TRAVAUX PRATIQUES I

Introduction à C

Si vous êtes sur une machine personnelle, faites tout de même la partie A sur une machine du lycée.

L'objectif de ce TP est d'apprendre les bases de la programmation en C : fonctions, types `int` et `bool`, structures `if` et `while`. Vous aurez besoin d'aller chercher sur Nuage le dossier associé au TP. Voici un raccourci vers Nuage (il ne marchera pas éternellement, je vous encourage à noter le lien complet sur une clef usb ou en marque-page) :

`https://link.infini.fr/tp-info-cg`

1. Télécharger les fichiers du TP. Si vous avez téléchargé un zip, vous pouvez les dézipper à l'aide de la commande `unzip` .

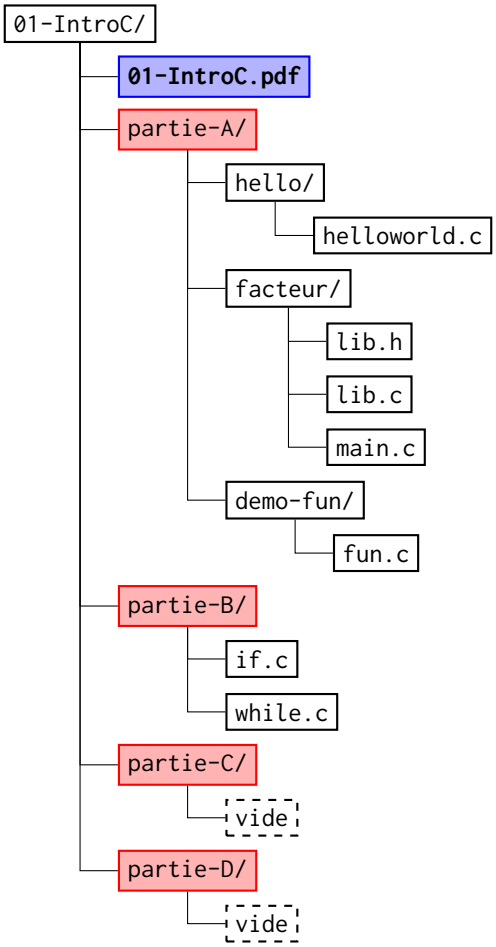

FIGURE I.1 – Organisation du dossier du TP

Convention 1 (Guillemets méta).
J'utilise les chevrons `<...>` comme des guillemets qu'il ne faut pas recopier dans les commandes ou dans le code. Il faut remplacer leur contenu par ce que l'on veut y mettre.

Sur les machines du lycée, nous allons utiliser `gedit` (aussi appelé « éditeur de texte ») pour éditer du texte. Il est bien pour débuter. [1] [2]

1. Vous pouvez utiliser `nano` si vous le souhaitez.
2. Si vous utilisez un éditeur plus avancé, *n*'utilisez *pas* le gros bouton magique qui compile pour vous.

A Fondamentaux

A.1 Compilation

Définition 2 (gcc, introduction).
Pour *compiler* un ou plusieurs fichiers C en un programme exécutable, on utilise la commande suivante :

```
gcc <fichier0.c> <fichier1.c> ... <fichierX.c>
```
Terminal

L'exécutable créé est nommé `a.out` . On le lance via la commande `./a.out` . Pour donner un nom à l'exécutable, on ajoute l'option `-o` :

```
gcc <fichier0.c> ... <fichierX.c> -o <prgm.exe>
```
Terminal

Pour avoir aider au débugage, on peut demander au compilateur des avertissements avec `-Wall` (Warning `all`) `-Wextra` (**W**arning **extra**) :

```
gcc <fichier0.c> ... <fichierX.c> -o <prgm.exe> -Wall -Wextra
```
Terminal

Un et un seul des fichiers compilés doit contenir une fonction `main` (cf suite du TP).

2. Allez dans le dossier `IntroC/partie-A/hello/`. Compilez le fichier `helloworld.c` . Testez le programme créé : que fait-il ?
3. Allez dans le dossier `IntroC/partie-A/facteur/`. Compilez ensemble les fichiers `lib.c` et `main.c` . Testez le programme créé : que fait-il ?

