TP #1 – APL2

5 novembre 2018
Erwan Kerrien

Rappel : si vous avez des questions sur ce TP ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr (je consulte plus rarement mon mail Erwan.Kerrien@univ-lorraine.fr).

Nous travaillons sous Linux. Je suppose dans chaque TP que vous avez créé un répertoire adéquat pour le module APL2 et/ou chaque TP. Il est admis que vous savez le créer et que vous savez y aller (voir les commandes `mkdir` et `cd`). Je ferai référence à ce répertoire par `REP`.

1 Introduction

Le module APL2 n'est pas un cours de C. Il ne s'agit donc pas de vous enseigner ici l'intégralité du langage C. L'objectif de ce TP est de vous donner les éléments de base qui vous permettront d'implémenter en C les structures de données vues en cours et manipulées en TD.

Il se peut donc que les notions vues en TP le soient trop rapidement. Je vous incite par conséquent fortement à vous référer au cours d'Anne Canteaut disponible sur `https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/cours.pdf` qui est un cours complet sur ce langage. La seule différence est que le cours ci-dessus suit la norme C90. Nous suivrons le standard le plus récent du langage (C11), qui autorise par exemple la déclaration de variables à tout endroit dans le code source (et non seulement en début de bloc comme c'était le cas avec C90).

Je suppose que vous êtes déjà familiers avec le langage Python qui est lui aussi un langage séquentiel et à ce titre a recours à des structures algorithmiques similaires comme les boucles (`for`, `while`) ou les expressions conditionnelles (`if ... else`), les fonctions, et structure également le code en blocs d'instructions : là où Python identifie un bloc par le niveau d'indentation des lignes qu'il contient, C enserrera un bloc d'instructions entre accolades, le bloc commençant par { et se finissant par }. Nous en verrons de nombreux exemples dans ce qui suit.

2 Prise en main : premier programme, compilation et exécution

Python est un langage interprété : une fois le programme Python écrit, il suffit alors de l'exécuter dans une fenêtre de commande par la commande `python <programme>`.

C est quant à lui un langage compilé (comme java). Entre l'écriture du code et l'exécution du programme, il y a donc une phase intermédiaire dite de compilation (même si cela recouvre diverses phases, voir le cours d'A. Canteaut, pages 9 et 10). La compilation a pour but de transformer le code source (fichier de texte) en un exécutable (fichier binaire). Pour lancer le programme, il suffit alors d'entrer le nom de l'exécutable dans une fenêtre de commande.

Exercice : Premier programme

Votre premier exercice consiste à effectuer ces étapes.

1. Ouvrez votre éditeur de texte préféré (si possible de code source, avec un mode C. Le mien est emacs), créez un fichier nommé `hello.c` dans le répertoire `REP`. L'extension `.c` est importante pour que d'une part vous trouviez facilement vos codes sources en C et d'autre part pour que votre éditeur de texte charge le bon mode d'édition.
2. Recopiez le code source suivant dans votre éditeur de texte puis sauvegardez le fichier `hello.c` :

```
1 #include <stdio.h>
2
3 int main()
4 {
5   printf("Hello world\n");
6   return 0;
7 }
```

3. Ouvrez une fenêtre de commande et déplacez-vous dans `REP` (commande `cd`). Vérifiez (commande `ls`) qu'il contient bien le fichier `hello.c`.

4. Compilez le programme `hello` en exécutant dans le terminal la commande : `gcc -o hello hello.c`. Vérifiez (commande `ls`) que le fichier `hello` a bien été créé. Si ce n'est pas le cas, un message d'erreur a certainement été émis par la commande de compilation. Analysez ce message, corrigez les éventuelles erreurs de recopie et recommencez.

5. Exécutez le programme en entrant dans le terminal la commande `./hello` : le message `Hello world` doit s'afficher.

Analyse du code

Ce code est la base de tout programme C. À la ligne 3, il déclare une fonction appelée `main` : elle ne prend aucun paramètre [1], puisqu'il n'y a rien entre les parenthèses ouvrante et fermante ; et elle renvoie un entier (type `int`). **Exécuter un programme C signifie exécuter le code de la fonction `main`.** Par conséquent, écrire un programme C revient à écrire cette fonction `main`. Tant que vous n'avez pas cette fonction, vous n'avez pas de programme.

Les lignes 4 à 7, donnent sa définition, toujours incluse dans un bloc d'instructions : en C un bloc d'instruction est composé de lignes encadrées par des accolades (`{` et `}`). Chaque instruction se termine pas le caractère point-virgule `;`. Contrairement à Python, l'indentation n'est pas importante. Il n'est même pas nécessaire d'écrire une instruction par ligne puisque le séparateur d'instructions est le point-virgule (`;`). Cependant, pour faciliter la lecture du code source, vous ne mettrez qu'une instruction par ligne (se terminant par `;`) et indenterez chaque ligne en fonction de la hiérarchie des blocs d'instructions. Ce dernier point ne pose en général pas de problème puisque les éditeurs proposant un mode C le font automatiquement.

