Tutoriel et évaluation du niveau de connaissance en programmation en C

ENSIIE FISA IAP 2019-2020

Cette première séance du cours d'Algorithme et Programmation est un tutoriel de programmation en C. Durant ce tutoriel, vous allez survoler les concepts qui seront abordés dans ce cours. L'objectif est, premièrement, de vous donner un support dans lequel vous trouverez une base pour programmer en C ; et, secondement, d'évaluer le niveau de chacun. À l'issue de cette séance, si vous n'aveiz jamais codé en C avant, vous aurez normalement une bonne intuition de ce qu'est un programme et des bases de l'algorithmique et de la programmation impérative en général.

Pour chaque section, faites les différents exercices dans l'ordre. Si vous réussissez sans difficulté à faire 2 exercices d'une section, vous pouvez, si vous le souhaitez, passer à la section suivante. Pensez tout de même à lire les énoncés des autres exercices, vous pourriez apprendre un truc ou deux. Chaque section est découpée en 2 partie : une partie tutoriel pour apprendre et une partie exercices pour pratiquer.

Ce TP se fera sous linux, normalement avec une distribution Ubuntu qui a été installée sur votre machine. Si vous avez décidé d'installer un autre système d'exploitation, vous serez livrés à vous même pour tout ce qui concerne l'exécution du programme.

1 Obtenir de l'aide

Durant ce TP, vous aurez peut être besoin d'aide. Votre chargé de TP sera peut être occupé à aider une autre personne. Vous êtes bien entendu autorisés à utiliser votre moteur de recherche préféré pour chercher des réponses. Cependant :

- restez critiques quant aux réponses que vous trouverez en ligne. Attention à ne pas tomber sur une source obsolète ou fausse.
- vous avez, en théorie, toutes les explications nécessaires pour résoudre un exercice en début de chaque section. Si la solution que vous trouvez implique d'utiliser un outil plus évolué, de télécharger une librairie tierce, ce n'est probablement pas ce qui est attendu. Plus généralement, évitez de copier coller du code trouvé en ligne sans comprendre de quoi il s'agit.
- vous disposez normalement de la commande *man* sous linux. Utilisez la commande `man 3 FONCTION` pour avoir de l'aide sur la fonction `FONCTION`. Le manuel est souvent verbeux mais a l'avantage d'être exhaustif. Attention, toutes les fonctions ne disposent pas d'un manuel.

2 Rappels de Shell

Vous pouvez sauter cette partie si vous savez utiliser les commandes `ls` et `cd`, bref si vous savez utiliser une console sous linux.

Lorsque vous utilisez votre ordinateur, vous avez deux modes : graphique et console. On va s'intéresser à l'exploration des fichiers. En mode graphique, l'explorateur de fichier ressemble généralement à la figure 1.

Vous disposez de fichiers, qui sont rangés dans des dossiers, qui sont eux-mêmes rangés dans des dossiers, ... Généralement, on parle d'arborescence de fichiers pour désigner la manière dont les fichiers sont rangés dans une machine. On parle d'arborescence car on peut redessiner les fichiers sous la forme de la figure 2, comme les branches d'un arbres qui se séparent et se ramifient jusqu'à atteindre les feuilles.

Il est possible, en mode graphique de parcourir ces fichiers et ces dossiers en cliquant sur les dossiers. Vous pouvez faire la même chose en mode console avec 2 commandes, `ls` et `cd`.

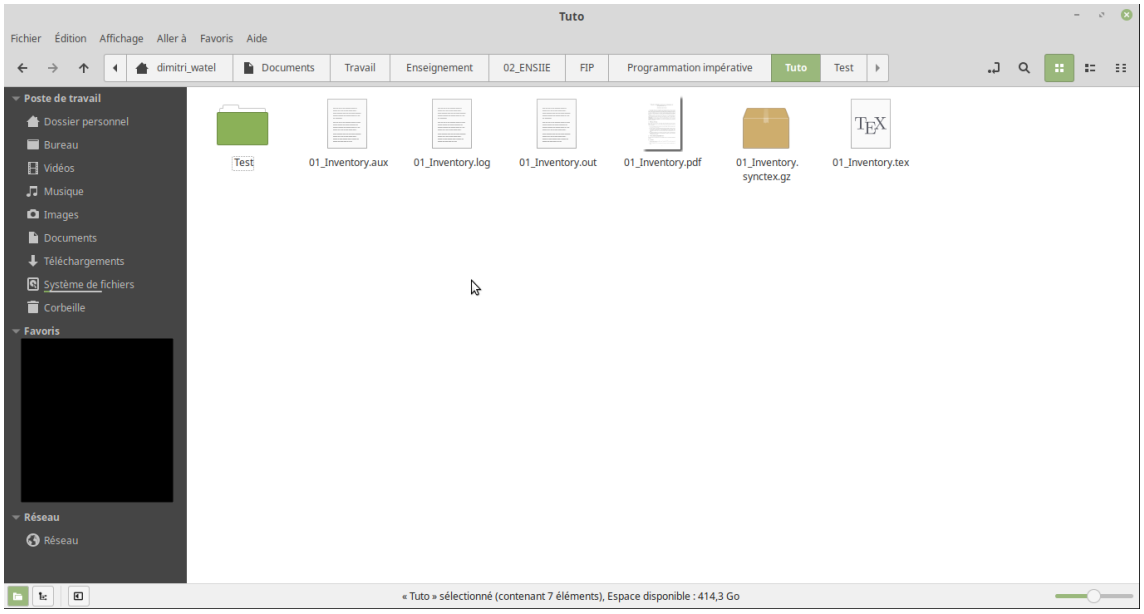

FIGURE 1 – Explorateur de fichiers

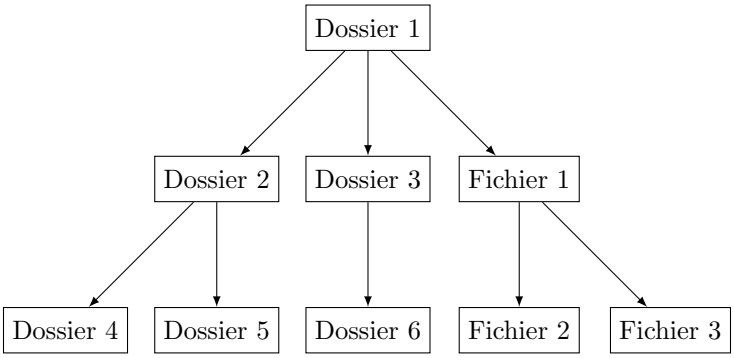

FIGURE 2 – Arborescence de fichiers

2.1 Tutoriel

Commencez par ouvrir un terminal. Sous Ubuntu, il suffit d'ouvrir le menu (icone Ubuntu de mémoire en haut à gauche, ou touche windows) et de chercher Terminal dans la barre de recherche. Vous pouvez aussi appuyer sur Ctrl+Alt+T. Lorsque vous ouvrez ce terminal, vous êtes dans un explorateur de fichiers textuel (*i.e.* on ne peut pas cliquer dedans) et dans lequel vous pouvez faire plein d'opérations dont la portée dépassent le cadre de ce cours. On ne va s'intéresser qu'à 2 choses dans ce cours : reproduire ce qu'on sait faire en graphique et compiler/exécuter du C.

En ouvrant le terminal, vous devriez voir quelque chose comme

```
> machin:~$
```

Sous linux, l'arborescence est constituée généralement de deux parties distinctes : le dossier de l'utilisateur et le reste. Le dossier de l'utilisateur est ce qu'on appelle son dossier `home` ; généralement représenté par une tilde `~`. Quand vous ouvre votre terminal, vous êtes donc dans votre dossier `home`.

Vous pouvez afficher ce qui se trouve dans votre dossier home avec la commande `ls`.

```
> machin:~$ ls
> Bureau Images Documents Enregistrements Musique Telechargements text.txt
> machin:~$
```

Vous pouvez vous déplacer dans un dossier de `~` en utilisant la commande `cd`.

```
> machin:~$ cd Documents
> machin:~/Documents$ ls
> Autres Personnel Travail img.png poeme.txt
```

Vous vous trouvez maintenant dans le dossier `~/Documents` c'est-à-dire le dossier qui se nomme `Documents` et qui se trouve dans `~`.

Pour remonter d'un cran, utilisez la commande `cd ..`

```
> machin:~/Documents$ cd ..
> machin:~$
```

Vous pouvez enchaîner pour aller plus vite.

```
> machin:~$ cd Documents/Personnel
> machin:~/Documents/Personnel$ cd ../../Bureau
> machin:~/Bureau$
```

Enfin, vous pouvez créer un nouveau dossier avec la commande `mkdir` et un nouveau fichier avec la commande `touch` :

```
> machin:~/Bureau$ ls
>
> machin:~/Bureau$ mkdir Jeux
> machin:~/Bureau$ ls
> Jeux
> machin:~/Bureau$ touch fichier.txt
> machin:~/Bureau$ ls
> Jeux fichier.txt
> machin:~/Bureau$ cd Jeux
> machin:~/Bureau/Jeux$
```

Manipulez ces commandes tant que possible si vous n'y êtes pas habitué, c'est très pratique.

Vous êtes perdus ? : utilisez la commande `cd` seule depuis n'importe où pour retourner dans votre dossier `home`.

```
>   machin:~/Bureau/Jeux$ cd
>   machin:~
```

3 Votre premier programme en C

Vous pouvez sauter cette première partie si vous avez déjà codé en C et déjà compilé un programme avec `gcc`.

Sachez juste que vous devez compiler tous vos programmes avec `gcc -Wall -Wextra -std=c99` sans erreur ni warning.

3.1 Tutoriel

1. Ouvrez un terminal.
2. Créez un répertoire `IAP` que vous utiliserez pour l'ensemble des TP du cours avec la commande `mkdir IAP`.
3. Dans ce répertoire, créer un sous-dossier pour cette séance nommé `TP1` avec la commande `mkdir IAP/TP1`.
4. Déplacez vous dans ce dossier avec la commande `cd IAP/TP1`.
5. Créez un nouveau fichier `helloWorld.c` (avec la commande `touch helloWorld.c`).
6. Ouvrez ce fichier avec l'éditeur de votre choix (`vim`, `emacs`, `nano`, `gedit`, ...). Si vous n'avez pas l'habitude du mode console, je vous conseille `gedit helloWorld.c` pour ce TP.

7. Remplissez le fichier avec le contenu suivant et sauvegardez :

```
1    #include <stdio.h>
2    int main(void){
3      // La ligne suivante affiche Hello world
4      printf("Hello World\n");
5      return 0;
6    }
```

8. Ouvrez un second terminal ou un second onglet dans le premier terminal et placez vous dans le même sous-dossier.
9. Tapez la commande suivante : `gcc -Wall -Wextra -std=c99 -o helloWorld helloWorld.c`
10. A l'aide de la commande `ls`, vous pourrez voir qu'il y a un nouveau fichier `helloWorld` qui vient d'apparaître.
11. Enfin tapez la commande `./helloWorld`. Votre terminal devrait ressembler à ceci :

```
> machin:~$ cd ~/IAP/TP1
> machin:~/IAP/TP1$ gcc -Wall -Wextra -std=c99 -o helloWorld helloWorld.c
> machin:~/IAP/TP1$ ls
> helloWorld helloWorld.c
> machin:~/IAP/TP1$ ./helloWorld
> Hello World
```

Félicitations, vous avez écrit votre premier programme en C, le *classique* Hello World. Qu'avons-nous fait ?

