Initiation C

AliENS

18 mars 2015

1 Introduction

Le langage C est un langage très ancien, remontant à 1972 [1], et très proche de la machine. Il a l'avantage par rapport à ses prédécesseurs d'avoir une syntaxe très lisible et de ne pas (forcément) dépendre de la machine sur laquelle on le lance, et il est bien plus simple que ses successeurs, ce qui permet de le connaître en détail. De plus, il est très explicite, et il y a très rarement ambiguïté à la lecture. C'est pour cela que le langage est toujours utilisé aujourd'hui, pour Linux et beaucoup d'autres programmes système par exemple.

Tout au long de ce sujet, vous travaillerez sur un mini-projet : un morpion de taille $n \times n$ quelconque. Les joueurs placent une croix ou un cercle chacun leur tour, et le premier qui arrive à en aligner trois (quelque soit n) a gagné.

2 Premier programme !

2.1 Le code

Ouvrez un fichier texte, par exemple `initiation.c`, et recopiez ceci :

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Attention, le C est très capricieux sur la syntaxe, n'oubliez pas les points-virgules en fin de ligne.

Pour compiler et lancer le programme, faites : `gcc initiation.c -o initiation`

Puis pour l'exécuter : `./initiation`

Youhou ! Vous venez d'écrire votre premier programme C !

2.2 Explications

Expliquons le code ligne par ligne :

```c
#include <stdio.h>
```

Ici, on inclut un fichier standard, nommé stdio.h, avec notre fichier. C'est-à-dire que lorsque notre programme sera compilé, le préprocesseur ajoutera tout son contenu au début de notre fichier, avant de le compiler.

Ceci est en réalité nécessaire parce que le compilateur C ne connait que les fonctions dont on a défini le nom et les arguments **avant** (au dessus dans le code) de l'utiliser. Nous reverrons cela par la suite.

```c
int main(void) {
```

1. Bien qu'il en existe des versions plus récentes, C89, C99 et même C11, avec des fonctionnalités souvent inconnues par la plupart des gens.

Ici, on définit une fonction, nommée `main`. C'est une fonction qui doit nécessairement exister dans votre code, avec ce nom-ci : c'est la fonction qui est exécutée lorsque votre programme est lancé ; s'il n'y en a pas, le compilateur va râler car il ne sait pas quoi faire. [2]

Cette fonction ne prend rien en argument (d'où le `void` ; on aurait simplement pu faire `int main()` mais cela aurait voulu dire "`main` prend un nombre arbitraire d'arguments", ce qui est moins précis et peut dans certains cas rendre le code moins clair (ou ne pas renvoyer d'erreur même si on donne trop d'arguments, par exemple). Elle peut éventuellement prendre des arguments, mais ce n'est pas important pour l'instant.

Elle renvoie un `int`, c'est-à-dire un entier, correspondant à la valeur renvoyée par votre programme : en général on met 0 (ou `EXIT_SUCCESS`) s'il n'y a pas eu de problème, et n'importe quelle autre valeur (ou `EXIT_FAILURE`) s'il y a un problème. Les programmes beaucoup utilisés en ligne de commande ont des codes d'erreurs bien définis.

```
printf("Hello, world!\n");
```

C'est la ligne la plus importante du programme : on appelle la fonction `printf`, qui affiche du texte dans la sortie standard (ce qui s'affiche dans une console). La chaîne de caractère est mise entre guillemets, et le `\n` permet de revenir à la ligne. La ligne finit par un point-virgule, de même que toutes les instructions en C, et on utilise des parenthèses (obligatoires même quand il n'y a pas d'arguments) pour appeler la fonction.

```
return 0;
```

Comme on l'a dit, la fonction `main` renvoie un entier. Ici, on renvoie 0, pour signaler que tout va bien.

```
}
```

On ferme l'accolade de définition de la fonction main, on sort de la définition.

Pour le morpion : modifiez le message en un message d'accueil.

3 Les variables

3.1 Les types de variables

Les variables sont une sorte d'espace de stockage : elles ont une valeur.