Le bloc d'instructions qui définit le corps de la fonction `main` contient ici deux instructions. La dernière (ligne 6) doit être systématiquement `return 0;` : elle indique que la fonction `main` va renvoyer l'entier (`int`) 0, et donc le programme va se terminer avec le code 0. En shell (sous Linux), ce code de terminaison signifie qu'un programme s'est déroulé sans problème (code de succès). À l'inverse, renvoyer un code non nul signifie qu'il y a eu un problème et vous pouvez dès lors utiliser toute une panoplie de codes non nuls pour identifier les différents problèmes (par exemple renvoyer 1 en cas de problème d'ouverture de fichier, 2 si un paramètre n'est pas valable, etc.).

La ligne 5 contient l'autre instruction qui appelle la fonction `printf`, que nous verrons progressivement plus en détail dans ce qui suit. Cette fonction permet d'afficher une chaîne de caractères : ici c'est `Hello world`. **Attention :** en C, une chaîne de caractères est **toujours** encadrée par des doubles guillemets (`"`). Les simples guillemets (`'`) servent à définir un caractère. Par exemple `"Hello world"` est une chaîne de caractères et `'H'` est le caractère H. La notation `'Hello world'`, qui serait acceptée en Python, est invalide en C. Notez enfin, le caractère spécial `\n` qui code un retour chariot (Enter).

La fonction `printf` n'est pas une instruction de base de C. En fait, C ne dispose que de très peu d'instructions de base (voir le cours d'A. Canteaut, p. 12). De manière un peu similaire à Java, tout en restant bien plus raisonnable que ce dernier langage, C dispose d'une bibliothèque de fonctions standard par défaut. La fonction `printf` est définie dans le module `stdio` (pour *standard input/output* – entrées-sorties standard). Afin de pouvoir appeler `printf`, il faut inclure le code qui la déclare, ce qui est fait par la commande de pré-processeur `#include <stdio.h>` en ligne 1.

3 Variables

Une variable a un nom, un type et une valeur.

Les noms de variables suivent a peu près les mêmes règles que dans tout langage, avec plus de restrictions que Python3 cependant. Au final, un nom de variable en C ne peut comporter que des lettres (minuscules ou majuscules, mais non accentuées... bref, dans les 128 premiers caractères ASCII), des chiffres, et le tiret bas (*underscore* en anglais). Un nom de variable ne peut pas commencer par un chiffre.

Contrairement à Python où un nom de variable peut désigner des objets de types différents au sein d'un même bloc d'instructions, une variable en C est toujours associée au même objet et donc au même type. La «durée de vie» d'une variable est limitée au bloc d'instructions dans lequel elle est déclarée : on peut donc avoir deux variables de même nom et de type différents si elles appartiennent à deux blocs d'instructions disjoints. Les types de base sont :

1. Nous verrons plus tard une autre manière de la définir avec des paramètres

Type	Valeur
`char`	entier ou caractère (code ASCII) codé sur un octet
`short`	entier codé sur deux octets
`int`	entier codé sur 4 octets
`float`	nombre décimal codé sur 4 octets
`double`	nombre décimal codé sur 8 octets

Les types entiers sont signés par défaut. Des entiers non signés peuvent être déclarés grâce au mot clé `unsigned` devant leur type. Par exemple une variable de type `char` pourra prendre des valeurs entières de -128 à 127 (valeur signée sur 8 bits en complément à deux), alors qu'une variable de type `unsigned char` pourra coder des valeurs de 0 à 255. Des entiers longs peuvent aussi être définis grâce au préfixe `long` : par exemple un `long int` codera un entier sur 8 octets. De même une variable `long double` codera un nombre décimal sur 16 octets. La commande `sizeof` prend en paramètre un type ou une variable et renvoie le nombre d'octets nécessaires pour stocker une variable de ce type ou cette variable.

Dès qu'on déclare une variable, elle est créée en mémoire. On peut déclarer une variable sans lui donner une valeur initiale (on dit *initialiser*). Par exemple la ligne

```
char c;
```

déclare une variable e nom `c` et de type `char`. Ce faisant elle est créée en mémoire et prend donc une valeur. Cependant cette valeur n'est pas forcément 0. On peut assigner une valeur initiale à une variable lors de sa déclaration par exemple suivant la ligne

```
char c='a';
```

où la variable `c` est initialisée au caractère `'a'`.

