Programmer en langage C
Cours et exercices corrigés [1]

Gabriel Dauphin
L2TI, Institut Galilée, Université Paris 13 Sorbonne Paris Cité
99, Avenue Jean-Baptiste Clément 93430 Villetaneuse, France
gabriel.dauphin@univ-paris13.fr

September 4, 2024

[1]Ce document a été rédigé pour les étudiants de Sup Galilée en Télécom 2 dans le cadre d'un cours d'harmonisation en C composé de 5 séances d'une heure trente. Il s'appuie sur un polycopié de C ainsi qu'un complément sur les listes chaînées. Les travaux pratiques sont organisés en six séances de deux heures correspondant chacune à une section de ce document. Il est nécessaire de finir tous les exercices non-supplémentaires avant de passer à la séance suivante.

Contents

Chapter 1

Types, variables, opérateurs, exécution conditionnelles, boucles, affichage des résultats

À propos de la saisie

Dans tous les exercices qui suivent, les valeurs des variables ne sont pas déterminées à l'exécution par un appel à une entrée clavier. Elles sont définies dans la fonction `main` et cette fonction doit résoudre l'exercice et afficher le résultat.

À propos des accents

Dans le cadre de cette formation, nous n'utiliserons pas les accents. Le C gère les accents uniquement en tant que possibles éléments d'une liste de 256 caractères incluant ou non des caractères spécifiques à chaque pays. Une solution possible serait de les gérer avec `wchar_t`, il faudrait aussi adapter le reste (inclusion de bibliothèques, utilisation d'autres fonctions que `printf` et `scanf_s`, utilisation d'autre format que `%s` et `%c`, place mémoire plus grande, et gestion de l'interface avec la fenêtre où le message doit être affiché ou saisi).

À propos de l'affichage et des types

Ce chapitre s'appuie sur les parties 2.1 et 2.2 ainsi que sur la page 58 du polycopié de C. La fonction `printf` qui permet d'afficher est rappelée dans la partie 5.1 du polycopié de C.

L'adéquation entre le format et le type de la variable à afficher n'est pas vérifié par le compilateur. À l'exécution, cela ne provoque pas d'erreur mais l'affichage est absurde et c'est une erreur parfois difficile à trouver. Il est donc nécessaire de vérifier systématiquement une fois que le programme compile tous les fonctions `printf` (et aussi les éventuelles fonctions `scanf` et `sprintfs` qui seront utilisées ultérieurement). L'erreur peut consister à utiliser `%d` au lieu de `%lf`, mais cela peut être aussi de cherche à afficher ce qu'on croit être une valeur et qui est en fait une adresse mémoire.

À propos de l'affichage de caractères spéciaux

Lorsque l'on souhaite afficher les caractères spéciaux tels que `%`,
ou le passage à la ligne, on utilise les caractères suivants

```
printf("%% \\ \n");
```

À propos de l'utilisation de `getchar()`

Il est conseillé de terminer la fonction `main` avec l'instruction `getchar();`. Cette instruction permet en effet d'attendre que l'utilisateur ait appuyé sur la touche `Return` avant de permettre éventuellement que le programme se termine et supprime l'affichage réalisé.

1.1 Algorithme utilisant une variable temporaire

On est généralement amené à utiliser une telle variable lorsqu'on souhaite écrire le résultat du calcul sur les mêmes variables que celles qui contiennent les valeurs en entrée. L'utilisation d'une variable temporaire permet d'écrire sur une variable sans que son contenu soit perdu.

Exercice 1 *On cherche à effectuer une permutation circulaire de trois nombres* a, b, c

$$a' = b, \quad b' = c, \quad c' = a$$

1.2 Algorithme de parcours séquentiel avec un nombre d'itérations connues

L'implémentation de ce genre d'algorithmes se fait avec `for` qui est rappelé dans la section 2.2 du polycopié de C. Les questions à se poser sont les suivantes :

1. Que doit calculer la boucle `for` ?

   Quelle est la signification de la variable introduite dans l'instruction `for`, s'agit-il d'un compteur du nombre d'itérations effectués, de l'indice n d'une suite ou d'une autre notion ? Cette variable ne doit pas être modifiée à l'intérieur de la boucle for, plus précisément c'est le troisième argument de l'instruction `for` qui détermine cela. Pour aider le lecteur, il est préférable que la condition d'arrêt de la boucle `for` soit formulée avec une inégalité stricte, les instructions utilisées dans la boucle `for` peuvent s'appuyer justement sur le fait que cette condition est forcément vérifiée.

2. Écrire le code dans la boucle `for`

Quelles sont les quantités dont la boucle devra avoir calculée ? Ces quantités doivent être initialisées. Ces quantités ne coïncident pas toujours avec les quantités demandées par l'exercice. Ainsi pour un programme demandant de calculer la moyenne, il est nécessaire de réaliser une boucle qui calcule la somme pour ensuite dans le cadre d'un post-traitement de déduire de cette somme la moyenne.

3. Écrire le code dans la ligne formée par l'instruction `for`

   En fonction des quantités à calculer et des relations imposées entre une itération et la suivante, quelles sont les quantités à calculer et à retenir pour l'itération suivante ?

4. Vérifier la premier et la dernière instruction

   Si la première itération n'est pas identique aux autres, il est préférable d'implémenter cette différence avec une expression conditionnelle plutôt que d'avoir du code similaire à deux endroits différents.

Exercice 2 *On considère une suite définie par* $u_{n+1} = au_n + b$ *et* $u_0 = 1$. *Connaissant* a, b *et* N *on cherche à calculer* u_N *et* $\Sigma_{n=0}^{N} u_n$

1.3 Algorithme avec un parcours séquentiel avec un nombre maximum d'itérations inconnues

Ce genre d'algorithmes peut s'implémenter avec un `while` ou un `do while` suivant que la condition pour répéter les instructions se fait avant ou après l'itération. Les questions à se poser sont les suivantes :

1. Que doit faire la boucle `while` ?

   Quelles sont les entrées et les sorties ? Il est possible d'avoir à effectuer un traitement avant ou après la boucle. Quelles sont les quantités à calculer ? Ces quantités doivent être initialisées.

2. Écrire le code dans la boucle `while`

   Le sens des variables n'est pas le même à gauche et à droite du signe `=`.

   La tâche à itérer doit à terme pouvoir modifier la condition qui contrôle la répétition.

3. Rajout d'un compteur

   Il peut être nécessaire d'introduire un compteur du nombre de fois que la boucle est réalisée. Le nom du compteur est important pour la compréhension du code. Il doit y avoir un code pour initialiser le compteur avant la boucle et un code pour l'actualiser dans la boucle.

4. Écrire le test dans l'instruction `while`

   La condition qui entraîne la répétition ne doit en général pas se limiter au fait de constater que le problème n'est pas résolu, il faut aussi s'assurer qu'il n'y ait pas répétition infinie de l'exécution des instructions.

   Il importe aussi de se souvenir que la négation d'un OU entre deux conditions est un ET entre les deux négations des deux propositions.

   Si cette condition est compliquée à implémenter, elle peut être effectuée dans la boucle `while` et sauvegardé dans une variable bouléenne à initialiser et à contrôler dans l'instruction `while`.

5. Vérifier que la dernière itération est correcte ?

   Est-ce que la dernière condition est correcte ? Est-ce qu'elle ne modifie pas les quantités é calculer d'une fois en trop ? Est-ce que les traitements faits à la dernière itération sont autorisés ?

6. Vérifier que la première itération est correcte ?

   Comment traiter la première itération ? Si la première itération n'est pas identique aux autres, il est préférable d'implémenter cette différence avec une expression conditionnelle plutôt que d'avoir du code similaire à deux endroits différents.

Exercice 3 *On considère une suite définie par* $u_{n+1} = u_n^2 + u_n + 1$ *et* $u_0 = 1$. *On se donne un seuil noté* s, *calculez la plus petit valeur de* n *telle que* $u_n \geq s$. *Cet exercice peut se voir comme une modification de l'exercice 2.*

Ce genre d'algorithmes de cette section ou de la section 1.2 peut aussi s'appliquer lorsqu'il s'agit de faire un traitement sur des grandeurs. Plus précisément, une première analyse du problème montre que des grandeurs peuvent être modifiées par un certain nombre de relation. Ensuite une deuxième analyse montre qu'une utilisation particulière de ces modifications peut conduire au résultat souhaité. C'est le cas de l'exercice 5 (p. 19) ou de l'exercice 17 (p. 46).

Un outil fréquemment utilisé pour les exercices sur les nombres entiers est le calcul du modulo, il s'agit du reste de la division euclidienne de a par b et qui est implémenté par % : $7\%2 = 1$ signifie que $7 = 3 \times 2 + 1$. Il est par exemple utilisé dans l'exercice 5 (p. 19) et 15 (p. 46). Le quotient de la division euclidienne de `a` par `b` est implémenté avec `/` lorsque `a` et `b` sont des entiers : `7/2=3`.

1.4 Algorithme utilisant une structure conditionnelle imbriquée

On peut traiter ce problème de deux façons.

- Le premier point de vue consiste à considérer qu'un certain nombre d'événements

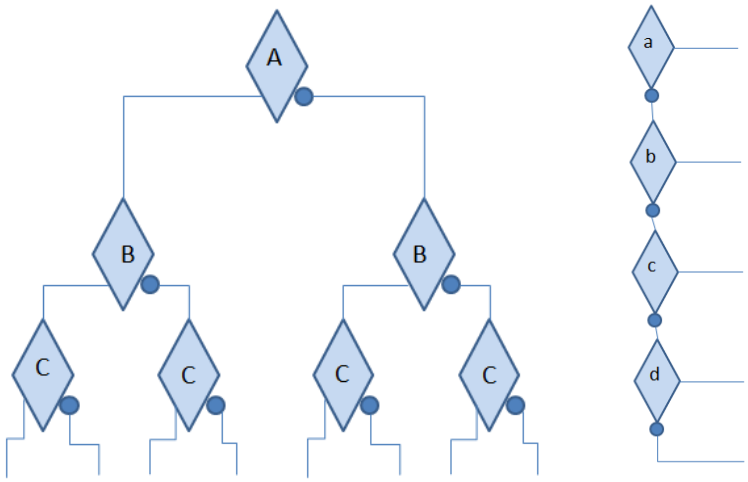

Figure 1.1: Deux schémas de structures conditionnelles

indépendants les uns des autres peuvent se comporter de telle ou telle façon. Dans ce cas on peut dessiner un arbre d'événements comme sur la gauche de la figure 1.1. Les losanges représentent des tests, l'issue défavorable est indiquée par un petit cercle suivi d'un trait tandis que l'issue favorable est indiquée par l'absence de petit cercle. Le sens de la lecture est du haut vers le bas et de la gauche vers la droite. L'implémentation se fait alors avec un ensemble d'instructions `if` et `else` et les instructions sont dans l'ordre du parcours de l'arbre en explorant en profondeur : le parcours se fait vers la gauche tant que c'est possible puis vers le noeud voisin puis vers le noeud père.

```
if (A)
```

```
    if (B)
      if (C)
        ...
     else
        ...
    else
      if (C)
        ...
    else
        ...
else
    if (B)
      if (C)
        ...
    else
        ...
    else
      if (C)
        ...
    else
        ...
```

- Le deuxième point de vue consiste à organiser les tests à partir des résultats recherchés. Pour chaque conséquence, on cherche à écrire la condition globale éventuellement en utilisant les opérateurs || et &&. On peut écrire alors l'ensemble des conditions sous la forme de la partie droite de la figure 1.1. Les losanges représentent des tests, l'issue défavorable est indiquée par un petit cercle suivi d'un trait tandis que l'issue favorable est indiquée par l'absence de petit cercle.

```
if (a)
    ...
else if (b)
    ...
else if (c)
    ...
else if (d)
    ...
else
    ...
```

  Cette organisation se rapproche de l'utilisation d'un **`switch`**.

Quand l'ensemble des conditions devient complexe, il est alors préférable de séparer une partie du traitement et de mettre cette partie dans une fonction ainsi qu'il est décrit dans la section 2.4 (p. 37).

Exercice 4 *On cherche à situer un nombre donné parmi trois valeurs a, b, c classées par ordre croissant.*

- *Combien y a-t-il de classements possibles ?*
- *En déduire un premier algorithme sans boucle ni variables intermédiaires comportant un nombre de tests égal aux nombres de classements ?*
- *Écrire le programme correspondant.*

On cherche maintenant à mettre des tests en commun.

