TP #13&14 – DEV

14 et 15 novembre 2022
Erwan Kerrien

Rappel : si vous avez des questions sur ce TP ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr

Nous travaillons sous Linux, avec Visual Studio Code (ou VSCodium) comme éditeur et gcc comme compilateur.

Une archive est disponible sur arche en accompagnement de ce TP. Elle contient les codes C donnés en exemple.

Le TP est à rendre pour demain matin (15 ou 16 novembre) 8h : seules les questions notées sont à me rendre (voir les sections "Questions notées").

- Nous verrons ensemble les sections 1, 2, 3 et 4.
- Vous me rendrez un texte de réponse aux deux questions de la section 4.1
- Vous me rendrez un programme codé en C **et commenté** pour répondre à la deuxième question section 4.1, ainsi qu'aux deux questions de la section 5.5
- **Vos programmes devront obligatoirement avoir le nom de fichier indiqué dans la question**. Par exemple, vous me rendrez un fichier nommé `swap.c` pour répondre à la deuxième question de la section 4.1
- Les questions de la section 6.3 sont optionnelles. Cependant, je vous encourage fortement à faire cette section 6 avant le prochain TP.
- la section 7 est un bonus pour celles et ceux qui souhaiteraient approfondir la notion de pointeur et expérimenter avec l'accès bas niveau à la mémoire que les pointeurs autorisent.

Ces fichiers sont à déposer sur arche (séparés ou sous forme d'archive zip). En cas de souci, vous pouvez les envoyer sur mon email Erwan.Kerrien@inria.fr

Prenez bien garde à ne pas m'envoyer autre chose qu'un fichier pour le texte de vos réponses (.txt, .doc, .docx, .odt ou .pdf), et 3 fichiers C. Ne m'envoyez pas les exécutables générés.

1 Objectifs du TP

Ce TP a pour objectif de vous présenter une notion du langage C appelée *pointeurs*. Les pointeurs sont la manière dont C implémente un mécanisme plus abstrait appelé *référence*, que l'on retrouve dans tous les langages. La première utilisation des pointeurs est donc le passage de variables par référence dans une fonction. Nous verrons comment cela est fait en C. La deuxième utilisation est pour l'implémentation des tableaux, que vous avez vu il y a 3 TP de cela. Enfin, la dernière utilisation est pour partager de la mémoire. C'est en particulier utile lors de la manipulation de structures qui peuvent s'avérer très lourdes en mémoire, ce qui rend leur copie coûteuse. Vous verrez un exemple de structure lors du prochain TP sur les fichier (la structure `FILE` qui permet de stocker diverses informations sur un fichier ouvert). La notion plus générale de structure sera vue lors du TP qui viendra après. Bref, vous l'aurez compris : les pointeurs sont très utiles, puissants et c'est pour cela qu'on les retrouve partout quand on programme en C.

Le travail demandé consiste en un premier temps à bien comprendre la notion de pointeurs, ce qui se fera par l'analyse d'exemples, sur lesquels des questions vous seront posées. Puis, dans un deuxième temps, nous l'appliquerons à la lecture de données depuis le clavier et plus généralement depuis un fichier.

2 Pointeur : variables et mémoire

2.1 Organisation de la mémoire

Un pointeur est un type de variables qui permet de stocker une adresse dans la mémoire. Par souci de concision, et abus de langage, on dira "un pointeur" pour une variable de type pointeur (comme on dit "un entier" pour une variable de type entier...).

Pour aller plus loin, il faut préciser ce qu'est une adresse mémoire. Il faut s'imaginer que la mémoire est un grand alignement de boîtes, contiguës, toutes de même taille (voir la figure 1). Chaque boîte a un numéro qui va de 0 pour la première jusqu'au plus grand nombre possible, soit $2^{64} - 1$ sur un ordinateur 64 bits. Une adresse mémoire est donc le numéro d'une boîte. Aller lire ce qui est en mémoire à l'adresse n revient donc à aller regarder ce que contient la boîte n°n. Aller écrire, revient à mettre quelque chose dans la boîte.

FIGURE 1 – La mémoire est composée de boîtes contiguës, toutes de même taille, numérotées.

2.2 Variables et mémoire

Il ne s'agit pas ici de faire un exposé précis et détaillé de la manière dont la mémoire est organisée. Vous avez un cours pour ça. La présentation qui suit est donc volontairement simplifiée pour mettre en avant les concepts essentiels qu'il faut comprendre pour maîtriser les pointeurs, et au-delà avoir une compréhension fine de ce qu'est une variable et notamment un type.

En première simplification, nous n'allons considérer que ce qui se passe quand on appelle une fonction. Vous savez qu'un programme C revient à appeler une fonction particulière appelée `main`. Se restreindre aux fonctions n'est donc pas une grosse contrainte.

Une fonction est composée de deux choses : ses variables et son code. Il est assez simple de comprendre que le code, une fois compilé est de taille connue (vous pouvez utiliser la commande shell `ls -l` pour voir la taille du fichier exécutable). Il faut aussi comprendre que les variables sont toutes de taille connue a priori. Ceci grâce à leur déclaration qui précise leur type.

2.2.1 Variables et types

En informatique, il n'y a qu'une chose qui existe physiquement pour stocker de l'information : le bit, donc seulement des 0 et des 1. Dans l'immense majorité des ordinateurs actuels, l'unité de base est en fait même un bloc de 8 bits, ce qu'on nomme un octet : c'est la taille de chaque boîte en mémoire.

De base, on ne peut donc stocker que des nombres et il faut tout traduire en nombre. L'idée géniale est qu'on peut faire tout ce qu'on veut en définissant un ensemble de types qui est très restreint en C. Ces types sont dits *primitifs*.

- Si l'information est un nombre entier positif, la traduction est simple puisqu'on a un simple codage binaire. La question se pose de combien d'octets utiliser. Pour cela, C définit plusieurs types :
  - `unsigned char` : tient sur 1 octet
  - `unsigned short` : tient sur 2 octets
  - `unsigned int` : tient sur 4 octets
  - `unsigned long` (ou `unsigned long int`) : tient sur 8 octets
  - `unsigned long long` : tient sur 8 ou 16 octets en fonction de l'architecture de l'ordinateur

  Le premier TP vous a montré comment afficher ces tailles grâce à la fonction `sizeof`. Notez bien les deux exemples suivants : `unsigned int i=1;` et `unsigned long i=1;`. Dans les deux cas, on stocke la valeur 1 dans une variable `i`. Sauf que dans le premier cas on utilisera 4 octets pour ce faire, et 8 octets dans le deuxième cas.

— Si l'information est à présent un nombre entier qui est potentiellement négatif, c'est simplement le codage qui va changer : on va utiliser un complément à 2. C définit ces types dits *signés* pour indiquer un tel encodage :
  — `char` : tient sur 1 octet
  — `short` : tient sur 2 octets
  — `int` : tient sur 4 octets
  — `long` : tient sur 8 octets
  — `long long` : tient sur 8 ou 16 octets en fonction de l'architecture de l'ordinateur

  Je vous renvoie à votre cours de système pour le complément à 2.
— Mais les encodages précédents ne permettent pas d'exprimer des nombres réels, à virgule flottante. Ici, on va utiliser l'encodage en mantisse et exposant (cf encore une fois le cours de système). Là encore C définit des types primitifs qui suivent cet encodage mais permettent d'utiliser plus ou moins d'espace mémoire :
  — `float` : tient sur 4 octets
  — `double` : tient sur 8 octets
  — `long double` : tient sur 16 octets
— Un autre type d'information très largement utilisé est le texte. C utilise le type `char` pour stocker une lettre via son code ASCII. Seuls les codes de 0 à 127 sont exploitables. Donc pas de nouveau type : toute manipulation de caractère se fait comme on manipule des nombres (mettre une lettre en minuscule revient à ajouter 32 à la variable de type `char`) et seules les fonctions `printf` et `scanf` (et consorts, voir section 6) vont vraiment traduire le "dessin" de la lettre affichée sur l'écran, à partir de son code ASCII.

  Par conséquent aussi, pas de lettre accentuée par défaut : un code du type `char c='é';` générera une erreur à la compilation `error: character too large for enclosing character literal type` indiquant qu'il ne peut stocker une lettre accentuée sur un `char`. Les éléments de rang 128 à 255 dans la table ASCII servent en effet à stocker les lettres spécifiques à la langue du système. Lorsque le système est français, on y trouve donc la lettre 'é'. Cependant cette partie de la table va par conséquent varier suivant l'ordinateur et C interdit donc de l'utiliser afin de ne produire que des programmes portables. Pour employer ces caractères spécifiques, il faut utiliser l'encodage utf-8 par exemple qui est multi-octets. Cela dépasse le cadre de ce cours introductif. Gardez donc en mémoire une chose simple : pas de lettre accentuée (ou autres du style ç) en C pour éviter les problèmes quand vous faites du bas niveau.
— On peut créer des nouveaux types en rassemblant plusieurs variables avec un type primitif au sein de ce qu'on appelle une structure (`struct` en C). Nous verrons cela lors d'un prochain TP. Mais on peut déjà comprendre que si on regroupe un `int` et un `char` dans une même structure, la taille de la structure sera de 4+1=5 octets. Là encore la taille est connue.
— **Tableau n'est pas un type**. C'est une structure de données, c'est-à-dire un moyen de rassembler plusieurs éléments. Contrairement à la structure, tous les éléments doivent ici être d'un même type. Dès lors qu'on spécifie ce type, on a alors un nouveau type, par exemple "Tableau d'entiers". Mais contrairement à la structure aussi, le nombre d'éléments à stocker n'est pas fixé a priori. La taille du tableau peut être connue au moment de la compilation. Par exemple par une déclaration comme `float tab[10];` ou `long l[]={1, 18, 3};` : dans le premier cas on a besoin de 10×4 octets et 3×8 dans le deuxième. Mais ce n'est pas toujours le cas. En C, une variable servant à référencer un tableau sera un pointeur qui stockera l'adresse du premier élément du tableau. Nous reprendrons ça en section 5.
— Une chaîne de caractères est un tableau de `char` (c'est son type) qui se termine par le caractère `'\0'`, dont le code ASCII est 0.
— Restent les pointeurs. Un pointeur est une adresse mémoire. C'est donc un entier positif sur suffisamment d'octets pour encoder toute adresse mémoire. Sur un système 64 bits, il faut 8 octets (8 fois 8 bits). On utilise donc le type `unsigned long`. Mais sur une machine 32 bits, il suffira de 4 octets (4 fois 8 bits). On peut donc n'utiliser qu'un `unsigned int`. Pour rendre le code compatible avec ces différentes variantes, C définit le type `size_t` qui est absolument équivalent à un `unsigned long` en 64 bits (donc toute machine avec un système d'exploitation récent).
— Nous n'abordons pas ici les `union` qui permettent de stocker une variable dont le type peut varier selon les besoins. Leur définition repose sur les types primitifs et leur taille est donc aussi connue a priori (taille du plus grand type de l'union). Les unions sont très peu utilisées aujourd'hui mais peuvent être utiles, notamment en programmation système.
— Restent aussi les `enum` qui permettent de définir un ensemble de valeurs admissibles. Le type sous-jacent est `int`, donc rien de nouveau non plus du côté de la taille.