Exercice : Types

Recopiez le programme suivant et exécutez-le : il affiche la taille mémoire (en octets) nécessaire pour stocker les types simples en C.

```
#include <stdio.h> /* fournit la fonction printf */
int main()
{
   printf("Taille d'un char: %ld\n", sizeof(char));                    /* -> 1 */
   printf("Taille d'un unsigned char: %ld\n", sizeof(unsigned char));  /* -> 1 */
   printf("Taille d'un short: %ld\n", sizeof(short));                  /* -> 2 */
   printf("Taille d'un unsigned short: %ld\n", sizeof(unsigned short)); /* -> 2 */
   printf("Taille d'un int: %ld\n", sizeof(int));                      /* -> 4 */
   printf("Taille d'un unsigned int: %ld\n", sizeof(unsigned int));    /* -> 4 */
   printf("Taille d'un long int: %ld\n", sizeof(long int));            /* -> 8 */
   printf("Taille d'un long long int: %ld\n", sizeof(long long int));  /* -> 8 */
   printf("Taille d'un float: %ld\n", sizeof(float));                  /* -> 4 */
   printf("Taille d'un double: %ld\n", sizeof(double));                /* -> 8 */
   printf("Taille d'un long double: %ld\n", sizeof(long double));      /* -> 16 */
   return 0;
}
```

Analyse

Le résultat affiché par chaque instruction `printf` est indiqué en commentaire (commence à `/*` et finit à `*/`. On peut aussi utiliser le `//` en début de commentaire si celui-ci tient sur une ligne). On peut remarquer comment on peut moduler le contenu de la chaîne de caractères envoyée à l'affichage via `printf` : chaque élément commençant par `%` sera remplacé par la valeur d'une variable (ou résultat d'une expression). La liste des variables est donnée à la suite de la première chaîne de caractères et elles seront insérées, dans l'ordre, dans la chaîne de caractères. Les caractères `ld` qui suivent le `%` indiquent qu'il est attendu ici une variable de type `long int`, plus grand entier disponible sur la machine.

Exercice : Variables

Recopiez le programme suivant et exécutez-le : il déclare, et initialise, une variable pour chaque type simple en C, puis affiche la taille nécessaire pour la stocker en mémoire (en octets).

```c
#include <stdio.h> // fournit la fonction printf
int main()
{
  char c='a'; // le caractere a
  unsigned char uc=0; // un char peut aussi stocker un nombre entier
  short s=-1; // entier signe
  unsigned short us=65535;  // valeur maximale possible sur 2 octets
  int i=0;
  unsigned int ui=-1; // attention, ici j'initialise a une mauvaise valeur
  long int li=0;
  float f=0.5; // valeur decimale
  double d=1e10; // 10 puissance 10
  long double ld=-5e-3; // -5 10 puissance -3 = -0.005
  printf("char\t\t: valeur=%c, taille=%ld\n",c,sizeof(c));              // valeur=a, taille=1
  printf("unsigned char\t: valeur=%d, taille=%ld\n", uc, sizeof(uc)); // valeur=0, taille=1
  printf("short\t\t: valeur=%d, taille=%ld\n", s,sizeof(s));           // valeur=-1, taille=2
  printf("unsigned short\t: valeur=%d, taille=%ld\n", us,sizeof(us)); // valeur=65535, taille=2
  printf("int\t\t: valeur=%d, taille=%ld\n", i,sizeof(i));             // valeur=0, taille=4
  printf("unsigned int\t: valeur=%u, taille=%ld\n", ui,sizeof(ui));    // valeur=4294967295, taille=4
  printf("long int\t: valeur=%ld, taille=%ld\n", li,sizeof(li));       // valeur=0, taille=8
  printf("float\t\t: valeur=%f, taille=%ld\n", f,sizeof(f));           // valeur=0.500000, taille=4
  printf("double\t\t: valeur=%lf, taille=%ld\n", d,sizeof(d));         // valeur=10000000000.000000,
                                                                       // taille=8
  printf("long double\t: valeur=%Lf, taille=%ld\n", ld,sizeof(ld));    // valeur=-0.005000, taille=16
  return 0;
}
```

Analyse

On voit ici comment déclarer les variables de type simple, et les initialiser (lignes 4 à 13). Le `=` simple en C est symbole d'affectation (il faudra utiliser le double égal `==` pour le test d'égalité).

Les lignes suivantes vous donnent divers codes pour insérer la valeur de ces variables dans une chaîne de caractères à afficher par `printf` (par exemple, ligne 14, `%c` pour un caractère, ligne 17, `%d` pour un entier, ligne 19, `%u` s'il est non signé, ligne 21, `%f` pour un `float`, et ligne 22 `%lf` pour un `double`).

On remarquera aussi un nouveau caractère spécial `\t` qui encode une tabulation.

Enfin, j'ai volontairement initialisé la variable `ui`, de type `unsigned int` à la valeur -1, invalide pour un type non signé (ligne 9). Le compilateur ne dit rien et aucune erreur ou avertissement n'est remonté : le programme va traduire le plus simplement possible cette valeur selon son code binaire : -1 en binaire et complément à deux, sur un entier (soit 4 octets) se code comme 32 "1" successifs. Ici, ligne 9, je demande au programme d'interpréter ce nombre comme un entier non signé. Il le traduit alors comme l'entier 4294967295, soit la valeur entière maximale qu'on peut stocker sur 4 octets ($2^{32}-1$).