— Dans le programme, nous avons indiqué à la ligne 4 d'afficher "Hello World" puis de sauter une ligne (avec le symbole `\n`).

— Pour cela, nous avons utilisé la *fonction* `printf`, qui nous servira dans toute la suite du tutoriel à afficher les résultats des programmes. C'est grâce à la ligne 1 que la machine sait quelle est la signification et le rôle de la fonction `printf`.

— La ligne 3 est un *commentaire*. Un commentaire est toujours ignoré par la machine. Il sert à donner des informations aux personnes qui reliront le code.

— Les lignes 2, 5 et 6 définissent le début et la fin du programme. Lorsque vous exécutez un programme, la machine cherche la ligne `int main(void)`. `main` est ce qu'on appelle une fonction. Tout programme C démarre son exécution par la fonction `main` : on éxécute toutes les instructions écrite entre les deux caractères { et }. La ligne 5 signifie que tout s'est déroulé normalement (on y reviendra une autre fois).

Ce fichier `helloWorld.c` n'est qu'un fichier texte. Il ne peut être lu tel quel par la machine. Nous avons donc, à l'étape 9, *compilé* ce programme ; autrement dit nous avont créé un nouveau fichier `helloWorld` lisible par la machine (ce fichier est appelé *binaire* ; si vous l'ouvrez vous ne verrez que des symboles illisibles).

Pour le compiler nous avons utilisé la commande `gcc` (pour GNU Compiler Collection). On peut lire cette commande ainsi :

— le `helloWorld.c` qui se trouve à la fin est le fichier que l'on souhaite compiler

— `-o helloWorld` permet de définir le fichier dans lequel on va compiler `helloWorld.c`. Le `-o` signifie tout simplement `output`.

— Les options `-Wall -Wextra -std=c99` servent notamment à définir les erreurs et warnings.

Enfin, la commande `./helloWorld` permet d'*exécuter* le programme.
Il y a donc 3 parties :

— le programme

— la compilation

— l'exécution

3.2 Exercices

Exercice 1 — *Affichage basique*

Créer, compilez et exécutez un programme `lorem.c` qui affiche le texte suivant (en respectant les sauts de ligne)

```
> Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
    labore et dolore magna aliqua.
> Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
    dolore eu fugiat nulla pariatur.
>
> Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
    anim id est laborum.
```

Exercice 2 — *Caractère spéciaux*

Certains charactères ne peuvent être affichés facilement. Par exemple, si vous souhaitez afficher le caractère `"`, vous aurez un problème car ce caractère délimite le début et la fin du texte à afficher. Pour l'afficher, il faut insérer `\"` dans le texte. Si vous souhaitez afficher une tabulation, il faut utiliser `\t`. Si vous souhaitez afficher le caractère `\`, il faut utiliser `\\`.

Créer un programme `lorem.c` qui affiche le texte suivant (en remplacant `[tab]` par une tabulation)

```
> \ [tab] "backslash"
> \\ [tab] "real backslash"
> \\\ [tab] "real real backslash"
> \\\\\\\\\\... [tab] "the true name of Ba'al, the soul eater"
```

Exercice 3 — *Erreur de compilation*

Copiez le programme `helloWorld.c` du tutoriel dans un fichier `helloWorldError.c` avec la commande `cp helloWorld.c helloWorldError.c`. Supprimez la première ligne et compilez pour constater que le programme ne reconnait plus la fonction `printf`.

Exercice 4 — *Commentaires et norme ANSI*

On utilie depuis le début le standard C99 (avec `-std=C99`). Il existe d'autres standards.

- Copiez le programme `helloWorld.c` du tutoriel dans un fichier `helloWorldComments.c`.
- Compilez `helloWorldComments.c` avec la commande `gcc -Wall -Wextra -ansi -o helloWorld helloWorld.c` pour constater que la norme `ansi` ne reconnait pas les commentaires de type `//`.
- Remplacez ce commentaire par `/* La ligne suivante affiche Hello world */` pour constater que ça fonctionne de nouveau.

Les commentaires écrits avec `//` doivent tenir sur une ligne. Ceux écrits entre `/*` et `*/` peuvent tenir sur plusieurs lignes.

4 Variables

4.1 Tutoriel

Une variable est, en quelque sorte, une boîte dans laquelle on peut mettre une valeur. Un peu à l'image des variables en mathématiques, on peut facilement accéder à cette valeur et l'utiliser pour faire des calculs. Il y a toutefois une différence fondamentale entre une variable mathématique et une variable informatique. La première se nomme variable car il s'agit d'un symbole capable de représenter une valeur constante inconnue parmi de nombreuses valeurs constantes (d'ailleurs on

parle aussi d'*inconnue* plutôt que de variable). En informatique (plus exactement dans les langages de programmation impérative comme le C), on nomme cet objet variable car sa valeur varie au cours du temps.

On peut créer une variable et l'utiliser très simplement en C :

```c
#include <stdio.h>
int main(void){
  int x;
  x = 2;
  int y = 3;
  int z = x + y;
  return 0;
}
```

Nous venons de créer 3 variables de type *entière* (`int`), les variables `x`, `y` et `z` ayant pour valeurs respectives 2, 3 et 2+3=5.

Le ligne `int x;` permet de *déclarer* la variable `x`, c'est-à-dire indiquer au programme qu'on souhaite utiliser une variable portant le nom `x` étant de type `int`. Puisqu'elle est de type `int`, elle ne pourra avoir que des valeurs entières (positives ou négatives) et non des valeurs telles que `3.14`.

La seconde ligne `x = 2;` est une *affectation* de la variable, c'est à dire qu'on lui affecte la valeur 2. Tant qu'on ne lui affecte pas une autre valeur, elle aura pour valeur 2.

La troisième ligne `int y = 3;` fusionne les deux concepts précédents. Il peut être conventionnel de déclarer toutes ses variables avant de commencer les calculs et donc d'éviter les lignes de ce genre. Cela permet de mieux organiser son programme et de le rendre plus lisible. Certains langages, comme PASCAL, forcent cette convention.

Le programme précédent n'affiche rien. Si vous souhaitez afficher une variable, il faut utiliser un peu plus en profondeur la fonction `printf`.

```c
#include <stdio.h>
int main(void){
  int x;
  x = 2;
  int y = 3;
  int z = x + y;

  // Affichage de x, y et z
  printf("x : %d\n", x);
  printf("y : %d\n", y);
  printf("z : %d + %d = %d\n", x, y, z);
  return 0;
}
```

Ce programme affiche :

```
>x : 2
>y : 3
>z : 2 + 3 = 5
```

`printf` est une fonction qui peut prendre autant de paramètres qu'on le souhaite. Le premier paramètre est le *format* qui indique à quoi doit ressembler la sortie. Une partie de ce format est indéterminée, il s'agit des symboles `%d`. Les paramètres qui suivent vont indiquer à `printf` quelle est la valeur que doit prendre cette partie indéterminée.

Il existe d'autres types de variables. On peut déjà lister 3 types principaux :

- `int` pour représenter les entiers compris entre -2147483648 et 2147483647.
- `char` pour représenter les caractères (lettres, chiffres, symboles, ...)
- `double` pour représenter (une partie finie) des réels (on parlera plutôt de *flottants* en informatique).

Chaque type a son propre mot-clef pour le format dans `printf` :

```c
#include <stdio.h>
int main(void){
  int x = 2;
  char y = 'c';
  double z = 2.23;

  // Affichage de x, y et z
  printf("x : %d\n", x);
  printf("y : %c\n", y);
  printf("z : %lf\n", z);
  return 0;
}
```

Il en existe d'autres, qui seront vu ultérieurement. En C, il est obligatoire de définir le type d'une variable. On dit qu'il est *typé statiquement*. Ce n'est pas le cas de tous les langages, Python, Php ou Ruby en sont des exemples.

Opérations arithmétiques Pour faire des calculs avec les variables, nous avons vu qu'il est possible d'additionner des variables. On peut faire plus d'opérations :

On suppose ici que `x` et `y` sont de type `int`.

- `x + y` renvoie un `int` de valeur égale à la somme des valeurs de x et y
- `x * y` renvoie un `int` de valeur égale au produit des valeurs de x et y
- `x - y` renvoie un `int` de valeur égale à la différence des valeurs de x et y
- `x / y` renvoie un `int` de valeur égale au quotient de la division euclidienne de x par y. Autrement dit $\frac{x}{y}$ arrondi à l'entier inférieur (sa partie entière).
- `x % y` renvoie un `int` de valeur égale au reste de la division euclidienne de x par y. Autrement dit `x - (x / y) * x`. **En particulier**, si la valeur de `x` est divisible par celle de `y`, alors `x % y` a pour valeur 0.

Si `x` ou `y` est de type `double`.

- `x + y` renvoie un `double` de valeur égale à la somme des valeurs de x et y
- `x * y` renvoie un `double` de valeur égale au produit des valeurs de x et y
- `x - y` renvoie un `double` de valeur égale à la différence des valeurs de x et y
- `x / y` renvoie un `double` de valeur égale à la division des valeurs de x et y.

On reviendra sur le type `char` plus loin.

4.2 Exercices

Exercice 5 — *Somme et produit*

Écrire un programme `sommeproduit.c` qui déclare et défini trois entiers x, y et z de votre choix et affiche leur somme et leur produit sur la même ligne.

Exercice 6 — *Les `int` ne sont pas des `double`.*

1. Que se passe-t-il si vous affichez une variable `x` de type `int` avec le format `%lf` ?

```c
int x = 2;
printf("%lf\n", x);
```

On peut *convertir* (on parlera de transtypage ou de cast) `x` en `double` avec `(double)x`. Que se passe-t-il avec le code suivant ?

```c
int x = 2;
printf("%lf\n", (double)x);
```

Attention `(double)x` ne transforme pas `x` en `double`, le type de `x` reste `int`. On lit juste la variable `x` comme si elle avait été déclarée en `double` avec la même valeur numérique.

2. Créer un programme `inverse.c` qui déclare, défini et affiche un `int a` > 1 de votre choix et le flottant (donc de type `double`) ayant pour valeur $\texttt{a}^{-1}$.

Exercice 7 — *Milieu*

Écrire un programme `milieu.c` qui déclare et défini quatre flottants x_1, x_2, y_1 et y_2 de votre choix et affiche sous la forme suivante les coordonnées du point (x_m, y_m) situé au milieu des points (x_1, y_1) et (x_2, y_2).

```
> Point p1 : (x1, y1)
> Point p2 : (x2, y2)
> Milieu de p1 et p2 : (xm, ym)
```

Exercice 8 — *Triangle rectangle*

Écrire un programme `triangle.c` qui déclare et défini deux entiers a et b et affiche un entier $c2$ égal au carré de l'hypoténuse d'un triangle rectangle de côtés a et b.

Exercice 9 — *Sphère*

Écrire un programme `sphere.c` qui déclare et défini un entier `r` et affiche le volume d'une sphère de rayon r égal à $\frac{4}{3} \cdot \pi r^3$. On posera $\pi = 3.14$.