A.2 Fonctions

Commençons par présenter la syntaxe de base d'une fonction en C. Voici un exemple typique :

partie-A/demo-fun/fun.c

```c
int produit(int a, int b) {
  return a*b;
}
```

Déchiffrons :

- Le premier mot-clef est `int` : il indique la nature de la sortie (`int` signifie « entier »).
- Ensuite vient le nom de la fonction : `produit` .
- Ensuite, entre parenthèses, les entrées de la fonction et leur nature. Elles sont séparées par des virgules.

On peut ensuite appeler cette fonction. Ainsi, `produit(5,3)` est l'entier renvoyé par cette fonction avec 5 et 3 comme entrées.

Définition 3 (Fonctions C).
Voici la forme générale d'une fonction C :

```c
<type_de_sortie> <nom_de_fonction>(<type0 arg0>, <type1 arg1>, <...>) {
  <corps_de_la_fonction>
}
```

L'instruction pour renvoyer est `return <sortie>;` .
Tous les arguments qui ne sont pas des tableaux ou des pointeurs[i] sont passés par valeur.

Pour appeler une fonction, on écrit `<identifiant>(<arg0>, <arg1>, <...>)` .

i. Cf plus tard dans l'année

Remarque. Les retours à la ligne et l'indentation [3] ne sont pas obligatoires *mais* sont très fortement recommandés pour la lisibilité du code. Pour indenter, on utilise la touche Tabulation [4]

4. Allez voir la fonction `carre` de `fun.c`. Quelles sont ses entrées et leurs types ? Quel est le type de sa sortie ? Modifiez ensuite son corps pour qu'elle renvoie bien le carré d'un entier.
   NB : les opérateurs d'addition, soustraction, multiplication et division [5] sont bel et bien `+` `-` `*` et `/` .

A.3 Commentaires

Il est très important qu'un code soit lisible. Pour cela, il est utile d'ajouter des **commentaires** : des explications en français qui l'expliquent à la personne qui relit, mais qui sont ignorés par le compilateur (et ne font donc pas partie du programme créé).

Définition 4 (Commentaires).
Les commentaires C débutent par `/*` et terminent par `*/` . Ils peuvent s'étendre sur plusieurs lignes.
On peut aussi faire des commentaires qui débutent par `//` et terminent par un retour à la ligne. Ils sont donc limités à au plus une ligne.

5. Retournez dans le fichier `helloworld.c`, et ajoutez des commentaires (peu importe ce qu'ils disent : vous n'avez pas à comprendre ce code pour l'instant). Vérifiez que le code compile encore et que le programme n'a pas changé de comportement.

A.4 Point-virgule

Toutes les instructions d'un code C se terminent par un `;` . C'est pour cela que vous en voyez à la fin des `return`, par exemple.

Notez que les structures (fonctions, if-then-else, boucles) ne sont pas des instructions mais des structures et n'ont donc pas besoin de `;` final. Similairement, les directives d'inclusion (`#include`, que nous verrons plus tard) n'en ont pas besoin non plus.

A.5 Variables

Définition 5 (Variables C).
Pour définir une nouvelle variable en C, on utilise `<type> <identifiant> = <valeur_initiale>;` . Le type d'une variable est sa nature.
Pour modifier une variable existante, on utilise `<identifiant> = <nouvelle_valeur>;` .

Exemple.

```c
int x = 0;
x = 3*x+1;
int y = (4*x)*2 -5;
```

À la fin de ce code, `x` vaut 1 et `y` vaut 3.

6. Modifiez la fonction `ex_def4` pour y implémenter l'exemple de la définition 4 du chapitre 0 du cours. Vous renverrez `x` à la fin de la fonction.
   Si vous n'avez pas fait NSI en 1ère, appelez-moi pour validation avant de continuer.
7. Crééez une fonction `polynome` qui calcule $x \mapsto \left\lfloor \frac{x^7+2x^3-2x+3}{x^6+1} \right\rfloor$. Il est recommandé d'utiliser des variables pour stocker des réusltats intermédiaires et rendre le code plus lisible. Modifiez ensuite la fonction `main` pour y ajouter un test de `polynome` (vous pouvez vous contenter de copier/coller et adapter le test de `carre`).
   L'opérateur `/` fait une division entière, c'est à dire qu'il renvoie la partie entière inférieure de la division.
   Il n'existe pas d'opérateur de puissance en C.
   Pour vous aider à déboguer, voici un tableau des images de ce polynôme :

$-3 \mapsto -3$	$1 \mapsto 2$
$-2 \mapsto -2$	$2 \mapsto 2$
$-1 \mapsto 1$	$3 \mapsto 3$
$0 \mapsto 3$	$4 \mapsto 4$

3. Le décalage horizontal du corps.
4. au-dessus de la touche « Verr. Maj. », à gauche de « A », et en dessous de « ² » sur un calvier azerty standard.
5. Division entière, c'est à dire quotient.

