Programmation C
TP n° 1 : Introduction

Compilation de vos programmes

Un programme écrit en langage `C` doit être compilé pour produire un exécutable. Pour les TP de ce cours, nous vous laissons le choix entre :

- écrire vos programmes dans un éditeur, et les compiler et les exécuter depuis le shell,
- ou utiliser un environnement de développement vous permettant d'éditer, compiler et exécuter vos programmes avec une unique interface.

Compilation et exécution en ligne de commande

L'utilisation d'un éditeur tel qu'`emacs` pour l'écriture de vos programmes offre un avantage que ne proposent pas tous les IDE : celui de la tabulation automatique du code au fur et à mesure de son écriture, qui facilite la détection d'erreurs de syntaxe de base. (*e.g.* oublis de points-virgules, de parenthèses, d'accolades, de guillemets fermants, etc.).
La compilation en ligne de commande permet d'autre part de forcer le compilateur à signaler tous les éléments du code qui, sans rendre le programme incompilable, lui semblent suspects (*e.g.* erreurs de typage probables, parties du code inutiles ou manquantes, etc).
Le fichier de votre programme doit avoir l'extension `.c`. Une fois sauvegardé sous `emacs`, pour compiler et exécuter ce programme, placez-vous dans le terminal dans le répertoire de sauvegarde et entrez la commande suivante :

`gcc -Wall` *monprog.c* `-o` *monprog*

où *monprog.c* est le nom du fichier et *monprog* le nom de l'exécutable à créer (l'option `-Wall` force l'affichage des « warnings », les messages d'alerte indiquant des erreurs probables). Pour lancer l'exécutable, invoquez dans le même répertoire la commande suivante (l'ajout de `./` avant le nom de l'exécutable, sans aucun espace, est indispensable) :

`./`*monprog*

Environnement de développement (IDE)

Si vous préférez programmer dans un IDE, vous pouvez par exemple vous servir de *Geany* (`https://www.geany.org/`), compatible avec les systèmes d'exploitation Linux, MacOs et Windows. Une fois installé et lancé sur votre machine, Geany s'utilise de la manière suivante :

1. Dans le menu `Fichier`, choisir `Nouveau` pour éditer un nouveau fichier dans lequel vous écrirez votre programme. Vous pourrez sauvegarder ce fichier en sélectionnant dans le menu `Fichier` : `Enregistrer sous`.
2. Pour compiler le fichier, cliquez sur le bouton : `Construit le fichier courant` ( ).
3. Pour lancer l'exécutable, cliquer sur : `Exécuter ou voir le fichier courant` ( ).

Affichage et lecture de valeurs

La fonction prédéfinie `printf` permet l'affichage de chaînes de caractères, ainsi que l'insertion de une ou plusieurs valeurs d'expressions dans ces chaînes. Pour vous servir de cette fonction (et de la fonction `scanf` ci-dessous), ajoutez en première ligne de votre programme :

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

Les expressions insérées peuvent être des variables, des constantes ou des expressions combinées à l'aide d'opérateurs. Les points d'insertions s'écrivent `%d` pour des expressions à valeurs entières, et `%lf` pour des valeurs de type `double`. Chaque `\n` rencontré dans la chaîne entraîne un retour à la ligne. On peut par exemple écrire (le sens de la dernière instruction `return 0` sera précisé plus tard) :

```
  #include <stdio.h>
  int main() {
    int n = 42;
    int m = 2;
    double x = 3.14;
                                          // (dans le terminal :)
    printf("%d\n", n + 1);                // 43
    printf("%d + %d = %d\n", n, m, n + m);   // 42 + 2 = 44
    printf("%d / %lf = %lf\n", n, x, n / x); // 42 / 3.140000 = 13.375796

    return 0;
  }
```

Le code ci-dessous montre la manière dont on peut lire une valeur de variable entière au clavier, à l'aide de la fonction prédéfinie `scanf`. L'appel de `scanf` suspendra temporairement l'exécution du programme en redonnant la main au terminal. L'utilisateur pourra alors rentrer un nombre, suivi d'un retour-chariot (touche "Entrée") : le programme reprendra alors la main, et stockera le nombre lu dans la variable `v`.

```
#include <stdio.h>
int main () {
  int v;

  scanf ("%d", &v);
  printf ("v : %d\n",v);

  return 0;
}
```

Notez le symbole `&` précédant le nom de la variable. Le sens cet opérateur vous sera expliqué plus tard. La variable `v` ne prendra une valeur que si l'utilisateur rentre effectivement une suite de caractères formant un nombre entier signé en décimal et restera de valeur indéfinie sinon.