5 Condition

5.1 Tutoriel

Dans la plupart des langages de programmation, il est possible d'exécuter une instruction si une condition est vérifiée et d'en exécuter une autre sinon.

C'est le mot-clef `if` qui permet cela. Par exemple le code suivant permet d'afficher `x est PAIR` si la variable entière `x` est paire et `x est IMPAIR` sinon. On rappelle que `x` est divisible par 2 si `x % 2` vaut 0.

```
1    if(x % 2 == 0){
2     printf("%d est PAIR", x);
3    }
4    else{
5     printf("%d est IMPAIR", x);
6    }
```

Tout ce qui est entre les premiers symboles { } est exécuté si `x` est pair et tout ce qui est entre les symboles { } après le `else` est exécuté sinon.

`else` n'est pas obligatoire. Il n'est pas nécessaire d'avoir un bloc `else`. Si la condition est fausse, alors il ne se passe rien. Par exemple si `x` est impair ici, il ne se passe rien.

```
1    if(x % 2 == 0){
2     printf("%d est PAIR", x);
3    }
```

Enchaîner les if. S'il y a plus que 2 possibilités vous pouvez enchaîner les conditions.

```
    if(x % 3 == 0){
      printf("%d est DIVISIBLE PAR 3", x);
    }
    else if(x % 3 == 1){
      printf("%d - 1 est DIVISIBLE PAR 3", x);
    }
    else{
      printf("%d - 2 est DIVISIBLE PAR 3", x);
    }
```

Les accolades { } ne sont pas (toujours) obligatoires. Lorsqu'il n'y a qu'une ligne entre les symboles { }, on peut les enlever. Par exemple le programme suivant fait la même chose que le premier :

```
   if(x % 2 == 0)
     printf("%d est PAIR", x);
   else
     printf("%d est IMPAIR", x);
```

Attention, c'est une source classique d'erreurs. Il est recommandé de toujours mettre les symboles { } pour éviter tout problème bête.

Et ce == alors ? Il existe en C ce qu'on appelle des opérateurs de comparaison qui permettent de comparer deux variables et de renvoyer une valeur (valant vrai ou faux). Ces opérateurs sont les suivants et fonctionnent aussi bien sur des `int` que sur des `double`.

- `x == y` renvoie vrai si `x` et `y` ont la même valeur.
- `x < y` renvoie vrai si `x` est strictement plus petit que `y`.
- `x > y` renvoie vrai si `x` est strictement plus grand que `y`.
- `x <= y` renvoie vrai si `x` est plus petit ou égal à `y`.
- `x >= y` renvoie vrai si `x` est plus grand ou égal à `y`.

Et et Ou. À ces opérateurs s'ajoute la possibilité de les associer. Par exemple

- `x == y && x == z` renvoie vrai si `x` et `y` ont la même valeur **et** si `x` et `z` ont la même valeur.
- `x == y || x == z` renvoie vrai si `x` et `y` ont la même valeur **ou** si `x` et `z` ont la même valeur.

5.2 Exercices

Exercice 10 — *Positivité*

Créer un programme `positif.c` qui affiche un entier `x` de votre choix puis affiche `POSITIF` si `x` est positif ou nul et `NEGATIF` sinon.

Exercice 11 — *Comparaison de `double`*

Attention à la comparaison de `double`,

1. Créez deux `double x` et `y` avec le programme suivant

```
      double x, y;
      x = 340282346638528859811704183484516925440.0 - 1;
      y = 340282346638528859811704183484516925440.0 - 10;
```

   Complétez le programme pour vérifier si `x` et `y` sont égaux. Que constatez vous ?

2. Créez deux `double` `x` et `y` avec le programme suivant

```
1       double x, y;
2       x = 1/34028234.0;
3       y = ((1/34028234.0) + 3) - 3;
```

   Complétez le programme pour vérifier si `x` et `y` sont égaux. Que constatez vous ?

Autant on ne peut éviter le premier problème, autant on peut éviter le second : au lieu de comparer `x` et `y` avec `x == y`, on peut vérifier si `abs(x - y)` n'est pas plus petit que 0.001.

3. Écrire un programme `compareDoubles.c` qui déclare et défini deux entiers `x` et `y` et vérifie si `abs(x - y)` est plut petit que 0.001.
4. Vous avez peut-être utilisé deux `if` dans le programme précédent. Si c'est le cas, refaite le avec un seul `if` et en utilisant les opérateurs `&&` et `||`.

6 Boucles

6.1 Tutoriel

Il existe en C, et dans la plupart des langages de programmation, deux manières de répéter un certain nombre de fois les mêmes opérations : les boucles `for` et `while`.

6.1.1 Boucle `for`

Un exemple classique de programme avec une boucle `for` est le suivant :

```
1   printf("Table de 2\n");
2   for(int i = 0; i < 5; i = i + 1){
3     printf("2 * %d = %d\n", i, 2 * i);
4   }
```

Ce programme affiche la sortie suivante :

```
>Table de 2
>2 * 0 = 0
>2 * 1 = 2
>2 * 2 = 4
>2 * 3 = 6
>2 * 4 = 8
```

On peut lire la ligne 1 du programme comme : *Pour tout i allant de 0 à 4, exécuter les instructions suivantes ...*

Remarquez la présence des accolades pour délimiter les instructions de la boucle. Vous pouvez mettre plusieurs instructions entre ces accolades sous une boucle `for`. Comme pour le mot-clef `if`, les accolades sont facultatives s'il n'y a qu'une instruction à l'intérieur. Mais il est recommandé de toujours les mettre.

Vous pouvez constater la présence du type `int` dans la boucle. C'est parce qu'on déclare la variable `i` au sein de la boucle. Il est possible de déclarer la variable avant la boucle. Cela est nécessaire si vous souhaitez déclarer toutes vos variables avant le début de votre programme.

```
1   int i
2   for(i = 0; i < 5; i = i + 1){
3     printf("2 * %d = %d", i, 2 * i);
4   }
```

Les valeurs chiffrées ne sont pas nécessairement des constantes, vous pouvez les définir comme des variables :

```
1  int i, n, m;
2  n = 0;
3  m = 5;
4  for(i = n; i < m; i = i + 1){
5    printf("2 * %d = %d", i, 2 * i);
6  }
```

Cela a bien entendu peu d'intérêt dans cet exemple, mais en aura dans les sections ultérieures.

Dernier détail, la variable `i` est ici incrémentée de 1 à chaque itération grâce à la partie `i = i + 1`. Il est possible d'augmenter `i` de plus que 1 ou de le diminuer en changeant ce morceau :

```
1  for(i = 0; i < 12; i = i + 3){
```

```
1  for(i = 12; i >= 0; i = i - 1){
```

Généralement, `i = i + 1` est écrit de manière plus compacte sous la forme `i++` ou `i += 1`. Choisissez la forme qui vous convient le mieux.

6.1.2 Boucle `while`

La boucle `while` permet de répéter des instructions comme le fait la boucle `for`.

Combien de fois pouvez-vous diviser un entier positif par 2 (arrondi à l'inférieur) avant d'atteindre 1 ? C'est la définition du log en base 2.

```
1  int n, log;
2
3  log = 0;
4  n = 28;
5  while(n > 1){
6    n = n / 2;
7    log = log + 1;
8  }
9  printf("Le log en base 2 de %d est %d.\n", n, log);
```

Ce programme affiche

```
>Le log en base 2 de 28 est 4.
```

Que fait ce programme ?

La ligne 6 indique *Tant que `n` est strictement plus grand que 1, faire ...* Tant que c'est vrai, on va diviser `n` par 2 et augmenter `log` de 1. La variable `log` compte donc le nombre de fois qu'on a divisé `n` par 2. Et on s'arrête quand on atteint 1. Ainsi `log` suit bien la définition indiquée plus haut.

Comme pour la boucle `for`, s'il n'y a qu'une seule instruction sous la boucle `while`, on peut enlever les accolades, mais il est conseillé de les garder.

Vous pouvez mettre dans une boucle `while` n'importe quelle condition que vous auriez mis avec le mot-clef `if`. Vous pouvez donc vous servir de tous les opérateurs de comparaisons et des opérateurs `&&` et `||`.

6.1.3 Différence entre une boucle `while` et une boucle `for`

Techniquement, en C, il n'y a pas de différence. Tout ce que vous pouvez faire avec une boucle `for`, vous pouvez le faire avec une boucle `while` et inversement.

Cependant, ces deux boucles existent dans pratiquement tous les langages de programmation, quand bien même elles sont redondantes. La raison est une question de lisibilité, on utilise chacune des boucles dans des situations différentes et il est conventionnel de ne pas inverser ces usages.

— Une boucle `for` est une boucle **bornée** : on connait exactement le nombre d'itérations d'une telle boucle et ce nombre ne doit pas être infini.

— Une boucle `while` est une boucle **non bornée** : on ne connait pas aisément le nombre d'itérations d'une telle boucle et ce nombre peut être infini.

Dans les exemples plus haut, on peut déduire de la ligne `for(i = 0; i < 5; i = i + 1){` qu'il y aura 5 itérations dans la boucle `for`. À l'inverse, la ligne `while(n > 1){` ne nous permet pas de savoir combien de fois on va diviser par 2 avant de s'arrêter.

Il vous est demandé dans ce cours de respecter ce principe.

6.2 Exercices

N'utilisez pas une boucle `while` si la boucle `for` est plus adaptée.

Exercice 12 — *Table de m*

Créer un programme `tablemultiplication.c` qui affiche un entier `m` de votre choix et affiche la table de multiplication de cet entier.

Exercice 13 — log *en base m*

Quelle est la définition du log en base m d'un entier n ? Créer un programme `logm.c` qui affiche deux entiers `n` et `m` de votre choix et affiche le log en base `m` de `n`.

Exercice 14 — *Somme des entiers de 1 à n*

Créer un programme `somme.c` qui affiche un entier `n` > 10 de votre choix et la somme des entiers de 1 à `n`.

Exercice 15 — *Puissances de 3*

Créer un programme qui affiche l'entier `i` tel que $\texttt{i}^3 \leq 1548 < (\texttt{i}+1)^3$.

Exercice 16 — *int*

On a vu plus haut que le type `int` représentait les entiers entre -2147483648 et 2147483647, que se passe-t-il si vous compilez et exécutez le programme suivant ?

```
1     printf("%d\n", 2147483647 + 1);
```

Créer un programme `maxPow2.c` qui, à l'aide d'une boucle, affiche l'entier 2^i tel que $2^i \leq MI < 2^{i+1}$ où MI est la plus grande valeur qu'une variable de type `int` puisse avoir.