- *Dessiner un arbre associé aux tests successifs qu'il faudra faire ?*
- *Écrire le programme correspondant aux tests de l'arbre.*

Valeur absolue

La valeur absolue se code avec `fabs` de la librairie `math.h` et non `abs` qui réalise d'abord une conversion en entier avant de calculer la valeur absolue.

1.4.1 Conseils de style

1.5 Conseils de style

Pour le placement des accolades, on pourra utiliser le schéma suivant

```
int main(void)
{
  int i;
  for(i=0;i<taille;i++) {
    printf("%d ",i);
  }
}
```

Dans la mesure du possible, quand on fait une instruction de test avec `==`, on cherche à mettre la constante à gauche du `==` Par exemple le code suivant

```
#include <stdio.h>
int main(void)
{
  int i=2;
  if (1==i) printf("Bizarre\n");
}
```

Dans ce code, si jamais on a mis `=` au lieu de `==`, il y aura détection d'une erreur à la compilation, alors que le code suivant affichera Bizarre.

```
#include <stdio.h>
int main(void)
{
  int i=2;
  if (i=1) printf("Bizarre\n");
}
```

1.6 Exercices supplémentaires

Exercice 5 *On cherche à implémenter un algorithme appelé* multiplication à la russe. *Observant que*

$$si\ x\ est\ paire,\ \ xy = \left(\frac{x}{2}\right)(2y)$$
$$si\ x\ est\ impaire,\ xy = (x-1)\,y + y$$

Écrivez un programme permettant de faire la multiplication entre deux entiers. Pour cela vous pourrez utiliser la fonction modulo qui s'implémente pour les en-

tiers positifs avec un pourcentage, x est paire si $x\%2 == 0$ et x est impaire si $x\%2 == 1$.

Exercice 6 *On cherche à calculer le terme u_N d'une suite définie par $u_{n+2} = u_{n+1} + u_n$ connaissant u_0, u_1 et N.*

Exercice 7 *On cherche à effectuer un classement de trois nombres a, b, c, les valeurs une fois classée sont notées a', b', c' et vérifient $a' \leq b' \leq c'$. En utilisant des fonctions* min *et* max, *cet exercice pourrait se résoudre de la façon suivante :*

$$a' = \min(a,b,c), \quad b' = a+b+c-\min(a,b,c)-\max(a,b,c), \quad c' = \max(a,b,c)$$

Mais ici, on ne souhaite pas utiliser de telles fonctions, et utiliser à la place des instructions de tests.

- *Combien y a-t-il de classements possibles ?*
- *Écrire un programme sans boucle ni variables intermédiaires comportant un nombre de tests égal aux nombres de classements ?*

On cherche maintenant à mettre des tests en commun.

- *Dessiner un arbre associé aux tests successifs qu'il faudra faire ?*

- *Écrire le programme correspondant à l'arbre de décision.*

Chapter 2

Tableaux, tableaux de tableaux, pointeurs, fonctions

Dans cette partie, la fonction `main` génère les valeurs et les tableaux. On suppose que l'on connaît à l'avance les tailles des tableaux et que donc on peut utiliser une allocation statique des tableaux. Cette fonction `main` transmet ces valeurs à une fonction qui elle réalise la tâche indiquée dans l'exercice. Une fois la tâche réalisée la fonction retransmet les valeurs obtenues à la fonction main.

On ne doit pas utiliser de variables globales, celles-ci doivent donc être définies dans les fonctions. En revanche les commandes `#include`, `struct`, `enum`, `typedef` et `#define`

doivent être en début de fichier en dehors des fonctions.

2.1 Utilisation de tableaux, chaînes de caractères et de fonctions

L'utilisation des tableaux est décrite dans le polycopié de C section 2.31 é 2.34 (p. 20 é 24).

L'utilisation des chaînes de caractères est décrite sur le polycopié de C en section 2.34 (p. 24-25). A ceci prés, que dans le cadre de ce cours, on s'interdira d'utiliser `const char * m="bonjour";` ou `char * m="bonjour";`[1] La déclaration d'une chaîne de caractère peut se faire par une des façons suivantes :

```
char mot[10];
char mot[10]="bonjour";
const char mot[]="bonjour";
```

Dans le premier cas, on alloue de la place pour une chaîne de caractère de 9 lettres. Le deuxième cas est similaire au premier, mais on remplit les 8 premiers octets avec les

[1]La raison poudeuxièmer laquelle on s'interdit cela ici, est dans le premier cas uniquement pour simplifier l'utilisation des chaînes de caractères, les rendre plus similaires au tableau de chiffres et aider à l'utilisation de la fonction `qsort` ; dans le deuxième cas, l'écriture induit le lecteur en erreur, car comme dans le premier cas `m` pointe sur une zone de mémoire non-modifiable et que `m[0]='u'` provoque une erreur à la compilation.

caractères bonjour\0. Dans le troisième cas on alloue 8 octets pour y mettre les 7 lettres de bonjour et le mot ne peut être modifié.

L'utilisation des fonctions est décrite dans le polycopié de C section 2.4 à 2.43 (p. 26 é 29).

Une fonction reçoit en entrée des valeurs, des tableaux et des chaînes de caractères. Elle peut faire sortir une valeur ou modifier une des valeurs, tableaux ou chaînes de caractères transmis en entrée. Dans le cadre de ce cours, on évite de retransmettre un tableau, une chaîne de caractère ou un ensemble de valeurs à travers la sortie de la fonction. Et dans le cas où on transmettrait un tableau ou une chaîne de caractère, cela signifierait que ce tableau aura été alloué à l'extérieur de la fonction, l'adresse correspondante aura été transmise dans les entrées[2]. On s'interdit ici de réserver un ensemble de places mémoire dans une fonction à moins qu'il n'y ait une façon habituellement utilisée pour déreserver ces places mémoire, c'est le cas des listes chaînées[3]. Au lieu de cela, la fonction qui appelle doit allouer la place mémoire sur laquelle la fonction appelée écrira, il est d'ailleurs souvent possible d'écrire les données sur les données transmises et l'absence d'un `const` dans la déclaration pour cette donnée peut faire penser que c'est ce que la fonction va faire. Ces informations que la fonction reçoit en entrée s'appellent la liste des arguments, elles sont dans les parenthèses lors de l'appel.

Dans l'utilisation d'une fonction, on distingue, la déclaration, la définition et l'appel.

[2]Un exemple de fonction qui utilise ce genre de syntaxe en C est `strstr`. Le problème que cela pose est qu'il y a ambiguïté sur le fait de savoir le programme qui a appelé la fonction est chargée de désallouer la place mémoire

[3]Un exemple de fonction en C qui utilise ce type de syntaxe est `fopen` et `fclose` ou `malloc` et `free`.

- La déclaration permet d'indiquer au compilateur l'existence d'une fonction utilisant tel ou tel argument et retournant tel argument.
- La définition donne au compilateur l'ensemble des instructions qui doivent être exécutés lorsqu'une autre instruction appelle la fonction.
- L'appel se fait dans une instruction et consiste en le nom de la fonction suivi de la liste des arguments entourée de parenthèses.

L'appel et la définition/déclaration dépendent du type d'argument [4].

- L'argument correspond à un entier, un caractère ou un réel.

  Si on ne souhaite pas modifier cette valeur alors l'appel se fait avec le nom de la variable. La définition ou la déclaration se font avec `const` suivi du type et suivi de `a`.

  Si on souhaite que cette valeur soit modifiée, alors on transmet l'adresse de la variable. L'appel se fait avec l'adresse de la variable (`&a`). La définition ou la déclaration se fait avec le nom du type suivi de `*` et suivi du nom du pointeur de la variable, c'est ce pointeur qui est alors utilisé dans la fonction.

- L'argument correspond à un tableau d'entiers ou de réels

  Pour transmettre un tableau, il faut aussi transmettre la taille de ce tableau qui identifient les cases que la fonction pourra utiliser.

[4] On appelle ainsi une donnée fournie à une fonction ou transmise par une fonction.

- Dans l'appel on met le nom de la variable associée au tableau ainsi que la taille.
- Si la fonction ne modifie pas les valeurs du tableau, l'argument est constitué de `const` suivi du type suivi du nom de la variable suivi de `[]`, ainsi que de `const` suivi de `int` suivi du nom de la variable associé à la taille.
- Si la fonction modifie les valeurs du tableau, l'argument est constitué du type du nom de la variable suivi de `[]`, ainsi que de `const` suivi de `int` suivi du nom de la variable associé à la taille.

- L'argument correspond à un tableau de caractères

  Une chaîne de caractère est similaire à un tableau, à ceci prés que la taille peut se déduire du fait que la dernière case est occupée par `\0`, il n'y a pas de taille à transmettre en revanche il faut prévoir une case en plus de la longueur de la chaîne que l'on veut stocker.

  - Dans l'appel on met le nom de la variable associée à la chaîne de caractère.
  - Si la fonction ne modifie pas la chaîne de caractères, l'argument est constitué de `const` suivi de `char` suivi du nom de la variable suivi de `[]`.
  - Si la fonction modifie les valeurs de la chaîne de caractères, l'argument est constitué de `char`, suivi du nom de la variable suivi de `[]`.

- L'argument est un pointeur de pointeur

C'est le cas d'une des implémentations des tableaux 2D (voir le polycopié de C section 2.3.5 p. 25 et ici dans la section 2.3 et 34), c'est aussi le cas des tableaux de chaînes de caractères (voir la section 2.6 et 40). C'est aussi le cas pour permettre qu'une fonction manipule des valeurs d'un tableau sans que la déclaration de cette fonction n'est à préciser le type de ces valeurs (voir la section 4.3 et 73). C'est aussi le cas si l'allocation du tableau se fait dans la fonction.

Par ailleurs si on souhaite retourner une valeur, alors dans l'appel on pourra récupérer cette valeur en mettant par exemple `=` à gauche du nom de la fonction. Dans la déclaration et la définition, on met à gauche de la fonction le type correspondant à la valeur que l'on souhaite retourner. Dans la définition, la valeur transmise est indiquée par l'argument placé à droite de la fonction **`return`**. Si on ne souhaite pas retourner de valeurs alors dans la définition et la déclaration, on met **`void`** à gauche du nom de la fonction. Dans la définition, la fonction **`return`** n'est pas forcément présente et quand elle l'est, il n'y a pas d'arguments.

Le qualificatif **`const`** ne garantit pas que la variable ainsi qualifiée ne soit pas modifiée, elle garantit qu'il n'y a aucune instruction qui puisse la modifier. Mais rien n'interdit que cette variable ne soit modifiée par l'intermédiaire d'une instruction agissant sur un autre nom de variable. [5]

Ce qualificatif permet de mieux comprendre le fonctionnement d'une fonction à partir de sa déclaration, en effet les arguments qui contiennent ce qualificatif ne peuvent être

[5] La modification d'une chaîne de caractère définie comme constante est possible avec `strcpy_s` et avec `memcpy_s`, cette modification entraîne seulement un Warning de type `C4090` indiquant `'function' : different 'const' qualifiers`.

modifiés par la fonction, ainsi dans l'instruction `strcpy`, si l'on hésite entre ce qui est copié et ce sur quoi la copie est faite, le fait de savoir où est placé le qualificatif `const` permet de lever l'ambiguïté.

Exercice 8 *On cherche à effectuer une permutation circulaire de trois nombres* a, b, c

$$a' = b, \quad b' = c, \quad c' = a$$

On cherche à implémenter cette fonction de façon à ce que l'action de cette fonction modifie les valeurs des variables a, b, c *du programme. La syntaxe de la fonction à utiliser est :*
`permutation(double * pa,double * pb,double * pc)` *Vous pouvez reprendre l'exercice 1 (p. 9)*

L'algorithme suivant est similaire à celui de la section 1.2 et 1.3. La variable à retenir discutée à la question 3 p. 10 peut être une entier ou un Bouléen. Un Bouléen est une variable qui ne peut deux valeurs possibles vrais faux. Ce type n'existe pas dans la version de base du C. On se propose de le construire en faisant appel au type `enum` avec la syntaxe suivante

```
typedef enum Boul {
  FALSE=0,
  TRUE=1
```

```
} Boul;
```

Mais comme ce type n'est pas reconnu comme un Bouléen par le C, dans le cadre de ce cours, on évitera de chercher à faire correspondre le résultat une valeur de ce Bouléen. Au lieu de cela, il suffit de faire un test[6]. Ainsi si l'on souhaite affirmer que la variable Bouléenne `test` est vraie quand $3 > 2$, il suffit d'écrire

```
Boul test;
if (3>2) test=TRUE;
else test=FALSE;
```

De même la valeur prise par cette variable Bouléenne n'est pas reconnue comme la valeur d'un Bouléen aussi pour la tester il suffit de faire

```
if (TRUE==test) ...
```

Exercice 9 *Écrire un traitement qui informe si un un tableau envoyé en argument est formé ou non d'éléments tous rangés en ordre croissant.*

[6] L'alternative au test consiste à faire une conversion de type en mettant devant l'expression à évaluer `(Boul)`

Allocation très particulière des chaînes de caractères

Quand on utilise de `const char * m="bonjour";`, il n'y a pas allocation de la mémoire. Il se trouve simplement qu'une mémoire est de fait allouée quand dans le programme une chaîne de caractère est écrite et `m` pointe sur cette chaîne de caractère. Cette place mémoire sera désallouée à la fin de l'exécution du programme et il ne faut donc pas la désallouer avec un `free`. Cette zone mémoire est non-modifiable.