Il en existe de nombreux types. Les principaux sont :

- `(signed) int` : contient des entiers signés sur 32 bits, entre -2^31 et 2^31-1 inclus (le `signed` est facultatif)
- `unsigned int` : contient des entiers non signés sur 32 bits, entre 0 et 2^32-1
- `(signed) short` : contient des entiers signés sur 16 bits (pour les limites : exercice !)
- `unsigned short` : contient des entiers non signés sur 16 bits
- `char` : contient des caractères, selon la norme ASCII, notés `'a'`, `'\n'`
- `signed char` : contient des entiers signés sur 8 bits, entre -128 et 127 [3]
- `unsigned char` : contient des entiers non signés sur 8 bits
- `(signed) long long` : contient des entiers signés sur 64 bits
- `unsigned long long` : contient des entiers non signés sur 64 bits
- `(unsigned/signed) long` : contient des entiers sur 32 bits ou sur 64 bits, selon votre architecture. Bref, à éviter !
- `float` : contient des nombres flottants (« réels ») sur 32 bits
- `double` : contient des nombres flottants sur 64 bits
- `long double` : contient des nombres flottants sur 128 bits

Vous utiliserez principalement `int` pour les nombres et `char` pour les caractères, le reste a été listé pour que vous ne soyez pas surpris si vous tombez dessus. (Les `float` vous seront peut-être utiles, mais en général on ne les garde qu'en dernier recours à cause de leur précision qui n'arrange pas toujours.)

Il existe d'autres types de variables, composés, que l'on verra plus loin.

2. En réalité, c'est une fonction nommée `_start`, présente dans `crt1.o`, un fichier déjà compilé toujours lié (*linké*) avec vos fichiers, qui va lancer `main`. D'où une erreur de compilation car le compilateur ne connait pas la fonction `main` à laquelle on fait référence dans CRT.

3. Il y a une légère différence avec `char`, car a priori on ne connait pas le "signe" d'un caractère. En pratique, on utilise `char` pour les caractères mais aussi à la place de `signed char` quand on n'utilise pas simplement un `int`.

3.2 Comment utiliser une variable

Pour définir une variable, on écrit simplement :

```
type maVariable;
```

Ceci permet d'utiliser `maVariable` plus loin dans le code. Attention cependant, sa valeur n'est a priori pas initialisée par défaut, il faut donc lui associer une valeur initiale dans de nombreux cas. Pour lui assigner une valeur, on fait :

```
maVariable = 57;
```

ou la version abrégée deux-en-un :

```
type maVariable = 57;
```

On peut également faire ce genre de choses :

```
type maVar1 = 3, maVar2 = 5;
```

pour définir plusieurs variables, ou :

```
maVar1 = maVar2 = maVar3;
```

pour mettre la valeur de `maVar3` dans `maVar1` et `maVar2`.

Parfois, une conversion d'un type vers un autre ne se fait pas toute seule ; dans ce cas, on utilise la syntaxe `(type)maVariable`.

3.3 Comment afficher une variable

Pour afficher une variable, on peut utiliser `printf`. Celui-ci prend au moins un argument, mais autant que l'on souhaite. On lui passe une chaîne de caractère, suivie des éléments que l'on souhaite insérer dans celle-ci.

Pour cela, on dispose du caractère spécial %. Suivi d'un ou plusieurs autres caractères, il sera remplacé par les variables placées en argument. Les plus importants sont `%d` (pour un entier), `%f` (pour un flottant), `%c` (pour un caractère, on affiche le caractère ASCII correspondant au nombre en question), `%s` (pour les chaînes de caractères, que l'on verra plus tard).

Il y a également des commandes d'échappement, telles que `\n` pour revenir à la ligne. Il y a également `\t` pour faire une tabulation. Les autres sont peu utiles (mis à part `\\` pour écrire simplement `\`, et de même pour `%`).

Exemples à tester (à mettre dans le `main` bien sûr) :

```
int a = 7, b = 3, c = 4;
a = b = c;
printf("Coucou, le resultat est %d, %d, a = %d\n", b, c, a);

char c = 'x';
printf("%d%c\n", c, c);
```

Attention à ne pas faire ce genre de bêtise :

```
printf("%d\n");
```

Dont le comportement est indéfini !

Pour le morpion : récupérez la taille de la grille demandée par l'utilisateur et affichez-la. Pour cela, on utilisera :

```
int tailleGrille;
scanf("%d", &tailleGrille);
```

Cette fonction attendra de recevoir un entier et le mettra dans tailleGrille (elle utilise un pointeur, nous verrons plus loin ce que c'est).

4 Les opérations et structures de contrôle

4.1 Opérations

Il existe de nombreuses opérations :

— Les opérations arithmétiques, données par `+`, `-`, `*` et `/` définies pour les entiers et les flottants (`/` est le quotient dans la division euclidienne pour les entiers, c'est-à-dire la partie entière supérieure du calcul exact)

— `a % b` donne le reste dans la division euclidienne de `a` par `b` (avec des entiers)

— Les opérations logiques bit à bit, pour les entiers : `&`, `|`, `^` et `~` (respectivement and, or, xor, not) et `>>` et `<<` (décaler d'un certain nombre de bits à gauche ou à droite un entier)

— Les opérations de comparaison : elles renvoient 1 si la condition est vérifiée, 0 sinon. `a && b` vérifie si `a` et `b` sont tous les deux non nuls (ce qui signifie qu'ils sont tous les deux vrais, par convention), `a || b` si `a` ou `b` est non nul, et `!a` si `a` est non nul (`a` et `b` doivent être entiers). On a ensuite `<`, `>`, `<=`, `>=`, `==` et `!=` pour la comparaison, stricte puis large, l'égalité et la différence.

