Chapitre 2 : Notions d'Algorithmique et de Programmation en C

12 septembre 2023

1 Langage C

Dans ce chapitre et dans les TP associés, on utilisera le langage C. C'est un langage de bas niveau, créé dans les années 1970. Sa popularité sur le long-terme et ses propriétés en font l'un des langages les plus utilisés en informatique.

Le langage C est un langage **impératif** : la programmation impérative est un paradigme de programmation pour lequel les opérations sont des séquences d'instructions modifiant l'état d'un programme. Les notions présentées ici sont donc communes à d'autres langages utilisant ce paradigme (Perl, Python, PHP, Java...)

Des langages peuvent être multi-paradigmes. Par exemple, OCaml permet de faire de la programmation impérative, mais est essentiellement vu comme un langage **fonctionnel** (c'est-à-dire dans lequel les calculs sont considérés comme des évaluations de fonctions mathématiques). Pour une même situation, selon le paradigme utilisé, un programmeur sera encouragé (ou contraint) à utiliser une méthode de programmation différente.

2 Variables et Types

Lors de l'exécution d'un programme informatique, des données sont stockées en mémoire. Elles sont représentées dans un programme par des **variables**. Une variable est caractérisée par :

— son **identifiant** (nom donné dans le programme)

— son **type**

— sa **valeur** dans l'ensemble des valeurs possibles pour son type

Le 1er TP indique quelles doivent être les composantes d'un programme élémentaire en C, qui contient une fonction principale `main` dans laquelle se situent les instructions (situées dans un bloc). L'une des instructions élémentaires de C est l'attribution d'une valeur à une variable donnée. Le C est un langage typé, et pour attribuer une valeur à une variable `v` donnée en C, il faut deux choses :

— **déclarer la variable** en indiquant son type. La déclaration d'une variable `v` de type `int` se fait avec la syntaxe suivante :

```
int v;
```

— **assigner** une valeur à la variable une fois que celle-ci est déclarée. L'assignation d'une valeur à une variable déjà déclarée se fait avec la syntaxe suivante :

```
v = 10;
```

La déclaration et l'assignation d'une valeur initiale à une variable peuvent être faits simultanément avec cette syntaxe :

```
int v = 10;
```

2.1 Types élémentaires en C

Les quatre type élémentaires utilisés en C sont les suivants :

1. `int` pour les **nombres entiers**, sur lesquels on peut effectuer les opérations arithmétiques de base (`+`, `-`, `*`, `/` et `%`)
2. `float` pour les « nombres à virgule flottante » (ou, par abus de langage, nombres **flottants**) qui servent à représenter les réels, également avec les opérations `+`, `-`, `*` et `/`
3. `bool` pour les **booléens** (en important `stdbool.h`) : les booléens sont initialement représentés par les entiers 1 (pour Vrai) et 0 (pour Faux), avec les opérations logiques de négation (`!`), de conjonction (`&&`) et de disjonction (`||`). Les opérations de comparaisons entre valeurs numériques `==`, `!=`, `<`, `>`, `<=` et `>=` permettent également de prendre en entrée deux entiers ou flottants et de rendre un booléen valant 1 ou 0 selon que la comparaison soit vraie est non. Importer `stdbool.h` permet de définir le type `bool`, pour que `true` représente le booléen vrai (1) et `false` le booléen faux (0).
4. `char` pour les **caractères**. Un caractère est noté entre deux guillemets simples (`'a'`).

Le langage C est un langage à **typage statique**, c'est-à-dire que le type d'une variable est connu au moment de la compilation et doit être explicité par le programmeur. Cela permet notamment de détecter les erreurs de type au moment de la compilation, avant l'exécution du code.