Cela complète le tour d'horizon des types en C. En résumé on a donc des types primitifs entiers et flottants, dont on connaît la taille. Il servent pour encoder les caractères (`char`) ou les pointeurs (`unsigned long`). On peut regrouper des élements en tableaux, s'ils sont tous du même type, et là leur nombre peut varier ; ou bien en structures s'ils sont de types différents, mais là leur nombre est fixe. Puis, on peut travailler de manière récursive, en faisant des tableaux de structures, ou en ayant un tableau dans une structure, ou une structure dans une structure ou finalement un tableau de tableaux, ce qui permet de représenter tout ce qu'on veut.

2.3 Fonction et mémoire

Quand votre code appelle une fonction, de l'espace mémoire est demandé au système d'exploitation pour stocker son code, mais également ses variables, c'est-à-dire ses **variables locales** composées des variables d'entrée, des variables de sortie et des variables intermédiaires. Pour cela, il faut que la liste des variables, ainsi que leur type soient connus (raison pour laquelle on vous embête à longueur de TD avec ça !). Le code s'exécute ensuite, modifiant éventuellement les variables locales, c'est-à-dire manipulant l'espace mémoire qui leur est dédié, puis lorsque la fonction se termine, **cet espace mémoire est libéré** (une **valeur** étant éventuellement renvoyée par la fonction). Afin de visualiser cela, nous allons utiliser un outil qui se nomme "C tutor".

2.3.1 Exemple de base : variable locale

Allez sur le site `https://pythontutor.com/c.html`. Vous avez un éditeur de code C. Remplacez le texte par défaut par le suivant :

```c
void simple()
{
        int i;
        i=42;
}

int main()
{
        simple();
        return 0;
}
```

Ce code appelle juste une fonction `simple` qui ne fait rien d'autre que définir une variable locale `i` de type `int` et l'initialise en y stockant la valeur 42.

Cliquez ensuite sur **Visualize Execution**. Une nouvelle page se charge (figure 2).

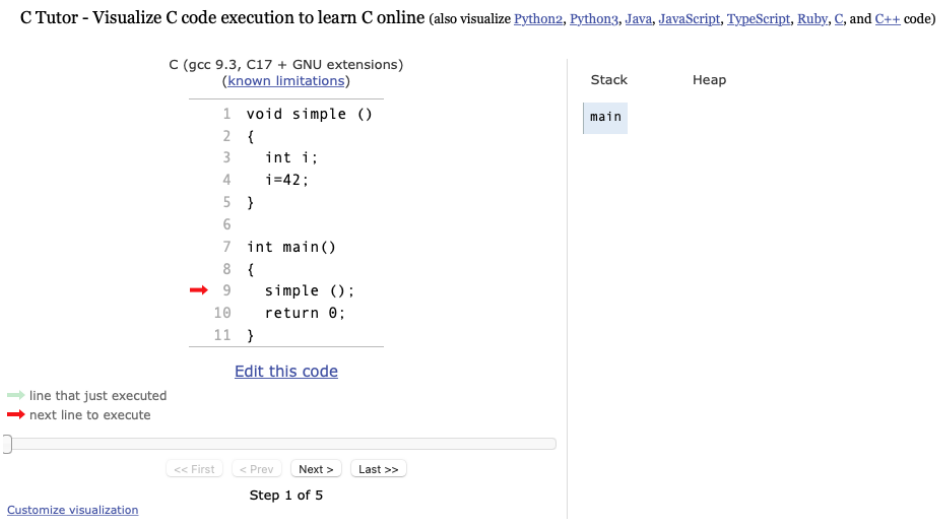

FIGURE 2 – Début de la visualisation de l'exécution

Vous voyez le code dans la partie gauche avec une flèche rouge qui vous indique à quelle ligne l'exécution se trouve (prochaine instruction à exécuter). À droite, vous avez l'état de la mémoire. Seule la partie *stack* (*pile* en français) nous intéresse pour l'instant. Nous verrons la partie *heap* plus tard (*tas* en français).

Le rectangle dans lequel `main` est inscrit correspond à la partie de la mémoire réservée pour l'exécution de la fonction `main`.

Cliquez sur **Next**. La page se met à jour (figure 3).

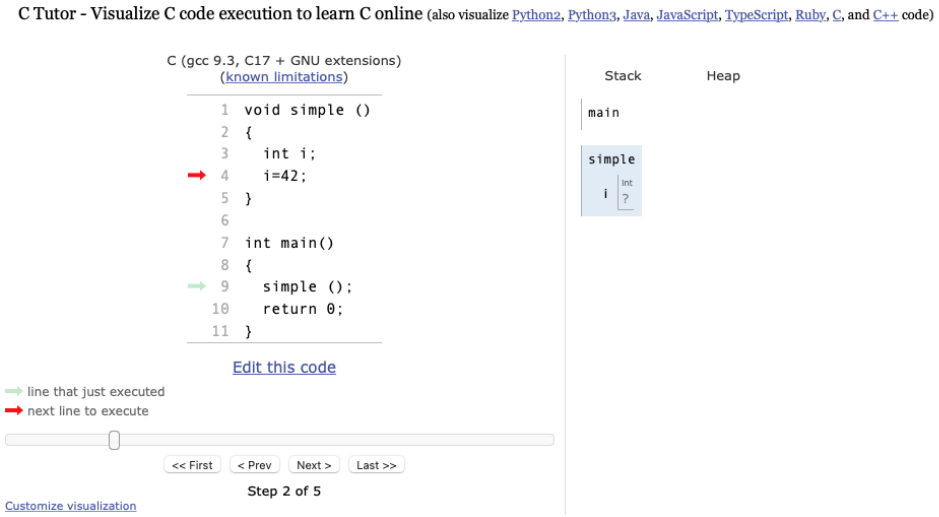

FIGURE 3 – Début de l'exécution de la fonction `simple`

La flèche rouge se positionne d'emblée en ligne 4, après la déclaration de la variable locale `i`. On voit dans la partie droite l'impact qu'a eu l'appel de la fonction : une zone mémoire a été réservée pour la fonction `simple` et le fait d'avoir déclaré la variable `i` de type `int` a permis de lui allouer une zone mémoire de 4 octets. Notez que cette zone mémoire est totalement séparée de celle réservée pour `main`. Remarquez également qu'un ? est indiqué comme valeur pour `i` : il y a forcément une valeur dans ces 4 octets (les bits sont forcément à 0 ou 1) mais cette valeur n'est pas valable tant que la variable `i` n'a pas été initialisée (l'allocation mémoire ne remet pas les bits à 0). C'est ce que nous allons faire en cliquant sur **Next**.

La ligne 5 est exécutée et on peut voir dans la partie droite (figure 4) que 42 a été stocké dans l'espace mémoire alloué pour `i`.

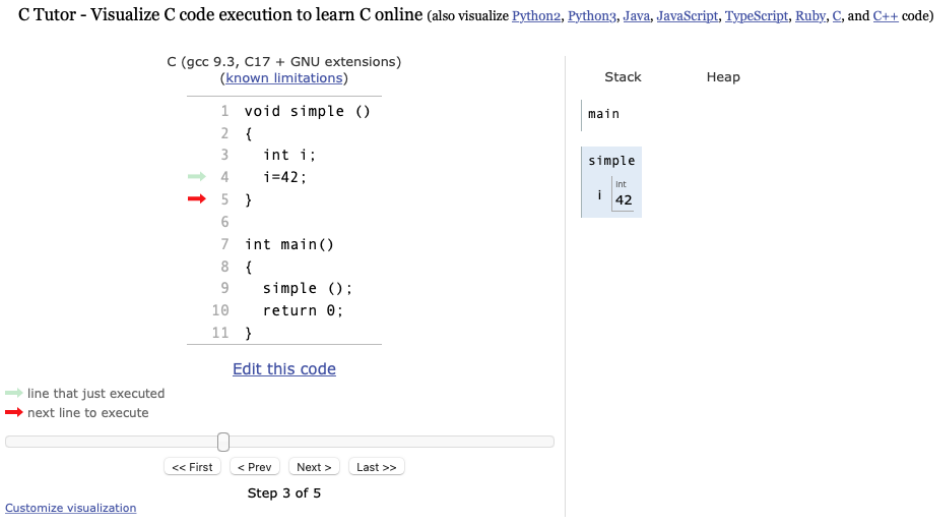

FIGURE 4 – Exécution de la ligne 5 qui stocke la valeur 42 dans l'espace mémoire alloué pour la variable `i`

La flèche rouge se positionne sur l'accolade fermant la fonction `simple`, ce qui indique qu'il faut exécuter cette fermeture. Cliquez sur **Next** pour le faire.

On voit que la flèche bouge très légèrement (figure 5) : il n'y a pas de valeur à renvoyer, mais il y a implicitement une instruction `return` à exécuter. Cliquez sur **Next** pour finaliser la fermeture de la fonction.

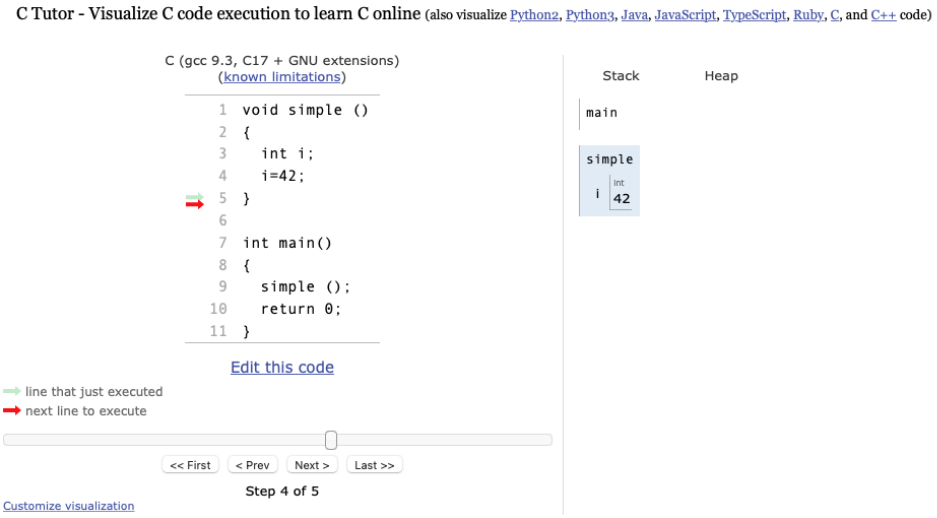

FIGURE 5 – Première étape de la fermeture de la fonction : une instruction `return` (implicite ici) est exécutée

La fonction est fermée, ce qui se voit par la disparition de l'espace mémoire réservé à la fonction `simple` à droite (figure 6) : on dit que la mémoire est *libérée*. La flèche vient se positionner en ligne 10 pour poursuivre l'exécution de la fonction `main`.