On se référera au cours d'A. Canteaut, pages 19 à 24 pour la liste des opérateurs disponibles pour manipuler les variables en C. Ceux-ci sont assez proches de ceux utilisés en Python, aussi ne sont-ils pas approfondis dans ce TP. Nous verrons certaines particularités de C au fur et à mesure du besoin que nous aurons, notamment concernant l'incrémentation et la décrémentation qui sont notamment utiles pour les boucles.

4 Fonctions

La définition [2] d'une fonction suit une syntaxe similaire à celle d'une variable (une fonction peut d'ailleurs être manipulée comme une variable en C) : une fonction a un nom, qui suit les mêmes règles que pour les variables, a un type qui est composé du type de la valeur qu'elle renvoie, ainsi que de la séquence de paramètres qu'elle accepte, avec leurs types, et une fonction a une valeur, qui est son code source défini au sein d'un bloc d'instructions.

2. Il existe une différence entre la déclaration d'une fonction, qui donne son prototype (quel est son nom, quels paramètres elle accepte, avec quels types, et quel type renvoie-t-elle) et sa définition (le code source qu'elle exécute). Cette distinction est surtout utile dans le cadre d'une programmation modulaire qui dépasse le cadre de ce cours. Aussi, parlerons-nous toujours ici de déclaration de fonction.

Exercice : Définitions de fonctions

Dans ce qui suit, vous implanterez tous les codes donnés et les exécuterez.

Première fonction Le code suivant définit la fonction `suivant` qui prend en paramètre un entier signé (`int`) et renvoie un entier signé qui est le suivant, à savoir l'entier en entrée + 1.

```
1  #include <stdio.h>
2
3  int suivant(int i)
4  {
5    int j=i+1;
6    return j;
7  }
8
9  int main()
10 {
11   int k=3,j;
12
13   j=suivant(k);
14   printf("Le nombre suivant %d est %d\n", k, j);
15   return 0;
16 }
```

Analyse On peut faire les remarques suivantes :

- la fonction `suivant` est définie de la ligne 3 à 7. Le paramètre est indiqué entre parenthèses après le nom de la fonction. En plus du type `int`, on indique un nom de variable (ici `i`) : ce nom sera utilisé localement pour faire référence au paramètre dans le code de la fonction. Par exemple, cette variable est utilisée ligne 5 pour initialiser la variable `j`.
- La variable `j` est une variable locale à la fonction : elle est allouée lors de sa déclaration en début de bloc (ligne 5) et est libérée en fin de bloc d'instructions (ligne 7). Elle n'a rien à voir avec la variable également appelée `j` mais définie dans une autre bloc d'instructions, celui de la fonction `main` (déclaration ligne 11).
- la ligne 11 vous donne un exemple de déclaration simultanée de plusieurs variables d'un même type. Ici `k` est initialisée à 3, tandis que `j` n'est pas initialisée.
- la non initialisation de `j` dans `main` n'est pas un souci car une valeur lui est aussitôt assignée en ligne 13.
- la ligne 13 montre un exemple d'appel de la fonction `suivant` avec comme paramètre la valeur de la variable `k`

Paramètres multiples Le code suivant définit la fonction `somme` qui ajoute 3 nombres : le premier est un entier non signé (`unsigned int`) le deuxième un entier court (`short`) et le dernier un nombre décimal simple précision (`float`). Leur somme est renvoyée sous la forme d'un nombre décimal simple précision (`float`)

```
1  #include <stdio.h>
2
3  float somme(unsigned int a, short b, float c)
4  {
5    return a+b+c;
6  }
7
8  int main()
9  {
10   unsigned int v1=70000;
11   short v2=-16;
12   float v3=0.5;
13
14   printf("La somme de %u, %d et %f est %f\n", v1, v2, v3, somme(v1,v2,v3));
15   return 0;
16 }
```

Analyse On peut remarquer les choses suivantes :

- Les trois paramètres sont de types simples qui encodent des nombres. Leur somme est donc valide.
- On peut passer une expression à l'instruction `return` (ligne 5), sans avoir besoin de passer par une variable intermédiaire comme dans le code précédent.
- Le résultat est juste parce que nous nous sommes assurés que le type renvoyé par la fonction `somme` était le plus "fin", à savoir qu'il pouvait gérer toutes les valeurs possibles étant donnés les paramètres en entrée. À titre d'exercice, modifiez ce code ligne 3 et remplacez le type `float` renvoyé par la fonction `somme` en un type `int`. Il vous faudra aussi modifier le dernier `%f` en `%d` dans l'instruction `printf` afin d'être cohérent avec le nouveau type renvoyé. Vous remarquez qu'alors le résultat n'est plus exact.
- reprenez le code original. Et modifiez l'ordre des paramètres dans l'appel à `somme` en ligne 14. L'appel à `somme(v2,v1,v3)` ne renvoie pas le bon résultat car la variable `v2` est vue par `somme` comme une entier non signé, alors que sa valeur est négative. De plus, la variable `v1` est vue comme un short. Or sa valeur de 70000 est supérieure au maximum possible pour un short (32767). Le programme ne peut donc pas bien interpréter les valeurs qui lui sont passées. De même, l'appel à `somme(v3,v1,v2)` renvoie encore une valeur différente car de plus la variable `v3` est vue comme un entier par la fonction `somme`, et sa valeur de 0,5 est donc interprétée au plus proche entier comme 0.