3 Fonctions

On a vu que dans un programme élémentaire, on avait une fonction `main` contenant un bloc d'instructions, qui était la fonction principale lue pendant l'exécution du programme. Au sein d'un seul programme, il est possible de définir plusieurs fonctions, qui pourront éventuellement être *appelées* dans les instructions d'autres fonctions (et en particulier dans `main`). Cela permet en particulier d'éviter de réécrire un ensemble d'instructions utilisé plusieurs fois au sein d'un même programme.

Définition

La **définition** d'une fonction est la donnée de :

- sa **signature** (type de retour, nombre et types de ses paramètres en entrée)
- son **corps** (bloc d'instructions)

3.1 Syntaxe

La syntaxe d'une fonction en C est la suivante :

```
type_R nom_fonction(type_1 arg_1,...,type_N arg_N)
    {
    // [bloc d'instruction]
    return valeur_R;
    }
```

- `valeur_R` correspond à la valeur de retour et `type_R` à son type ;
- `nom_fonction` correspond à l'identificateur de la fonction ;
- `arg_1`, ...`arg_N` correspondent aux paramètres en entrée de la fonction, et `type_1`, ...`type_N` à leurs types.

Exemple. La fonction `pgcd` ci-dessous renvoie, pour deux entiers naturels non-nuls a et b, le PGCD de a et b en effectuant l'algorithme d'Euclide (en utilisant une boucle `while`) :

```
int pgcd(int a, int b)
    {
    int r;
    while (b != 0){
        r = a\%b;
        a = b;
        b = r;
    }
    return a;
    }
```

La signature de la fonction (aussi appelée type, par abus de langage) est indiquée dans l'*En-tête* : la fonction prend en entrée un couple d'entiers et renvoie un entier. Elle peut se noter `int` $\times$ `int` $\to$ `int`.

Remarque. Le `return` (qui fait sortir de la fonction) peut ne pas se situer à la fin du bloc d'instruction, et être présent à plusieurs endroits. Il peut par exemple y en avoir dans les blocs d'instruction du `if` et du `else` d'une condition. Les différents `return` doivent cependant tous renvoyer une valeur du type donné. Aussi, s'il n'y en a pas à la fin, il doit y en avoir sur chaque « chemin » que peut emprunter la fonction (branches du flot de contrôle), sans quoi le compilateur C renvoie un message d'avertissement.

Remarque. Il est possible de définir des fonctions qui ne renvoient rien (par exemple une fonction constituée exclusivement d'instructions d'affichage), à l'aide d'un type spécifique qui est le type `void`.

Voici un programme contenant la fonction `pgcd` définie précédemment :

```
#include <stdio.h>

int pgcd(int a, int b)
    {
    int r;
    while (b != 0){
        r = a%b;
        a = b;
        b = r;
    }
    return a;
    }

int main(){
    int c = 35;
    int d = 21;
    int e = pgcd(c,d); // appel de pgcd sur les valeurs 35 et 21
    printf("%d",e);
    return 0;
}
```

Dans ce programme, la fonction `pgcd` est *appelée* dans la fonction `main`, sur les variables `c` et `d`. Ici :

- `a` et `b` sont les **paramètres** de la fonction : ce ne sont pas des variables avec des valeurs définies.
- `c` et `d`, dont les valeurs sont définies pendant l'exécution de la fonction `main` sont les **arguments** sur lesquels s'applique la fonction `pgcd`.

3.2 Portée des Variables

Définition

En informatique, la **portée** d'un identifiant correspond à la partie d'un programme au sein de laquelle l'identifiant existe et peut être utilisé.

En C, on peut avoir des variables **locales** ou **globales**, selon que leur portée se limite à une fonction dans laquelle elles sont définies ou à la totalité du programme.

Si une variable est déclarée à l'intérieur d'une fonction donnée, lorsqu'on appelle cette fonction, la variable n'existera que pendant l'appel de la fonction.

