractère “backslash" ;
- `\'`, l’apostrophe ;
- `\"`, les guillemets.

Instruction `for`

```
for (<initialisation>; <condition>; <incrementation>) {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
}
```

- `<initialisation>` est évaluée une seule fois, avant l'exécution de la boucle.
- `<condition>` est évaluée lors de chaque passage, avant d'exécuter les instructions dans le corps de la boucle ;
- `<incrémentation>` est évaluée lors de chaque passage, après avoir exécuté les instructions dans le corps de la boucle.

Différence avec Java

- Attention, on ne peut déclarer le type de l'itérateur dans l'initialisation qu'avec le standard C99.
- Par exemple, le fragment de code suivant ne compile pas avec le standard ANSI :

```
for (int i = 0; i < 10; ++i) {
    printf("Valeur %d du tableau : %d", i, tab[i])
}
```

- Il faut plutôt écrire

```
int i;
for (i = 0; i < 10; ++i) {
    printf("Valeur %d du tableau : %d", i, tab[i])
}
```

Instructions `if`, `else if` and `else`

```
if (<condition>) {
    <instruction>
}

if (<condition>) {
    <instruction 1>
} else {
    <instruction 2>
}

if (<condition 1>) {
    <instruction 1>
} else if (<condition 2>)
    <instruction 2>
}
```

Blocs

- Un bloc est un ensemble d'instructions délimitées par des accolades ;
- Les accolades sont facultatives dans les structures conditionnelles s'il n'y a qu'une seule instruction ;
- Ainsi, les fragments suivants sont équivalents :

```
if (!valide) printf("ERREUR");
```

```
if (!valide)
    printf("ERREUR");
```

```
if (!valide) {
    printf("ERREUR");
}
```

Instruction `switch`

```
switch (<variable>) {
    case <valeur 1> : <instruction 1>
    case <valeur 2> : <instruction 2>
    ...
    case <valeur n> : <instruction n>
    default : <instruction n + 1>
}
```

- Les instructions `case` sont parcourues séquentiellement, jusqu'à ce qu'il y ait une correspondance.
- Si c'est le cas, l'instruction correspondante est exécutée, ainsi que toutes les instructions suivantes, tant que le mot réservé `break` n'est pas rencontré.
- L'ordre d'énumération n'est pas important si on trouve une instruction `break` dans chaque cas.
- Le cas `default` est optionnel.

Boucles `while` et `do-while`

Syntaxe :

```
while (<condition>) {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
}

do {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
} while (<condition>);
```

Instruction `break` et `continue`

- `break` permet de sortir de la boucle ;
- `continue` permet de passer immédiatement à l'itération suivante ;
- Il est généralement à éviter d'utiliser plusieurs instructions `break` et `continue` dans la même boucle.