Exercice 17 — *Rectangle*

Créer un programme `carre.c` qui connaissant un entier `n` de votre choix, affiche en console des caractères `*` organisés sous forme d'un carré de largeur et longueur n. Par exemple pour $n = 4$:

```
> ****
> *  *
> *  *
> ****
```

7 Fonctions

7.1 Tutoriel

Toute personne qui souhaite coder professionnellement et partager son code cherche avant tout à éviter la répétition de code. Le code suivant par exemple semble un peu rébarbatif :

```c
int n, s;

n = 3;
s = 0;
for(int i = 1; i <= n; i++){
  s += i * i;
}
printf("somme des carres de 1 a 3 : %d", s);

n = 10;
s = 0;
for(int i = 1; i <= n; i++){
  s += i * i;
}
printf("somme des carres de 1 a 10 : %d", s);

n = 5;
s = 0;
for(int i = 1; i <= n; i++){
  s += i * i;
}
printf("somme des carres de 1 a 5 : %d", s);
```

Il existe en programmation la notion de *fonction*, qui a plus ou moins le même objectif que les fonctions en mathématiques : définir un objet qui prend en entrée un ou plusieurs paramètres, qui effectue un ensemble d'instructions ou de calculs et qui renvoie une ou plusieurs valeurs en sortie.

Par exemple en mathématiques, la fonction log prend en entrée un réel et renvoie un réel. Attention, comme pour les variables, les fonctions en mathématiques et en informatiques sont différentes. Mais on y reviendra une autre fois (voire dans un autre cours). Pour être exact, on devrait plus souvent parler en informatique de *Procédure* plutôt que de *Fonction*, car la procédure fait mention d'un effet mécanique de calcul par une machine alors que la fonction fait plus penser à un objet abstrait.

Pour définir une fonction il faut lui donner un nom, définir ce qui est donné en entrée et ce qui est donné en sortie, puis décrire comment calculer la sortie à partir de l'entrée. Par exemple, ci-dessus, on pourrait définir la fonction `sommeCarres` et y faire appel 3 fois, ce qui évite de recopier 3 fois le même code.

```c
int sommeCarres(int n){
  s = 0;
  for(int i = 1; i <= n; i++){
    s += i * i;
  }
  return s;
}

int main(void){
  printf("somme des carres de 1 a 3 : %d\n", sommeCarres(3));
  printf("somme des carres de 1 a 10 : %d\n", sommeCarres(10));
  printf("somme des carres de 1 a 5 : %d\n",sommeCarres(5));
  return 0;
}
```

Si on décortique la fonction `int sommeCarres(int n){` :

- `sommeCarres` est le nom de la fonction.
- Cette fonction prend en entrée un entier, qui sera, dans la fonction, une variable nommée `n`.
- Elle renvoie en sortie un entier. Cette entier est calculé à l'aide de la variable `s`, puis renvoyé avec `return s;`.

Remarquez la présence des accolades. Contrairement aux accolades des conditions et des boucles, celles des fonctions sont obligatoires.

Une fonction peut prendre plusieurs paramètres, et les types des paramètres et de la sortie peuvent être différents. On peut mettre plusieurs `return` dans la fonction. Dès qu'on rencontre un `return`, la fonction s'arrête et renvoie la valeur indiquée.

```
int sommes1SurI(int n, double d){
  double s = 0.0;
  if(n <= 0)
    return -1;
  for(int i = 1; i <= n; i++)
    s += 1/(double)i;
  if(s < d)
    return 1;
  else
    return 0;
}

int main(void){
  printf("%d %lf %d\n", 30, 23.45, sommes1SurI(30, 23.45));
  printf("%d %lf %d\n", -2, 1000.0, sommes1SurI(-2, 1000.0));
  printf("%d %lf %d\n", 16, 2, sommes1SurI(16, 2));
  return 0;
}
```

Ceci renvoie

```
> 30 23.450000 1
> -2 1000.000000 -1
> 16 2.000000 0
```

On pourrait imaginer enfin de mettre l'appel de `printf` dans une fonction, pour éviter de recopier cet appel 3 fois :

```
int sommes1SurI(int n, double d){
  double s = 0.0;
  if(n <= 0)
    return -1;
  for(int i = 1; i <= n; i++)
    s += 1/(double)i;
  if(s < d)
    return 1;
  else
    return 0;
}

void compareSommes1SurI(int n, double d){
  printf("%d %lf %d\n", n, d, sommes1SurI(n, d));
}

int main(void){
  compareSommes1SurI(30, 23.45);
  compareSommes1SurI(-2, 1000.0);
  compareSommes1SurI(16, 2);
  return 0;
}
```

Remarquez le type `void` renvoyé par la fonction `compareSommes1SurI`. Ce type signifie que la fonction ne renvoie rien. Elle se contente de faire un calcul et de l'afficher ou de modifier un état. Ce

type de fonction n'existe pas en mathématique. C'est ce qu'on entend par *Procédure* généralement.

Il est aussi possible pour une fonction de ne rien prendre en entrée. C'est le cas pour la fonction `main`. Le `void` explicite le fait qu'elle ne prenne aucun argument en entrée.

Il n'est pas possible pour une fonction en C de renvoyer plusieurs paramètres.

7.2 Exercices

Exercice 18 — *Gravité*

Dans un fichier `gravite.c`, créer une fonction qui, connaissant trois `int` $m1$, $m2$ et d, renvoie un `double` égal à la valeur de la force de gravité entre deux corps de masses $m1$ et $m2$ éloignés d'une distance d égale à $G \cdot m1 \cdot m2/d^2$, avec $G = 6.67 \cdot 10^{-11}$.

Exercice 19 — *Cylindre*

Dans un fichier `cylindre.c`, créer 3 fonctions. Une fonction qui, connaissant un `double r`, calcule le périmètre d'un cercle de rayon `r`. Une fonction qui, connaissant un `double r`, calcule l'aire d'un cercle de rayon `r`. Et une fonction qui, connaissant deux **double** `r` et `h` **affiche** l'aire et le volume d'un cylindre de rayon `r` et de hauteur `h`. L'aide vaut $2\mathcal{A}(r) + \mathcal{P}(r) \cdot h$ et le volume vaut $\mathcal{A}(r) \cdot h$ où $\mathcal{A}$ et $\mathcal{P}$ calculent respectivement l'aire et le périmètre d'un cercle. On prendra $\pi = 3.14$.

Exercice 20 — *Retour arrière*

Créez un fichier `boucles.c` avec 2 fonctions. Une fonction qui connaissant un entier `n` affiche la table de multiplication de `n` ; et une fonction qui connaissant deux entiers `n` et `d` renvoie le logarithme en base `d` de `n`.

Exercice 21 — *Formule usuelles de mathématique*

Il existe de nombreuses fonctions mathématiques déjà codées. Vous pouvez les utiliser. Pour ce faire, il y a 2 modifications à apporter à vos fichiers :

- ajouter `#include <math.h>` en haut de votre fichier.
- ajouter `-lm` à la fin de la commande pour compiler avec `gcc` ; ce qui donne `gcc -Wall -Wextra -std=c99 -o sortie file.c -lm`

Vous pouvez maintenant appeler des fonctions de la bibliothèque de mathématiques : `https://fr.wikipedia.org/wiki/Math.h`.

Dans un fichier `nlogn.c`, créer une fonction qui, connaissant un entier `n`, renvoie $\texttt{n}\ln(\texttt{n})$.

Exercice 22 — *Triangle*

Créer un fichier `triangle.c` contenant une fonction qui, connaissant un entier n, affiche en console des caractères `*` organisés sous forme d'un triangle isocèle de hauteur n, de base de taille $2n - 1$ et pointant vers le bas. Par exemple pour $n = 4$, le programme dervait afficher

```
> *******
>  *   *
>   * *
>    *
```

Exercice 23 — *Norme 1*

Créer un fichier `norme1.c` contenant une fonction qui, connaissant un entier n, affiche en console des caractères `*` organisés sous forme d'un losange de rayon n (affiche toutes les `*` dont la distance en norme 1 au centre est inférieure ou égale à 4). Par exemple pour $n = 3$, le programme dervait afficher

```
>    *
>   ***
>  *****
> *******
>  *****
>   ***
>    *
```

8 Caractère et chaîne de caractères

8.1 Tutoriel

On l'a vu plus haut, le type `char` représente un caractère, c'est-à-dire un symbole de texte, lettre ou chiffre ou ponctuation ou autre. Un caractère s'écrit entre guillemets **simples**. Il existe en C 256 caractères manipulables.

```
1     char c = 'a';
2
```

Une chaîne de caractères est un type de données représentant une suite de caractères ; dit autrement, une chaîne de caractères est une manière de représenter simplement du texte. Vous en avez déjà vu et manipulé sans le savoir tout au long de ce tutoriel. Une chaîne de caractères s'écrit entre guillemets **doubles** et s'utilise par exemple avec la fonction `printf`. Le type associé à la chaîne de caractères est `char*` ou `char[]`. Il existe une différence entre les deux. Nous y reviendrons plus loin.

Une première manière simple de générer une chaîne de caractères est de le faire en dur dans le texte :

```
1     char* text = "O blas bougriot glabouilleux\nTes micturations me
      touchent\nComme des flatouillis slictueux\n";
```

La manipulation des chaînes de caractères passe par des opérations de bases et des fonctions de la bibliothèque `string.h`. Deux manipulations basiques : accéder au i^{e} caractère de la chaîne et connaître sa taille.

```
1     #include <stdio.h>
2     #include <string.h>
3
4     int main(void){
5       char* slogan = "Le cheval c'est trop genial.\n";
6       char c;
7       c = slogan[0]; // Renvoie 'L', le 1e caractere de la chaine
8       c = slogan[3]; // Renvoie 'c', le 4e caractere de la chaine
9       int s = strlen(slogan); // Renvoie la taille de la chaine
10      c = slogan[s - 1]; // Renvoie '\n' le dernier caractere de la
      chaine
11      c = slogan[s]; // Renvoie '\0' un caractere special situe en
     fin de chaine
12      c = slogan[s + 1]; // ERREUR (probablement)
13      c = slogan[-3]; // ERREUR (probablement)
14      return 0;
15    }
```

Comme vous pouvez le constater

— `\n` est un caractère à part entière, ce ne sont pas 2 caractères distincts.

— le caractère spécial `\0` suit le dernier caractère de la chaîne. Cela permet de savoir où se trouve la fin de la chaîne, en particulier on peut recoder la fonction strlen en cherchant ce caractère. Si vous essayez d'afficher ce caractère en console, vous ne verrez rien.

— vous ne devez pas accéder à un caractère au delà de la fin de la chaîne ou en deçà du début. Le comportement n'est pas spécifié, le programme peut continuer à fonctionner ou crasher, on ne sait pas. Il serait très malvenu de parier que le programme ne crasherait pas.

Afficher un caractère ou une chaîne de caractères

Pour afficher un caractère ou une chaîne de caractères, il faut utiliser, comme pour tout autre donnée la fonction `printf`. Dans le format, les symboles `%c` et `%s` correspondent respectivement aux caractères et aux chaînes.

```
1  char* s1 = "le cheval";
2  char* s2 = "c'est trop genial\n";
3  printf("%s %s %s", s1, s1, s2);
4
5  printf("La %d-ieme lettre de l'alphabet est %c.\n", 3, 'c');
6
```

qui affiche

```
>le cheval le cheval c'est trop genial
>La 3-ieme lettre de l'alphabet est c.
>
```

Modifier une chaîne

Pour modifier un caractère d'une chaîne, il suffit d'utiliser l'instruction suivante :

```
1      s[3] = 'a';
2
```

Attention, cette instruction ne fonctionne pas tout le temps. Vous ne **pouvez pas** modifier une chaîne qui a été initialisée dans le code sous la forme suivante :

```
1      char* s1 = "le cheval";
2      s1[2] = 's' // ERREUR
3
```

Par contre, si vous l'avez initialisé avec le type char[] cela fonctionne :

```
1      char s1[100] = "le cheval";
2      s1[2] = 's' // ok
3      printf("%s\n", s1); // Affiche "lescheval".
4
```

Il existe d'autres méthodes, on y reviendra plus tard en cours si le temps le permet.