Mentionnons une notion sur laquelle on reviendra plus tard. Ne vous prenez pas la tête avec maintenant, je la mets ici principalement au cas où vous auriez des ”bugs”liés à cela :

Définition 6 (Portée).

- Une variable définie dans un corps est dite **locale**. Elle n'existe plus à la fin de ce corps. Si c'est le corps d'une boucle, elle n'existe plus à la fin de l'itération (et est recréé au début de l'itération suivante).
  Les arguments d'une fonction sont des variables locales au corps de la fonction.
- Une variable qui n'est pas définie dans un corps est dite **globale**. Elle existe de sa déclaration jusqu'à la fin du fichier.
- Aucune variable n'existe avant sa déclaration.
- Il n'est pas possible de créer plusieurs variables ayant le même nom dans le même corps. Si en imbriquant des corps (par exemple des Si dans des Si) on créé plusieurs variables ayant le même nom, seule la plus « récente » est accessible.

B Structures conditionnelles `if` et boucle `while`

Faire ce TP sur sa machine, ou chez soi

Si vous avez Linux (ou MacOS) installé : utilisez le terminal comme sur les machines du lycée. Sinon, on va utiliser un compilateur en ligne : `https://www.onlinegdb.com/online_c_compiler` .

Je vous conseille de coder sur votre machine puis de copier/coller dans onlinegdb pour compiler. Cliquez que « Run » ou sur la touche « F9 » pour compiler et exécuter.

Cela ne vous dispense pas de devoir savoir utiliser le terminal et compiler dans celui-ci.

B.1 Structure `if`

Définition 7 (`if`).
Voici comment traduire un saut conditionnel en C :

si *valeur logique* **alors**
| corpsDuAlors
sinon
| corpsDuSinon

Pseudo-code

```c
if (<valeur logique>) {
  <corpsDuAlors>
} else {
  <corpsDuSinon>
}
```

Code C

Si le corps du `else` est vide, on peut simplement ne pas écrire `else { <...> }` .

Remarque. Là aussi, les retours à la ligne et indentations sont optionnels mais très fortement recommandés.

La valeur logique à mettre dans le `if` est généralement d'un test, par exemple `b > a` .

Elle peut aussi être une variable de type `bool`, qui contient le résultat d'un test (cf plus loin). Nous verrons également dans cette section comment faire le ET, le OU, etc de plusieurs valeurs logiques.

Définition 8 (Comparaisons).
Voici les opérateurs que l'on peut utiliser pour comparer des nombres :

Opérateur	Rôle
`==`	Égalité.
`!=`	Non-égalité.
`<`	Inférieur strict.
`<=`	Inférieur large.
`>`	Supérieur strict.
`>=`	Supérieur large.

Exemple.

partie-B/if.c

```c
/** Renvoie le maximum entre a et b */
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

8. Modifiez le code de la fonction `affiche_est_majeur` dans le fichier `partie-B/if.c` afin qu'il affiche si la personne entrée est majeure ou non.
   Appelez moi si vous ne comprenez pas comment répliquer l'affichage.
9. On dit qu'une année est bissextile si elle est multiple de 4 mais pas de 100. Les années multiple de 400 sont des exceptions et sont toujours bissextiles.
   Modifiez le code de `est_bissextile` pour qu'il renvoie `true` si une année est bissextile et `false` sinon. Vous pouvez utiliser l'opérateur `%` : `x % y` est le reste de la division euclidienne de `x` par `y`. Ainsi, pour tester si une variable `x` est divisible par 4, il suffit de tester `x % 4 == 0`.
   Les valeurs logiques Vrai/Faux sont des valeurs qui existent au même titre que les entiers 0, 1, 2, etc. « Vrai» s'écrit `true` *. « Faux » s'écrit* `false` *.*
   Vous utiliserez des `if` *imbriqués et non des opérateurs logiques.*
10. Modifiez `affiche_est_majeur` en mettant `true` comme valeur logique dans son `if`. Expliquez le nouveau comportement du programme. Idem avec `false`.

B.2 `while`

Définition 9 (`while`, `break`).
Voici comment traduire une boucle à précondition en C :

tant que *valeur logique* **faire**
  corpsDuTantQue

Pseudo-code