Il ne faut pas utiliser la séquence

```
char * m="bonjour";
```

L'écriture induit le lecteur en erreur, car comme dans le premier cas `m` pointe sur une zone de mémoire non-modifiable et que `m[0]='u'` provoque une erreur à la compilation.

2.2 Algorithmes utilisant des tableaux

Il est fréquent qu'un programme ait à gérer un tableau. Parfois aussi il est utile d'introduire un tableau pour réaliser une tâche qui pourtant ne porte que sur des données individuelles, un tel tableau peut ainsi permettre de stocker des données spécifiques (voir l'exercice 17 p. 46). Parmi les algorithmes utilisant des tableaux, on distingue les algorithmes suivants

2.2.1 Les algorithmes de parcours simple avec éventuellement une structure conditionnelle

Ces algorithmes ont été vus en section 1.2 et section 1.3 (p. 9 et 11). Lorsqu'on utilise ces algorithmes pour des tableaux, il faut aussi prévoir de sortir de la boucle si le fait de poursuivre amène à sortir des zones mémoires allouées.

Se rajoutent à ces algorithmes, ceux où la tâche à réaliser dans la boucle comporte également une expression conditionnelle décrite par la section 1.4 (p. 13). On peut aussi être amené à rajouter des variables temporaires dans les différentes tâches, mais si cette variable temporaire est un tableau, il est préférable d'utiliser des fonctions pour agir dessus, ce qui est décrit dans la section 2.4 (p. 37).

Un algorithme de recherche dichotomique peut se voir comme un cas particulier de ceux définis dans la section 1.3 (p. 11). Une itération est caractérisée par deux bornes repéré par deux indices délimitant dans un tableau la zone recherchée. Ces deux indices sont initialisés au premier et dernier élément du tableau. A chaque itération, l'algorithme fait la comparaison de la valeur indiquée à la case située au milieu des deux bornes et la valeur cible. Les deux indices sont modifiés en fonction du résultat de cette comparaison. L'algorithme prend fin lorsque les deux indices sont successifs. Il y a alors deux possibilités, le tableau contenait cet élément ou ne le contenait pas.

2.2.2 Les algorithmes de parcours avec une boucle imbriquée dans une boucle

Il est fréquent d'utiliser un algorithme formé d'une boucle imbriquée dans une autre boucle. Le bon fonctionnement repose sur ces idées.

- Il ne doit pas y avoir de chevauchements entre les deux boucles.
- Les éléments de la boucle interne (initialisation et progression des variables, condition de répétition, tâches é effectuer) peuvent dépendre de la boucle externe, en revanche l'inverse n'est pas possible.

Ce genre d'algorithmes peut permettre par exemple de faire un déplacement d'un caractère dans une chaîne de caractère en choisissant de répéter une action de déplacement le long d'un parcours particulier du tableau. Le tri à bulle peut se voir aussi comme un cas particulier de cet algorithme : la tâche répétée consiste en un échange entre termes successifs quand ils ne sont pas ordonnés. Cette tâche est répétée en parcourant autant de fois le tableau qu'il n'y a de cases dans le tableau (en fait on peut réduire un peu le nombre de fois que l'on répète cette tâche). Une petite variante de cet algorithme consiste en deux boucles successives imbriquée dans une autre boucle.

Exercice 10 *On se donne un tableau de nombre, ce tableau contient une case en plus des cases remplies. Modifiez ce tableau de façon à insérer un nouvel élément à une position particulière. Proposez deux solutions, l'une en parcourant*

en parcourant une partie du tableau dans l'ordre inverse, l'autre en utilisant deux variables temporaires.

2.2.3 Algorithmes parcourant de multiples tableaux ordonnés composés d'éléments n'ayant a priori aucun lien entre eux

Ne sont donc pas concernés ici le cas d'une liste de fiches dans chacun des champs serait stocké sous la forme d'un tableau. En effet si les différents tableaux ont une indexation commune,c'est-à-dire des cases de même rang correspondent aux mêmes fiches, il suffit de faire un parcours séquentiel. En fait, il est préférable de relier entre eux les tableaux d'une manière fixe soit en utilisant un tableau comportant plusieurs lignes soit en utilisant de structures définies dans la section 3.3 (p. 60). Si au contraire il n'y a pas d'indexation commune, mais que les tableaux ne sont pas ordonnés, il est nécessaire d'envisager toutes les associations possibles et cela se fait avec l'algorithme d'une boucle imbriquée dans une boucle décrit dans la section 2.2.2 (p. 32) : la boucle principale parcourt le premier tableau, à l'intérieur de cette boucle principale une deuxième boucle parcourt le deuxième tableau. S'il y a plus que deux tableaux, il faudrait faire un boucle encore à l'intérieur, mais en pratique il est préférable de créer une fonction ce qui est décrit dans la section 2.4 (p. 37)

L'algorithme parcourt les tableaux en faisant progresser les indices correspondant à des cases du tableau. L'algorithme consiste en une boucle dont la condition de répétition

porte sur ces indices. La tâche à réaliser au sein de la boucle consiste en un traitement qui notamment décide le tableau dont l'indice va progresser en tenant compte des valeurs maximales des indices.

2.3 Utilisation de tableaux à deux dimensions

Ce qu'on a vu jusqu'ici sont en fait des tableaux à une dimension. Les tableaux à 2 dimensions permettent par exemple de définir une matrice. On distingue le nombre de lignes L et le nombre de colonnes C. Il y a deux implémentations possibles pour un tableau en 2 dimensions. Pour illustrer on suppose qu'il s'agit d'un tableau d'entiers.

- La première implémentation d'un tableau à 2 dimensions consiste à définir un tableau à une dimension de taille $L \times C$. La déclaration se fait

  ```
  int tab[L*C];
  ```

  Pour l'accès à la case de ligne i et de colonne j, il y a deux conventions suivant qu'on lise la matrice 2D suivant les colonnes ou suivant les lignes. Avec la première convention, l'accès se fait avec `tab[i+j*L]` (i.e. c'est le choix fait dans beaucoup de fonctions Matlab). Avec la deuxième convention, l'accès se fait avec `tab[j+i*C]`. [7]
  Il est conseillé de réaliser des fonctions `getM` et `setM` pour accéder à ces données, par exemple avec la deuxième convention :

[7] Pour éviter de se tromper remarquez que dans ces deux formules, on ne multiplie pas un compteur de ligne avec un nombre de ligne ou un compteur de colonne avec un nombre de colonne.

```
double getM(const double tab[],int C,int i,int j)
{
  return tab[j+i*C];
}
void setM(double tab[],const int C,const int i,const int j,const double valeur)
{
  tab[j+i*C]=valeur;
}
```

- La deuxième implémentation d'un tableau à 2 dimensions consiste à définir un tableau à une dimension dont les cases sont des pointeurs vers une deuxième tableau constitué des lignes du tableau à deux dimensions. Cette implémentation est décrite dans le polycopié de C en section 2.3.5 (p. 25 et 26).

  – La déclaration du tableau modifiable est

    ```
    int tab[2][3]={{1,2,3},{4,5,6}};
    int taille=2;
    ```

    En fait dans cet exemple `taille` devra toujours être inférieur à 2.

  – La déclaration du tableau non-modifiable est

    ```
    const int tab[2][3]={{1,2,3},{4,5,6}};
    const int taille=2;
    ```

L'utilisation pour lire ou pour écrire se fait avec

```
tab[i][j]
```

– Si la fonction modifie le contenu du tableau, l'argument dans une définition ou une déclaration d'une fonction est

```
int tab[][3]
```

– Si la fonction ne modifie pas le contenu du tableau, l'argument dans une définition ou une déclaration d'une fonction est

```
const int tab[][3]
```

On observe ainsi qu'un tableau à deux dimensions est de fait un pointeur sur pointeur, mais il est particulier au sens où les valeurs prises par `tab[i]` qui sont les adresses des différents tableaux sont des *constantes* et en fait peuvent être déduites à partir de `tab` en utilisant la longueur des lignes, d'où l'importance de repréciser cette longueur dans chaque déclaration ou définition.[8]

Exercice 11 *On considère un tableau à deux dimensions. Calculez la somme des éléments du tableau. Proposez deux solutions, la première utilisant une implémen-*

[8] Ceci n'est plus vrai pour une allocation dynamique.

tation 1D et la deuxième utilisant une implémentation 2D.

2.4 Créations de fonctions pour réaliser des algorithmes plus complexes

Il faut éviter de proposer un algorithme plus complexe que ceux mentionnés dans les sections 2.2.1, 2.2.2 et 2.2.3 (p. 31, p. 32, p. 33). La solution consiste à définir une nouvelle fonction éventuellement appelée dans une des tâches incluse dans un ensemble de boucles avec éventuellement une structure conditionnelle. Cette fonction peut aussi comporter un ensemble de boucles et une structure conditionnelle. Cette fonction doit avoir une définition précise, telle qu'on puisse tester et garantir son fonctionnement avant que le programme global soit terminé. Trouver quelle fonction construire est une réelle difficulté. Il est souvent possible de s'inspirer de fonctions déjà existantes notamment celles qui existent pour les chaînes de caractères et qui sont rappelés brièvement en section 3.1 (p. 49). Plus généralement, l'idée est qu'il est préférable de chercher à constituer des fonctions qui servent à manipuler des données particulières sans nécessiter la connaissance de l'ensemble des données du problème, cette façon de choisir les fonctions amènent à raisonner de manière orientée objet.

Une autre raison de créer une fonction est d'éviter d'avoir des successions de deux ou plus de lignes de code similaires à deux endroits différents du programme.

Une source d'erreur lorsqu'on utilise les fonctions est de se tromper dans l'ordre des

arguments, d'autant qu'il peut y en avoir beaucoup. Voici un ensemble de règle qui permet plus ou moins de fixer un ordre pour les arguments :

1. La taille d'un tableau suit le tableau, si deux tableaux ont la même taille, la taille suit les deux tableaux.

2. Les données qui seront modifiées sont mises au début et les données constantes sont mises en fin de liste d'argument.

3. Les données plus importantes en mémoire sont mises au début.

2.5 Algorithme du tri à bulle

Cet algorithme simple permet d'ordonner un tableau d'entier. L'algorithme consiste à faire une succession de successions d'échanges de termes successifs dans un tableau. Ces échanges n'ont lieu qui si les termes successifs ne sont pas dans le bon ordre. Il s'agit d'une succession de successions car si on ne parcourt qu'une fois le tableau, cela ne suffit pas pour ordonner le tableau.

Cet algorithme est décrit :

```
En entrée~: tableau de n cases
Répéter n fois
  Pour i de 0 à n-1
```

```
  si tableau[i]>tableau[i+1] alors échanger tableau[i] avec tableau[i+1]
Fin pour
Fin répéter
```

Voici une illustration de l'algorithme.

```
tab=(5,4,3,2,1)
tab=(4,5,3,2,1)
tab=(4,3,5,2,1)
tab=(4,3,2,5,1)
tab=(4,3,2,1,5)

tab=(3,4,2,1,5)
tab=(3,2,4,1,5)
tab=(3,2,1,4,5)
tab=(3,2,1,4,5)

tab=(2,3,1,4,5)
tab=(2,1,3,4,5)
tab=(2,1,3,4,5)

tab=(1,2,3,4,5)
tab=(1,2,3,4,5)
```

```
tab=(1,2,3,4,5)
```

2.6 Utilisation de tableaux de chaînes de caractères

Un tableau de chaîne de caractères est un tableau à deux dimensions conforme à la deuxième implémentation de la section 2.3 (p. 34), à ceci prés qu'il n'est pas nécessaire de connaître la taille d'une chaîne de caractère pour la lire, la présence du caractère \0 permet de retrouver cette taille.