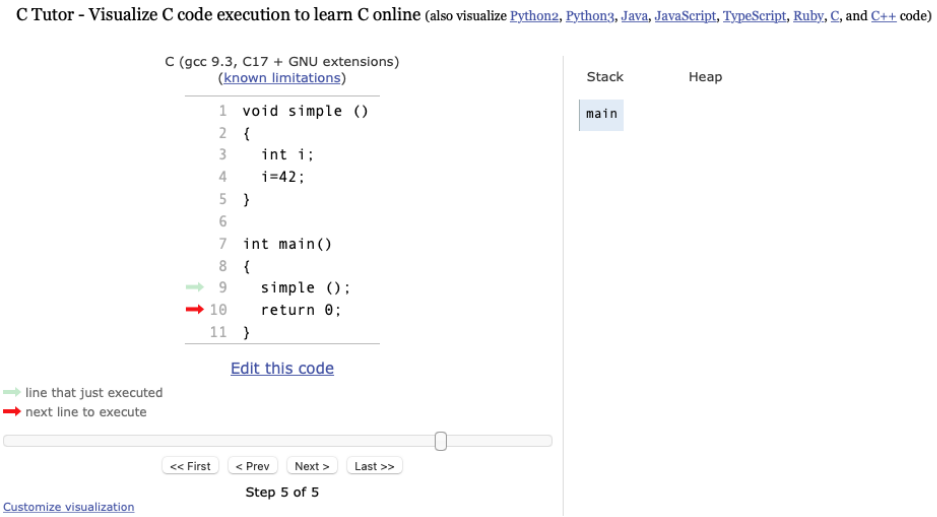

FIGURE 6 – Finalisation de la fermeture de la fonction : l'espace mémoire alloué pour la fonction est libéré et on revient à l'exécution de la fonction appelante

On peut remarquer que l'espace mémoire associé à la variable `i`, locale à la fonction `simple` n'est plus accessible.

De la mémoire est allouée lorsque la fonction est appelée : on parle d'*allocation dynamique*. Mais de plus, cette allocation ne demande pas d'instruction spécialisée puisqu'on sait d'emblée de quel espace mémoire on a besoin pour les variables : on parle dès lors d'*allocation automatique*. En regard de cette allocation automatique, il y a une libération automatique de la mémoire qui a lieu quand la fonction se termine : on ne peut pas libérer cette mémoire avant cela.

2.3.2 Cas des variables d'entrée

Une variable d'entrée est gérée exactement comme une variable locale (allocation en entrée de fonction, libération en sortie). La seule différence est dans l'initialisation qui est faite automatiquement grâce aux valeurs passées lors de l'appel de la fonction. Revenez à l'éditeur de code en cliquant sur **Edit this code** et modifiez le code pour obtenir le suivant :

```
1  void simple(int k)
2  {
3          int i;
4          i=42;
5  }
6
7  int main()
8  {
9          simple(3);
10         return 0;
11 }
```

Puis lancez la visualisation en cliquant sur **Visualize Execution**. Le point de départ est quasiment le même. Débutez l'exécution de la fonction `simple` en cliquant sur **Next**.

Là apparaît une différence (figure 7).

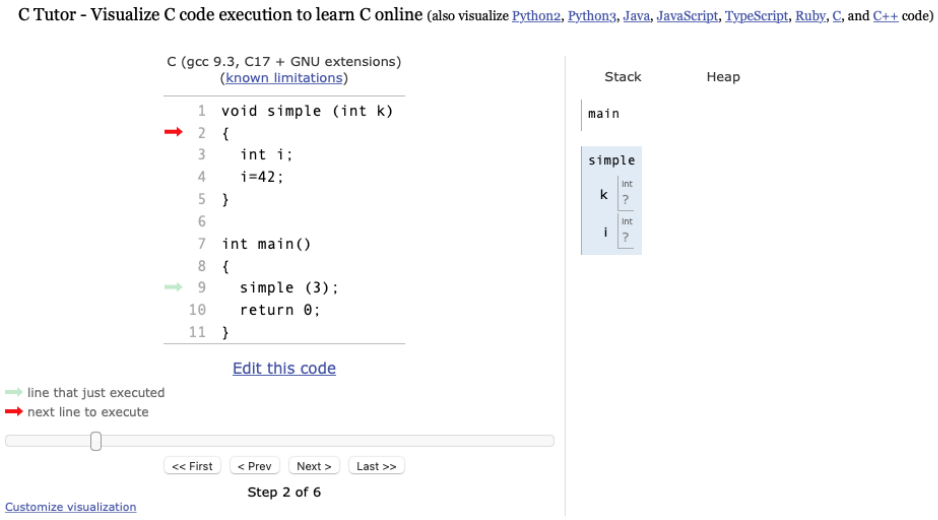

FIGURE 7 – Démarrage de l'exécution d'une fonction avec variable d'entrée

À droite, on voit que l'espace mémoire alloué à la fonction `simple` comporte de l'espace pour deux variables `k` et `i` qui sont toutes deux vues comme locales à la fonction. Le contenu de ces deux variables est indéfini. De plus, la flèche rouge ne se positionne plus sur l'initialisation de la variable `i` (ligne 4) mais sur la première accolade (ligne 2) : il y a une étape à réaliser avant de commencer l'exécution du code de la fonction. Cliquez sur **Next**.

On voit que la variable `k` est initialisée à 3, la valeur passée en appel de la fonction `simple`, puis la flèche se déplace en ligne 4 pour en débuter l'exécution (figure 8).

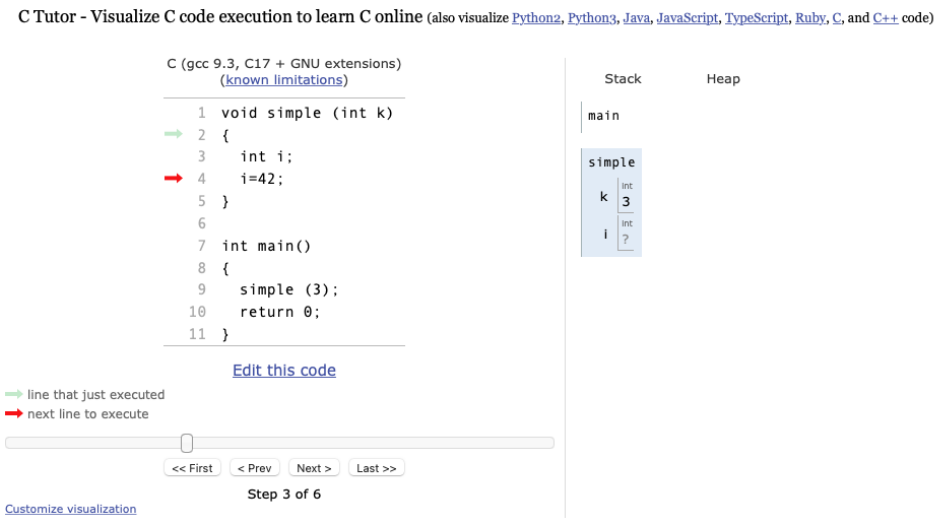

FIGURE 8 – Première étape de l'exécution d'une fonction avec variable d'entrée : recopier les valeurs passées en paramètres dans les variables locales correspondantes

Le reste de l'exécution est similaire. Notez à nouveau la libération de la mémoire réservée aux variables locales de `simple`, y compris la variable d'entrée `k`, qui ne sont plus accessibles une fois `simple` terminée.

2.3.3 Cas d'une valeur renvoyée

Que se passe-t-il à présent quand la fonction renvoie une valeur ? En C, on ne peut pas faire comme en python : on ne peut renvoyer qu'une seule valeur qui doit être d'un type connu. Ce type sera donc soit un des types primitifs donnés plus haut (y compris pointeur), soit un type structure que nous verrons lors d'un prochain TP.

Revenez à l'éditeur de code et modifiez le code pour obtenir le suivant :

```
1  float simple(int k)
2  {
3          int i;
4          i=42;
5          return 1.5;
6  }
7
8  int main()
9  {
10         float r;
11         r=simple(3);
12         return 0;
13 }
```

Veuillez notez deux choses. D'une part, je vous montre ici que c'est bien une valeur qu'on renvoie et non une variable. Si j'avais utilisé une variable locale (par exemple `tmp`) et que j'avais terminé ma fonction par `return tmp;`, alors c'est bien la valeur stockée dans la variable locale qui aurait été renvoyée et non pas la variable elle-même. D'autre part, je dois déclarer une variable locale à la fonction `main` (que j'appelle `r`) afin d'avoir un espace mémoire où stocker la valeur que renverra la fonction `simple`.

Passons en revue les différentes étapes du code (avec le bouton **Next**).

- Le programme débute avec un espace mémoire pour la fonction `main`. Remarquez qu'un espace a été automatiquement alloué à la variable `r`.
- à la ligne 11, la fonction `simple` est appelée. Remarquez que la valeur de la variable `r` n'est toujours pas définie. Elle ne le sera qu'après l'exécution de la fonction `simple`.

— le déroulement de la fonction est identique à ce qui précède. Notez qu'aucun espace n'est réservé pour la valeur de retour car je n'ai pas utilisé de variable locale de sortie.
— à la fin de l'exécution de `simple`, rien ne semble fait quand `return 1.5;` est exécuté. La valeur renvoyée est en fait stockée dans un registre du processeur qui est invisible ici. Puis l'espace mémoire réservé à la fontion `simple` est libéré.
— Enfin, l'instruction d'affectation en ligne 11 peut terminer son exécution en copiant cette valeur dans l'espace mémoire alloué à la variable `r`.

3 Pointeurs

Ce qui précède doit vous convaincre que toute déclaration de variable implique qu'un espace mémoire lui est alloué. La taille de cet espace mémoire est déterminée par le type de la variable. Toute variable a donc une adresse en mémoire : si plusieurs octets (cases mémoire) sont alloués, cette adresse est celle du premier octet ; les autres sont forcément contigus (collés à la suite) de ce premier octet.