Exemple. Voici un programme qui appelle une fonction dans laquelle est déclaré et défini un entier `x`, et où il est fait référence à `x` en-dehors de l'appel de la fonction.

```
#include <stdio.h>

void auxiliaire(){
    int x = 50;
}

int main(){
    auxiliaire();
    printf("%d",x);
}
```

Ce programme renvoie un message d'erreur : la variable `x` appelée à la ligne `printf("%d",x);` n'est pas déclarée, car le `x` de la fonction `auxiliaire` n'existe plus une fois l'appel à la fonction auxiliaire terminé.

Pour qu'une variable puisse exister au sein de plusieurs fonctions, il faut la déclarer à l'extérieur du corps des fonctions : ce sont alors des variables **globales.** Voici une modification du programme précédent où `x` est une variable globale : elle existe en vaut 50 après l'appel de la fonction `auxiliaire`.

```
#include <stdio.h>

int x;

void auxiliaire(){
    x = 50;
}

int main(){
    auxiliaire();
    printf("%d",x);
}
```

3.3 Passage en Valeur

On a vu que l'utilisation d'une fonction auxiliaire permet de modifier la valeur d'une variable globale. Que se passe-t-il si l'on crée une fonction auxiliaire qui prend un paramètre en entrée et modifie sa valeur, et que l'on appelle cette fonction sur un argument donné ? Prenons par exemple le programme ci-dessous :

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

```
void auxiliaire(int x){
    x = x + 10;
}

int main(){
    int y = 50;
    auxiliaire(y);
    printf("%d",y);
}
```

On pourrait s'attendre à ce que la valeur affichée de `y` soit 50+10=60 ; or la valeur de `y` n'a pas changé.

En fait, lorsqu'on appelle une fonction de paramètre `x` sur un argument `y`, voici ce qui se passe lors de l'exécution du programme :

- Une variable temporaire `x` est créée, à laquelle on affecte la valeur de l'argument `y` ;
- Les instructions de la fonction sont appliquées sur la variable `x` ;
- La variable `x` cesse d'exister une fois l'appel terminé.

Exemple. pour le programme appelant une fonction `pgcd`, des variables `a` et `b` ont été créées, prenant les valeurs des `c` et `d` (35 et 21). A la fin des calculs, la variable `a`, qui valait 7, a été renvoyée par la fonction ; la commande `e = pgcd(c,d)` a donc affecté la valeur renvoyée par la fonction (7) à la variable

Exemple. pour le programme ci-dessus, la fonction `auxiliaire` crée une variable `x` et lui affecte la valeur 50 ; avec l'instruction `x = x + 10`, la variable `x` change de valeur et vaut maintenant 60 ; elle disparaît ensuite. La variable `y` n'a pas été modifiée car toutes les opérations ont été faites sur une copie.

Cette procédure s'appelle le **passage par valeur**. Comme on l'a vu, **la fonction appelée ne peut pas modifier directement la valeur des arguments**, les opérations étant faites sur une copie de ceux-ci.

Une autre syntaxe permet à une fonction de modifier la valeur de ses arguments. On parle alors de **passage par adresse** : c'est ce que l'on a fait avec la fonction `scanf`, où les arguments étaient notés `&x` au lieu de `x`.

4 Conditions

Le langage C permet d'écrire des instructions conditionnelles, c'est-à-dire des instructions qui ne sont exécutées que lorsqu'une condition est vérifiée. Ces conditions sont exprimées sous la forme de booléens : elles peuvent donc notamment être construites à l'aide des opérateurs de comparaisons entre nombre, et des opérations booléennes. Si l'on souhaite par exemple exécuter une instruction seulement si une variable `x` est positive et une variable `y` vaut 3, on notera la condition `x > 0 && y == 3`. Il est également possible de proposer plusieurs blocs d'instructions selon plusieurs conditions possibles, jusqu'à ce qu'une d'entre elle soit vérifiée (si c'est le cas) avec `else if`, et un bloc d'instruction si aucune des conditions précédentes n'est vérifiée, avec `else`.

Pour la syntaxe des conditions, voir TP01.