- La déclaration du tableau modifiable en valeurs et en nombre de ligne est[9]

  ```
  char tab[][30]={"maison","arbre","voiture"};
  int taille=3;
  ```

  En fait dans cet exemple `taille` devra toujours être inférieur à 3 et la longueur des mots ne devra pas dépasser 29 caractères.

- La déclaration du tableau non-modifiable est

[9]Ici encore on s'interdit d'utiliser l'instruction char * tab[]={"maison","arbre","voiture"} ou char * tab[3]={"maison","arbre","voiture"}

```
const char tab[][30]={"maison","arbre","voiture"};
const int taille=3;
```

L'utilisation pour lire ou pour écrire une chaîne de caractère se fait avec `tab[i]`. Si l'on souhaite lire ou modifier un caractère en particulier, l'accès se fait avec `tab[i][j]`.

- Si la fonction modifie le contenu du tableau, l'argument dans une définition ou une déclaration d'une fonction est

  ```
  char tab[][30]
  ```

- Si la fonction ne modifie pas le contenu du tableau, l'argument dans une définition ou une déclaration d'une fonction est

  ```
  const char tab[][30]
  ```

Exercice 12 *Définissez dans la fonction* `main` *un tableau de mots et affichez le dans une fonction* `affichageMots`.

2.7 Priorités entre opérateurs et de l'usage des parenthèses

En C, une expression est analysée en tenant compte de niveaux de priorités entre les opérateurs. Ainsi `a=b+1;` est analysé comme d'une part `b` est ajouté à `1` puis affecté à `a`. Cependant si `=` avait été prioritaire par rapport à `+`, cette expression aurait été analysée comme d'une part `b` est affecté à `a` et le résultat de cette affectation (en l'occurrence la nouvelle valeur de `a`) aurait été ajouté à `1`.

Voici un ensemble d'expressions où il n'est pas nécessaire d'utiliser les parenthèses :

```
c<=a+1, c+=a+1,
0==a&&1==b  //en fait si a est non-nul,
            //il n'évalue pas
            //la deuxième expression.
*a+1 //cela signifie que la somme de 1 et
     //de la valeur pointée
     //par le pointeur appelé a
tab[2]++ //cela signifie l'incrémentation
         //du troisième élément
         //du tableau
&tab[2] //cela signifie l'adresse
        //du troisième élément du tableau
```

```
        //dans le cadre de ce cours
        //on n'utilise pas tab+2
&tab[2][3]
```

En revanche il faut des parenthèses dans les cas suivants

```
(*a)++ //cela signifie l'incrémentation de la valeur
       //pointée par le pointeur a
       //dans le cadre de ce cours
       //on n'utilise pas a++
       //quand a est un pointeur ou un tableau
(int)(x+0.4) //Dans cette expression,
             //x+0.4 est d'abord évalué
             //et est un {\tt double}
             //puis le résultat est convertit
             //en {\tt int}.
             //Lors d'une conversion dans un type énuméré,
             //par exemple {\tt Boul}, il faut aussi
             //mettre des parenthèses
             //autour de l'expression à convertir.
!(x==1) //en effet !x==1
        //est vrai quand x==0
        //est faux quand x différent de 0
```

2.8 Conseils de style

Le fait d'éviter d'utiliser le signe `-` dans le deuxième argument de `for`, permet de rendre plus visible la façon dont on peut accéder aux éléments du tableau.

```
#include <stdio.h>
int main(void)
{
  const int tab[]={1,3,5}; const int taille=3;
  int i;
  for(i=0;i+1<taille;i++) {
    if (tab[i]<tab[i+1]) printf("1 ");
  }
}
```

2.9 Utilisation de `sizeof`

`sizeof` permet de récupérer la taille en octet. Cette fonction peut prendre en argument un type et dans ce cas elle renvoie la taille mémoire utilisée pour les variables de ce type. Cette fonction peut aussi s'utiliser sur une variable de ce type et elle renvoie aussi la taille de cette variable. On peut l'utiliser aussi pour un tableau, mais **seulement** quand ce tableau a été alloué **statiquement** et **dans cette fonction**. Dans ce cas la fonction

`sizeof` renvoie la place mémoire utilisée par ce tableau, c'est-à-dire le nombre de cases du tableau multiplié par la place occupée par chaque case du tableau.

```
#include <stdio.h>
int main(void)
{
  int tab[] = { 1, 2, 3, 4 };
  printf("le tableau est de taille %d\n", sizeof(tab) / sizeof(tab[0]));
  getchar();
  return 0;
}
```

2.10 Exercices supplémentaires

Exercice 13 *On cherche un nombre dans un tableau trié. On utilise pour cela une recherche dichotomique : à chaque itération on restreint la zone recherchée en testant la case au milieu de la zone recherchée, en comparant la valeur de cette case avec le nombre recherché et en déterminant une nouvelle zone où doit se trouver le nombre recherché.*

Exercice 14 *Appliquez l'algorithme de tri à bulle sur un tableau d'entiers.*

Exercice 15 *On cherche à reproduire le fonctionnement de la preuve par neuf. Pour chaque nombre N, on peut calculer un entier noté $r(N)$ qui est obtenu en écrivant ce nombre sur une base 10 et en ajoutant les composantes de ce nombre sur cette base et é nouveau en écrivant le nombre ainsi obtenu sur une base 10 et en ajoutant encore les composantes jusqu'à ce que ces composantes soient entre 0 et 9. Cette technique permet de détecter s'il y a une erreur dans une multiplication car $r(N_1N_2) = r(r(N_1)r(N_2))$. Utilisez cette technique pour vérifier des multiplications de différents nombres. Une indication sur l'exercice peut être trouvée dans la section 1.3 (p. 11).*

Exercice 16 *On considère deux tableaux triés. Calculez un tableau obtenus en fusionnant les deux tableaux de façons à contenir les valeurs de chacun des tableaux et à ce que le tableau obtenu soit trié.*

Exercice 17 *On considère deux dates, on cherche le nombre de jours séparant les deux dates, (i.e. en incluant que le premier jour de ces deux jours). On suppose que les deux dates sont dans la même année qui n'est pas bissextile, c'est-à-dire que l'on suppose que le mois de février compte 28 jours. On suppose que la deuxième*

date est postérieure à la première date. Pour simplifier on suppose que la date est identifiée par un numéro de mois et de jour. Le programme utilise une table contenant le nombre de jours par mois. Les sections 1.3 et 2.2 comportent des indications sur cet exercice.

Chapter 3

Structures de données, stockage dynamique et gestion des chaînes de caractères

3.1 Algorithmes utilisant les fonctions spéciales pour les chaînes de caractères

Le langage C dispose d'un grand nombre de fonctions dédiées aux chaînes de caractères qui évitent d'avoir à recréer des algorithmes complexes. Ces fonctions sont regroupées

dans `string.h`, bibliothèque à inclure. L'évolution du C a amené à modifier un certain nombre de ces fonctions, leurs nouveaux noms est formé de l'ancien nom suivi de `_s`. Cette évolution vise à permettre au programmeur d'indiquer un nombre maximum de caractères à écrire dans une chaîne de caractères, ceci permettant d'éviter d'écrire en dehors de la zone allouée. Il est possible de ne pas utiliser ces nouvelles fonctions en utilisant `#define _CRT_SECURE_NO_WARNINGS`. Mais dans le cadre de cette formation, nous utiliserons les instructions ces nouvelles fonctions.

```
#include <string.h>
```

- `int strcpy_s(char *dest,int dest_size,const char *src);` : copie du deuxième argument sur le premier argument.
- `int strncpy_s(char *dest,int dest_size,const char *src,int n);` : copie des `n` premiers caractères du deuxième argument sur le premier argument.
- `int strcat_s(char *dest,int dest_size,const char *src);` : concaténation de deux chaînes de caractères.
- `int strcmp(const char *,const char *)` : comparaison entre deux chaînes de caractères.
- `int strlen(const char *)` : longueur d'une chaîne de caractère.

- `char * strchr(char *,int c)` ou `const char * strchr(const char *,int c)` : adresse mémoire de la premiére occurence de `c` dans une chaîne de caractère.
- `char * strstr ( char * str1, const char * str2 )` : renvoie l'adresse de l'endroit dans str1 où se trouve str2.
- `char* strtok_s(char* cs,const char* ct,char** context);` est illustré en annexe B.1.1 (p. 106). L'ancienne version de cette fonction est exposée dans le polycopié de C à la section 5.2.3 (p. 63).
- `double atof (const char* str)` : conversion d'un nombre écrit sous forme d'une chaîne de caractère en un `double` (voir section 5.2.3 du polycopié de C).
- `int atoi (const char* str)` : conversion d'un nombre écrit sous forme d'une chaîne de caractère en un `int`.
- `int sprintf ( char * str, const char * format, ... );` est similaire à `printf`, il permet de faire l'opposé de `atoi` et `atof` en écrivant ces valeurs dans une chaîne de caractère.

Les fonctions `atoi` et `atof` sont dans la librairie `stdlib.h`, les autres sont dans `string.h` Un certain nombre de ces fonctions existent aussi dans des versions qui limitent leur fonctionnement é un nombre donné de caractères, de telles fonctions sont utiles pour éviter d'écrire en dehors des zones allouées pour les chaînes de caractères.

Remarquez que dans ces fonctions les caractères sont le plus souvent transmis en tant que case d'un tableau de `char`, cependant dans le cas où ils sont transmis en tant que caractère, ils ne sont pas transmis avec le type `char` mais avec le type `int` qui est occupe plus de mémoire.

Ces fonctions renvoient une adresse mémoire sur une chaîne de caractère déjà allouée. Il ne faut donc pas allouer de la place mémoire, mais déclarer un pointeur sur caractère et y stocker cette adresse mémoire en sachant que ce pointeur peut être utilisé comme une chaîne de caractère. Ce pointeur peut être déclaré de deux façons suivant qu'il pointe sur une chaîne de caractère variable ou fixe

```
char *mot;
const char * mot=...
```

Dans le deuxième cas, il faut mettre juste après et sur la même ligne la fonction qui va donner l'adresse mémoire qui va être affecté à ce pointeur.
Important : On ne peut pas copier une chaîne de caractère sur une variable avec `=`, ni tester le fait qu'une chaîne de caractère est identique à une autre avec `==`. Dans le premier cas, il convient d'utiliser `strcpy_s` et dans le deuxième cas `strcmp`.

Exercice 18 *On considére un tableau de mots. Comptez le nombre de mots finissant par* tion. *On suppose ici que les mots ne sont pas suivis d'espaces. Une solution consiste à se positionner au début des quatre dernières lettres de chaque mot et de vérifier que ce qui suit cette position est la chaîne de caractère* `tion`. *Cela*

se fait d'une part en utilisant `strlen` *en ajoutant la valeur obtenue à l'adresse du tableau et en retranchant* 4 *à cette valeur et d'autre part avec* `strcmp` *pour tester la chaîne de caractère résultante.*

Voici la fonction `main` *à utiliser.*

```
int main(void){
  char tab[][LG]={"arbre","demonstration","finition","tion_tion"};
  printf("Nombre de mots %d\n",compteMots(tab,4));
  return 0;
}
```

3.2 Allocation dynamique

Dans le cadre de ce cours, l'allocation dynamique n'est utilisée que pour les tableaux ou les conteneurs plus complexes comme les listes chaînées. Elle sert lorsqu'on ne connaît pas un ordre de grandeur du nombre d'éléments contenus dans le tableau. Elle sert aussi au sein d'une fonction qui reçoit un tableau en argument mais qui pour le traitement a besoin de créer d'autres tableaux de la même taille.

Une allocation dynamique doit toujours être suivie d'une désallocation (i.e. un `free`). Entre l'allocation et la désallocation, il ne doit y avoir aucun `return`[1].

Il n'est pas possible d'allouer de la place mémoire à une adresse donnée (i.e. c'est la

[1]Cela entraÏne la non-désallocation de la mémoire et donc ce qu'on appelle une fuite de mémorie

fonction qui alloue qui détermine l'adresse). En fait dans le cadre de ce cours, il n'y a pas besoin de stocker des adresses et de déclarer des pointeurs sans y mettre immédiatement une adresse mémoire dans le cadre de l'allocation dynamique, sauf dans la section 3.1 (p. 49) et au sein des fonctions `comparer` de la section 4.3 (p. 73).

Sauf pour faire de la réallocation de mémoire (voir section 3.2.1 p. 54), on ne doit jamais écrire sur une adresse mémoire sur un pointeur auquel on vient déjà d'allouer de la place mémoire.