```c
while (<valeur logique>) {
  <corpsDuTantQue>
}
```

Code C

L'instruction `break;` permet de quitter immédiatement la boucle en cours.

11. Dans `partie-B/while.c`, codez une fonction nommée `td1a` qui implémente le « calcul de 42-i sans soustraction » de l'exercice 1 du TD d'introduction.
    Vous devrez peut-être modifier le test de `td1a` *dans la fonction* `main`.
12. Dans `main`, écrivez une boucle qui fait des tests pour la fonction `sup_log2` . Vous pouvez par exemple regarder les valeurs de 0 à 20. Expliquez ce que fait `sup_log2` (sans mentionner de logarithme ;)).

13.
- Déboguez la fonction `is_prime` de `while.c`.
- En cours de maths, vous prouverez qu'au lieu de tester tous les $d \in [\![2; n[\![$, il suffit de tester tous les $d \in [\![2; \sqrt{n}]\!]$. En conséquence, modifiez `is_prime` pour que si $d^2 > n$, on quitte la boucle immédiatement à l'aide d'un `break`.

C Autour de la notion de types

Les valeurs que l'on manipule sont *typées* : elles ont une nature. Nous avons beaucoup utilisé `int` jusqu'à présent, le type des entiers.

Lorsque l'on déclare une variable, il faut indiquer le type de la variable. On ne pourra ensuite mettre *que* des valeurs de ce même type dans cette variable. On dit que le langage C est un **langage typé**. On verra plus tard qu'il est **faiblement typé**.

Lorsque l'on passe un argument à une fonction, il faut que cet argument ait le type attendu : si une fonction demande en entrée un entier `int` et que vous passez `3.14`, cela ne marchera pas.

C.1 `void`

`void` est le type du « rien ». On l'utilise dans la déclaration de fonctions, soit comme type de sortie (pour indiquer que la fonction ne renvoie rien [6]), soit comme type d'entrée (pour indiquer que la fonction ne prend pas d'arguments).

C.2 `bool`

Définition 10 (Booléen).
Une valeur logique est appelé un **booléen** et se note `bool`. Il n'y a que deux valeurs possibles : Vrai et Faux, notés `true` et `false` en C.
Pour utiliser ce type, il faut mettre `#include <stdbool.h>` au début du fichier où l'on programme.

Remarque.
- Ce n'est pas parce qu'il n'y a que deux valeurs possibles qu'il booléen est forcément défini comme `bool b = true;` ou `bool b = false;`. On peut tout à fait écrire `bool b = 3 == 2;`, ou encore `bool b = <fonction>(<trucs>);`
- Le nom provient de George Boole, un des fondateurs de l'étude mathématiques du calcul logique.

Définition 11 (Opérateurs logiques).
Voici les opérateurs que l'on peut utiliser pour combiner des booléens :

Opérateur	Rôle	Explication
`&&`	Et logique.	`a && b` est Vrai si et seulement si à la fois `a` et `b` sont Vrais.
`||`	Ou logique.	`a || b` est Vrai si et seulement si `a` et `b` ne sont pas tous les deux Faux.
`!`	Négation logique.	`!a` est Vrai si et seulement si `a` est Faux.
`^`	Ou exclusif.	`a ^ b` est Vrai si et seulement si soit `a` est Vrai soit `b` est Vrai mais pas les deux à la fois.

Nous verrons en cours des priorités entre ces opérateurs.

Remarque. Notez que comme les tests d'égalité, différence, etc sont des valeurs logiques, ce sont des booléens ! On peut donc les combiner avec ces opérateurs. Par exemple : `(0 <= x) && (x < 100)`.

6. Ne rien renvoyer n'est pas ne rien faire d'utile : il peut y avoir des effets secondaires !

14. Copiez `partie-B/if.c` dans `partie-C/`. Refaites la question sur les années bissextiles, mais sans utiliser de `if` imbriqués. Vous utiliserez à la place les opérateurs ci-dessus.
Pour rappel, en terminal, on peut copier un ou plusieurs fichiers avec `cp` :