Créer des chaînes contenant d'autres valeurs

La fonction `printf` qui permet d'afficher un texte formaté (print format) a un pendant relatif aux chaînes de caractères, la fonction `sprintf`.

```c
#include <stdio.h>

int main(void){
  int n = 3;
  char c = 'c';

  char s[100];
  sprintf(s, "La %d-ieme lettre de l'alphabet est %c.", n, c);
  printf("La chaine s est %s\n");
}
```

```
>La chaine s est La 3-ieme lettre de l'alphabet est c.
```

Remarquez qu'on a dû préciser ici le type de `s` sous la forme char[] plutôt que char*. C'est une manière d'initialiser une chaîne de caractères vide contenant au plus 100 caractères (dont le caractère spécial `\0` qu'on a vu tout à l'heure). On peut le faire avec char* mais on verra ça ultérieurement.

8.2 Exercices

Exercice 24 — *Duplicat*

Créer un programme `duplicat.c` qui contient une fonction prenant en entrée deux chaînes de caractères `s` et `t` et renvoie en sortie une chaîne contenant `s` puis `t`. Créer ensuite une seconde fonction, utilisant la première, prenant en entrée une chaîne `s` et un entier `n` et renvoie une chaîne contenant `n` fois de suite la chaîne `s`.

Exercice 25 — *affichage*

Créer un programme `printf2.c` qui contient une fonction prenant en entrée une chaîne de caractères `s` et qui, à l'aide d'une boucle `for` et de la fonction `strlen`, affiche chaque caractère de `s` dans l'ordre sur des lignes distinctes. Faire une seconde fonction qui a le même comportement, mais qui utilise une boucle `while` et qui n'utilise pas la fonction `strlen`.

Exercice 26 — *Oublions le \0*

Créons une chaîne à partir de rien ainsi :

```c
      char s1[10];
      s1[0] = 'a'
      s1[1] = 'b'
      s1[2] = 'c'
      s1[3] = 'd'
      printf("%s\n", s1);

```

Que se passe-t-il ? A votre avis pourquoi, comment peut-on y remédier ?

Exercice 27 — *palindrome*

Créer un programme `palindrome.c` qui contient deux fonctions. La première reçoit une chaîne de caractères et vérifie si cette chaîne est un palindrome. Si c'est le cas elle renvoie 1 et sinon elle renvoie 0. La seconde reçoit une chaîne de caractères `s` et en renvoie une autre `s2` en sortie. La chaîne `s2` est un palindrome formé en mettant à la suite `s` et la chaîne `s` renversée.

Exercice 28 — *Chaîne entière*

Créer un programme `isnumber.c` qui contient une fonction qui, connaissant une chaîne de caractère, vérifie si cette chaîne est un entier (donc constitué uniquement de numéros). Vous pouvez utiliser la fonction `isdigit(char c)` qui renvoie 1 si `c` est un chiffre et 0 sinon.

Pour pouvoir utiliser `isdigit`, vous devez ajouter `#include <ctype.h>` au début du fichier.

9 Tableaux

Un tableau est une manière de représenter une liste ordonnée d'objets. Par exemple, on pourrait représenter le tableau suivant sous forme d'un tableau en c :

| 7 | 3 | 32 | 21 | 2 | 7 | 6 |

Il est possible de faire 2 actions avec un tableau : récupérer la valeur du case et modifier la valeur d'une case. On peut donc voir un tableau comme un ensemble de variables.

Voici un exemple d'utilisation de tableau :

```c
int tab[7];
tab[0] = 7;
tab[1] = 3;
tab[2] = 32;
tab[3] = 21;
tab[4] = 2;
tab[5] = 7;
tab[6] = 6;
for(int i = 0; i < 7; i = i + 1){
  printf("tab[%d] = %d\n", i, tab[i]);
}
```

qui afficherait

```
> tab[0] = 7
> tab[1] = 3
> tab[2] = 32
> tab[3] = 21
> tab[4] = 2
> tab[5] = 7
> tab[6] = 6
```

Une chaîne de caractères est un tableau comme les autres contenant des caractères, avec la spécificité de se terminer par le symbole spécial `\0`. Il est donc normal de retrouver des similitudes entre ce qu'il est possible de faire avec un tableau et une chaîne de caractères.

Un tableau est une variable comme les autres, qui doit être déclarée puis définie. Lorsque vous déclarez un tableau, vous définissez sa taille et son type. Un tableau ne peut pas contenir 2 types différents. Si vous initialisez un tableau d'entiers, il ne contiendra que des entiers.

Attention, lorsque vous initialisez un tableau, par exemple avec `int tab[7];`, son contenu n'est pas spécifié, il peut contenir n'importe quoi. De même, vous avez le droit de remplir un tableau partiellement, mais les cases non remplies ne contiennent aucune valeur spécifique. Le programme suivant est valide. Par contre si vous tentez d'accéder à la case `tab[1]`, la valeur que vous obtiendrez n'est pas spécifiée.

```c
int tab[7];
tab[0] = 7;
tab[2] = 32;
tab[3] = 21;
```

Ainsi, quand vous créez une chaîne de caractères avec la commande `char s[100] = "abcde";`, vous créez un tableau de 100 char où seules les 6 premières cases sont définies (en comptant le caractère `\0` à la fin). Toutes les autres cases peuvent contenir n'importe quoi.

Pourquoi faire une chose pareille plutôt que de toujours définir la taille exacte du tableau ? Tout simplement parce qu'on ne connait pas toujours la taille exacte du tableau. Supposons que vous écriviez un programme qui prend en entrée un entier et renvoie un tableau contenant la liste des diviseurs de cet entier. Si c'est un autre utilisateur qui vous donne cet entier, vous ne pouvez pas deviner la taille du tableau à l'avance. Vous pouvez alors déclarer un tableau dont la taille est une grande valeur constante, en espérant que cette valeur soit supérieure à la taille requise, et si ce n'est pas le cas, vous renvoyez une erreur. Ou vous pouvez déclarer un tableau de taille variable *Variable lenght Array ou VLA*.

Depuis que la norme C99 existe, vous pouvez déclarer les tableaux ainsi :

```
1     int tab[n];
```

où `n` est un entier. Ça fonctionne mais votre programme est (un peu) plus lent. En effet, si, dans votre code, il est écrit `int tab[100];`, on connaît la taille du tableau en lisant le code. Si vous écrivez `int tab[n];`, on ne connait cette taille que lorsqu'on exécute le code. Donc dans le premier cas, dans l'idéal, le tableau pourrait être déjà créé au moment de la compilation ; et pas dans le second cas, ce qui ralenti son exécution.

Quelle que soit la solution que vous choisissiez, il n'est pas dit que la taille que vous déclariez soit la taille effective du tableau. Si on reprend l'exemple de la liste des diviseurs, vous ne connaissez pas le nombre de diviseurs d'un entier avant de les avoir calculé, donc avant d'avoir créé et rempli le tableau. Vous savez qu'il y a moins de `n / 2` diviseurs de `n` (probablement beaucoup moins). Vous pouvez donc initialiser le tableau avec une taille au moins égale à `n / 2`. Mais vous n'utiliserez pas toutes les cases du tableau. Sa taille effective est donc moins élevée que prévu. Autrement dit, il faudrait avoir créé le tableau pour connaître sa taille effective.

Comment connaître la taille effective d'un tableau ? C'est impossible. Deux solutions envisageables : ajouter à la fin une valeur spéciale, comme c'est le cas des chaînes de caractère. Par exemple pour les diviseurs, vous pouvez terminer par `-1`, puisqu'on considère généralement uniquement les diviseurs positifs. Autre solution, vous pouvez, dans le programme, toujours associer la variable `tab` à une autre variable entière `size` qui contiendra la taille du tableau. Une dernière solution serait de mettre la taille du tableau dans le tableau, mais ça ne fonctionne que s'il s'agit d'un tableau d'entier.

On verra dans la suite une manière plus élégante de régler ce problème.

9.1 Exercices

Exercice 29 — *Appartenance*

Créer un programme `tableau.c` contenant une fonction qui, connaissant un entier `size`, un tableau `tab` contenant `size` entiers et un entier `m`, renvoie 1 si `m` appartient au tableau `tab`.

Exercice 30 — *Dichotomie*

Dans le fichier `tableau .c`, ajoutez une fonction qui, connaissant un entier `size`, un tableau `tab` contenant `size` entiers triés et un entier `m`, effectue une recherche dichotomique de `m` dans le tableau.

Exercice 31 — *Passage par valeur et passage par référence, comment renvoyer un tableau ?*

Qu'affiche le programme suivant ?

```c
void f(int x){
  x = 1;
}
void g(int t[]){
  t[0] = 1;
}
int main(void){
  int x = 3;
  f(x);
  int t[1];
  t[0] = 3;
  g(t);
  printf("%d\n", x);
  printf("%d\n", t[0]);
}
```

Dans le fichier `tableau.c`, ajouter une fonction qui, connaissant un entier `size` et deux tableaux `tab` et `tab2` contenant `size` entiers, écrase le contenu de `tab2` avec le contenu `tab` inversé.

Dans le fichier `tableau.c`, ajouter une fonction qui, connaissant un entier `size`, un tableau `tab` contenant `size` entiers, écrase `tab` avec son propre contenu inversé.

Remarque : En C, renvoyer un tableau est complexe. On va donc éviter de le faire.

10 Entrée, Sortie, Erreur, argc, argv

Vous pouvez imaginer un programme comme une grosse fonction, il peut prendre des paramètres en entrées et envoyer des sorties. Il existe plusieurs moyens pour un programme de communiquer avec l'extérieur. On va s'intéresser ici à comment l'utilisateur pourrait décrire les entrées du programme, comment le programme pourrait décrire des sorties. On ne s'intéressera pas aux fichiers ici.

10.1 Tutoriel : les sortie et erreur standards

Vous connaissez une première forme de communication avec l'extérieur : la fonction `printf` qui écrit dans la console. La console ne fait pas partie du programme, la console est votre interface avec le programme.

On dit que `printf` écrit dans *la sortie standard*, il s'agit d'un canal existant dans la majorité des langages de programmation (très probablement la totalité). Lorsque vous exécutez un programme, l'outil que vous utilisez pour lancer ce programme dispose généralement d'un moyen de lire ce canal pour utiliser tout ce qu'il voit dessus, qui consiste à afficher ce qui passe sur ce canal.

Il existe une seconde sortie standard, nommée *erreur standard*. Ce canal n'a aucune différence avec la sortie standard, excepté son nom. Conventionnellement, il est utilisé pour décrire des erreurs, alors que la sortie standard est utilisée pour décrire des données normales.