Type `void` Certaines fonctions ne sont pas faites pour renvoyer une valeur. Plutôt que de renvoyer une valeur arbitraire, C a défini un type particulier `void` fait pour ces fonctions. L'instruction `return` ne prend alors pas de paramètre et est même optionnelle dans ce cas [3].

Le code suivant implante la fonction `message` qui affiche des messages de différents types.

```
1  #include <stdio.h>
2
3  void message(char t[], char s[])
4  {
5    printf("Message␣(%s)␣:␣%s\n", t, s);
6    return; // optionnel
7  }
8
9  int main()
10 {
11   message("Erreur", "Ceci␣est␣une␣erreur␣grave!!");
12   message("Avertissement", "Bon␣je␣vais␣laisser␣courir␣pour␣cette␣fois.");
13   return 0;
14 }
```

Analyse On peut voir ligne 3 comment déclarer des variables de type chaîne de caractères : ce sont des tableaux de caractères ainsi que nous l'avons vu lors du premier TD. C fait l'hypothèse qu'une chaîne de caractères se termine par le caractère nul (type `c`, valeur=0, codage=`'\0'`). On voit également ligne 5 comment encoder dans une chaîne passée à `printf` des remplacements de chaînes de caractères (emploi de `%s`).

Passage par valeur Le code suivant implante la fonction `sommeProduit` qui prend en entrée deux nombres entiers et les remplace, le premier par leur somme, et le deuxième par leur produit (remarquons l'emploi d'une variable temporaire `k` en ligne 5 pour ce faire).

3. elle peut être intéressante cependant si on a besoin de prévoir une sortie prématurée de la fonction

```c
#include <stdio.h>

void sommeProduit(int i, int j)
{
  int k=i * j; /* stockage temporaire de la valeur qui ira dans j */
  i = i + j;   /* i recoit la somme des parametres */
  j = k;       /* j recoit le produit des parametres (stocke dans k) */
  return;
}

int main()
{
  int i=3; /* declaration de i et initialisation a 3 */
  int j=4; /* declaration de j et initialisation a 4 */
  printf("i=%d ; j=%d\n",i,j); /* affiche : i=3 ; j=4 */
  sommeProduit(i,j); /* appel de la fonction : on pourrait s'attendre
                        a ce que i vaille 7 et j vaille 12 en sortie */
  printf("i=%d ; j=%d\n",i,j); /* affiche encore : i=3 ; j=4 */
  return 0;
}
```

Analyse Lorsque vous exécutez ce code, vous pouvez voir qu'il ne fait pas ce qui était prévu et en particulier les valeurs stockées dans les variables `i` et `j` de la fonction `main` ne sont pas affectées par la fonction `sommeProduit`. La raison est que le passage des paramètres en C se fait **par valeur** : la première chose que fait `sommeProduit` quand on l'appelle est de déclarer deux variables (ici `i` et `j`, en accord avec la définition de `sommeProduit` ligne 3) **qui lui sont locales**, puis aussitôt il **recopie les valeurs** des paramètres passés en entrée (ici 3 et 4) **dans ces variables locales**. Ce sont donc des variables locales qui sont manipulées dans le code de `sommeProduit` lignes 4 à 9. Du coup, les variables de la fonction `main`, même si elles sont passées en paramètres de `sommeProduit`, ne sont pas du tout affectées par ce qui s'y passe.

Passage par variable La manière de corriger cela est de faire un passage **par variable**. En C, ceci se fait en passant en paramètres non pas les variables, mais les **adresses des variables**. Le code suivant effectue un passage par variable des paramètres de la fonction `sommeProduit`, et le programme a du coup le comportement souhaité.

```c
#include <stdio.h>

void sommeProduit(int *i, int *j)
{
  int k=(*i) * (*j); /* stockage temporaire de la valeur qui ira dans j */
  *i = *i + *j;      /* *i recoit la somme des parametres */
  *j = k;            /* *j recoit le produit des parametres (stocke dans k) */
  return;
}

int main()
{
  int i=3; /* declaration de i et initialisation a 3 */
  int j=4; /* declaration de j et initialisation a 4 */
  printf("i=%d ; j=%d\n",i,j); /* affiche : i=3 ; j=4 */
  sommeProduit(&i,&j); /* appel de la fonction : on passe ici les adresses
                          de i et j, obtenues par indirection */
  printf("i=%d ; j=%d\n",i,j); /* affiche alors : i=7 ; j=12 */
  return 0;
}
```

Analyse Lors de l'appel à `sommeProduit` à la ligne 16, on passe l'adresse des variables `i` et `j` locales à `main` : on obtient l'adresse d'une variable en faisant précéder son nom de l'opérateur `&` dit **adresse-de**.