3.2.1 Allocation dynamique d'un tableau 1D

L'allocation dynamique est présentée dans le polycopié de C au chapitre 4 (p. 45-51). Dans le cadre de ce cours, l'allocation dynamique d'une variable, d'un tableau, d'une chaîne de caractère ou d'une structure qui se fait dans une fonction devra être désallouée (i.e. utilisation de `free`) dans la même fonction. Dans le cas d'une liste chaînée ou d'un arbre (voir section 5.1), c'est l'ensemble de la liste chaînée ou de l'arbre qui sera désallouée dans la fonction qui a créé cet objet complexe.

Pour un tableau de type `int` alloué dynamiquement, si la taille est fixe, l'allocation dynamique se fait ainsi :

```
int taille=5;
int * const tab=(int *)malloc(taille*sizeof(int));
```

Si la taille est variable

```
int taille; taille=5;
int * tab=(int *)malloc(taille*sizeof(int));
```

La réallocation[2] se fait alors ainsi

```
taille=6;
int * tab=(int *)realloc(taille*sizeof(int));
```

Il est nécessaire à chaque fois qu'il y a une allocation dynamique de vérifier si cette allocation a pu se faire en testant que le pointeur créé ne pointe pas sur `NULL`. Par exemple avec l'instruction suivante

```
if (NULL==tab) {
  printf("echec malloc\n");
  getchar();
  exit(EXIT_FAILURE);
}
```

La désallocation de la mémoire se fait avec `free`. La fonction `calloc` s'utilise de la même façon que `malloc` et offre en plus l'avantage de mettre à zéro toutes les cases mémoires allouées.

Remarquez que dans tous ces exemples et contrairement à la norme définissant le langage C, il y a une conversion de type à la sortie des fonctions de mémoire[3]. En effet le

[2]Le contenu du tableau est préservé même si le tableau se trouve à un autre endroit de la mémoire.

[3]i.e. à gauche de `malloc` ou de `realloc`, il y a en effet l'expression `(int *)` qui signifie conversion de ce qui était un pointeur sur rien en un pointeur sur entier.

compilateur généralement utilisé est en fait celui de C++ qui respecte pratiquement tous les éléments de la norme de C, mais refuse une conversion implicite du type `void *` en un autre type de pointeur.

Exercice 19 *Calculez le produit scalaire de deux vecteurs de même taille. La taille des vecteurs doit être définie dans le main, il est donc nécessaire d'utiliser une allocation dynamique.*

Le produit scalaire de deux vecteurs de composantes a_i et b_i vaut $\Sigma_i\, a_i b_i$.

3.2.2 Allocation dynamique d'un tableau 2D

Si le tableau 2D est en fait implémenté sous la forme d'un tableau 1D, il suffit d'allouer et déreserver conformément é la section 3.2.1 (p. 54).

Si le tableau 2D est un tableau de tableaux 1D alors il faut allouer le tableau de pointeur et ensuite parcourir toutes les cases du tableau et y affecter l'adresse obtenue en allouant un tableau 1D. Ainsi pour allouer un tableau d'entiers de taille $L \times C$, les instructions sont

```
int ** tab = (int **) malloc(L*sizeof(int*));
if (NULL==tab) {
printf("echec malloc\n"); getchar(); exit(EXIT_FAILURE);
}
```

```
for(int i=0; i<L; i++) {
  tab[i]=(int *)malloc(C*sizeof(int));
if (NULL==tab[i]) {
  printf("echec malloc\n"); getchar(); exit(EXIT_FAILURE);
}
}
```

Dans la pratique ces lignes devraient être aussi complétées d'une vérification de ce que les adresses allouées ne sont pas nulles.

La déréservation se fait aussi en parcourant chaque case du tableau d'adresses et en désallouant chaque tableau associé puis en désallouant le tableau d'adresses.

```
for(int i=0; i<L; i++)
  free(tab[i]);
free(tab);
```

On transmet l'ensemble du tableau dans une fonction en mettant `tab` dans l'argument, que ce tableau soit modifié ou non par la fonction. Si cette fonction ne modifie pas le tableau alors la syntaxe est `const int * const * tab`. Si cette fonction modifie le tableau alors la syntaxe est `int ** tab`. On ne peut pas utiliser le même prototype pour la fonction lorsqu'il s'agit d'un tableau 2D alloué de façon statique et d'un tableau alloué de façon dynamique.[4]

[4]Si l'on veut rentrer plus dans les détails, on peut distinguer plusieurs types de protection, `const int * const *`

Exercice 20 *Utilisez une allocation dynamique 2D pour allouer la matrice et le vecteur suivant. La fonction réalisée doit dépendre du nombre de ligne et colonnes. Elle doit fonctionner pour une matrice rectangle. Faites le produit matriciel du premier par le deuxième*

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \tag{3.1}$$

3.2.3 Allocation dynamique d'un tableau de chaînes de caractère

On souhaite ici allouer dynamiquement un tableau contenant L chaînes de caractère, chacune de longueur C. La seule différence avec la section 3.2.2 (p. 56), est que le type utilisé est `char` et qu'il faut prévoir une case en plus pour y mettre le caractère de fin de chaîne `\0`. L'allocation se fait donc de cette façon :

`tab` qui garantit que ni les valeurs des tableaux 2D ne seront pas modifiés ni les emplacements des lignes du tableau, `const int * const * const tab` qui garantit qu'en plus l'emplacement du tableau ne sera pas modifié, `int * const * const tab` qui garantit que l'emplacement des lignes et du tableau ne seront pas modifiés et enfin `int ** const tab` qui garantit que l'emplacement du tableau ne sera pas modifié. L'utilisation du prototype `const int ** tab` provoque une erreur car il est possible de modifier les valeurs d'un tableau en agissant sur les lignes. L'utilisation du `const` à droite n'a pas d'impact sur le résultat final dans la mesure où le fait que la fonction déplace l'emplacement du tableau dans la mémoire n'aura pas d'impact hors de la fonction puisque de toute façon la fonction n'a accès qu'à une copie de l'emplacement du tableau. Cette remarque est aussi valable pour les tableaux de chaînes de caractères

```
char ** tab = (char **) malloc(L*sizeof(char *));
if (NULL==tab) {
  printf("echec malloc");
  getchar();
  exit(EXIT_FAILURE);
}
for(int i=0; i<L; i++) {
  tab[i]=(char *)malloc((1+C)*sizeof(char));
if (NULL==tab[i]) {
    printf("echec malloc");
    getchar();
    exit(EXIT_FAILURE);
  }
}
```

La désallocation se fait ainsi :

```
for(int i=0; i<L; i++)
  free(tab[i]);
free(tab);
```

On transmet l'ensemble du tableau dans une fonction en mettant `tab` dans l'argument, que ce tableau soit modifié ou non par la fonction. Si cette fonction ne modifie pas le tableau alors la syntaxe est `const char * const * tab`. Si cette fonction ne modifie

pas le tableau alors la syntaxe est `char ** tab`. On ne peut pas utiliser le même prototype pour la fonction lorsqu'il s'agit d'un tableau 2D alloué de façon statique et d'un tableau alloué de façon dynamique.

3.3 Structures

Les structures sont définies dans le polycopié de C dans la section 3 (p. 33 é 42). On distingue le fait de définir une structure (ce qui se fait dans ce cours avec `typedef`, `struct`, les champs entourés d'accolades et le nom du nouveau type associé é la structure), le fait d'allouer de la place mémoire pour une structure, de lire ou modifier les valeurs contenues dans une structure, le fait de mentionner cette structure dans un argument lors de la définition ou de la déclaration d'une fonction. Dans le cadre de ce cours, on n'utilisera pas de structure comme valeur retournée d'une fonction sauf dans le cadre d'un algorithme récursif (voir dernière séance) et en s'interdisant dans ce cas le fait que la structure contienne un pointeur.

- Si la structure est simplement lue, elle figure en argument sous la forme de `const` suivi du nom du type de la structure et suivi du nom de la structure.
- Si cette structure est modifiée dans la fonction, alors il faut utiliser un passage par adresse, l'argument est constitué du nom du type de la structure suivi de `*` suivi du nom du pointeur vers la structure

Différents types de structures

On distingue trois types de structures :

- La structure transmise ne contient que des objets de types définis en C

  Lorsque les champs de cette structure sont de types `char, int, double,...` la lecture dans la fonction se fait par passage par copie de la structure. L'écriture se fait par passage par adresse de la structure.

- La structure transmise contient un tableau alloué statiquement

  Lorsque un des champs de cette structure contient un tableau alloué statiquement, alors la lecture dans la fonction se fait par passage par copie de la structure, la lecture et l'écriture du tableau se font dans la fonction en utilisant l'adresse lue sur la structure. On peut remarquer que la fonction `sizeof` appliquée au tableau donne alors la même taille dans la fonction et à l'extérieur. Cette fonction appliquée à la structure donne aussi la même taille dans la fonction et à l'extérieur et cette taille tient compte de la taille du tableau.

- La structure transmise contient un tableau alloué dynamiquement

  Lorsque cette structure contient un tableau alloué dynamiquement, on a deux situations :

– Dans la fonction on ne fait que de la lecture ou de l'écriture, alors il suffit de passer par copie la structure et d'utiliser l'adresse transmise pour lire et écrire sur le tableau.

  L'instruction `sizeof` appliquée au tableau dans la fonction ne donne plus la taille du tableau. Cette instruction appliquée à la structure donne la même taille qu'à l'extérieur mais cette taille ne tient pas compte de la taille du tableau.

– Dans la fonction on réalise aussi l'allocation du tableau, alors il faut transmettre l'adresse de la structure pour changer la valeur du pointeur associé au tableau lors de l'allocation.

  L'instruction `sizeof` appliquée au tableau dans la fonction ne donne plus la taille du tableau. Cette instruction appliquée à la structure ne donne plus la taille de la structure.

Exercice 21 *Définissez un tableau contenant deux triplets de double.*

```
Triplet tab[]={{1,2,3},{2,3,4}};
```

Les triplets sont une nouvelle structure que vous définissez qui contient trois doubles. Ensuite vous utilisez sans aucune modification la fonction permutation définie lors de l'exercice 8 pour permuter chacun des triplets du tableau et vous affichez le résultat.

Exercice 22 *On considère un tableau 1D de taille 100 et on cherche comme dans l'exercice* **??** *à savoir si tous les éléments sont rangés en ordre croissant. Pour cela créez une structure appelée* `Tab_Cpl` *contenant un tableau alloué statiquement de taille 100, la taille (qui doit être inférieur à 100), un indicateur de position sur ce tableau et un objet de type* `enum` *indiquant si le tableau est rangé par ordre croissant, si on ne le sait pas ou s'il n'est pas rangé (*`ON_NE_SAIT_PAS, OUI,NON`*). Créez une fonction* `test_est_range` *qui utilise un objet ainsi définie et qui vérifie, si effectivement tous les éléments sont rangés dans le sens croissant, seulement dans le cas où le type enum vaut* `ON_NE_SAIT_PAS`*.*

Exercice 23 *On utilise la structure suivante :*

```
typedef struct Tab {
  double * tab;
  int taille;
} Tab;
```

Créez les fonctions `allouer,desallouer,initialiser,afficher` *pour gérer ce type de tableau. La fonction* `initialiser` *affecte une même valeur à toutes les cases du tableau, cette valeur est transmise en argument à la fonction. La fonc-*

tion `allouer` *alloue de la place mémoire en fonction d'un argument* `taille`*. La définition de la fonction* `allouer` *est :*

```
void allouer(Tab * tab,int taille);
```

Exercice 24 *Il s'agit de simuler la gestion de comptes. On se donne un certain nombre de mouvements (dépôt ou retrait, numéro de compte et somme) et on en déduit le solde de différents comptes, certains ayant été créé lorsqu'ils n'existent pas. La fonction* `main` *créé un tableau de comptes vide mais avec suffisamment d'emplacements mémoire, elle créé un certain nombres de mouvements qui sont ensuite exécutés et finalement elle affiche le solde de tous les comptes créés. Les structures à utiliser sont définies de la façon suivante :*

```
typedef struct Compte {
  int compte;
  int solde;
} Compte;
typedef enum Type {
  DEPOT,
  RETRAIT,
} Type;
typedef struct Mouvement {
```

```
  Type t;
  int compte;
  int somme;
} Mouvement;
```

Exercice 25 *On considère un tableau de mots alloué dynamiquement. Adaptez le programme réalisé lors de l'exercice 18 (p. 52) et comptez le nombre de mots finissant par* tion.