La fonction suivante écrit une erreur dans l'erreur standard si l'entier `n` est négatif, et décrit la parité de `n` en sortie standard sinon.

```c
#include <stdio.h>
void parite(int n){
  if(n < 0){
    fprintf(stderr, "L'entier %d est negatif.", n);
  }
  else if(n % 2 == 0){
    printf("L'entier %d est pair.", n);
  }
  else{
    printf("L'entier %d est impair.", n);
  }
}
```

Le mot-clef `stderr` est le canal d'erreur standard. De même, vous pouvez utiliser `stdout` pour écrire sur le canal de sortie standard. Les fonctions `printf(...` et `fprintf(stdout, ...` font la même chose.

10.2 Tutoriel : indiquer des entrées au programme

De même qu'il existe une sortie et une erreur standard, il existe une *entrée standard*. On utilise en C la fonction `scanf` pour lire l'entrée standard. De manière équivalente, vous pouvez écrire `fscanf(stdin, ...)` pour lire l'entrée standard.

Si vous écrivez et exécutez le programme suivant, vous remarquerez qu'il se bloque :

```c
#include <stdio.h>
int main(void){
  int x;
  scanf("%d", &x);
  printf("Vous avez ecrit %d.\n", x);
}
```

Vous devez alors écrire dans la console un entier qui sera lu en entrée standard.

```
> machin:~ ./testScanf
> 42
> Vous avez ecrit 42.
```

Il faut bien noter dans cet exemple que le premier 42 n'a pas été écrit par le programme mais par vous.

Vous pouvez mettre plus d'un entier.

```c
#include <stdio.h>
int main(void){
  int x, y, z;
  scanf("%d %d %d", &x, &y, &z);
  printf("Vous avez ecrit %d %d %d.\n", x, y, z);
}
```

Vous pouvez mettre un format.

```c
#include <stdio.h>
int main(void){
  int x;
  scanf("Hey %d", &x);
  printf("Vous avez ecrit %d.\n", x);
}
```

Mais attention à respecter le format :

```
> machin:~ ./testScanf
> Hey 1
> Vous avez ecrit 1.
> machin:~ ./testScanf
> 1
> Vous avez ecrit 32767.
```

Vous pouvez aussi donner des doubles et des caractères.

```c
#include <stdio.h>
int main(void){
  double x;
  char c;
  scanf("Hey %lf %c", &x, &c);
  printf("Vous avez ecrit %lf %c.\n", x, c);
}
```

Et les chaînes de caractères ?

C'est compliqué ! `scanf` ou `fscanf` s'attendent à un format. `scanf("%s", s);` ne fonctionnera pas nécessairement comme vous le souhaitez, notamment avec les espaces. Le programme peut également attendre car il n'a pas moyen de savoir si vous avez terminé de remplir le chaîne de caractères. Un autre comportement, par exemple avec le programme suivant, est assez inattendu :

```c
#include <stdio.h>
int main(void){
  while(1){ // Repete la boucle indefiniment.
    char x[100];
    scanf("Nom : %10s", x);
    printf("Vous avez ecrit %s.\n", x);
  }
}
```

```
> machin:~ ./testScanf
> Nom : abc
> Vous avez ecrit abc.
> Vous avez ecrit abc.
> Vous avez ecrit abc.
> Vous avez ecrit abc.
> Vous avez ecrit abc.
> ...
```

Des explications ici :

`https://stackoverflow.com/questions/16882489/why-does-scanf-get-stuck-in-an-infinite-loop-on-invalid-input`

Bref, comment faire ? Vous pouvez : utiliser la fonction `fgets`.

```c
#include <stdio.h>
int main(void){
  while(1){ // Repete la boucle indefiniment.
    char x[100];
    fgets(x, 100, stdin);
    printf("Vous avez ecrit %s.", x);
  }
}
```

10.3 Arguments d'entrée

Votre programme peut prendre des arguments en entrée sans utiliser l'entrée standard.
Par exemple :

```c
#include <stdio.h>
int main( int argc, char* argv[] )  {
  printf("Nombre d'arguments : %d\n", argc);
  for(int i = 0; i < argc; i = i + 1){
    printf("%s\n", argv[i]);
  }
}
```

```
> machin:~ ./testArgs 1 3 2
> Nombre d'arguments : 4
> ./testArgs
> 1
> 3
> 2
> machin:~ ./testArgs abc
> Nombre d'arguments : 2
> ./testArgs
> abc
> machin:~ ./testArgs
> Nombre d'arguments : 1
> ./testArgs
>
```

Remarquez qu'il y a toujours au moins un argument, le nom du programme lui-même.

Problème : tous les arguments sont des chaînes de caractère. Comment faire pour transformer une chaîne en entier ou en double ?

Vous pouvez utiliser la commande `sscanf` qui fonctionne exactement comme la fonction `scanf`. Au lieu de lire une entrée standard formatée elle lit une chaîne de caractères formatée.

```c
#include <stdio.h>
int main( int argc, char* argv[] )  {
  char* p1 = argv[1];
  char* p2 = argv[2];
  int x, y;
  sscanf(p1, "%d", &x);
  sscanf(p2, "%d", &x);
  printf("%d + %d = %d\n", x, y, x + y);
}
```

```
> machin:~ ./argsSum 1 2
> 1 + 2 = 3
```

Il existe d'autres méthodes, bien entendu.

Dernier détail. Autant en parler ici, il existe une fonction `ssprintf` pour remplir une chaîne de caractères avec un certain format.

10.4 Exercices

Exercice 32 — *Arithmétique, version standard*

Écrire un programme `arithmstd.c` qui déclare deux entiers x et z, les définit à partir de l'entrée standard affiche leur somme, leur produit, leur différence et leur rapport sur la même ligne.

Modifier ce programme pour afficher une erreur si la division ne peut être calculée.

Exercice 33 — *Réciproque de Pythagore*

Écrire un programme `pythareci.c` qui prend 3 paramètres `a`, `b` et `c` en entrée (avec `argc` et `argv`). En supposant que ces paramètres soient entiers, le programme doit indiquer s'il existe un triangle rectangle de côté `a`, `b` et `c`.

Modifier ce programme pour afficher une erreur s'il n'y a pas exactement 3 paramètres en entrée. Le modifier ensuite pour vérifier si les 3 paramètres sont des entiers (rappel : un exercice dans la partie sur les chaînes de caractères concerne ce point).

Exercice 34 — *Votre première ligne de commande*

Écrire un programme `afficher.c` qui prend entre 0 et 4 paramètres en entrée. Ces paramètres peuvent être :

— `-v`

— `-x` suivi d'un entier `N`

— `-h`

Ce programme a pour simple but d'afficher un entier n.

Si `-x N` est donné, le programme considère que $n =$ `N`. Sinon il considère que $n = 1$.

Si `-v` est donné, le programme affiche `"Voici un entier: `n`"`. Sinon il affiche simplement `"`n`"`.

Si `-h` est donné, le programme n'affiche pas l'entier et affiche à la place un message d'aide pour expliquer comment fonctionne le programme (vous pouvez recopier l'énoncé).

Par exemple : `./afficher.c -v -x 3` affiche `"Voici un entier : 3"`.

Le programme doit vérifier qu'aucun paramètre n'est donné en double et qu'aucun mauvais paramètre n'est donné. Dans ce cas, il doit afficher une erreur. Vous pouvez pour cela créer une fonction pour vérifier si deux chaînes sont égales et réutiliser la fonction permettant de savoir si une chaîne est un entier.

11 Structure

11.1 Tutoriel

Les structures permettent de créer des types complexes en associant d'autres types.

Par exemple, vous voulez représenter un rationnel, une fraction $\frac{p}{q}$ où p et q sont entiers. Vous pouvez utiliser le type `double` mais ce type est moins précis que celui des `int`. On pourrait donc préférer d'avoir deux entiers pour représenter cette fraction plutôt qu'un `double`.

On peut écrire le programme suivant :

```
1   #include <stdio.h>
2
3   struct rat{
4     int p;
5     int q;
6   }
7
8   int main()  {
9     struct rat r;
10    r.p = 3;
11    r.q = 2;
12  }
```

Le type `struct rat` peut maintenant être utilisé comme n'importe quel type. Attention, le type n'est pas `rat` mais bien `struct rat`. Il peut être donné en entrée et en sortie de fonction. Il ne peut par contre pas être donné tel quel aux fonctions `printf`, `scanf`, ...

Grâce aux structures, on peut associer un tableau avec sa taille effective. Dans l'exemple suivant, le type `struct tabsize` contient un tableau avec au plus 100 entiers, et un autre entier représentant sa taille effective (inférieure à 100).

```c
#include <stdio.h>

struct tabsize{
  int size;
  int tab[100];
}

int main()  {
  struct tabsize t;
  t.size = 3;
  for(int i = 0; i < t.size; i++){
    t.tab[i] = 0;
  }
}
```

11.2 Exercices

Exercice 35 — *Complexe*

Dans un fichier `complexes.c`, créer une structure `struct complexe` avec 2 entiers `p` et `q` représentant le complexe `p + i · q`.

- Créer une fonction qui affiche un nombre complexe.
- Créer une fonction qui additionne deux nombres complexes.
- Créer une fonction qui calcule l'opposé d'un nombre complexe.
- Créer une fonction qui calcule le conjugué d'un nombre complexe.
- Créer une fonction qui calcule le produit de deux nombres complexes.
- Créer une fonction qui calcule le module d'un nombre complexe.
- Créer une fonction qui calcule l'argument d'un nombre complexe.
- Créer une fonction qui, connaissant le module et l'argument, renvoie le nombre complexe associé.

Exercice 36 — *L'école, version IPI*

Les membres de l'école peuvent être soit des étudiants, soit des administratifs, soit des enseignants-chercheurs.

Un étudiant est caractérisé par son nom et son numéro d'inscription.

Un administratif est caractérisé par son nom et sa catégorie administrative (A, B ou C).

Un enseignant-chercheur est caractérisé par son nom et le nom du laboratoire de recherche auquel il est rattaché.

Dans un fichier `personnel.c`,

- À l'aide d'une structure, définissez un type `personne` pour représenter un membre de l'école. Réflechissez à comment faire pour intégrer, dans une seule structure, les 3 types possibles de personne ?
- Écrivez une fonction `listing` qui prend un tableau de personnes en argument et sa taille et rend la liste des noms de tous les membres de l'école.
- Écrivez une fonction `separe` qui prend un tableau de personnes et sa taille en argument et qui affiche d'une part la liste des noms des étudiants, et d'autre part la liste des noms des administratifs et des enseignants-chercheurs.
- Écrivez une fonction `effectifs` qui prend un tableau de personnes en argument et qui calcule le nombre de personnes pour chacune des trois catégories.

12 Pointeur

12.1 Tutoriel

Un pointeur est un type de donnée. Il s'agit d'une adresse mémoire.

Dans la partie sur les variables, nous avons vu qu'une variable peut être représentée comme une boîte dont on peut regarder et modifier le contenu. Lorsque, dans le programme, on fait appel à la variable `x`, on regarde le contenu de la boîte `x`. Mais où se trouve cette boîte ? On parle alors de l'adresse de la variable.