Il y a également des opérateurs effectuant une affectation en même temps qu'ils renvoient un résultat ! `x *= y` multiplie `x` par `y`, stocke le résultat dans `x` et le renvoie. De même, `=` renvoie aussi le résultat de l'affectation, et on a la même chose pour `*=`, `/=`, etc.

`x++` augmente la valeur de `x` de 1, mais renvoie cette fois l'ancienne valeur, contrairement à `++x` qui renvoie la nouvelle.

Il existe également l'opérateur ternaire `?`, tel que `a ? b : c` vaut `b` si `a` est vrai, et `c` sinon.

Des exemples tordus (essayez de prévoir le résultat avant de tester !) :

```
int x = 5, y = 7;
printf("%d\n", x+++--y);

printf("%d\n", !!57);

int x = 0;
printf("%d\n", x++ || --x);
```

Oups... Cet exemple est bien trop tordu, il dépend de quel côté est évalué en premier. Il y a certainement des règles de priorité, mais mieux vaut laisser tomber !

4.2 Structures de contrôle

Ça y est, on va enfin faire de vrais programmes ! Pour cela, on a les structures de contrôle habituelles : conditions et boucles.

```
if (test1)
  block1
else if (test2)
  block2
else
  block3
```

`block1`, `block2` et `block3` sont des blocs, ils sont constitués soit d'une unique instruction, soit d'une ou plusieurs instructions placées entre accolades.

De même, on a :

```
while (test)
  do_something
```

qui exécute une instruction tant que `test` est vérifié, et :

```
do
  do_something
while (test)
```

qui fait la même chose mis à part qu'il exécutera toujours `do_something`, et enfin :

```
for (instr1; cond; instr3)
  do_something
```

qui est une version courte de :

```
instr1;
while (cond) {
  do_something
  instr3;
}
```

On a également :

```
switch (val) {
  case 1:
    do_1;
    break;
  case 2:
    do_2;
    break;
  default:
    do;
    break;
}
```

qui permet d'effectuer une action selon la valeur de `val`, `default` étant le cas par défaut. Si les `break` ne sont pas mis, le cas suivant sera également exécuté (à éviter).

Pour le morpion : écrivez la boucle principale du programme. Pour l'instant, comme on ne gère pas le jeu en lui-même, on va supposer que le joueur 1 gagne immédiatement, et il faut permettre aux joueurs de rejouer ou quitter, selon s'ils rentrent 1 ou 2 (ou 'r' ou 'q', qui nécessiteront d'utiliser `%c` au lieu de `%d`).

4.3 Les fonctions

Pour éviter d'avoir à refaire plusieurs fois la même chose, il existe les fonctions. Celles-ci prennent un certain nombre d'arguments, et en renvoient un, par exemple :

```
int plusUn(int n) {
  return n + 1;
}

int main(void) {
    printf("%d %d\n", plusUn(5), plusUn(85));
    return 0;
}
```

Une fonction doit être définie avant d'être utilisée : sinon, on insère avant de l'utiliser ce qu'on appelle un prototype : la fonction sans son contenu.

```
int plusUn(int n);
```

5 La mémoire : pointeurs, tableaux

5.1 Fonctionnement de la mémoire

La mémoire, dans un ordinateur, est une fonction qui associe à une adresse (un entier naturel) une donnée (en général un entier). C'est là que sont stockées les variables.

Il y a plusieurs types d'allocations mémoire, pour réserver de l'espace mémoire :

— L'allocation statique, qui se fait avant l'exécution du programme. La place réservée doit être constante. C'est ce qu'il se produit lorsqu'on déclare une variable globale (hors de toute fonction).

— L'allocation dynamique, qui se fait soit sur la pile d'appel, et automatiquement (mais la place réservée doit être constante aussi), soit sur le tas, ce qui permet d'avoir une taille déterminée lors de l'exécution.

Pour l'allocation statique et sur la pile d'appel, ce sont des déclarations de variables classiques. Pour l'allocation dynamique, on appelle une fonction appelée `malloc`.