3.1 L'opérateur *addresse-de*

Cet opérateur permet de récupérer l'adresse d'une variable, quelle qu'elle soit. Il se note par un `&` placé juste devant le nom de la variable. Par exemple, le code suivant définit une variable `i` de type `int` et en affiche l'adresse grâce à un `printf`.

```
1  // fichier adressede.c
2  #include <stdio.h>
3
4  int main()
5  {
6          int i=42;
7          printf("%p\n", &i);
8          return 0;
9  }
```

Compilez ce code est lancez-le plusieurs fois. Vous pouvez noter que l'adresse change à chaque exécution.

3.2 L'opérateur d'*indirection* (ou *déréférencement*).

Une fois qu'on a récupéré une adresse, donc un numéro de boîte, on peut accéder à cette boîte pour voir ce qui s'y trouve (accès en lecture) voire en modifier le contenu (accès en écriture). L'opération d'accès à la valeur stockée s'appelle *indirection* ou *déréférencement*, et il se note par une `*` placée devant la variable de type pointeur (par exemple `*p` si `p` est une variable de type pointeur). On peut modifier l'exemple précédent pour afficher la valeur stockée à l'adresse de `i`, ce qui se note `*(&i)`.

```
1  // fichier deref1.c
2  #include <stdio.h>
3
4  int main()
5  {
6          int i=42;
7          printf("%d\n", *(&i));
8          return 0;
9  }
```

Compilez et exécutez ce code. Vous remarquerez l'usage du format `%d` dans le `printf` car c'est une bien une valeur de type `int` qu'on veut afficher. Cet exemple n'est pas très utile. On va plutôt chercher à manipuler des variables dont les valeurs sont des adresses mémoire, ce qu'on appelle des pointeurs.

3.3 Déclaration d'une variable de type pointeur.

Il nous faut donc des variables de type pointeur, soit pour stocker l'adresse qu'on récupère via `&`, soit pour accéder au contenu d'une adresse via l'opérateur `*`. Cette dernière opération d'accès implique forcément deux choses : 1) on doit

pouvoir décoder le contenu de la mémoire (on a vu qu'on avait un encodage différent pour les entiers positifs, pour les entiers signés et pour les flottants) ; et 2) on doit pouvoir savoir combien d'octets sont concernés par cet encodage. Tout ceci est défini par le type. La déclaration d'un pointeur doit donc indiquer quel type est pointé (comment et sur quel espace est encodée la valeur stockée à l'adresse mémoire).

Un pointeur se déclare en indiquant le type pointé, suivi d'une étoile `*`. Par exemple :

```
1   int *pi;
2   char *pc;
```

déclare deux variables de types pointeurs : `pi` de type `int*`, qui est donc un pointeur vers un `int` ; et `pc` de type `char*` qui est donc un pointeur vers un `char`. On notera que les espaces avant et/ou après le caractère `*` importent peu. On aura aussi bien déclarer ces variables comme :

```
1   int* pi;
2   char* pc;
```

ou encore

```
1   int * pi;
2   char * pc;
```

On préférera cependant la première manière indiquée car elle permet de mieux comprendre une déclaration comme

```
1 int *pi, i;
```

qui, bien qu'horrible pour la lisibilité du code, est valide et déclare une variable `pi` de type pointeur vers un `int`, ainsi qu'une variable `i` de type `int`. De même, pour déclarer deux variables pointeurs vers un même type, il faudra les faire précéder à chaque fois d'un `*`. Ainsi pour déclarer `pi1` et `pi2`, deux pointeurs vers des `int`, on écrira :

```
1 int *pi1, *pi2;
```

On peut dès lors commencer à jouer avec les pointeurs. Compilez et exécutez le code suivant (disponible dans l'archive du TP).

```
1  // fichier pointeur1.c
2  #include <stdio.h>
3
4  int main()
5  {
6    int i;
7    int *pi;
8
9    i=3;
10   pi = &i;
11   printf("(via la variable) Valeur=%d; adresse=%p\n",i,&i);
12   printf("(via le pointeur) Valeur=%d; adresse=%p\n",*pi,pi);
13
14   *pi = 4;
15   printf("(via la variable) Valeur=%d; adresse=%p\n",i,&i);
16   printf("(via le pointeur) Valeur=%d; adresse=%p\n",*pi,pi);
17
18   return 0;
19 }
```

L'adresse de la variable `i`, de type `int` est prise en ligne 10 et stockée dans la variable `pi` de type `int*`, soit un pointeur sur un `int`. Les lignes 11 et 12 montrent que les deux variables se rapportent à la même zone mémoire et aux mêmes valeurs, et que les deux opérateurs adresse-de et indirection sont inverses l'un de l'autre. En ligne 14, on modifie la zone pointée par `pi` pour y stocker la valeur 4 via le mécanisme d'indirection. Les lignes 15 et 16 montrentque la variable `i` a bien été modifiée. Par ailleurs, on remarquera que le format `%p` permet d'afficher une adresse (au format hexadécimal) stockée dans un pointeur.

3.4 C tutor

Revenons à C tutor pour voir comment cela peut se visualiser. Entrez le code suivant :

```c
#include <stdio.h>

int main()
{
  int i,j;
  int *pi;

  i=3;
  pi = &i;

  j=*pi;

  *pi = 4;

  return 0;
}
```

L'espace mémoire pour la fonction `main` (figure 9) est alloué avec d'emblée un espace pour stocker deux variables `i` et `j` de type `int` et une variable `pi` de type `pointer`. Au niveau de la place mémoire, un pointeur est en effet une adresse, quelque soit le type pointé, et on a donc besoin de 8 octets pour la stocker (équivalent à un `unsigned long`). Ce n'est que lorsqu'on accède à la mémoire pointée (ici l'espace alloué à `i`) qu'on a besoin d'en connaître le type (ici `int`). Le bonus (section 7) explique comment on peut exploiter cela via la conversion de types.

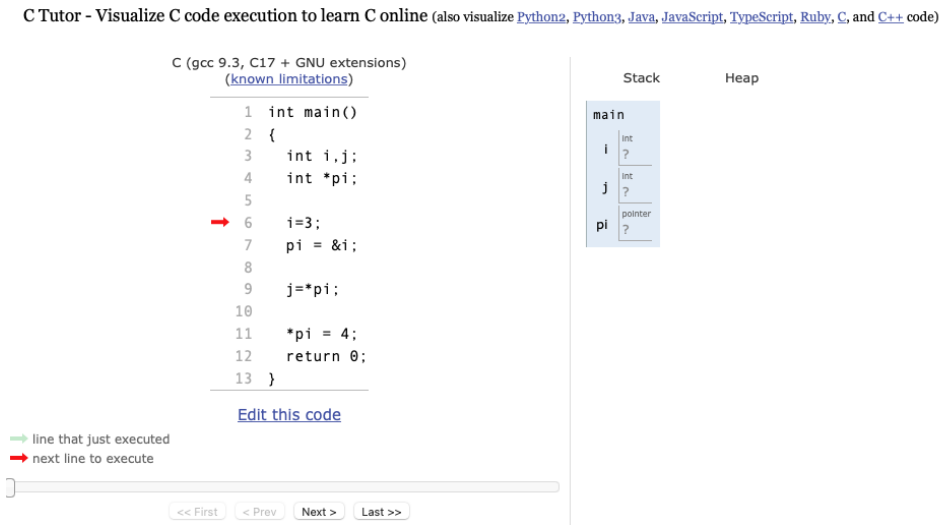

FIGURE 9 – Premier exemple avec pointeur. Début d'exécution.

Cliquez sur **Next** (figure 10). La ligne 6 est exécutée, affectant la valeur 3 à la variable `i`.

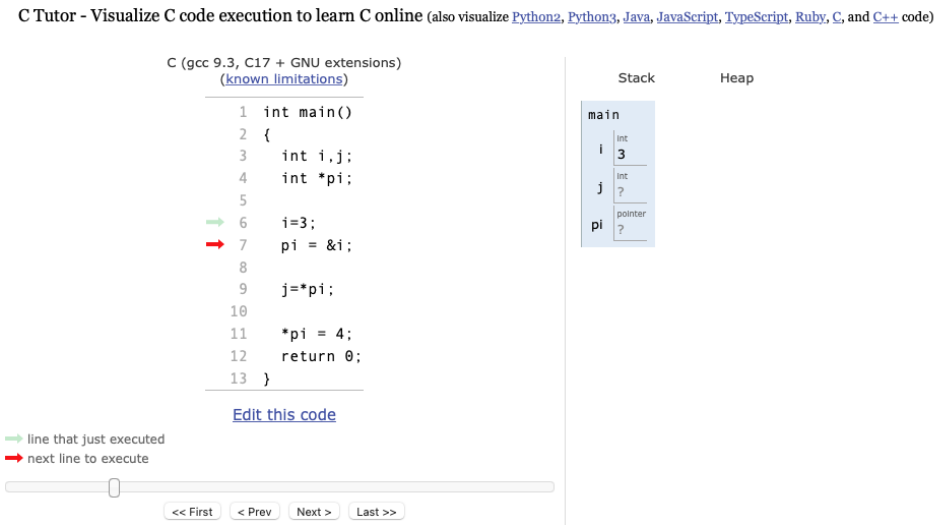

FIGURE 10 – La valeur de `i` est intialisée à 3.

Cliquez sur **Next** (figure 11). La ligne 7 est exécutée : la variable `pi` reçoit comme valeur l'adresse mémoire de la variable `i`. Ceci est valide puisque de l'espace mémoire a été alloué automatiquement pour `i` à l'appel de la fonction `main`. Vous voyez que ce lien est matérialisé par une flèche : il faudra suivre la flèche si on veut accéder à la valeur stockée à cette adresse en mémoire (indirection).

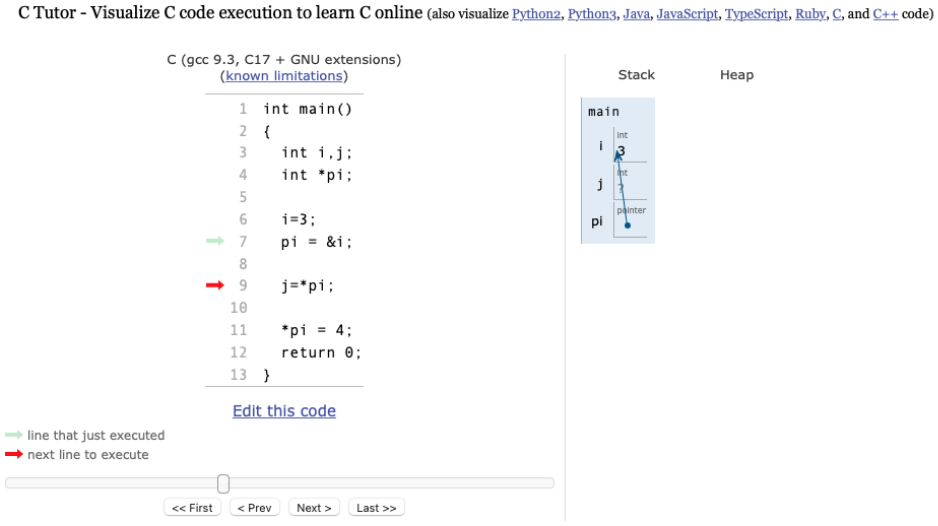

FIGURE 11 – L'adresse de `i` est stockée dans la variable `pi`.