```
cp <fichier0> <fichier1> <...> <fichierX> <dossier_destination/>
```
Terminal

Remarque. Les opérateurs `==` et `!=` fonctionnent sur tous les types, y compris les booléens. Cependant, ils sont presque inutiles sur les booléens ! En effet, `b == true` a la même valeur logique que `b` lui-même et on peut donc simplement écrire `b`. De même, `b != true` a la même valeur logique que `!b`.

15. De même, réécrivez `a == false` et `a != false`.

Remarque. **Préférez toujours les écritures plus booléennes que l'on vient de trouver plutôt que == et != sur des booléens.** En effet, faire le contraire est souvent perçu comme révélateur d'une mauvaise compréhension des booléens (on ne comprendrait pas que `b` est une valeur logique qui peut donc être utilisée dans un `if` ou `while`, on penserait à la place que seul `b == true` en a une).

C.3 `int`

Vous avez déjà vu les opérateurs des entiers dans ce TP :

- Les opérateurs d'arithmétique `+`, `-`, `*`, `/` (division *entière* !) et `%`.
- Les opérateurs de comparaisons `==`, `!=`, `<`, `<=`, `>`, `>=`.

Mentionnons en plus que sur laplupart des oridnateurs, `int` ne permet pas de travailler avec tous les entiers mais uniquement avec $[\![-2^{31}; 2^{31}[\![$.

16. Rappelez la différence entre `=` et `==`. Faites attention à ne pas les mélanger !

C.4 Autres types

Nous verrons d'autres types durant l'année. À titre indicatif, j'en donne ici la liste :

- (Tableaux)
- Autres types d'entiers relatifs : `int8_t`, `int16_t`, `int32_t`, `int64_t`.
- Types d'entiers naturels : `unsigned int`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `size_t`.
- Type des lettres `char`, et des suites de lettres (les textes) `char*`.
- Type des nombres dyadiques [i]
- Pointeurs `<type>*`.
- Variantes non-modifiables avec `const`.
- Structures `struct`.
- Plus si affinités.

i. « dyadique » = comme décimal, mais en base 2. C'est à dire les nombres à virgule qui ont une écriture finie en base 2.

D Fonctions `main`, `printf` et `scanf`

D.1 `main`

Définition 12 (main).

- La *seule* fonction lancée à l'exécution du programme est la fonction nommée `main` .

  Par conséquence, un code que l'on compile doit avoir une et une seule fonction nommée `main` .
- Elle *doit* avoir comme type de sortie `int` : la valeur qu'elle renvoie est communiquée au terminal, c'est un code indiquant si le programme a bien fonctionné ou non. Ces deux codes sont respectivement `EXIT_SUCCESS` et `EXIT_FAILURE`. Pour les utiliser, le début du fichier doit commencer par `#include <stdlib.h>`.
- On peut la définir comme ayant :
  - 0 entrées : son prototype d'entrée est alors `int main(void)` .
  - 1 entrée : cette entrée est obligatoirement un `int`, qu'il vaut mieux nommer `argc`. Cet valeur est le nombre de mots qui ont été tapés sur la ligne de commande qui a lancé le programme (**arg**ument **c**ount).

    Son prototype d'entrée est alors `int main(int argc)` .
  - 2 entrées : la première entrée est `argc`. La seconde est `argv`, un tableau de chaines de caractères[i]. Les cases du tableau contiennet en fait les mots de la ligne de commande qui a lancé le programme (**arg**ument **v**alue).

    Son prototype d'entrée est alors `int main(int argc, char* argv[])` .
  - Pas d'autres possibilités.

Pour l'instant, nous irons au plus simple : `int main(void)` .

i. Nous verrons cela plus tard dans l'année, c'est trop compliqué pour l'instant

D.2 `printf`

Vous devriez déjà savoir utiliser `printf`, plus ou moins. Voici des compléments qui pourront servir durant l'année. Il n'est pas nécessaire de connaitre le tableau par coeur.

Définition 13 (printf).
La fonction `printf` permet d'afficher dans le terminal. Elle prend un argument : un texte délimité par des guillemets doubles `"` qui est le texte à afficher.
Si l'on veut afficher des valeurs/variables dans ce texte, il faut mettre des **spécificateurs** dans le texte qui correspondent au type :

Spécificateur	Type
`%d` ou `%i`	Entier (affiché en base 10)
`%h`	Entier (affiché en base 16)
`%u`	Entier naturel (base 10)
`%lf`	Nombre à virgule (type `double`)

Spécificateur	Type
`%c`	Caractère (type `char`)
`%s`	Texte (type `char*`)
`%p`	Pointeur
`%ld` et `%lu`	comme `%d` et `%u` mais pour des entiers longs (> 32 bits)

Pour chaque spécificateur que l'on place dans le texte à afficher, il faut ajouter en argument (après le texte) la valeur correspondant (dans le même ordre que les spécificateurs). Par exemple :