Du coup, les types des paramètres attendus par `sommeProduit` ne sont plus des `int` mais des adresses pointant vers des `int`, ce qui s'écrit en C grâce à l'opérateur `*` placé devant le nom de la variable (voir ligne 3).

En conséquence, les variables `i` et `j`, locales à `sommeProduit` ne sont plus des entiers mais des **pointeurs vers des entiers**. Pour manipuler, en lecture ou en écriture, la zone mémoire vers laquelle ils pointent, il faut les **déréférencer**, qui se fait à nouveau par l'opérateur `*`. On peut voir lignes 5 à 7 son utilisation pour accéder aux valeurs `*i` et `*j` pointées par respectivement `i` et `j`. Nous approfondirons dans les prochains TP cette notion de pointeurs, à la fois fondamentale et particulière au langage C.

5 Expressions conditionnelles

Compilez et exécutez le code suivant :

```
1  #include <stdio.h>
2
3  int main()
4  {
5    printf("Youpi!\n");
6    printf("Tralala\n");
7    return 0;
8  }
```

Vous voyez à présent aisément que ce programme fait deux choses : afficher «Youpi !», puis, sur la ligne suivante «Tralala».

Imaginons à présent qu'on ne veuille afficher la première ligne que si une variable est paire. On peut utiliser le code suivant :

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i=3;
6    if (i%2 == 0)
7      printf("Youpi!\n");
8    printf("Tralala\n");
9    return 0;
10 }
```

L'important ici est l'introduction de l'expression conditionnelle `if`. Son usage est très similaire à Python avec la différence que l'expression testée doit être mise entre parenthèses. Je rappelle que l'indentation de la ligne qui suit le `if` n'est pas significative en C. On peut donc remarquer que seule l'instruction qui suit le `if` est exécutée de manière conditionnelle, l'instruction `printf` en ligne 8 étant toujours exécutée. Notez aussi qu'il n'y a pas de point-virgule `;` après le `if`. **Ajoutez-en un à la fin de la ligne 6 et retestez le code** : vous voyez que les deux `printf` sont toujours exécutés. Dans ce cas, en effet, le `;` indique la fin de l'instruction conditionnelle qui du coup est vide.

On pourra noter aussi l'emploi de l'opérateur `%` en ligne 6 qui permet d'avoir le reste de la division euclidienne (ici par 2).

Ce code n'est pas très intéressant puisque la variable `i` est fixée. Vous pouvez tester d'autres valeurs de `i` (par exemple 2 ?). C'est vite rébarbatif. Le code suivant le modifie donc pour faire cela de manière aléatoire :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main()
6  {
7    srand(time(NULL));
8    if (rand() % 2 == 0)
9      printf("Youpi!\n");
10   printf("Tralala\n");
11   return 0;
12 }
```

Plusieurs choses ont été introduites ici.

— la fonction `rand` (en ligne 8) renvoie un nombre entier aléatoire entre 0 et `RAND_MAX`. Cette fonction, ainsi que la constante `RAND_MAX` appartiennnent à la librairie standard et sont déclarées dans le fichier `stdlib.h`, qui est donc inclus ligne 2 [4].

— on peut remarquer que le résultat de la fonction peut directement être testé en parité en ligne 8, sans avoir besoin de le stocker dans une variable intermédiaire.

— la ligne 7 sert à initialiser la séquence aléatoire suivie par `rand`. On utilise le temps système afin d'avoir en initialisation différente à chaque exécution du programme. Le temps système est obtenu par l'appel à la fonction `time` qui est déclarée dans le fichier `time.h` (voir son inclusion ligne 3)

Et si maintenant, je voulais n'afficher «Tralala» que si je n'ai pas affiché «Youpi !». On va utiliser la même structure `if...else` qu'en Python, aux conventions d'écriture près. Le code suivant le réalise :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main()
6  {
7    srand(time(NULL));
8    if (rand() % 2 == 0)
9      printf("Youpi!\n");
10   else
11     printf("Tralala\n");
12   return 0;
13 }
```

On remarquera à nouveau qu'il n'y a pas de `;` après le `else`. De même que pour `if`, seule l'instruction qui suit le `else` est exécutée de manière conditionnelle.

Enfin, si on veut pouvoir exécuter plusieurs instructions de manière conditionnelle, il suffit de les regrouper dans un bloc d'instructions. C'est ce que montre le code suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main()
6  {
7    srand(time(NULL));
8    int i=rand();
9    if (i % 2 == 0)
10     {
11       printf("J'ai␣le␣nombre␣pair␣%d␣->␣", i);
12       printf("Youpi!\n");
13     }
14   else
15     {
16       printf("J'ai␣le␣nombre␣impair␣%d␣->␣", i);
17       printf("Tralala\n");
18     }
19   printf("Ceci␣est␣toujours␣execute\n");
20   return 0;
21 }
```

Le résultat de `rand` a cette fois été stocké dans une variable `i` afin de pouvoir être réutilisé (en lignes 11 et 16, en vue de l'afficher). On remarquera que le fait de ne pas mettre de caractère fin de ligne `'\n'` en fin de chaîne de caractères aux lignes 11 et 16, fait qu'il n'y a pas de retour chariot dans l'affichage.