Exercice 1 : Premier programme

Pour être simplement sûr que vous êtes prêts : compilez sans erreurs puis exécutez un premier programme affichant un message d'accueil dans le terminal. Complétez le code de manière à lire les valeurs de deux variables emtières au clavier, et affichez leurs valeurs et leur somme.

Exercice 2 : Boucles simples et imbriquées

1. Écrire un programme lisant un entier `n` positif au clavier et affichant la somme des cubes des `n` premiers entiers naturels positifs. Par exemple, si l'utilisateur entre `5`, votre programme devra afficher `225` :

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

2. Écrire un programme lisant un entier `n` positif au clavier, puis lisant successivement `n` entiers et enfin, affichant la somme (entière) et la moyenne (sous forme de `double`) des nombres lus.

3. Ecrire une fonction `int fact (int n)` calculant et renvoyant la *factorielle* de `n` (supposé positif). On a par définition $0! = 1$, et pour tout $n \geqslant 1$:

$$n! = 1 \times 2 \times \ldots \times n$$

   Compléter le code par un `main` lisant un entier positif au clavier, et affichant la factorielle de cet entier.

4. Modifier la fonction `fact` précédente afin qu'elle affiche, en une seule boucle, les factorielles de chaque entier compris entre `1` et `n` :

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
...
```

5. Dans le `main` d'un nouveau programme, déclarez un tableau intialisé explicitement par des valeurs quelconques, dont la taille sera déduite du nombre d'éléments de cette initialisation :

```
1  int tab[] = {42, 17 /* etc */ }
```

   (a) Complétez le code de `main` par les instructions nécessaires pour afficher : les valeurs de `sizeof(tab)`, de `sizeof(tab[0])` et de `sizeof(tab)/sizeof(tab[0])` ; la suite des éléments du tableau (séparés par des espaces). Testez le résultat.

   (b) Laissez la déclaration de `tab` dans `main`, mais transférez les instructions suivantes (sauf le `return 0`) dans une fonction `void affichage(int t[])`, en renommant `tab` en `t` dans celles-ci, Dans `main`, appelez cette fonction sur le tableau `tab`. Que constatez-vous [1] ? Comment régler le problème ?

1. La raison de ce comportement vous sera expliquée lorsque nous aborderons les pointeurs. Sans trop entrer dans les détails, malgré les ressemblances de syntaxe, `t` et `tab` n'ont pas le même type : `tab` est un tableau d'entiers, `t` n'est que l'adresse de ce tableau en mémoire.

Exercice 3 : suite de Syracuse

Étant donné un entier $n \geqslant 1$, la *suite de Syracuse* engendrée par n est la suite des valeurs prises par n dans le traitement suivant :

1. Si n n'est pas égal à 1, alors :
   (a) si n est pair, on le divise par 2,
   (b) sinon, on multiplie n par 3 et on lui ajoute 1,
   on revient à l'étape 1.

Le *temps de vol* de n est le nombre de fois où l'étape 1 est effectuée dans ce traitement. Le temps de vol de 1 vaut 0. Le temps de vol de 3 vaut 7, la suite de Syracuse de 3 étant :

$$3, 10, 5, 16, 8, 4, 2, 1.$$

La *conjecture de Collatz* est que pour tout entier n, la suite de Syracuse engendrée par n atteint toujours 1, *i.e.* le temps de vol de tout entier naturel est fini.

1. Ecrire un programme calculant et affichant la suite de Syracuse d'une constante `N` définie par un `#define` en début de programme (*e.g.* `27`) (noter que la simple terminaison du programme est la preuve de la conjecture pour cette constante). Les valeurs de cette suite seront affichées en les séparant par des espaces :

   ```
   27 82 41 124 62 ...
   ```

2. Compléter le programme de manière à calculer simultanément le temps de vol de `N`. Ce temps de vol sera stocké dans une variable dont on affichera la valeur en fin d'éxécution, sur une nouvelle ligne, et après celle de `N` :

   ```
   27 : 111
   ```

3. Modifier le programme afin de vérifier la conjecture de Collatz pour tous les entiers naturels de `1` à `N`. Le programme affichera simplement, sur des lignes distinctes, chaque entier suivi de son temps de vol

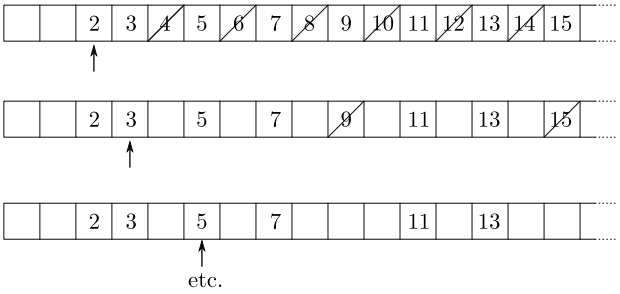
   ```
   1 : 0
   2 : 1
   3 : 7
   4 : 2
   5 : 5
   6 : 8
   7 : 16
   8 : 3
   ...
   ```

Exercice 4 : crible d'Ératosthène

Le *crible d'Ératosthène* est une méthode permettant de calculer tous les nombres premiers inférieurs à un entier *SUP* donné. Son principe est le suivant :

1. On écrit la liste de tous les entiers supérieurs ou égaux à 2 et inférieurs à *SUP*.
2. On effectue un parcours de cette liste. À chaque entier i rencontré, on supprime de la liste tous les entiers strictement plus grands que i et multiples de i encore présents.

En fin de traitement, les nombres encore présents dans la liste sont tous les nombres premiers inférieurs à *SUP*. Voici par exemple l'état de la liste après la rencontre de 2, puis 3 – à la dernière étape, la rencontre de 5 sera suivie de l'effacement de tous les multiples de 5 encore présents (*e.g.* 25), etc.



Implémentation de l'algorithme en C. L'entier *SUP* peut être représenté par une constante `SUP` supérieure ou égale à 2, définie par un `#define` en début de programme. La liste peut être représentée par un simple tableau d'entiers à `SUP` éléments, initialisé par des `0`. À chaque étape du traitement, on peut faire la convention suivante :

- A partir de la position 2, si la case de position `i` contient un `0`, alors l'entier `i` est encore présent dans la liste représentée par le tableau. Si elle contient une valeur non nulle (*e.g.* `1`), alors l'entier `i` a été supprimé de la liste.

Autrement dit, la notion de "suppression de l'entier i dans la liste" est représentée par l'écriture d'un `1` à la position `i` dans le tableau.

En fin de traitement, à partir de la position `2`, les valeurs de positions contenant encore un zéro seront celles de tous les nombres premiers inférieurs à `SUP`. Ces valeurs seront affichées au fur et à mesure du traitement :

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 ...
```