Cliquez sur **Next** (figure 12). La ligne 9 est exécutée. `*pi` à droite de l'affectation indique qu'on va déréférencer le pointeur `pi` pour accéder à la valeur stockée sous cette adresse. Cet accès se fait en lecture puisqu'on veut simplement aller voir quelle valeur s'y trouve (rôle de *right-value* dans une affectation = accès en lecture). La valeur est lue puis est stockée dans la variable `j` dont la valeur est enfin initialisée (voir la partie mémoire à droite). On remarque que c'est la même valeur que `i`.

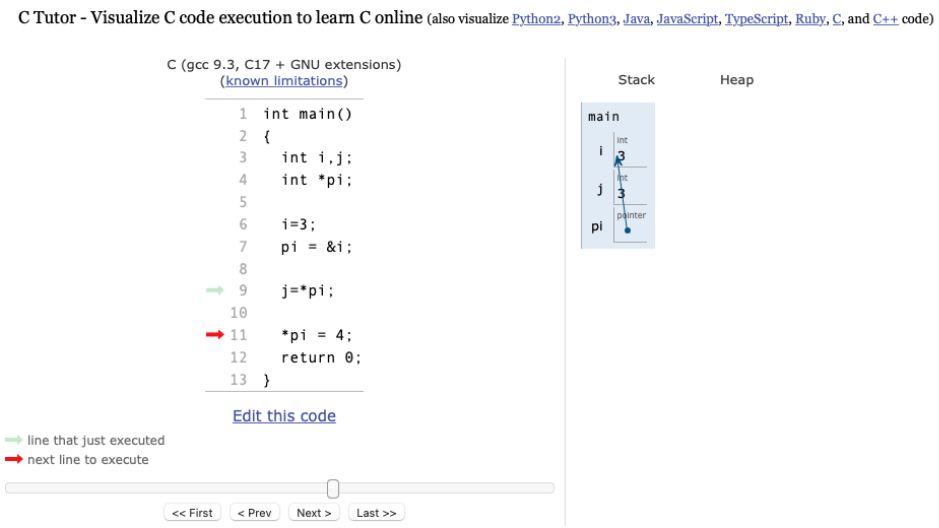

FIGURE 12 – Indirection : accès en lecture à l'adresse stockée dans `pi`

.

Cliquez sur **Next** (figure 13). La ligne 11 est exécutée. Ici `*pi` est à gauche de l'affectation (left-value) et on va déréférencer le pointeur `pi` pour accèder en écriture à cette adresse. Vous pouvez remarquer que `pi` n'est pas modifiée (elle pointe toujours vers le même endroit en mémoire, ce qui veut dire qu'elle contient toujours la même adresse). En revanche la mémoire allouée à la variable `i` a changé, ce qui signifie que la valeur stockée dans la variable `i` a été modifiée.

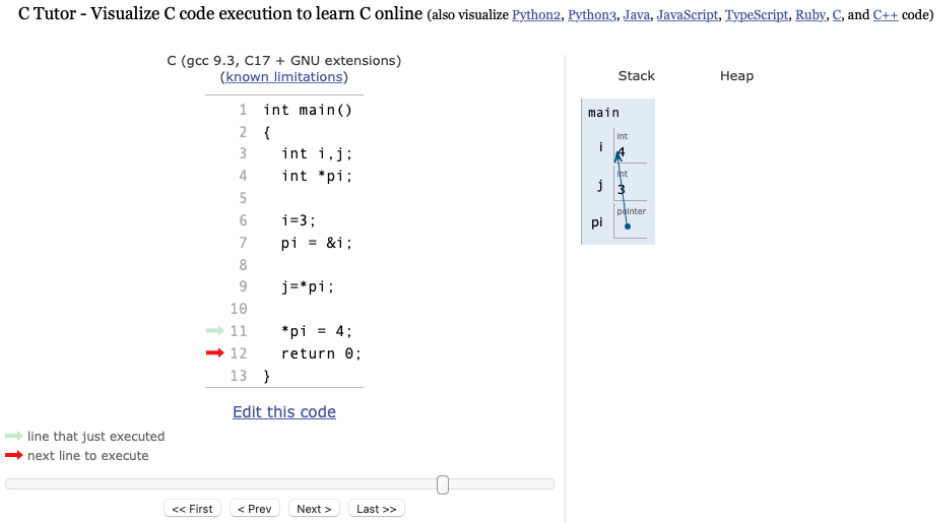

FIGURE 13 – Indirection : accès en écriture à l'adresse stockée dans `pi`

.

4 Passage par variable (ou par référence)

Dans les exemples précédents, la visualisation permise par C tutor vous a montré que les espaces mémoire réservés pour chaque fonction étaient séparés et bien distincts. Mais vous avez pu remarquer également que l'espace mémoire réservé à la fonction appelante (`main` dans ce qui précède) restait valide pendant l'exécution de la fonction appelée (`simple` dans ce qui précède). Or les pointeurs permettent a priori d'accéder à n'importe quelle adresse en mémoire, même celles en-dehors de la zone réservée à la fonction en cours d'exécution. On doit donc pouvoir notamment accéder à l'espace mémoire de la fonction appelante : pour cela on passe à la fonction appelée une adresse de l'espace réservé à la fonction appelante, et ce, grâce à un pointeur. C'est comme si la fonction `main` fournissait une clé `k` à la fonction `simple` pour l'autoriser à accéder à son espace mémoire.

Pour préciser les choses, entrez le code suivant sur C tutor.

```
1  void simple(int *k)
2  {
3          int i;
4          i=42;
5          *k = 21;
6  }
7
8  int main()
9  {
10         int j;
11         j=10;
12
13         simple(&j);
14
15         return 0;
16 }
```

Le début est classique maintenant : l'espace mémoire pour `main` est réservé avec l'espace pour une variable `j` de type entier. En cliquant deux fois sur **Next** (figure 14), la ligne 10 est exécutée, ce qui initialise le contenu de `j` à 10.

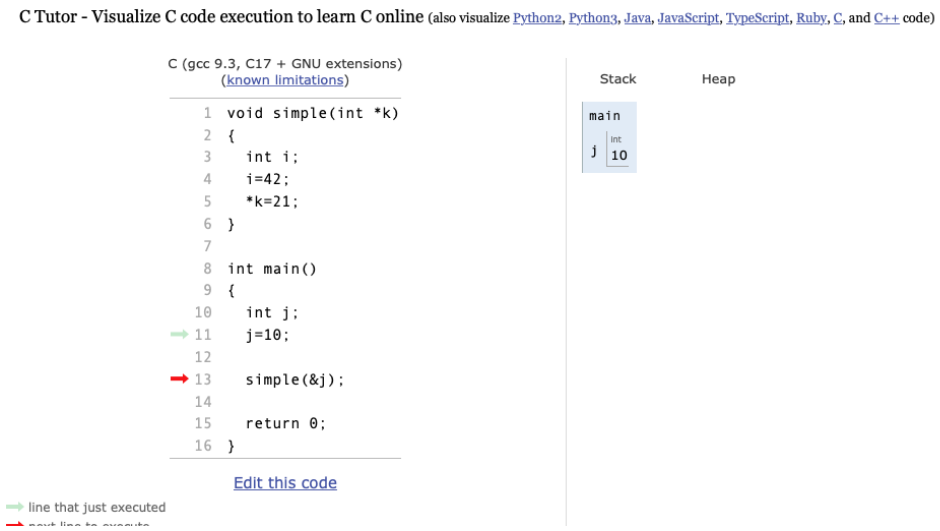

FIGURE 14 – Passage par variable : initialisation d'une variable `j` locale à `main`

Cliquez à nouveau sur **Next** (figure 15). L'espace mémoire est alloué pour la fonction `simple` avec notamment l'espace pour ses deux variables `k` (de type pointeur) et `i` (de type `int`). Vous pouvez remarquer que l'espace mémoire pour `main` est toujours actif.

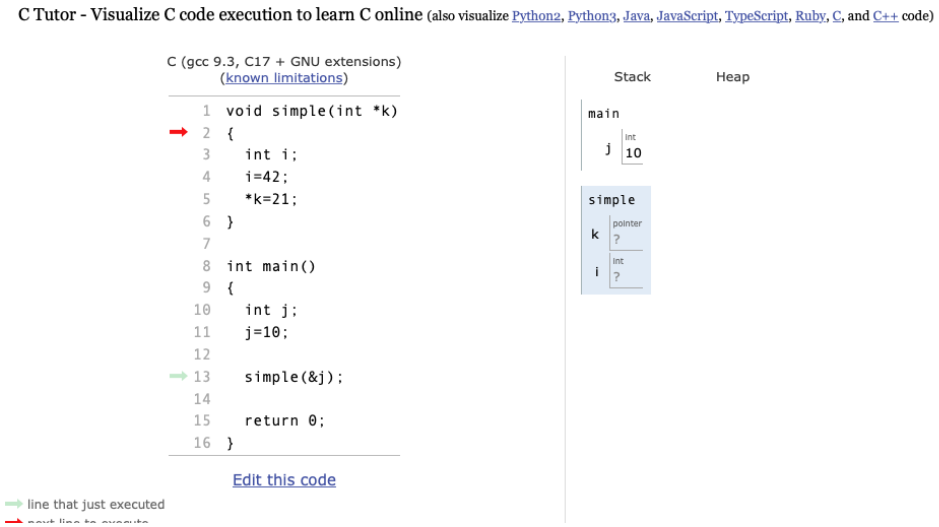

FIGURE 15 – Passage par variable : appel de la fonction `simple`

Cliquez sur **Next** (figure 16). Comme on a une variable en entrée, la valeur passée lors de l'appel à `simple`, à savoir `&j` (ligne 13), est recopiée dans la variable `k`. On passe donc la valeur d'une adresse, ce qui relie la variable `k` locale à la fonction `simple` avec la variable `j` locale à la fonction `main` : `k` est une référence à `j`, ce qui établit un lien entre deux espaces mémoire qui sont autrement séparés et distincts.