Concrètement, une adresse est un entier. Deux variables différentes ont des adresses différentes (sinon il s'agirait de la même boîte). Un pointeur est une variable contenant une adresse. On peut donc imaginer un pointeur comme une boîte contenant une boîte, mais la représentation classique est plutôt la suivante :

La flèche signifie que la valeur du pointeur est l'adresse de `x`.

Voici un exemple en C pour créer un tel pointeur.

```c
#include <stdio.h>

int main(void){
  int* px;
  int x = 2;
  px = &x;
}
```

Le pointeur `px` pointe sur la variable `x`.

Pour déclarer un pointeur en C, il faut déclarer vers quel type de variable il pointe. On utilise pour cela la notation `*`, que vous avez déjà aperçu avec `char*` pour les chaînes de caractères. On peut donc avoir des `int*` ou des `double*` ou même des `struct rat *`.

La notation `&x` que nous avons utilisée pour la fonction `scanf` signifie *l'adresse de* `x`. En écrivant `px = &x`, on écrit bien *dans la boîte du pointeur* `px`*, on range l'adresse de* `x`.

Il existe une notion duale de l'adresse qui est le *déréférencement*. C'est à dire accéder à la variable pointée par un pointeur.

```c
#include <stdio.h>

int main(void){
  int* px;
  int x = 2;
  px = &x;
  *px = 3;
  printf("%d\n", x);
}
```

Ce programme affiche

```
> 3
```

L'opération `*px =` change la valeur de la variable pointée par `px`.

On représente généralement les choses ainsi :

A quoi cela peut-il bien servir ? Si l'exemple précédent vous montre comment utiliser un pointeur, il n'était pas nécessaire d'utiliser ce pointeur pour changer la variable `x`. La commande `x = 3;` aurait fait la même chose.

Dans un exercice sur les tableaux, on a vu que les fonctions en C utilisait la notion de *passage par valeur*. Cela signifie qu'il est impossible de modifier une *variable* passée en paramètre. On a vu que c'était possible pour les tableaux, mais ce n'est pas le tableau lui-même qui est modifié c'est son contenu. On appelle ça le *passage par référence*. On peut faire pareil avec un pointeur. On verra en cours qu'il existe une forte similitude entre un pointeur et un tableau, mais que ces deux notions sont bien distinctes en C et qu'il ne faut pas les confondre.

```c
#include <stdio.h>

void incrVal(int xt){
  xt = xt + 1;
}

void incrRef(int* pxt){
  *pxt = *pxt + 1;
}

int main(void){
  int x = 2;
  incrVal(x);
  printf("%d\n", x);
  incrRef(&x);
  printf("%d\n", x);
}
```

```
> 2
> 3
```

Note : on a nommé les variables en entrée des fonctions ***`incrVal`*** *et* ***`incrRef`*** *ainsi pour les distinguer facilement de* ***`x`*** *et* ***`px`****. Mais on aurait très bien pu les nommer aussi* ***`x`*** *et* ***`px`****.*

Les deux fonctions semblent faire la même chose mais seule la seconde a un effet. Le passage par référence (le fait de fournir le contenant d'une variable plutôt que la variable elle-même en entrée) est le seul moyen en C de modifier une variable dans une fonction.

Que se passe-t-il concrètement ?

Quand on appelle la fonction `incrVal(xt);`, ce n'est pas la variable `x` qui se retrouve en entrée de la fonction, c'est sa valeur. Pour vous en convaincre, on aurait pu faire ça :

```c
  int x = 2;
  int y = 3;
  incrVal(x + y);
```

On comprend bien ici que `incrVal(x + y);` donne la valeur $2 + 3$ en entrée et non une variable. Sinon, quelle variable aurait été augmentée de 1 ? `x` ? `y` ? `x` et `y` ? Aucune de ces réponses n'aurait de sens.

Donc que se passe-t-il à l'exécution ? Lors de l'appel de la fonction `incrVal`, on crée une nouvelle variable temporaire, dont le nom est `xt` et donc la valeur est une copie de la valeur donnée en entrée. La variable `xt` n'est donc pas la même variable que la variable `x`. Modifier la première ne modifie donc pas la seconde.

Donc que se passe-t-il lors de l'appel de `incrRef` ? La même chose. Mais nous n'avons pas utilisé le même type de données. On a donné une adresse en entrée et pas un entier. Au moment de l'appel de la fonction, on crée une nouvelle variable temporaire, dont le nom est `pxt` et dont la valeur est une copie de la valeur donnée en entrée. Cette valeur est une adresse, l'adresse de la variable `x`. De même que pour `incrVal`, la variable `pxt` de la fonction n'est pas la même que la variable `px` qui a servi a appeler la fonction. Donc, même conclusion, modifier la première ne modifie pas la seconde. Pourquoi cela fonctionne-t-il alors ? Parce qu'on ne modifie pas la variable

`pxt`, on modifie la variable pointée par `pxt`, notée `*pxt`. Or l'adresse de `x` ne change jamais. Les deux variables `px` et `pxt` pointent sur la même adresse, donc elles pointent toutes les deux sur `x`. En modifiant `*pxt`, on modifie donc bien la variable `x`.

Attention, à l'inverse, la fonction suivante ne fait rien de plus que la fonction `incrVal`, elle ne modifie jamais `x`.

```
1    void incrRefRate(int* pxt){
2      int xt = *pxt + 1;
3      pxt = &xt;
4    }
```

12.2 Tutoriel : malloc et free

Dans cette partie on va s'intéresser à comment renvoyer un pointeur ?

On peut afficher un pointeur avec la fonction `printf` et le format `%p`. Par afficher un pointeur, on entend afficher l'adresse pointée par ce pointeur. Essayez maintenant de faire tourner le code suivant :

```
1    #include <stdio.h>
2
3    int* adress(int x){
4      return &x;
5    }
6
7    int main(void){
8      int x = 2;
9      int* px = &x;
10     int* px2 = adress(x);
11
12     printf("%p\n", px);
13     printf("%p\n", px2);
14   }
```

Vous devriez voir quelque chose comme ça :

```
> 0x4ffc2239
> (nil)
```

`0x...` est une adresse (écrite en hexadécimal). `(nil)` est aussi une adresse, celle du néant. Autrement dit `adress(x)` ne pointe sur rien. Selon les cas, `(nil)` peut aussi s'écrirre `NULL` ou `0` ou encore `0x0000000`.

Pourtant le pointeur créé par la fonction `adress` doit bien avoir une valeur. Pour s'en convaincre, rajoutons un `printf`.

```c
#include <stdio.h>

int* adress(int x){
  int* px = &x;
  printf("%p\n", px);
  return px;
}

int main(void){
  int x = 2;
  int* px = &x;
  int* px2 = adress(x);

  printf("%p\n", px);
  printf("%p\n", px2);
}
```

```
> 0x4ffc2498
> 0x4ffc2514
> 0x4ffc2514
```

Le résultat est différent. On a bien une adresse, mais cette adresse est différente de la première adresse affichée. Cela s'explique simplement par le fait que les variables `x` dans la fonction `main` et dans la fonction `adress` ne sont pas les mêmes. Donc leurs adresses sont différentes.

Le `(nil)` du premier programme peut s'expliquer par le fait que la variable `x` de la fonction `adress` est temporaire. Cette variable n'existe plus une fois la fonction close. Donc `&x` ne pointe plus sur rien une fois la fonction close. Plus exactement l'adresse contenue dans ce pointeur ne correspond plus à rien. L'exécuteur préfère donc renvoyer un pointeur sur rien, donc `printf` affiche `(nil)`.

Note du rédacteur : mon intuition est que dans la variable où on renvoie ***px****, le compilateur n'a pas détecté qu'on renvoyait un pointeur sur une variable qui allait disparaître alors que dans la version où on renvoie* ***&x****, il est évident que ce pointeur ne pointe plus sur rien, on peut donc sans crainte remplacer l'adresse retournée par rien.*

Ça n'a pas l'air bien grave. Changeons un peu le programme :

```c
#include <stdio.h>

int* adress(int x){
  return &x;
}

int main(void){
  int x = 2;
  int* px2 = adress(x);
  *px2 = 3;
}
```

```
> Erreur de segmentation (core dumped)
```

Nous en arrivons à la Némésis des programmeur en C, l'erreur de segmentation. Il s'agit d'une erreur qui vous annonce simplement que *Le programme a tenté d'écrire là où il n'avait pas le droit d'écrire, je préfère tout fermer et partir en pleurant. Bisous.* Aucune explication ? Même pas une petite ligne pour indiquer où l'erreur a eu lieu ? Rien ? Non, rien !

Vous ne pouvez pas utiliser un pointeur qui pointe sur une adresse inutilisable. Au mieux vous aurez cette erreur. Ce n'est pas le pire : le programme suivant compile et s'exécute sans erreur.

```c
#include <stdio.h>

int* adress(int x){
  int* px = &x;
  return px;
}

int main(void){
  int x = 2;
  int* px2 = adress(x);
  *px2 = 3;
}
```

Il n'y a pas d'erreur mais vous écrivez n'importe où. Ce n'est pas mieux.

Comment y remédier ? Nous allons utiliser la fonction `malloc`.

La fonction `malloc` est une fonction de `stdlib.h`, il faut donc inclure en haut du programme `#include <stdlib.h>`

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

int* notAdress(int x){
  int* px = (int*) malloc(sizeof(int));
  *px = x;
  return px;
}

int main(void){
  int x = 2;
  int* px2 = notAdress(x);
  printf("%p", px2);
}
```

`malloc` crée un pointeur et vous garanti que le contenu de l'adresse pointée ne sera pas effacé à la fin de la fonction, il ne sera jamais effacé sans votre accord avant la fin du programme.

Remarquez que j'ai changé le nom de la fonction, ce qui est renvoyé n'est pas du tout un pointeur contenant l'adresse de `x`*. C'est juste un pointeur sur une variable contenant la même valeur que* `x`*.*

Pour effacer un pointeur créé avec `malloc`, il faut utiliser la fonction `free`. On dit que le pointeur a été libéré.

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

int* adress(int x){
  int* px = (int*) malloc(sizeof(int));
  *px = x;
  return px;
}

int main(void){
  int x = 2;
  int* px2 = adress(x);
  printf("%p", px2);
  free(px2);
}
```

Une bonne pratique de programmation consiste à toujours écrire la libération d'un pointeur en même temps que le `malloc` associé.

Pourquoi faire des `free` ? Observons le programme suivant :

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

int main(void){
  while(1){
    int* px = (int*) malloc(sizeof(int));
    scanf("%d", px);
    printf("%p %d", px, *px);
  }
}
```

À chaque itération, le programme demande un entier à l'utilisateur. Cet entier est rangé à l'adresse pointée par `px`. Puis on affiche l'adresse et l'entier. Et on recommence.

À chaque itération, on réserve donc une nouvelle case pour le contenu de `px` qui ne sera jamais effacée. Mais la mémoire de votre ordinateur est limitée. Au bout d'un moment, le programme va s'arrêter car vous n'en aurez plus assez pour réserver une nouvelle case. En utilisant `free`, plus de problème. Le programme suivant peut tourner à l'infini.

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

int main(void){
  while(1){
    int* px = (int*) malloc(sizeof(int));
    scanf("%d", px);
    printf("%p %d", px, *px);
    free(px);
  }
}
```

12.3 Modifier une chaîne

On a vu dans la section des chaînes de caractères qu'il est possible de modifier une chaîne sauf si elle a été initialisée avec `char* c = "...";`.