3.4 Exercices supplémentaires

Exercice 26 *On considère une chaîne de caractère et on cherche à permuter l'ordre des lettres : la première lettre devient la deuxième, la deuxième lettre devient la troisième, ... , et la dernière lettre devient la premiere. Ainsi le mot* `arbre` *devient* `rbrea`*. En utilisant les fonctions sur les chaînes de caractères* `strlen`, `strcpy`, `strncpy`, *il est possible d'éviter l'utilisation d'une boucle* `for`*, d'autant que* `strncpy` *permet aussi de copier un certain nombre de lettres sur une chaîne de caractère déjà existante.*

Exercice 27 *Il s'agit ici de compléter l'exercice 24 (p. 64). L'ensemble des mouvements est donné par une ligne de commande constituée d'un juxtaposition de motif. Le motif est formé d'une lettre 'D' (pour dépôt) ou 'R' (pour retrait), d'un numéro de compte, d'une virgule, d'une somme et d'une virgule. Chaque motif codifie un mouvement. Dans cet exercice, les mouvements sont lues sur la ligne de commande et ensuite exécutée conformément à l'exercice 24 et affiche les différents comptes et leur solde.* `Type` *est ici un peu modifié.*

```
typedef enum Type {
  DEPOT='D',
  RETRAIT='R',
} Type;
```

Dans cet exercice, il est utile d'utiliser les fonctions définies dans la section 3.1 (p. 49).

Exercice 28 *On considère deux ensembles d'entiers A, B ne comptenant aucun doublon. Trouvez $A \backslash B$, c'est-à-dire un ensemble formé des éléments de A n'appartenant pas à B. Ici il ne s'agit pas de faire un tri des éléments au préalable, mais de parcourir les éléments de A et supprimer tous ceux qui seraient dans B.*

Chapter 4

Entrées, sortie, fichiers, fonction main, type générique et utilisation de qsort

4.1 Utilisation de l'entrée clavier

L'insertion d'informations au programme peut aussi se faire en transmettant des paramêtres au moment de l'appel à exécution du programme avec `scanf_s`, voir la section

5.2.1 du polycopié de C pour la définition de `scanf`. `scanf_s` diffère de `scanf` seulement par le fait que quand on lit une chaîne de caractère, il faut indiquer juste après cette chaîne de caractère le nombre maximal d'octet que l'on peut y mettre. Dans le cadre de ce cours, nous n'utiliserons pas le format `"%c"`.

Exercice 29 *Écrivez un programme qui saisit des notes entrées itérativement au clavier. Ce programme se termine quand l'utilisateur entre -1 et dans ce cas le programme affiche la moyenne des notes. Le programme ne retient pas l'ensemble des notes entrées au fur et à mesure, il retient leur somme et leur nombre. La moyenne est déterminée à la fin avec ces deux quantités.*

4.2 Utilisation des fichiers

L'utilisation des fichiers textes est décrite dans le polycopié de C en section 5.3 et conformément à ce polycopié, la lecture et l'écriture se fera dans ce cours avec les fonctions `fgets` et `fputs`.

```
char * fgets( char * str, int maxLength, FILE * stream );
int fputs( const char * string, FILE * stream );
```

Lorsqu'on ne met pas de répertoire dans le nom du fichier, ce fichier est créé ou est lu dans le même répertoire que le programme écrit en C lorsque le programme est exécuté à

travers l'IDE MSDN. Ce fichier est créé ou est lu dans le même répertoire que l'exécutable, lorsque c'est l'exécutable qui est exécuté.

Exercice 30 *Rédigez un programme qui écrit sur un fichier texte une liste de mots.*

4.2.1 Données entrées en ligne de commande

Dans cette partie, on exécute le programme depuis un terminal en se plaçant dans le répertoire où se trouve l'exécutable. L'annexe C.3.2 explique comment exécuter en ligne de commande un programme écrit avec l'interface de Microsoft Visual C. Pour se faire, on remplace `int main(void)` par `int main(int argc,char *argv[])`.

- `argc` est un plus le nombre de paramètre placé à droite du programme.
- `argv` est une table de chaîne de caractère. `argv` est censée être lue et non écrit. `argv[1]` est l'adresse de la chaîne de caractère associée à la liste de caractère placé à droite du nom du programme ; `argv[2]` est l'adresse de la chaîne de caractère associée à la liste de caractère placé à droite du nom du programme, etc...

Même quand les arguments placés à droite du nom du programme exécutable sont des nombres, ceux-ci sont stockés dans `argv` comme des chaînes de caractères.

Exercice 31 *Le programme à réaliser permet d'ajouter des nombres et d'afficher leurs sommes totales au fur et à mesure qu'on exécute le programme.*

- *L'exécution du programme en ligne de commande sans argument permet en apparence d'initialiser la somme en zéro. En réalité le programme créé un fichier texte et inscrit dedans la valeur nulle.*

- *L'exécution du programme en ligne de commande suivi d'un nombre permet en apparence d'ajouter ce nombre à la somme, le résultat de cet addition est affiché. En réalité le programme convertit ce nombre en un double qui est ajouté au double lu dans le fichier texte. La somme de ces deux doubles est affiché à l'écran et enregistré dans le ficher texte à la place de la précédente valeur.*

Voici un exemple d'exécution dans un fenêtre dos

```
c:\SIMU\EXERCICE\DEBUG>exercice

c:\SIMU\EXERCICE\DEBUG>exercice 3.5
3.500000
c:\SIMU\EXERCICE\DEBUG>exercice 6.7
10.200000
```

4.2.2 Polymorphisme en C

En informatique, Le polymorphisme signifie le fait de gérer de types différents avec des fonctions de même nom. La façon exacte dont la gestion est réalisée dépend du type et est détaillée dans le programme. Mais l'ensemble des traitements associés à chaque type est encapsulé en une fonction. Comme en général les différents types ne prennent pas la même quantité de mémoire, l'ensemble du traitement est gérer par des adresses. En C, ceci est possible en convertissant de façon explicite un `void *` en une adresse d'un certain type.

Exercice 32 *L'objet de ce programme est d'afficher la liste des participants à une réunion. Cette réunion est composée de différents types de personne : il y les membres de la famille (le père, la mère, le fils), il y les amis que l'on connaît par leur prénom et il y a les inconnus qui sont numérotés. Complétez le programme suivant.*

```
typedef enum   TYPE{Famille, Ami,Inconnu,}   TYPE;
typedef struct PERSONNE{void * personne; TYPE type;} PERSONNE;
typedef enum   FAMILLE{Pere,  Mere,  Fils,} FAMILLE;
typedef char    AMI [30];
```

```
typedef int     INCONNU;
void affiche(const PERSONNE tab[],int taille);
int main(void) {
  AMI ami1="Pierre",ami2="Julie";
  INCONNU inconnu1=1;
  FAMILLE famille1=Pere,famille2=Fils;
  PERSONNE reunion[]={{&ami1,Ami},{&ami2,Ami},{&inconnu1,Inconnu},{&fam
  affiche(reunion,sizeof(reunion)/sizeof(reunion[0]));
  getchar();
  return 0;
}
```

Voici un exemple d'affichage du programme

```
La reunion est compose de
Pierre est un ami
Julie est un ami
c'est l'inconnu numero 1
pere de la famille
fils de la famille
```

4.3 Utilisation de la fonction `qsort`

La fonction **`qsort`** de la libraire **`stdlib.h`** permet de trier des tableaux.
Le fait de trier un tableau permet de simplifier certains traitements de données :

- union, intersection : voir l'exercice 16 (p. 46).
- calcul d'histogramme : voir l'exercice 37 (p. 80).
- tirage aléatoire d'un ordonnancement : voir l'exercice 38 (p. 80).

La définition de cette fonction **`qsort`** est [1]

```
qsort(tableau,nombreDeCases,sizeof tableau[0],comparaison);
```

Elle repose sur une fonction, ici appelée **`comparaison`** qui effectue une comparaison entre deux éléments du tableau. La déclaration de cette fonction de comparaison garantit à travers l'utilisation du qualificatif **`const`** les valeurs du tableaux ne seront pas modifiées. Cette déclaration ne dépend pas du type des éléments du tableau[2]. La comparaison entre les éléments est le fait d'une fonction à transmettre à la fonction **`qsort`**, pour définir cette fontion il est nécessaire d'utiliser le type des données, cela se fait donc avec une conversion explicite[3]. On remarque que le type utilisé pour faire la conversion explicite

[1] Ceci reste vrai même pour une allocation dynamique.

[2] C'est ce qu'on appelle la programmation générique, la fonction **`qsort`** déplace des éléments dont elle ne connaît pas le type

[3] c'est-à-dire dans les exemples successifs avec `(const int *)`, `(const char * const *)`, `(const int (*)[2] )`.

est le même que le type de la variable dans laquelle on copie la donnée. Les données transmises aux fonctions de comparaison sont des pointeurs sur les éléments du tableaux, une fois leur conversion en un pointeur sur un type, le type doit avoir la même taille que la taille annoncée dans le troisième argument de la fonction **qsort**. Par ailleurs la fonction **qsort** ne fait que déplacer les données sans les modifier, elle impose que les fonctions de comparaison ne modifie pas les données, c'est pour cela que les données sont des pointeur sur rien constants. Et même après leur conversion explicite, ils doivent rester constants.

- Si le tableau est composé d'entiers

```
int comparer(const void * pa,const void * pb)
{
  const int * paInt=(const int *)pa;
  const int * pbInt=(const int *)pb;
  return *paInt-*pbInt;
}
```

- Si le tableau est à deux dimensions composé de 2 colonnes alloué statiquement et que la comparaison se fait sur la première colonne

```
int compValTab(const void * ptr1,const void * ptr2)
{
  const int (*tab1)[2]=(const int (*)[2] )ptr1;
```

```
  const int (*tab2)[2]=(const int (*)[2] )ptr2;
  return (*tab1)[1])-(*tab2)[1];
}
```

- Si le tableau est composé de chaînes de caractères alloué statiquement comme un tableau à deux dimensions avec `NB_C` déterminé par `#define`.

```
int compValTab(const void * ptr1,const void * ptr2)
{
  const char (*tab1)[NB_C]=(const char (*)[NB_C])ptr1;
  const char (*tab2)[NB_C]=(const char (*)[NB_C] )ptr2;
  return strcmp(* tab1, * tab2);
}
```

- Si le tableau est un tableau à deux dimensions alloué dynamiquement comme un tableau à deux dimensions et que la comparaison se fait sur la première colonne

```
int comparer(const void * a,const void * b)
{
  const int * const * tabA = (const int * const *) a;
  const int * const * tabB = (const int * const *) b;
  return (*tabA)[0]-(*tabB)[0];
}
```

- Si le tableau est composé de chaînes de caractères alloués dynamiquement

```
int comparer(const void * pa,const void * pb) {
  const char * const *paChar=(const char * const *)pa;
  const char * const *pbChar=(const char * const *)pb;
  return strcmp(*paChar,*pbChar);
}
```

Attention la fonction `comparer` doit retourner un entier même si la comparaison porte sur des doubles et dans ce cas il faut faire des tests sur la valeurs de la différence et retourner un entier.

Exercice 33 *Utilisez la fonction* `qsort` *pour trier un tableau d'entiers.*

4.4 Utilisation des nombres aléatoires

Pour générer des nombres aléatoires, il faut enclencher le générateur aléatoire avec un événement non-prévisible. Cela se fait avec les bibliothèques `time.h` et `stdlib.h`, et l'instruction suivante :

```
srand((unsigned int)time(NULL));
```

Cette instruction doit se trouver a priori au début du `main`, elle ne doit pas se trouver avant chaque tirage aléatoire.

On génère un `int` entre 0 et 999 de la façon suivante

```
printf("%d\n",rand()%1000);
```

Normalement, on peut générer une loi uniforme entre 0 et 1 avec l'instruction suivante [4]

```
(rand()+0.0)/RAND_MAX
```

4.5 Exercices supplémentaires

Exercice 34 *écrivez un programme qui permet de jouer au jeu du pendu. Le joueur 1 entre le mot, l'ordinateur l'enregistre et affiche une série de lignes blanches pour que le joueur 2 ne puisse pas lire le mot, il affiche ensuite une série d'étoiles qui correspond au nombre de lettres du mot à trouver. Le joueur 2 propose des caractères jusqu'à ce qu'il ait trouvé le mot, ou qu'il ait perdu (nombre de coups > 10). à chaque fois que le joueur 2 propose un caractère l'ordinateur affiche le mot avec des * et les caractères déjé trouvés*

Entrez un mot : secret

[4] Expérimentalement, j'ai pu constaté lors d'une expérimentation que le premier tirage aléatoire qui suit `srand((unsigned int)time(NULL));` est très proche de 0.5 mais que les valeurs suivantes semblent réellement suivre une distribution aléatoire.

```
******
caractere ? a
******
caractere ? e
*e**e*
caractere ? t
*e**et
caractere ? c
*ec*et
caractere ? s
sec*et
caractere ? r
secret
VOUS AVEZ GAGNE
```

Une solution possible est d'utiliser une chaîne de caractères temporaire contenant le mot où certains caractères sont remplacés par des étoiles. Le calcul de son contenu est réalisé par deux fonctions à utiliser successivement : `initialiser_avec_etoiles` *et* `remplacer_par_lettres`.