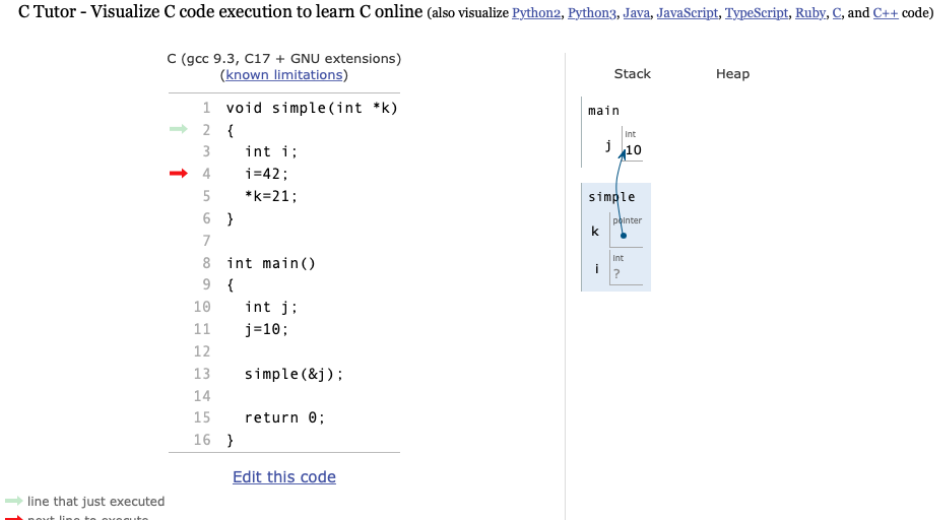

FIGURE 16 – Passage par variable : copie de l'adresse de `j` dans la variable locale `k`

Cliquez sur **Next** (figure 17) : le contenu de la variable `i` est initialisé à 42 (ligne 4).

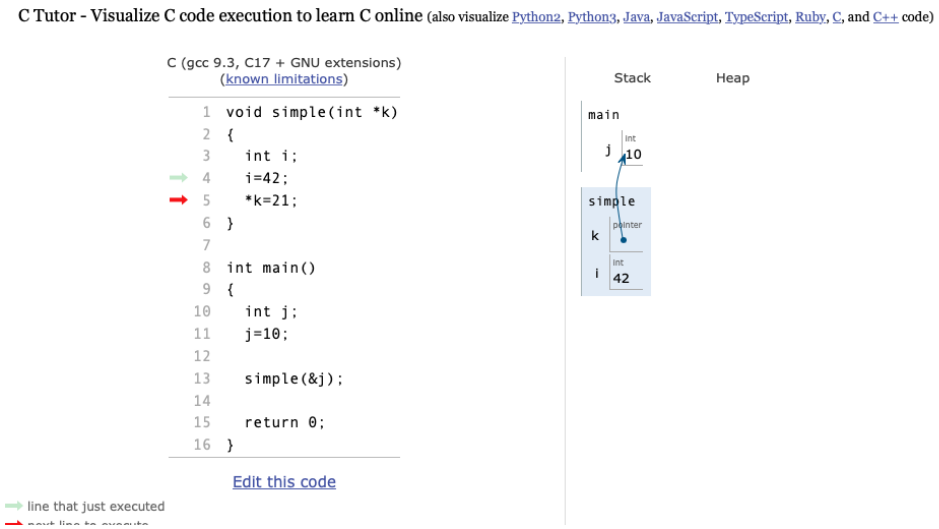

FIGURE 17 – Passage par variable : affectation de la valeur 42 à la variable `i`

Cliquez sur **Next** (figure 18) : la valeur 21 est stockée sous l'adresse pointée par `k` par indirection. Cette adresse est celle de la variable `j` qui passe de 10 à 21 : on a donc bien modifié l'espace mémoire de `main` (fonction appelante) depuis la fonction `simple` (fonction appelée).

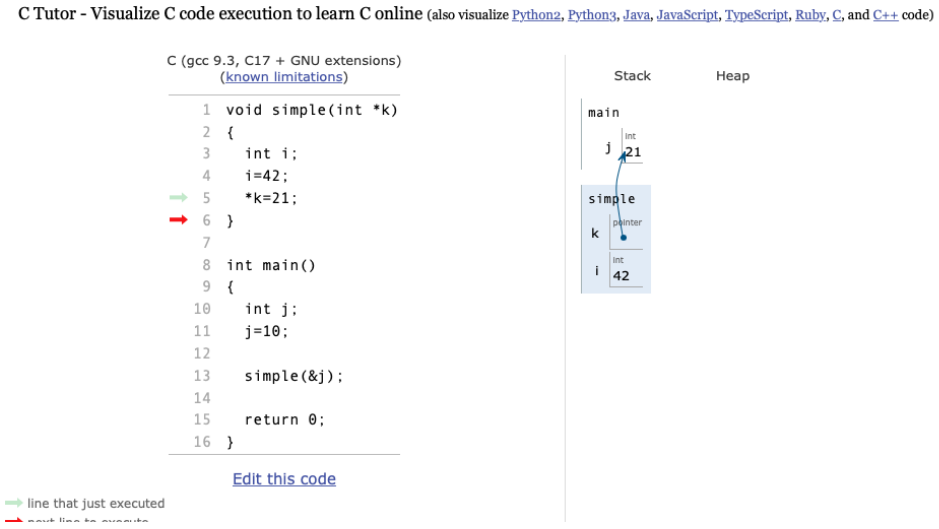

FIGURE 18 – Passage par variable : affectation de la valeur 21 à la variable `j`, par indirection du pointeur `k`

Cliquez deux fois sur **Next** pour terminer l'exécution de la fonction `simple` (figure 19) : l'espace mémoire de la fonction `simple` est libéré mais la modification effectuée dans l'espace mémoire de la fonction `main` (variable `j`) est conservée. On peut poursuivre et terminer l'exécution de la fonction `main`.

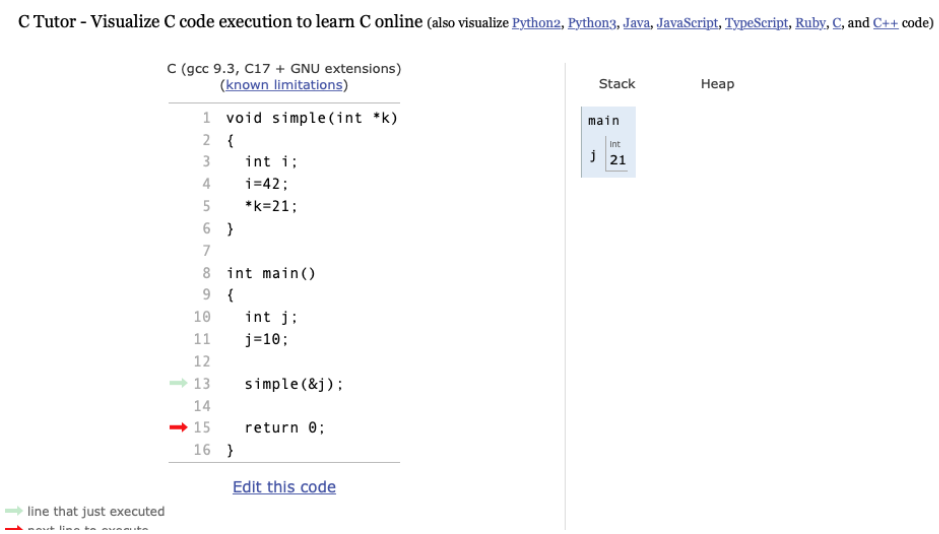

FIGURE 19 – Passage par variable : la fonction `simple` se termine, ce qui provoque la libération de l'espace mémoire qui lui était réservé ; la modification apportée au sein de l'espace mémoire pour la fonction `main` est conservé

En résumé, manipuler des adresses via des pointeurs permet d'accéder à n'importe quel espace mémoire a priori. En pratique, cet accès est réservé à l'espace mémoire lié au programme en cours : sous Unix, des mécanismes existent pour assurer la sûreté des processus, et un programme ne peut pas agir pour aller modifier la mémoire d'un autre programme. Si vous essayez de faire cela, le système arrêtera l'exécution de votre programme avec une erreur `Segmentation fault` (accès mémoire non autorisé). C'est pour cela qu'on n'affecte jamais directement un nombre à un pointeur, mais on prend l'adresse d'une variable déclarée afin d'être sûr d'avoir une adresse valide. La seule exception est l'utilisation de la valeur 0, qui est une adresse mémoire interdite à tout processus. Cela permet donc d'identifier les pointeurs invalides et d'éviter leur indirection. C définit même une constante `NULL` pour cette adresse mémoire 0. Par exemple, l'instruction `float *pf=NULL;` permet de déclarer un pointeur `pf` et d'indiquer qu'il ne peut peut pas être déréférencé (puisque nul). Ceci est très utile lors de l'allocation manuelle de mémoire au moyen de la fonction `malloc`. La mémoire ainsi allouée n'est en effet pas libérée automatiquement : comme elle a été allouée manuellement, elle doit être libérée manuellement. L'avantage est qu'on peut ainsi allouée de la mémoire dans une fonction et la garder allouée, même après la fermeture de la fonction. Cette libération mémoire se fait par la fonction `free`. Or cette fonction pré-suppose que l'adresse mémoire (pointeur) qui lui est passé en paramètre, est valide : si on essaie de libérer un espace mémoire non autorisé ou bien déjà libéré, le programme plantera avec un message d'erreur `Segmentation fault`. Il faut donc s'assurer au prélable de la validité du pointeur, ce qui, par convention, est le cas si le pointeur est non nul.