Il fallait alors utiliser le type `char c[] = "...";` à la place. Mais parfois, on a besoin d'un pointeur plutôt que d'un tableau de `char`. Une technique consiste à créer le `char*` avec un malloc puis à le remplir. Plutôt que de le remplir case par case, ce qui peut être fastidieux, on peut utiliser la fonction `strcpy`.

```
#include <stdio.h>
#include <string.h>
...
char * c = char *a = malloc(256 * sizeof(char));
strcpy(a, "Ceca est une chaine modifiable!");
c[3] = 'i';
...
free(c);

```

12.4 Tutoriel : pointeur sur plusieurs valeurs

Pourquoi utiliser `malloc` et `free` ? Pour pouvoir renvoyer des pointeurs à l'aide de fonction ? Pourquoi faire ? Il existe sans doute de très nombreuses réponses dont la première serait *et pourquoi pas ?*, mais une bonne raison est de pouvoir renvoyer facilement plusieurs valeurs en même temps sans utiliser de structure ni de tableau.

Reprenons la syntaxe de la fonction `malloc` :
`int* px = (int*) malloc(sizeof(int));`
`malloc` renvoie un pointeur. Le `(int*)` qui est devant est nécessaire pour rappeler au compilateur qu'on range le résultat dans un pointeur sur entier et non un pointeur sur autre chose.
`sizeof(int)` est la taille en mémoire de ce qui est pointé par le pointeur. On parlera de mémoire en cours mais sachez que `sizeof` vous renvoie la taille de n'importe quel type : `sizeof(double)`, `sizeof(char)`, `sizeof(char*)`, ... tout fonctionne.

On peut donc faire ça :
`int* px = (int*) malloc(2 * sizeof(int));`
Ce qui peut se lire comme *Soit `px` un pointeur sur 2 entiers.* On peut accéder au premier entier avec `*px` et au second avec `*(px + 1)`. Une syntaxe plus compacte et plus proche de la version tableau est `px[0]` et `px[1]`.

On peut aussi faire ça :
`int* px = (int*) malloc(n * sizeof(int));`
où `n` est une variable entière. Ce qui peut se lire comme *Soit `px` un pointeur sur n entiers.* On peut accéder au i^e entier avec `px[i - 1]`.
Donc si on veut une fonction qui renvoie la liste des `n` entiers entre 0 et `n - 1`, on peut le coder comme ça :

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

int* nPremiers(int n){
  int* nPremiers = (int*) malloc(n * sizeof(int));
  for(int i = 0; i < n ; i = i + 1)
    nPremiers[i] = i;
  }
  return nPremiers;
}

int main(void){
  int n = 10;
  int * p = nPremiers(n);
  free(p);
}
```

Remarquez que l'appel à `free` ne change pas, même si on pointe sur plusieurs éléments, il ne faut faire qu'un `free`. Pour être exacte : 1 `malloc` ⇒ 1 `free`.
Dernière remarque : si le pointeur sur plusieurs valeurs ressemble à un tableau, ce n'est pas exactement la même notion.

12.5 Exercices

Exercice 37 — *Echange*

Dans un programme `echange.c`, créer une fonction qui prend en entrée 3 entiers a, b et c. À l'issue de la fonction,

— b a la valeur que a avait au début de la fonction
— c a la valeur que b avait au début de la fonction
— a a la valeur que c avait au début de la fonction

Exercice 38 — *Initialisation, Fin*

Dans un programme `initfin.c`, créer une fonction qui crée et renvoie un pointeur sur un entier égal à 0, puis une autre fonction qui prend un pointeur en entrée et libère la mémoire associée à ce pointeur.

Exercice 39 — *Min et max*

Dans un programme `minmax.c`, créer une fonction qui prend en entrée un entier n, un tableau `tab` contenant `n` entiers et 2 pointeurs sur entiers `min` et `max`. Cette fonction doit mettre à l'adresse pointée par `min` et `max` respectivement la plus petite et la plus grande valeur des entiers contenu dans `tab`. La fonction ne renvoie rien.

Exercice 40 — *Liste de log*

Dans un programme `loglist.c`, créer une fonction qui prend en entrée un entier n et crée et renvoie un pointeur p sur n flottant et initialise `p[i]` avec $\log(i)$.

Exercice 41 — *Pointeur sur pointeur*

Comment feriez vous pour représenter le type pointeur sur pointeur sur entier ?

Dans un programme `pointeursType.c`, créer une fonction qui prend en entrée un entier n et un pointeur l sur n entiers. Renvoyez en sortie un pointeur sur pointeur d'entiers p représentant deux listes d'entiers. La première liste contient le nombre d'entiers pairs pointés par l, suivi de la liste de ces entiers. La seconde liste contient le nombre d'entiers impairs pointés par l, suivi de la liste de ces entiers.

Exercice 42 — *Structure*

Dans un programme `date.c`, créez une structure `date`. Une date est définie par le jour, le mois et l'année.

Créer une fonction qui prend en entrée une date et l'affiche.

Créer une fonction qui prend en entrée un pointeur sur `struct date` et qui incrémente la date de un jour. Par soucis de simplicité, on ignorera les années bissextiles.

On utilisera la notation `->` : soit un type `struct s` possédant l'attribut `a`, si `x` est une variable de type `struct s`, on peut accéder à l'attribut `a` de `x` avec `x.a`. Soit maintenant un pointeur `p` sur `x`, alors on peut accéder à `x.a` avec `(*p).a`. Il existe une notation pour améliorer la lisibilité : `p->a`.

Exercice 43 — *Retour au tuto*

1. Le code n'est pas correct, pourquoi ?

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

int* adress(int x){
  int* px = (int*) malloc(sizeof(int));
  px = &x;
  return px;
}

int main(void){
  int x = 2;
  int* px2 = adress(x);
  printf("%p", px2);
  free(px2);
}
```

2. Le code n'est pas correct, pourquoi ?

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

char* hello(){
  char* px = "hello";
  return px;
}

int main(void){
  char* s = hello();
  printf("%s", s);
  free(s);
}
```

13 Récursivité

La récursivité est la possibilité pour une fonction de s'appeler elle-même.

Reprenons notre exemple de logarithme en base 2. Le logarithme en base 2 d'un entier est le nombre de fois qu'il faut le diviser par 2 (arrondi à l'inférieur) pour arriver à 1.

Donc si on a $y = \lfloor \frac{x}{2} \rfloor$, par définition, on vérifie $\log_2(x) = \log_2(y) + 1$; sauf si $x = 1$ auquel cas $\log_2(x) = 0$. On a donc une formule de récurrence. Cette formule peut être codée à l'aide d'une fonction récursive.

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

int log2(int x){
  if(x == 1)
    return 0;
  else
    return log2(x / 2) + 1;
}

int main(void){
  printf("%d\n", log2(10));
}
```

L'avantage des fonctions récursives est qu'elles sont (généralement) plus lisibles voire plus simples à coder que les fonctions qui font la même chose sans récursivité (dites généralement les fonction *itératives*).

Il faut toutefois être prudent avec les fonctions récursives. En particulier, êtes-vous certains que la fonction s'arrête ? Dans le programme ci-dessus, la fonction s'arrête si `x` ≥ 1. Mais si $x \leq 0$, la fontion boucle à l'infini. Un simple test permet d'éviter ce cas. Une autre solution serait de préciser quelque part à l'utilisateur qu'il ne faut pas appeler cette fonction avec un entier négatif ou nul.

Un autre risque des fonctions récursives est de faire trop d'appels récursifs. On y reviendra.

13.1 Exercices

Exercice 44 — *Factorielle*

Ecrire un programme `fact.c` qui contient une fonction qui connaissant un entier `n` calcule sa factorielle récursivement. On rappelle que $n! = \prod_{i=1}^{n} i$.

Exercice 45 — *Dichotomie Récursive*

Ecrire un programme `dichoRec.c` qui contient une fonction qui connaissant un entier `min`, un entier `max` supérieur à `min`, un tableau `tab` contenant au moins `max` entier triés par ordre croissant et un entier `x`, vérifie si `x` est présent dans le tableau entre le `min`$^{\text{e}}$ élément et le `max - 1`$^{\text{e}}$ élément. Cette fonction doit être récursive.

Exercice 46 — *Remplacement de caractères*

Ecrire un programme `replace.c` qui contient une fonction qui connaissant une chaîne de caractères `s`, deux caractères `c` et `d` et un entier `i` remplace tous les caractères `c` situés après le `i`$^{\text{e}}$ caractère de la chaîne `s` par `d`. Cette fonction doit être récursive. (Attention à bien initialiser votre chaîne de caractères avec un tableau ou un `malloc`).

Exercice 47 — *Des maths !*

Ecrire un programme `exp.c` qui contient une fonction qui connaissant un `double` $0 < \texttt{x} < 1$ calcule $\log(\texttt{x}+1)$ comme suit :

Poser $l = 0$. Pour tout i de 1 à l'infini, si $\frac{z^i}{i} < 0.001$, renvoyer l. Sinon $l = l + \frac{z^i}{i}$.

Coder cette fonction sous forme itérative puis sous forme récursive. On pourra ajouter des paramètres à la fonction récursive si nécessaire.

14 Un peu de tout

Exercice 48 — *Jeu de mémo*

Le jeu de mémo consiste à retrouver les paires de tuiles identiques dans une grille carrée 2D.

Initialement, les tuiles sont retournées faces cachées. À son tour, le joueur choisit deux tuiles faces cachées, consulte leurs valeurs et :

— soit il s'agit d'une paire et le joueur marque un point, replace les tuiles faces visibles et rejoue ;

— soit il ne s'agit pas d'une paire, auquel cas les tuiles sont replacées faces cachées et on passe au joueur suivant.

Le jeu s'arrête lorsque toutes les paires ont été découvertes, et le joueur ayant le plus de points remporte la partie.

Ici, les valeurs des tuiles seront représentées par des chaînes de caractères. Pour modéliser le jeu, nous allons utiliser deux grilles :

— une grille `tuiles` contenant les valeurs de chaque tuile

— une grille `jeu` où chaque aura pour valeur `"*"` si la tuile correspondante est face cachée, et la valeur de la tuile si celle-ci est face visible.

Ces grilles seront de taille `SIZE` × `SIZE` où `SIZE` est une constante (variable globale).

— Écrire une fonction d'initialisation pour la grille `jeu`.

— Écrire une fonction d'initialisation pour la grille `tuiles`.

On demandera à l'utilisateur de fournir $\texttt{SIZE}^2/2$ chaînes de caractères. Pour chaque chaîne s, on placera aléatoirement deux tuiles de valeur s dans deux emplacements vides.

— Écrire une fonction qui demande deux cases à l'utilisateur et affiche (si possible) les valeurs des tuiles correspondantes.

Si les entrées sont valides et correspondent à des tuiles faces cachées, la fonction retournera 0. Sinon, la valeur de retour sera 1.

— Écrire une fonction simulant un tour de jeu. Cette fonction renverra 0 si le joueur courant marque un point et doit rejouer, et 1 sinon.

— Écrire une fonction permettant de simuler une partie entière à deux joueurs.