Pour cela, on a besoin d'un nouveau concept : les pointeurs. Un pointeur est une adresse vers une variable. On accède à l'adresse d'une variable avec `&maVariable`, et `type*` est le type associé à un pointeur sur une variable de type `type`. On utilise `*ptr` pour déréférencer un pointeur, c'est-à-dire obtenir la valeur contenue à son adresse.

Attention : si le pointeur n'est pas initialisé correctement, votre code ne fonctionnera pas. Le pointeur vient en général de l'adresse récupérée d'une variable, ou d'une valeur renvoyée par `malloc`.

```
int *x = malloc(sizeof(int) * 50);
```

Important : contrairement à l'allocation statique, lorsqu'on alloue dynamiquement à l'aide de `malloc`, il faut libérer la mémoire avec `free`. Sinon, on appelle ça une fuite mémoire (ou *memory leak*) : la mémoire n'est pas libérée à la fin du programme. En pratique ça ne change rien puisque le système d'exploitation est capable de le rattraper, mais c'est une très mauvaise habitude de laisser des fuites mémoire dans son programme.

```
free(x);
```

Exercice : écrire une fonction `void plusUn2(int *x, int *y)` qui augmente les valeurs contenues dans `x` et `y` de 1.

5.2 Tableaux

Il y a deux types de tableaux en C : les statiques, et les dynamiques. Pour les statiques, on fait :

```
int monTableau[5] = {1, 2, 3, 4, 5};
int v = monTableau[0];
```

Et pour les dynamiques :

```
int *monTableau = malloc(sizeof(int) * 5);
monTableau[0] = 1; monTableau[1] = 2;
int v = monTableau[0];
```

Une chaîne de caractères est un simple tableau de `char`, terminant par le caractère nul `'\0'`.

Pour le morpion : réservez la grille de la taille demandée par l'utilisateur, et initialisez-là. Écrivez deux fonctions, une qui affiche la grille (passée en argument), et une qui indique si le joueur actuel a gagné. Ensuite, utilisez ces fonctions pour écrire ce qu'il manque à la boucle principale : les joueurs jouent tour à tour en entrant des coordonnées (on pourra utiliser `scanf("%d %d", &x, &y);` pour lire les deux coordonnées, séparées d'une espace), dont il faudra vérifier la validité (case non occupée). Puis on cherchera si le joueur actuel a gagné après chaque tour.

6 Structures et énumérations

On peut rassembler plusieurs variables en une seule : c'est ce que l'on appelle une structure.

```
struct maStructure {
  int a;
  char b;
  int m[15];
};

struct maStructure coucou = {1, 'a', {5, 7}};
printf("%c\n", coucou.b);
```

Si on a besoin de définir des constantes (par exemple, différents types d'objets), on peut utiliser des énumérations.

```
enum niveau {
  NIVEAU_FACILE,
  NIVEAU_NORMAL,
  NIVEAU_DIFFICILE
};

enum niveau monNiveau = NIVEAU_NORMAL;
```

Les valeurs d'une énumération auront un type entier ; les valeurs n'importent pas normalement.

Pour le morpion : utilisez une structure de jeu stockant la grille, sa taille, et le joueur actuel. Ainsi, on peut passer en argument cette structure uniquement (ou, mieux, un pointeur vers celle-ci) plutôt que les arguments séparément. Utilisez une énumération pour les différents type de case.

7 Application des pointeurs : implémentation d'une liste chaînée

Pour le morpion : on peut avoir envie de stocker un historique des victoires des joueurs. Pour pouvoir ajouter chaque score en temps constant, on peut implémenter une liste chaînée.

Pour cela, vous pourrez utiliser une structure de cette forme :

```
struct Score {
    /* 1 si le joueur A a gagné, 0 sinon */
    int aWon;
    /* Pointeur vers le prochain score, ou NULL s'il n'y en a pas */
    struct Score *next;
};
```

Et implémenter ces fonctions :

```
/* Ajoute le score en tête de la liste chaînée. */
void addScore(Score **table, int aWon);
/* Affiche l'historique des scores. */
void displayScore(Score *table);
/* Libère la mémoire utilisée pour les scores. */
void freeScore(Score *table);
```

Pour la première fonction, on a besoin d'un double pointeur car on devra modifier le premier élément, qu'on allouera avec un malloc(). On veut récupérer un `Score*`, on a donc besoin d'un `Score**` pour le stocker.

Remarque : on aurait également pu considérer que la tête de la liste chaînée ne contient pas un vrai score, ou utiliser une autre structure de «table des scores», pour éviter d'utiliser un double pointeur. (Mais comprendre ce qu'est un double pointeur est très instructif !)