Un exemple d'utilisation correcte de ces fonctions est :

```c
#include <stdio.h> // pour printf
#include <stdlib.h> // pour malloc et free

int main()
{
  char *s=NULL; // initialisation du pointeur s
                // a NULL pour indiquer qu'il
                // est invalide

  s=malloc(6*sizeof(char)); // allocation d'un
                            // espace memoire de
                            // 6 caracteres
  s[0] = 'H'; // stockage des 5 caracteres du
  s[1] = 'e'; // mot Hello, suivi d'un caractere
  s[2] = 'l'; // nul pour terminer la chaine
```

```
16    s[3] = 'l';
17    s[4] = 'o';
18    s[5] = '\0';
19    printf("MESSAGE: %s\n", s); // affichage de la
20                                // chaine
21
22    if (s) free(s); // liberation de la memoire
23                    // allouee par malloc, mais
24                    // seulement si s est un
25                    // pointeur valide (non nul)
26    return 0;
27 }
```

4.1 Exercice : Questions notées

1. Reprenez le code de la page 14 sous C tutor et modifiez la ligne 5 en retirant l'indirection (`*`) : `k=21;`. Que se passe-t-il ? Pourquoi ?
2. Un exemple classique de passage par variable est une fonction qui échange le contenu de deux variables [1].
   C ne propose pas de mécanisme haut niveau comme Python et il faut donc faire les choses à la main. Je vous propose le code suivant pour la fonction d'échange entre deux variables.

```
1  // fichier swapBad.c
2  #include <stdio.h>
3
4  void swap(float x, float y)
5  {
6          float tmp=x;
7          x=y;
8          y=tmp;
9          return;
10 }
11
12 int main()
13 {
14         float a=2.3;
15         float b=-1.7;
16
17         swap(a,b);
18
19         printf("a=%f ; b=%f\n",a,b);
20
21         return 0;
22 }
```

   — Compilez et lancez ce programme. Fonctionne-t-il ? Pourquoi ? Vous pouvez utiliser C tutor pour comprendre ce qui se passe.

   — Corrigez ce code pour qu'il fonctionne et que le programme affiche `a=-1.700000 ; b=2.300000`. Vous me rendrez le résultat sous la forme d'un fichier nommé `swap.c`

1. En Python, on peut le faire simplement (`b,a = a,b`) mais c'est parce que Python implémente un ensemble de structures de données et de fonctions haut niveau qui sont appelées de manière sous-jacente et sans que le programmeur en ait conscience. Par exemple, le code python `b,a=a,b` exploite la structure de données `tuple` : un `tuple` est créé à partir des valeurs de `a` et `b`, puis on crée un deuxième `tuple` à partir du deuxième élément et du premier élément (dans cet ordre) du `tuple` précédent, puis le mécanisme haut niveau de dépaquetage (*unpacking*) d'un `tuple` est appelé pour distribuer les valeurs stockées dans le `tuple` sur les variables `b` et `a`.

5 Pointeurs et tableaux

5.1 Déclaration d'un tableau

Si on en connaît la taille, on peut déclarer un tableau de telle manière que la mémoire nécessaire soit automatiquement allouée. Il y a plusieurs manières de le faire en C. Elles sont regroupées dans le code suivant :

```
1  int main()
2  {
3          float tab[10];
4
5          int liste_nb[] = {1,3,-2,5};
6
7          char chaine[] = "Bonjour";
8
9          int N=4;
10         unsigned short quatre_ushorts[N];
11
12         return 0;
13 }
```

On atteint ici une limitation de C tutor qui ne sait pas bien analyser ces déclarations. Nous devons donc nous en passer [2]. Analysons donc ce code :

- En ligne 3, nous déclarons une variable de nom `tab` qui est un tableau de 10 `float`. Les éléments de ce tableau ne sont pas inialisés et donc le contenu du tabeau n'est pas valide. C'est la manière la plus répandue de déclarer un tableau dont la taille connue.
- En ligne 5, nous déclarons une variable `liste_nb` qui est un tableau (présence des crochets []). On n'a pas besoin de donner sa taille car il est initialisé avec une liste de 4 valeurs (1,3,-2,5) : on sait donc qu'il est de taille 4. Ici l'initialisation est faite et le contenu du tableau est valide.
- En ligne 7, nous déclarons une variable `chaine` qui est un tableau de caractères. En C, c'est ainsi que sont représentées les chaînes de caractères. La chaîne `"Bonjour"` à droite de l'affectation est donc bien interprétée comme un tableau de 8 caractères (ne pas oublier le `'\0'` final !). Ici encore l'initialisation permet de connaître la taille du tableau et il n'est donc pas besoin de l'indiquer entre les crochets [].
- En lignes 9 et 10, on montre qu'on peut aussi déclarer un tableau (ici la variable `quatre_ushorts`) dont la taille est donnée par la valeur d'une variable (ici `N`). Attention, toutefois : modifier la variable `N` par la suite ne changera pas la taille du tableau. C'est bien la valeur de la variable au moment de la déclaration qui est prise comme taille, fixe, du tableau.

Les lignes 5 et 7 sont des cas particuliers. En premier lieu, on peut se permettre d'omettre la taille du tableau (crochets vides []) car la valeur passée à l'initialisation l'indique. Par exemple une déclaration `char toto[];` ne sera pas valide et provoquera l'erreur `error: definition of variable with array type needs an explicit size or an initializer` indiquant qu'il faut absolument connaître la taille a priori du tableau.

Par ailleurs, les initialisations faites en lignes 5 et 7 ne peuvent être faites qu'au même moment où la variable est déclarée. Dans le premier cas, le code

```
1  int liste_nb[4];
2  liste_nb = {1,3,-2,5};
```

provoquera l'erreur `error: expected expression` : ce n'est pas valide en C. Il faut initialiser les éléments un par un. Et dans le deuxième cas, le code :

```
1  char chaine[8];
2  chaine = "Bonjour";
```

provoquera l'erreur `error: array type 'char [8]' is not assignable`.

2. Vous pouvez tester, mais faites attention à ce que C tutor vous montre. Ce n'est pas fiable.

Dans ce cas, on utilisera les fonctions de la librairie standard pour manipuler des chaînes de caractères et en particulier ici la fonction `strcpy` qui permet de recopier une chaîne de caractères :

```
1  #include <string.h>
2  int main()
3  {
4          char chaine[8];
5          strcpy(chaine,"Bonjour");
6          return 0;
7  }
```

Toutes ces déclarations allouent automatiquement de la mémoire pour les variables tableau, et par conséquent cette mémoire est automatiquement libérée est fin de fonction.

5.2 Lien entre tableau et pointeur

La dernière erreur donne une indication du type utilisé pour un tableau : on voit que `chaine` n'est pas de type `char` mais de type `char [8]`. Cela ressemble au type pour un pointeur vers des `char` : `char *`.

En effet, en C, un tableau est un pointeur. Souvenez-vous qu'un tableau est stocké en mémoire sur des cases contiguës. Une variable tableau contient l'adresse du premier élément. Pour vous en convaincre, compilez et lancez le code suivant :

```
1  // fichier tabptr.c
2  #include <stdio.h>
3  int main()
4  {
5          float tab[3];
6
7          printf("Valeur de tab: %p\n", tab);
8          printf("Adresse du premier element: %p\n",&tab[0]);
9
10         return 0;
11 }
```

Compilez et lancez-le plusieurs fois : vous verrez qu'à chaque fois les deux adresses affichées sont les mêmes. En ligne 7, on affiche la valeur stockée dans `tab`, et en ligne 8, on affiche l'adresse du premier élément de `tab`, c'est-à-dire de `tab[0]`.

5.3 Passage d'un tableau en argument

Un tableau étant d'emblée un pointeur, il est systématiquement passé par variable en argument d'une fonction. Cette fonction peut dès lors en modifier le contenu puisqu'elle sait où les informations sont stockées en mémoire. Par exemple le code suivant va initialiser chaque élément d'un tableau à la valeur `val` (ici 1). Ce code peut être analysé sous C tutor pour bien comprendre ce qui se passe.

```
1  void inittab(float tab[], int N, float val)
2  {
3          int i;
4          for (i=0; i<N; i++) tab[i] = val;
5          return;
6  }
7
8  int main()
9  {
10         float A[10];
11
12         inittab(A,10,1);
13
14         return 0;
15 }
```

Sauf cas particulier, modifier un tableau ne signifie pas changer l'endroit où il est stocké en mémoire, mais bien en modifier le contenu. On est bien dans un mécanisme de pointeur ou on indique un endroit en mémoire et on en modifie le contenu. Vous remarquerez que l'on passe la taille du tableau en argument. Comme le tableau n'est qu'un pointeur, C va

connaître la taille d'un élément du tableau (donné par le type pointé) mais ne peut pas connaître le nombre d'éléments et donc la taille totale qu'occupe le tableau en mémoire. On doit donc systématiquement indiqué la taille lorsqu'on passe un tableau en argument.

5.4 Renvoi d'un tableau

Comme la mémoire allouée automatiquement pour une fonction est automatiquement libérée lorsqu'elle se termine, on ne peut pas renvoyer un tableau qui a été déclaré localement. Essayez de compiler par exemple le code suivant (ne peut pas être analysé sous C tutor) :

```
1  // fichier rettab.c
2  float *inittab(int N, float val)
3  {
4          float tab[N];
5          int i;
6          for (i=0; i<N; i++) tab[i] = val;
7          return tab;
8  }
9
10 int main()
11 {
12         float *tab = inittab(10,1);
13         return 0;
14 }
```

Vous avez un avertissement `warning: address of stack memory associated with local variable 'tab' returned` vous indiquant que vous renvoyez l'adresse d'une variable allouée sur la pile (*stack*) mais qui n'est plus valide : en effet la variable `tab` est locale à `inittab` et la mémoire qui lui est réservée est donc libérée dès que la fonction se termine. Elle n'est donc plus valide une fois qu'on retourne à `main`.

5.5 Exercice : Questions notées

1. Écrivez une fonction qui prend en argument un tableau de `float` (ainsi que sa taille) et ajoute 1 à chaque élément. Vous me rendrez le résultat sous la forme d'un fichier nommé `ajoute.c`. Ce fichier doit pouvoir être compilé puis exécuté. Il doit donc avoir une fonction `main` en plus de la fonction demandée.
2. Faites en sorte que le renvoi de tableau du dernier exemple fonctionne. On utilisera pour cela la fonction `malloc`. Attention à bien utiliser en regard la fonction `free` au bon endroit dans votre fonction `main`. Vous me rendrez cette question sous forme du fichier nommé `alloue_tab.c`.

6 Fichiers (Optionnel)

6.1 Entrée/sortie standard

Nous avons déjà vu la fonction `printf` qui permet d'écrire une chaîne de caractères sur l'écran, autrement dit ce qu'on appelle la *sortie standard*. Le premier argument est appelé *format* et comprend des séquences commençant par `%` qui se verront remplacées par la valeur des variables mises dans la suite des arguments de `printf` (voir le TP précédent pour les différents formats, et la commande shell `man -s 3 printf` pour une documentation extensive).

Sa contrepartie est la fonction `scanf` qui permet de lire des valeurs sur l'entrée standard, c'est-à-dire entrées au clavier par l'utilisateur (un peu comme `input` en python). `scanf` prend les mêmes arguments que `printf`, à savoir une chaîne de caractères qui indique le format, puis une suite d'arguments qui sont chacun associés à une séquence commençant par `%`. La différence, est qu'ici la valeur de ces variables est lue dans la chaîne de caractères entrées au clavier. Du coup, ces variables doivent être passées par variable : autrement dit on indique un pointeur vers ces variables à la fonction `scanf`.