```c
printf("Bla %d %c", 10, 'l');
```

Remarque.

- Si l'on veut faire un retour à la ligne, on place `\n` dans le texte là où l'on veut faire le retour.
- `printf` est une fonction très spéciale en cela qu'elle prend un nombre variable d'arguments selon le contenu de son premier argument (le texte). On parle de *fonction variadique*. C'est un peu compliqué.
- Il n'y a pas de spécificateur de booléens. Utilisez `%d` , qui affichera `0` pour `false` et `1` pour `true`.
- ⚠ **Ne confondez JAMAIS**[7] **la sortie (`return`) d'une fonction avec un affichage (`printf`).** ⚠

7. JAMAIS !!

La sortie communique une valeur au reste du code, l'affichage est un effet secondaire qui modifie des pixels à l'écran.

17. Dans `partie-D/` , créez un programme qui affiche « Il est 7h48. ». Le 7 et le 48 ne seront pas écrit directement dans le texte (on va mettre des spécificateurs `%d` dans le texte à la place, et passer les nombres en arguments.)
Comparez ce qu'il se passe si vous mettez ou non un `\n` à la fin du texte à afficher.

Remarque. Nous verrons plus tard que `printf` peut faire plus qu'afficher sur le terminal. En fait, il *écrit* sur le terminal, et on peut le faire écrire ailleurs... par exemple dans un fichier que l'on veut modifier.

D.3 scanf

`printf` permet d'écrire sur le terminal. `scanf` est son symétrique et permet de lire ce que l'utilisateur tape dans le terminal.

Son fonctionnement est un peu technique, aussi je vais me contenter d'exemples. Dans l'idée, c'est le symétrique de `printf` :

- `scanf(" %d", &var);` attend que l'utilisateur entre un entier dans le terminal puis le stocke dans la variable `var` qui doit être de type `int`.
  Notez la présence de l'espace initial dans le texte (`" %d"`) et de l'esperluette `&` devant l'identifiant de variable (`&var`). Les deux sont importants [8].
- Les spécificateurs sont les mêmes que pour `printf`, et on doit donner des variables ayant le type associé.
- On peut lire deux valeurs d'un coup : `scanf(" %d %d", &var0, &var1);` . idem pour 3, 4, etc.
- On peut aussi faire cette lecture avec une boucle. Pour lire `len` valeurs :

```c
int i = 0;
while (i < len) {
  scanf(" %d", &variable);
  i = i +1;
}
```

L'utilisateur devra soit écrire toutes ces valeurs sur une même ligne séparéespar des espaces, soit les unes après les autres séparées par desretours à la ligne.

18. Copiez `partie-C/if.c` dans `partie-D/` . Modifiez le `main` pour qu'il demande une année à l'utilisateur et affiche en réponse si elle est bissextile ou non.
Vous continuerez à utiliser une fonction `bool est_bissextile(int annee)` .

Remarque.

- Si vous voulez taper un message à afficher à l'utilisateur avant une lecture de `scanf`, indiquez-le avec `printf` et non dans le texte de `scanf`. Ce dernier décrit ce que l'on s'attend à *lire*, et n'est pas affiché.
  C'est donc assez différent du comportement de `input` de Python.
- ⚠ J'insiste : ne confondez pas non plus les entrées d'une fonction avec l'utilisation de `scanf`. [9]

E Aide et fin

E.1 Aide !

Le site `https://pythontutor.com/c.html#mode=edit` propose de visualise l'exécution pas à pas d'un code C. N'hésitez pas à l'utiliser pour voir ce qu'il se passe lorsque votre programme s'exécute.

E.2 Pour occuper les plus rapides

Allez sur le site `https://www.france-ioi.org/algo/chapters.php`. Faites le chapitre 2 en C, ou allez directement débloquer le niveau 3 en C (auquel cas il faudra que vous regardiez comment fonctionnent les tableaux en C).

8. L'espace « absorbe » les éventuels retours à la ligne ou espaces en trop de l'utilisateur. Le & permet de passer par pointeur `var`
9. Si vous ne comprenez pas en quoi c'est différent, appelez-moi.