Exercice 35 *Complétez l'exercice 30 de façon à relire le fichier texte. Utilisez la fonction* `qsort` *ou un tri à bulle de façon à ordonner les mots de ce fichier et ensuite réenregistrez les mots une fois ordonnée.*

Exercice 36 *On considére un tableau et on cherche à réordonner les éléments. Placez les éléments pairs en début de tableau dans l'ordre croissant et les éléments impairs en fin de tableau en ordre inversé. Il s'agit ici d'utiliser la fonction* `qsort`*. La parité est testée sur la valeur des éléments en utilisant la notion de modulo*

```
0==n%2 //n est paire
1==n%2 //n est impaire
```

Voici ce que fait la fonction comparaison entre x et y :

- *si x et y sont paire, alors elle retourne $x - y$*
- *si x et y sont impaires, alors elle retourne $y - x$*
- *si x et paire et y est impaire, alors elle retourne -1*
- *si x est impaire et y est paire, alors elle retourne 1*

Exercice 37 *On considère un tableau composé d'un certain nombre d'entiers. Donnez le nombre de valeurs distinctes et le nombre d'occurrences de chaque valeurs distinctes. C'est ce qu'on appelle un histogramme. L'idée consiste à d'abord trier le tableau.*

Exercice 38 *On utilise ici le tri d'un tableau pour faire un tirage aléatoire d'un ordonnancement. L'objectif ici est que pour chaque mot entré par l'utilisateur, on affiche ce mot dans un ordre quelconque et ce jusqu'à ce qu'il entre le mot* `fin`*. Pour cela on créé un tableau ayant le même nombre de colonnes que le mot et ayant deux lignes. Sur la première ligne on tire des chiffre au hasard et sur la deuxième ligne on met les nombre de* 0 *à la longueur du mot moins* 1*. Ensuite on trie les colonnes du tableau de façon à ce que la première ligne soit constitué de nombre croissant. Enfin on affiche successivement les lettres du mots désignés par les nombre de la deuxième ligne du tableau.*

Chapter 5

Algorithmes plus élaborés

5.1 Conteneur génériques

Les piles sont un outil utiles pour implémenter un certain nombre d'algorithmes. Elles permettent de stocker un nombre variable d'informations en cours de traitement et de les retrouver par la suite quand l'ordinateur est à nouveau disponible pour les traiter. Le fonctionnement ainsi que leur implémentation sous la forme d'une liste chaînée est décrit dans le document `listes.pdf`. Les piles peuvent aussi s'implémenter à l'aide d'un tableau, comme le montre l'exercice suivant :

Exercice 39 *Créer une pile en utilisant un tableau de taille fixe égale à* 50 *dont les cases correspondent à des doubles. Les fonctions qui utilisent cette pile sont* `creerPile`, `empiler`, `dépiler` *et* `pileEstVide` *qui retourne un Bouléen* `TRUE` *si*

c'est vrai. La fonction `main` *à utiliser est :*

```
int main(void)
{
  double tab[TAILLE_TAB];
  int taille;
  creerPile(&taille);              affichePile(tab, taille);
  empiler(tab, &taille, 3); affichePile(tab, taille);
  empiler(tab, &taille, 4); affichePile(tab, taille);
  while (FALSE == pileEstVide(taille)) {
    printf("valeur retiree de la pile %1.0lf ", depiler(tab, &taille));
    affichePile(tab, taille);
  }
  affichePile(tab, taille);
  return 0;
}
```

Le résultat d'exécution est :

```
pile=()
pile=(3)
pile=(3,4)
valeur retiree de la pile 4 pile=(3)
```

```
valeur retiree de la pile 3 pile=()
pile=()
```

Exercice 40 *Utilisez la pile créée à l'exercice 39 pour implémenter une calculatrice utilisation la notation en polonaise inversée (notation postfixée). Cette calculatrice accepte des opérateurs unaires* `sqrt` *et des opérateurs binaires* $+, -, *, /$ *et de des doubles. La notation en polonaise inversée permet d'écrire des calculs sans utiliser des parenthèses, les opérateurs suivent les données. Ainsi l'expression*

$$\frac{4 + 2\sqrt{16}}{6}$$

est entrée dans la calculatrice avec une succession de champs suivant

```
{"16","sqrt","2","*","4","+","6","/"}
```

ou bien par les champs suivants

```
{"4","2","16","sqrt","*","+","6","/"}
```

Utilisez la pile définie dans l'exercice 39 pour implémenter cette calculatrice.

La fonction `main` *à utiliser est la suivante :*

```
int main(void)
```

```
{
  const char commande1 [][LG]={"16","sqrt","2","*","4","+","6","/"};
  const char commande2 [][LG]={"4","2","16","sqrt","*","+","6","/"};
  const int taille=8;
  double resultat;
  if (TRUE==executer(&resultat,commande1,taille)) printf("Le resultat e
  else printf("echec\n");
  if (TRUE==executer(&resultat,commande2,taille)) printf("Le resultat e
  else printf("echec\n");
}
```

Avec une liste chaînée, on peut aussi définir un arbre, il suffit pour cela de considérer que chaque élément de la liste peut pointer vers deux ou un plus grand nombre d'éléments. Lorsque chaque élément pointe vers deux éléments, on parle d'arbre binaire, c'est-à-dire d'un graphe où chaque noeud a au plus deux fils.

En fait un arbre et plus généralement un graphe peut aussi être implémenté sous la forme d'une matrice où l'existence d'un lien entre un noeud `i` et un noeud `j` est matérialisée par une valeur par une valeur particulière de la composante (i,j) de cette matrice. On peut aussi affecter un poids à chacune de ces relations entre les noeuds en affectant à chaque composante (i,j) le coût de ce lien. S'il n'y a pas de lien entre le noeud `i` et le noeud `j`, on affecte une valeur infinie (ou très grande). Le graphe est dit orienté si cette matrice n'est pas symétrique (i.e. le coût d'aller de `i` à `j` peut être différent du coût d'aller de `j` à `i`.

5.2 Algorithme récursif

Les algorithmes récursifs sont décrits dans le polycopié de C dans la section 2.4.4 (p. 29 et 30). On peut aussi considérer des implémentations plus complexe que l'exemple cité de la factorielle où la fonction s'appelle elle-même à deux reprises. La difficulté alors est de bien vérifier que les modifications sont prises en compte lors du deuxième appel, (dans le cas où ceci serait nécessaire).

Classiquement, l'implémentation d'un algorithme consiste à réaliser un calcul à partir de la valeur retour d'une fonction donnée. Ainsi la factorielle s'implémente ainsi

```
#include <stdio.h>
int factorielle(int n)
{
  if (0==n) return 1;
  return n*factorielle(n-1);
}
int main(void)
{
  int n=4;
  printf("factorielle(%d)=%d\n",n,factorielle(n));
  return 0;
}
```

Mais, quand il n'y a qu'UN appel par appel, on peut aussi le faire avec un pointeur [1]

```
#include <stdio.h>
void factorielle(int * res,int n)
{
  if (0==n)
    *res=1;
  else {
    factorielle(res,n-1);
    *res=n*(*res);
  }
}
int main(void)
{
  int n=4;
  int res;
  factorielle(&res,n);
  printf("factorielle(%d)=%d\n",n,res);
  return 0;
}
```

[1] Attention cependant au fait qu'ici il n'y a pas copie des grandeurs modifiées mais évolution d'une grandeur stockée en mémoire.

Exercice 41 *Utilisez un algorithme récursif pour implémenter l'exercice 40 (p. 83). Vous pourrez par exemple construire une fonction dont la déclaration est :*

```
double executer(Boul * estOk,const char commande [][LG],int * taille);
```

Cette fonction lit de droite à gauche la ligne `commande` *en faisant décrémenter* `*taille` *et en s'appelant récursivement une ou deux fois quand l'expression lue est celle d'un opérateur unaire ou binaire. La fonction* `main` *à utiliser est :*

```
typedef enum Boul {
  FALSE=0,
  TRUE=1,
} Boul;
typedef struct {
  double val; //valeur du résultat
  Boul   _ok; //est-ce que le résultat est valide
} Resultat;
int main(void)
{
  const char commande1 [][LG]={"16","sqrt","2","*","4","+","6","/"}; in
  const char commande2 [][LG]={"4","2","16","sqrt","*","+","6","/"}; in
```

```
  Resultat resultat;
  afficher(commande1,taille1);
  resultat=executer(commande1,&taille1);
  if (TRUE==resultat._ok) printf("Le resultat est %lf\n",resultat);
  else printf("echec\n");
  afficher(commande2,taille2);
  resultat=executer(commande2,&taille2);
  if (TRUE==resultat._ok) printf("Le resultat est %lf\n",resultat);
  else printf("echec\n");
}
```

Le résultat visible doit être le suivant :

```
(16,sqrt,2,*,4,+,6,/)
sqrt(16)=4
4*2=8
4+8=12
12/6=2
Le resultat est 2
(4,2,16,sqrt,*,+,6,/)
sqrt(16)=4
2*4=8
8+4=12
```

```
12/6=2
Le resultat est 2
```

La notation de polonaise inversée est équivalente à la récursivité quand on lit la chaîne par la fin et que l'on applique les opérandes dans un ordre inversé.

Dans cet exemple, il y a trois informations variables, la valeur du résultat, le fait de savoir s'il y a eu une erreur[2] et un curseur indiquant l'élément à lire dans la chaîne. Les deux premières informations ont la structure d'un arbre binaire et sont dans l'implémentation transmis en utilisant la récursivité. La troisième information est modifiée après chaque lecture et donc n'a pas besoin d'être transmis par récursivité. La transmission de deux informations en sortie d'une fonction se fait en définissant une structure en l'occurrence `Resultat` qui est le type de la sortie de la fonction `executer`.

Le fonctionnement de la récursivité s'implémente de la façon suivante

```
Resultat executer(char commande[][LG],int * taille) {
  //lecture du dernier élément de commande
  //modification de *taille
  //en fonction de la nature de cet élément appel à divers traitement
  //Chaque traitement a la syntaxe suivante
     return fct_traitement1(executer(commande,taille));
  // ou la syntaxe suivante
     return fct_traitement2(executer(commande,taille),executer(commande,
```

[2] C++ permet la gestion des exceptions `try` & `catch` qui aurait été une alternative.