Par exemple, testez le petit code suivant

```
// fichier input.c
#include <stdio.h>

int main()
{
    char c;
    int i;
    float f;
    char s[10];
    int ret;
    ret=scanf("Un char %c, un entier %d, un reel %f, une chaine %s et c'est tout.", &c, &i, &f, s);
    printf("Resultat (%d): %c // %d // %f // %s //\n", ret, c, i, f, s);
    return 0;
}
```

`scanf` renvoie le nombre d'arguments correctement lus (stocké dans `ret`). **On remarqura qu'on prend l'adresse de tous les paramètres de `scanf`, sauf `s`.** `s` est en effet une chaîne de caractères, donc un tableau et on cherche à renseigner non pas l'adresse de ce tableau mais bien tous les éléments qu'il contient. On passe donc simplement l'adresse de ces éléments, c'est-à-dire `s` elle-même.

Testez avec différentes entrées au clavier et vous verrez que cette fonction est assez délicate à manipuler. À noter toutefois que la forme `%*` permet d'indiquer que le format doit contenir un certain élément, dont on indique le format, mais dont la valeur ne nous intéresse pas : cette marque de formatage n'aura donc pas de variable associée dans la liste d'arguments. Par exemple :

```
char pays[256];
float temp;
scanf("%s %*s %f\n", pays, &temp);
```

Lira bien une ligne qui contient une première chaîne de caractères (sans espace, ni tabulation), qui correspond au nom d'un pays, puis une deuxième chaîne de caractères (sans espace, ni tabulation), mais qui ne nous intéresse pas, puis un réel qui indique une température. On remarquera l'allocation automatique de la variable `pays`, suffisamment grande pour accueillir un nom de pays (dont la longueur est a priori variable et dont on ne connaît pas la longueur maximale, il faut donc utiliser une borne haute raisonnable, ici 256).

6.2 Fichiers

Il existe les mêmes fonctions pour les fichiers. Elles ont pour noms `fprintf` et `fscanf`. Elles prennent les mêmes arguments que `printf` et `scanf`, sauf qu'il faut mettre en premier un nouvel argument : un pointeur vers un fichier. Une variable fichier est de type `FILE` : elle se crée par la fonction `fopen` qui prend en paramètre une chaîne de caractères (nom d'un fichier), ouvre ce fichier, et renvoie un pointeur vers un `FILE`. Le pointeur est libéré par un appel à `fclose` qui ferme également le fichier précédemment ouvert.

Un fichier peut être ouvert en lecture et/ou en écriture. Afin de spécifier le mode d'ouverture, la fonction `fopen` prend un deuxième argument sous forme de chaîne de caractères : "r" pour ouvrir en lecture seule, "w" pour ouvrir en écriture (écrase le contenu du fichier, ou crée le fichier s'il n'existe pas), ou "a" pour ouvrir en écriture (se positionne en fin de

fichier, ce qui permet donc d'en étendre le contenu). `fopen` renvoie le pointeur `NULL` en cas d'échec. Voir l'aide en ligne du shell par la commande `man -s 3 fopen`.

L'exemple suivant fait simplement une recopie d'un fichier, passé en argument du programme, dans le fichier passé en second argument du programme : on suppose juste ici que le premier fichier contient une liste de réels, ces réels étant recopiés, un par ligne, dans le fichier en sortie. Le fichier `reels.txt` nécessaire à ce programme est fourni dans l'archive.

```
1  // fichier copie.c
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6     char fichierIn[]="reels.txt";
7     char fichierOut[]="reels_copie.txt";
8
9     FILE *fIn, *fOut;
10    float val;
11
12    // ouvre le premier fichier en lecture
13    fIn = fopen(fichierIn, "r");
14
15    // test si tout s'est bien passe
16    if (!fIn)
17      {
18        fprintf(stderr, "Impossible d'ouvrir le fichier %s en lecture\n", fichierIn);
19        return 2;
20      }
21
22    // si on est la, c'est que tout va bien. On ouvre le deuxieme fichier en ecriture
23    fOut = fopen(fichierOut,"w");
24    // on test encore si tout est ok
25    if (!fOut)
26      {
27        fprintf(stderr, "Impossible d'ouvrir le fichier %s en ecriture\n", fichierOut);
28        return 3;
29      }
30
31    // realise la copie: on suppose que le fichier ne contient que des valeurs reelles
32    while (fscanf(fIn, "%f", &val) == 1)
33      fprintf(fOut, "%f\n", val);
34
35    // ferme les fichiers ouverts
36    fclose(fIn);
37    fclose(fOut);
38
39    return 0;
40 }
```

On remarquera l'usage de `stderr`, lignes 18 et 27 : il s'agit d'un fichier ouvert par défaut vers la sortie d'erreur standard. Il existe deux autres fichiers ouverts par défaut : `stdout` pour la sortie standard et `stdin` pour l'entrée standard, ce qui fait que `printf(...)` est équivalent à `fprintf(stdout, ...)` et `scanf(...)` est équivalent à `fscanf(stdin, ...)`. On remarquera par ailleurs l'usage de valeurs non nulles renvoyées par le programme en cas d'erreur. La boucle `while` en lignes 32 et 33 permet de réaliser la copie, celle-ci s'arrêtant quand on ne peut plus lire de réel. Une autre manière de tester si nous sommes arrivés à la fin d'un fichier serait d'appeler la fonction `feof(fIn)` qui renvoie 0 tant que nous ne sommes pas à la fin du fichier. Pour plus d'informations sur la lecture et écriture dans un fichier, je vous invite à aller aussi vous renseigner sur les fonctions `fwrite` et `fread`, et ne pas vous arrêter là car bien d'autres possibilités existent que nous n'avons pas le temps de voir ici.

6.3 Exercice : Question notée (Optionnelle)

Ecrivez un programme, que vous nommerez `readDB`, correspondant au fichier source `readDB.c`, qui prend en entrée un fichier contenant sur chaque ligne un nom, suivi d'un prénom, puis un âge et enfin un genre (M ou F). Le programme renverra :

1. Le nombre de personnes (autrement dit le nombre de lignes)
2. Le nombre d'hommes et de femmes
3. L'âge moyen de ces personnes

Par exemple si je donne le fichier suivant en entrée (fichier `DB.txt`) :

```
Enfant Hélène 44 F
Enfant Ludivine 47 F
Flaille Abdel 19 M
Flaille Akim 23 M
Flaille Yves 21 M
Neymar Jean 24 M
Titegoute Justine 78 F
Yapudebiairedenlefrigo Robin 67 M
```

Le programme répondra :

```
Le fichier contient 8 noms de personnes, dont 5 hommes et 3 femmes, avec un âge moyen de 40.375 ans.
```

7 Bonus : arithmétique des pointeurs et conversion

Nous avons vu l'opération d'indirection (*) sur un pointeur. Il existe une autre opération qu'on peut faire : c'est le décalage. Il permet de modifier l'adresse contenue dans le pointeur vers la fin de la mémoire (opération +) ou vers le début (opération -). Il est important de noter que le premier terme de l'addition (soustraction) est forcément une adresse (pointeur) et le second est un entier long. Cet entier long indique le nombre d'éléments dont il faut se décaler (et non pas le nombre d'octets : on se décale d'un nombre entier d'éléments).

Compilez et lancez le code suivant pour bien comprendre ce décalage :

```
1  // fichier arithptr1.c
2  #include <stdio.h>
3
4  int main()
5  {
6          int *tab;
7
8          printf("Pointeur : %p\n", tab);
9          printf("Decalage de +1: %p\n", tab+1);
10         printf("Decalage de -1: %p\n", tab-1);
11         return 0;
12 }
```

Remarques :

- Je n'ai pas besoin d'initialiser le pointeur en ligne 6 : je ne vais pas faire d'indirection et donc il n'y a pas de souci d'accès mémoire.
- Même si je ne l'initialise pas, `tab` contient quand même une valeur (ce que le dernier processus à accéder à cet espace mémoire y avait écrit là).
- Cette initialisation n'est pas à 0. Il faut le faire à la main si c'est important dans votre code. Initialisez vos variables !
- On remarque que le décalage est bien de 4 octets (en + ou en -), ce qui correspond bien à 1 `int`

Du coup, on peut comprendre comment se fait l'accès à un élément quelconque d'un tableau. Compilez et lancez le code suivant :

```
1  // fichier arithptr2.c
2  #include <stdio.h>
3
4  int main()
5  {
6          int tab[]={1,2,3,4};
7
8          printf("Pointeur : %p\n", tab);
9          printf("1er element : %p <-> %p\n", tab, &tab[0]);
10         printf("2e element : %p <-> %p\n", tab+1, &tab[1]);
11         printf("3e element : %p <-> %p\n", tab+2, &tab[2]);
12         printf("4e element : %p <-> %p\n", tab+3, &tab[3]);
13
14         printf("1er element : %d <-> %d\n", *tab, tab[0]);
15         printf("2e element : %d <-> %d\n", *(tab+1), tab[1]);
16         printf("3e element : %d <-> %d\n", *(tab+2), tab[2]);
17         printf("4e element : %d <-> %d\n", *(tab+3), tab[3]);
18         return 0;
19 }
```

Accéder à l'élément de rang `i` revient donc à décaler le pointeur de `+i` et déréférencer l'adresse obtenue.

Enfin, comme tout pointeur, quelque soit le type pointé, est une adresse, on peut sans aucun souci modifier le type pointé grâce à une conversion de type. Une conversion de type en C est très simple à faire (trop parfois, ce qui peut poser de sacrés soucis si on ne fait pas attention... n'hésitez pas à expérimenter !). Nous allons pouvoir utiliser ça pour regarder un à un les octets qui permettent de coder un entier. Compilez, lancez et analysez le code suivant :

```
1  // fichier byteperbyte.c
2
3  #include <stdio.h>
4
5  int main()
6  {
7          int i=1; // declaration d'un entier i initialise a 1
8          int *pi=&i; // stockage de l'addresse de i dans le pointeur pi
9          char *pc=(char *)&i; // idem mais avec un pointeur char *
10
11         int j;
12
13         printf("Valeur: %d\n", i);
14         printf("Via le pointeur entier: %d\n", *pi);
15         printf("Via le pointeur char: ");
16         // lecture octet par octet de l'espace memoire de i
17         for (j=0; j<sizeof(int); j++) printf("%d ",pc[j]);
18         printf("\n");
19         return 0;
20 }
```

Qu'en déduisez-vous ?

Changez le code pour initialiser i à -1. Qu'obtenez-vous ? Pourquoi ?