Le cours d'A. Canteaut liste les opérateurs relationnels et logiques possibles (p. 21 et 22) ainsi que leur ordre de priorité dans le tableau 1.4 p. 24.

4. A noter que la génération des nombres aléatoires par `rand` n'est pas d'une excellente qualité et on préférera faire appel à des bibliothèques scientifiques qui proposent des générateurs de meilleur qualité.

En p. 23 de ce même document, est exposé l'opérateur ternaire qui permet de faire des affectations conditionnelles, mais il sort du niveau visé par ce cours. On trouvera aussi en p. 25 l'instruction `switch` qui permet d'éviter les `if`... `else if`...`else` à rallonge.

6 Boucles

While

Comme en Python, les mots-clés `while` et `for` permettent de faire des boucles. L'usage de `while` est très similaire à celui qui en est fait en Python, aux mêmes conventions d'écriture près que pour le `if`, à savoir que l'expression testée comme condition de fin (ou plutôt de continuité pour le `while`) doit être mise entre parenthèses, et dans le cas où plusieurs instructions sont incluses sous le `while`, celles-ci doivent être regroupées au sein d'un bloc d'instructions (sinon seule la première est exécutée en boucle). Le code suivant donne un exemple d'usage du `while`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i=0;
6    while (i < 20)
7      {
8        printf("%02de␣instruction␣de␣la␣boucle\n", i);
9        i++;
10     }
11   return 0;
12 }
```

On remarquera en premier lieu l'opérateur d'incrémentation `++` en ligne 9. Cette opération ajoute 1 à la variable `i`. Il s'agit ici d'une *post-incrémentation* puisque les `++` sont placés après la variable, ce qui veut dire que l'expression `i++` renvoie la valeur de `i` **avant son incrémentation**. Par exemple, si `i` vaut 0, alors `i++` renverra la valeur 0, **puis** incrémentera `i`, ce qui veut dire que `i` vaudra 1 **après** cette instruction. Au contraire, l'expression `++i` est valide et réalise une *pré-incrémentation*, à savoir qu'elle **commence par incrémenter** la variable `i`, **puis** renvoie la nouvelle valeur de `i`. Ainsi si `i` vaut 0, `++i` renvoie 1. `i` vaudra aussi 1 après cette instruction. Pour plus de détails et des exemples pour jouer et tester ces notions, voir le cours d'A. Canteaut en p. 23.

On remarquera aussi le format `%02d` utilisé pour afficher l'entier `i` en ligne 8. Cela signifie que je veux afficher cet entier sur 2 chiffres, ajoutant éventuellement des 0 en début pour combler les espaces vides. Cela permet d'avoir un affichage bien aligné (essayez sans le 0, mais en laissant le 2 et vous verrez qu'alors par défaut, un espace est inséré avant le nombre). On notera enfin que le `e` qui suit le `%02d` ne gêne aucunement l'interprétation de la chaîne de caractères.

Do...while

Chose qui n'existe pas en Python, C propose aussi des boucles sous la forme `do`...`while`. La différence avec ce qui précède est que le bloc d'instruction de la boucle est exécuté systématiquement au moins une fois avant que le test de continuité ne soit fait par le `while`. La différence apparaît dans le code suivant qui inclus la même instruction dans une boucle `while`, telle que nous venons de la voir, puis dans une boucle `do`, `while`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i=0;
6    while (i < 0)
7      {
8        printf("(1) %02de instruction de la boucle\n", i);
9        i++;
10     }
11   i=0;
12   do
13     {
14       printf("(2) %02de instruction de la boucle\n", i);
15       i++;
16     }
17   while (i < 0);
18   return 0;
19 }
```

On notera que :

- bien que le test de continuité soit toujours faux, la séquence `do`,`while` en lignes 12 à 17 exécute au moins une fois le bloc d'instructions.
- en ligne 11, on réinitialise la variable `i` à 0, sans la redéclarer. Des malencontreux copier-coller peuvent mener à une duplication de la ligne 5 qui générerait alors une erreur (variable déclarée deux fois).
- ne pas oublier le `;` en fin de ligne 17 pour conclure l'instruction `do`, `while`. Ce point-virgule n'est pas nécessaire dans l'instruction `while` simple car la fin du bloc d'instruction en indique clairement la fin.