Remarque. Un entier j est multiple de i si et seulement si j modulo i est égal à 0. (l'opérateur de modulo s'écrit `%` en C), mais on a aussi : un entier $j > i$ est multiple de i si et seulement s'il est de la forme $2 \times i$, $2 \times i + i$, $2 \times i + i + i$ ....

Exercice 5 : (⋆) Questions supplémentaires

1. Deux nombres premiers sont *jumeaux* si la différence entre le plus grand et le plus petit est égale à 2 : 3 et 5, 5 et 7, 11 et 13, 17 et 19.... Une conjecture ancienne (Polignac, vers 1849) est qu'il existe une infinité de nombres premiers jumeaux.

   Modifier le programme de l'exercice précédent pour qu'il affiche seulement, pendant la construction du tableau, les couples de nombres premiers jumeaux entre `2` et `SUP - 2` inclus :

   ```
   (3 5) (5 7) (11 13) (17 19) (29 31) (41 43) (59 61) (71 73) ...
   ```

2. La conjecture de Goldbach (1742) est que tout nombre pair $n > 2$ est la somme de deux nombres premiers (nécessairement tous les deux inférieurs à n).

   Toujours à partir du programme de l'exercice précédent, trouvez un moyen simple de vérifier cette conjecture pour tous les nombres pairs entre `4` (inclus) et `SUP` (exclus).

Indication. Il faut au moins un second tableau, que vous remplirez pendant le remplissage du premier. La complexité globale du traitement devrait être quadratique. On peut aussi mettre en mémoire et afficher par relecture les premières possibilités trouvées, avec des tableaux supplémentaires :

```
4 = 2 + 2
6 = 3 + 3
8 = 3 + 5
10 = 5 + 5
12 = 5 + 7
14 = 7 + 7
16 = 5 + 11
18 = 7 + 11
20 = 7 + 13
22 = 11 + 11
24 = 11 + 13
26 = 13 + 13
28 = 11 + 17
30 = 13 + 17
...
```