5 Boucles

Les boucles constituent un autre type de structure de contrôle en C. Le principe de la boucle est de permettre de répéter plusieurs fois, éventuellement en fonction d'une variable dont la valeur pourra varier. Il existe deux types de boucles en C : les boucles non-bornées (`while`) et les boucles bornées (`for`)

5.1 Boucle While

Les boucles `while` ("tant que" en anglais) permettent de répéter un bloc d'instructions tant qu'une condition est vérifiée. A chaque itération de la boucle, le programme ré-évalue la condition, lance une nouvelle itération si elle est vérifiée et sort de la boucle sinon.

Comme pour les instructions conditionnelles, la condition de la boucle `while` s'exprime sous la forme d'un booléen. Celui-ci peut notamment dépendre d'une variable dont on fera varier la valeur à l'intérieur de la boucle, de telle sorte que la condition finisse par ne plus être vérifiée.

La syntaxe d'une boucle `while` est la suivante :

```
while (condition)
{
    [bloc d'instructions]
}
```

Attention

Rien ne garantit a priori que la condition va cesser d'être vérifiée à un moment, et qu'on va sortir de la boucle. Si ce n'est pas le cas, la boucle sera infinie et le programme ne s'arrêtera pas sans intervention de l'utilisateur.

Exemple. Le programme ci-dessous a une boucle infinie :

```
#include <stdio.h>

int main(){

    int x = 5;
    while (x>0){
        printf("%d \n",x);
    }
}
```

Les boucles `while` servent dans toutes les situations où l'on souhaite répéter une instruction jusqu'à ce qu'une situation terminale apparaisse. En mathématiques, cela peut être le calcul des termes d'une suite jusqu'à atteindre une certaine valeur, ou le calcul du plus grand terme en-dessous d'une certaine valeur dans une suite croissante. Elles servent également à donner une condition pour terminer un programme en fonction de l'action d'un utilisateur.

5.1.1 Incrémentation et Compteurs

Une opération courante sur les entiers en programmation est l'**incrémentation**, qui consiste à augmenter d'une unité la valeur d'une variable. On peut le faire avec la commande `variable = variable + 1` par exemple, mais il existe également un opérateur d'incrémentation, que l'on utilise en notant `variable++`. L'opérateur `--` existe également pour la décrémentation (baisse d'une unité).

Pour des incrémentations et décrémentations de variables avec un écart différent d'une unité, on peut utiliser les opérations `+=` et `-=` : `variable += 2` augmente la valeur de la variable de 2 unités.

Les opérations d'incrémentations sont notamment utilisées dans des boucles où l'on utilise un compteur, dont on fait varier la valeur entre une valeur initiale et une valeur finale. Voici un exemple d'utilisation de la boucle `while` pour ce type d'utilisation :

```
#include <stdio.h>

int main(){

    int compteur = 10;
    while (compteur<=20){
        printf("%d \n",x);
        x++;
    }
}
```

5.2 Boucle For

L'utilisation d'une boucle avec un compteur dont la valeur varie est fréquente en informatique. Elles ont trois caractéristiques :

- L'**initialisation** de la valeur d'une variable compteur ;
- La **condition** de la boucle while, indiquant la valeur maximale (ou minimale) à atteindre pour cette valeur ;
- L'**incrémentation** (ou la décrémentation) après l'exécution des autres instructions à l'intérieur de la boucle.

Les **boucles `for`** permettent d'avoir une écriture plus condensée pour ce type de boucles avec compteur. Leur syntaxe est la suivante :

```
#include <stdio.h>

int main(){

    for (initialisation ; condition ; incrementation){
        [instructions]
    }
}
```

Voici un programme équivalent au programme précédent, utilisant la boucle `for` :

```
#include <stdio.h>

int main(){

    int x;
    for (x = 10 ; x<=20 ; x++){
        printf("%d \n",x);
    }
}
```