For

La boucle `for` en C est en revanche très différente de celle de Python. Le code suivant en donne un exemple

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i;
6    for (i=0; i<10; i++)
7      printf("Valeur : %d\n", i);
8    return 0;
9  }
```

De même que pour `if` et `while`, seule l'instruction (`printf`, ligne 7) est exécutée dans la boucle. La partie difficile concerne l'instruction `for` elle-même, ligne 6, et notamment ce qu'il faut mettre entre les parenthèses. Trois instructions s'y trouvent, séparées par des `;` :

1. l'initialisation : instruction exécutée une fois avant d'exécuter la boucle. Ici, il s'agit d'initialiser la variable dite *de boucle* `i` à 0 (`i=0`)
2. le test de continuité : instruction exécutée en début de chaque itération. Si elle est vérifiée, on continue en exécutant l'instruction de boucle, sinon la boucle s'arrête.
3. la finalisation : instruction exécutée à la fin de chaque itération. Ici il s'agit d'incrémenter la variable de boucle (`i++`)

Ce code peut donc s'écrire sous forme de boucle `while` qui est totalement équivalente :

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i;
6    i=0;
7    while (i<10)
8      {
9        printf("Valeur : %d\n", i);
10       i++;
11     }
12   return 0;
13 }
```

La boucle `for` permet simplement une écriture plus concise de la boucle.

Tableaux

Un tableau est un ensemble d'éléments d'un même type. Le code suivant donne un exemple de déclaration d'un tableau, nommé `tab`, de 100 `float` (ligne 5), et le remplit avec les nombres de 0 à 1, tous les 0.01, en affichant à chaque fois la valeur insérée.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    float tab[100];
6    int i;
7    for (i=0; i<100; i++)
8      {
9        tab[i] = (float)i/100;
10       printf("Valeur inseree en position %d : %f\n", i, tab[i]);
11     }
12   return 0;
13 }
```

On notera l'usage des crochets `[]` contenant un indice entier pour accéder à un élément par son indice `i`, soit en lecture (dans le `printf`, ligne 10), soit en écriture dans l'affectation ligne 9. Ainsi que l'indique l'initialisation de la boucle, le premier élément d'un tableau en C a pour indice 0. Le dernier indice valide pour `tab` est donc 99.

On remarquera aussi l'usage d'accolades pour exécuter le bloc de deux instructions (lignes 9 et 10) dans la boucle, ainsi que la conversion de type en ligne 9 : `(float)` permet de considérer la valeur stockée dans `i` et de la convertir en un `float` (le type de `i` reste inchangé). Ceci permet à la division par 100 d'être faite entre un nombre décimal (`(float)i`) et un nombre entier. On appelle cela un *cast*. Omettre ce cast aurait fait que la valeur 0 aurait été stockée à chaque endroit de `tab` puisqu'on aurait toujours eu le résultat de la division entière entre l'entier `i`, inférieur à 100 par définition de la boucle, et l'entier 100.

Un exemple plus complexe est donné par le code suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  float aleat()
6  {
7    return (float)rand() / RAND_MAX;
8  }
9
10 int main()
11 {
12   srand(time(NULL)); // initialisation du generateur de nombres aleatoires
13
14   int N=rand()%200+1; // generation d'un nombre aleatoire entre 1 et 200 inclus
15
16   float tab[N]; // declaration d'un tableau de N float
17
18   // remplissage du tableau avec des nombres entre 0 et 1
19   int i;
20   for (i=0; i<N; i++)
21     tab[i] = aleat();
22
23   // calcul de la moyenne des nombres dans le tableau
24   float m=0;
25   for (i=0; i<N; i++)
26     m += tab[i];
27   m /= N;
28   printf("La moyenne est : %f\n", m);
29
30   // recherche du premier element plus grand que 0.9
31   for (i=0; tab[i] < 0.9; i++);
32   if (i < N)
33     printf("Element trouve %f en position %d\n", tab[i], i);
34   else
35     printf("Aucun element trouve\n");
36
37   return 0;
38 }
```

On remarquera la possibilité en C moderne de déclarer un tableau dont la taille est stockée dans une variable (ligne 16). On notera l'emploi des opérateurs d'affectation dits *composés* `+=` et `/=` en lignes 26 et 27, qui mettent à jour la variable `m` en lui appliquant l'opération indiquée (`+` et `/`). Enfin, à noter que le test de continuité dans une boucle `for` (mais c'est pareil pour une boucle `while`) peut être quelconque : en ligne 31, on teste la valeur courante de `tab[i]` et on continue tant qu'elle est inférieure à 0.9.

Ces tableaux se comportent comme des variables : ils sont alloués automatiquement en mémoire lors de leur déclaration, et sont détruits en sortie du bloc d'instructions dans lequel ils ont été déclarés. Ils ne peuvent pas être agrandis ni rétrécis. Nous verrons dans le prochain TP que la notion de pointeur est utilisée pour pallier ces défauts.