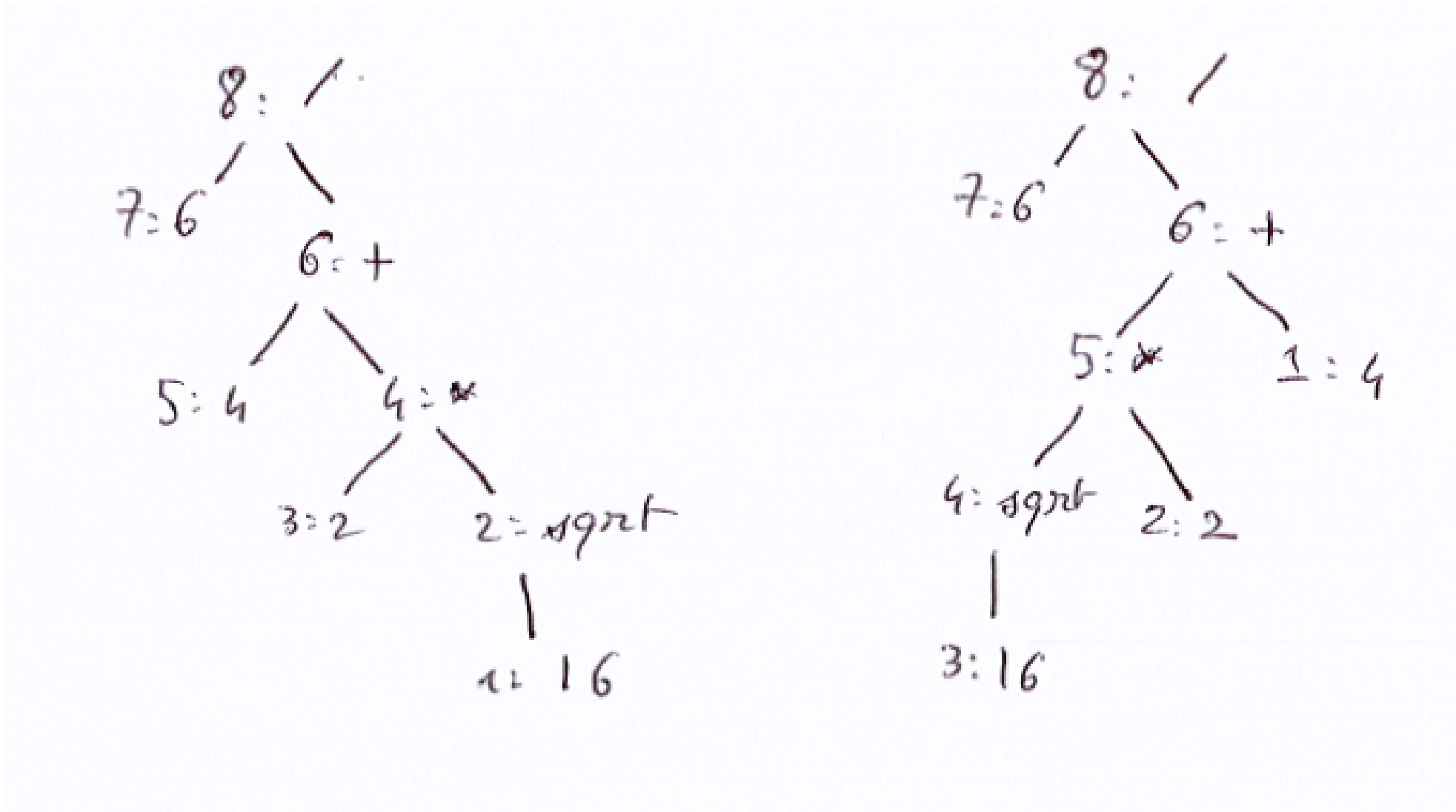

Figure 5.1: Illustration du fonctionnement de la récursivité. Dans chaque noeud de l'arbre, le chiffre à gauche est le numéro de la case dans le tableau commande et le message à droite est le contenu de cette variable.

```
}
```

Lors de l'exécution du programme la récursivité fonctionne en deux temps. Dans un premier temps, le programme met dans une pile les différents appels. Dans un deuxième temps, les traitements sont effectués et les différents calculs sont alors effectués comme le montre la figure 5.1.

5.3 Algorithmes utilisant un automate fini

L'état est en général un chiffre plutôt qu'une chaîne de caractère. Ce chiffre est en général manipulé sous la forme d'un groupe de lettre majuscule quand on utilise `#define` (voir le polycopie de C p. 21). Il est conseillé de définir la variable représentant l'état comme une énumération dont voici la syntaxe. On définit d'abord l'énumération en dehors de toute fonction, conformément à cet exemple.

```
typedef enum Jour {lundi=1,mardi,mercredi,jeudi,vendredi,samedi,dimanche
```

Dans une fonction qui se trouve après sur le même fichier ou qui suit une inclusion de fichier contenant cette définition, on peut alors l'utiliser ainsi

```
Jour j;
```

L'utilisation normal d'une énumération est de faire des affectations et des lectures des données sous la forme des éléments énumérés en l'occurrence les jours de la semaine. En

fait dans cet exemple il y a équivalence entre les jours de la semaine et les chiffres de 1 à 7. Mais pour utiliser cette équivalence, il faut faire une conversion explicite en mettant (Jour) juste avant le chiffre que l'on souhaite convertir.

La définition d'états permet dans l'algorithme de définir des tâches distinctes associées aux différents états et des tâches communes qui s'exécutent indépendamment de l'état. On peut alors considérer deux types d'algorithmes.

- Pour le premier type d'algorithme, le nouvel état est déterminé à l'issue de l'éxecution de la tâche spécifique relative à l'état en cours. L'ensemble de l'algorithme consiste donc en une initialisation qui notamment spécifie un état initial puis une boucle qui répète l'exécution tant qu'on n'a pas atteint un état final. Au sein de la boucle il y a une tâche spécifique qui est exécutée en fonction de l'état. Le branchement vers la tâche spécifique s'effectue généralement avec un `switch` décrit dans le polycopié de C en section 2.2 (p. 18).

- Pour le deuxième type d'algorithme, la tâche commune détermine une action et l'information combinée de l'action et de l'état détermine l'état suivant. Cette table à deux entrées est appelée une matrice de transition :

$$E' = T_{AE}$$

  $[T_{ij}]$ est la matrice de transition, A est la valeur associée à l'action, E est la valeur de l'état et E' est la nouvelle valeur de l'état. L'ensemble de l'algorithme consiste donc en une initialisation qui notamment spécifie un état initial puis une boucle qui

répète l'exécution tant qu'on n'a pas atteint un état final. Au sein de la boucle il y a une tâche spécifique qui est exécutée en fonction de l'état et une tâche générale qui détermine une action. La matrice de transition permet d'en déduire le nouvel état.

Exercice 42 *L'objectif est d'utiliser un automate pour programmer la recherche d'une solution à une variante du jeu des chiffres et des lettres. La fonction principale tire aléatoirement une valeur cible entre 0 et 1000 et* 8 *valeurs entre 0 et 100. A partir de ces* 8 *valeurs le programme doit trouver une façon de les combiner avec les opérandes* $+, -, *, /$ *de façon à obtenir la valeur la plus proche de la valeur cible. Concrètement le programme tire aléatoirement différentes séquences d'opérations à réaliser et utilise l'inplémentation de la calculatrice déjà faite lors de l'exercice* **??** *pour calculer les valeurs associées à chaque séquences d'opérations, la meilleure séquence est retenue.*

Voici un exemple de résultat :

```
En utilisant les nombres suivants
(3,1,45,20,67,70,45,62)
la cible a atteindre est 212
c'est la valeur 212 qui est atteinte
avec la séquence suivante
20 20 / 70 +3 * 45 + 1 / 45 - 1 -
en 641 essais
```

Ici le programme a très naïvement constaté que $212 = (70 + 1) * 3 - 1$

L'utilisation d'un automate permet ici de simplifier l'utilisation de boucles, ainsi le corps du programme devient

```
Etats etat=DEB;
while(FIN!=etat) {
  switch(etat) {
  ...
  }
  etat=transition[message][etat];
}
```

Chaque valeur de `etat` correspondent à une tâche à effectuer. Lorsque cette tâche est une condition, le résultat de cette tâche est inscrite dans la valeur de `message`. La table de transition implémentée par un tableau 2D `transition` détermine la tâche suivante. Le type `Message` est ainsi défini

```
typedef enum Message {
  NON=0,
  OUI=1,
} Message;
```

Le type `Etats` est défini avec une écriture condensée pour rendre le programme plus lisible. La convention utilisée ici est que si le premier caractère est le caractère souligné alors il s'agit d'un test.

```
typedef enum Etats {
  DEB    =0,          //début
  SQ_NV  =1,          //nouvelle séquence
  NB1    =2,          //tirage du premier nombre
  NB2    =3,          //tirage du deuxième nombre
  OP     =4,          //tirage de l'opérateur
  SQ_EV  =5,          //évaluation de la séquence
  _ME_SQ =6,          //est-ce que la séquence est meilleure
                      //que celles calculées jusqu'ici
  ME_SQ  =7,          //actualisation de la meilleure séquence
  _CI_OK =8,          //la cible est-elle atteinte
  _LG_SQ =9,          //est-ce que la séquence est trop longue
  _GD_ESS=10,          //est-ce que le nombre d'essais est trop grand
  FIN    =11,         //fin
  ERR    =12,         //erreur
} Etats;
```

Le programme fonctionne conformément au graphe représenté sur la figure 5.2. Ce graphe est implémenté de la façon suivante :

```
const Etats transition[2][13]=\
//DEB  ,SQ_NV,NB1,NB2,OP   ,SQ_EV ,_ME_SQ,ME_SQ ,_CI_OK,_LG_SQ ,_GD_ESS,FIN,ERR
{{SQ_NV,NB1  ,NB2,OP ,SQ_EV,_ME_SQ,_LG_SQ,_CI_OK,_LG_SQ,NB2    ,SQ_NV  ,FIN,ERR},  //NO
 {SQ_NV,NB1  ,NB2,OP ,SQ_EV,_ME_SQ,ME_SQ ,_CI_OK,FIN   ,_GD_ESS,FIN    ,FIN,ERR}}; //OU
```

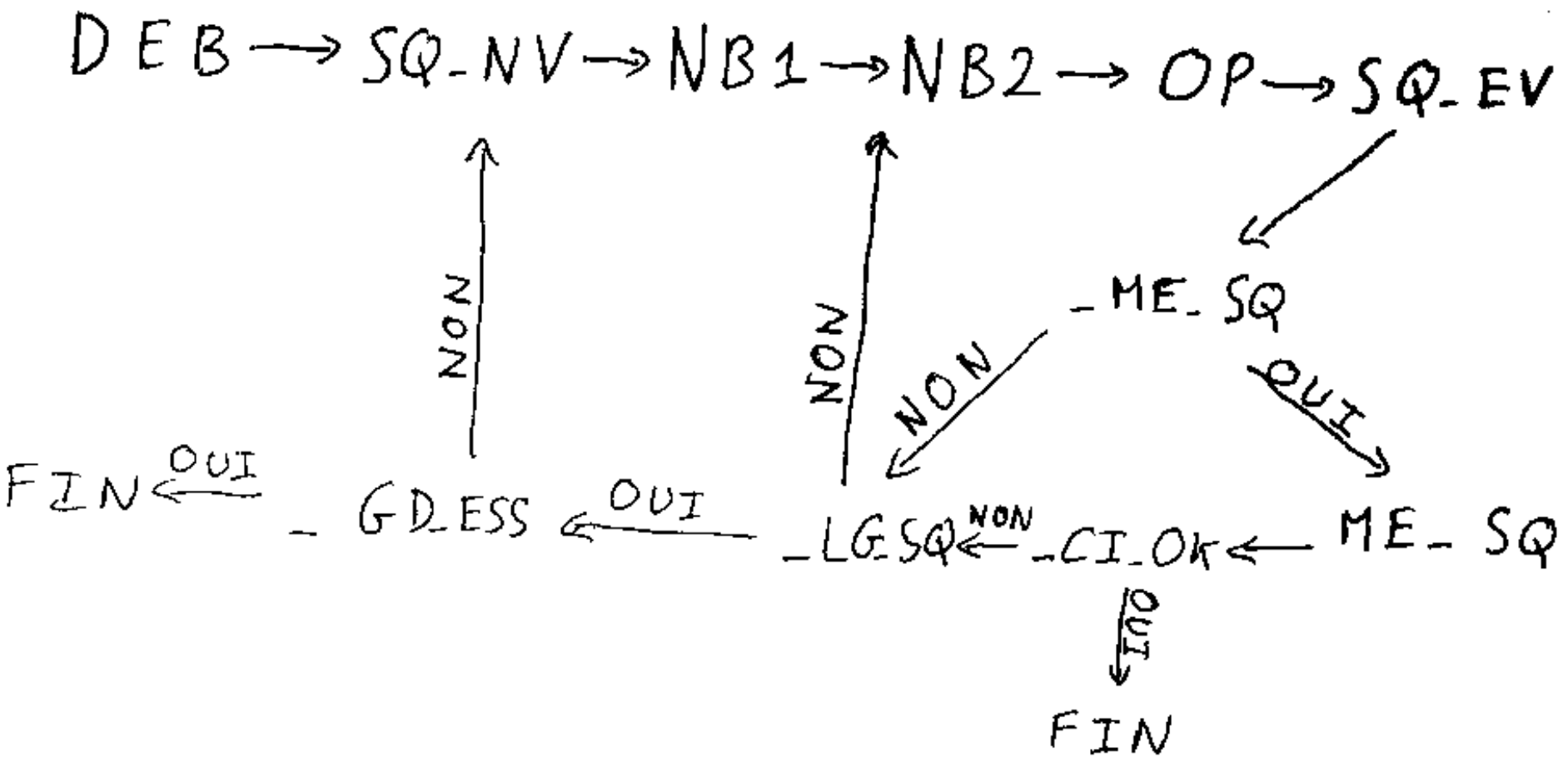

Figure 5.2: Graphe correspondant à l'automate permettant d'implémenter la solution de l'exercice sur les chiffres et les lettres.

Comme le programme devient un peu long il est conseillé de le séparer en différents fichiers sources.

5.4 Algorithme de Dijkstra

Cet algorithme est bien décrit sur internet par exemple sur

`https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`

On se donne un graphe non-orienté de noeuds et pour chaque paire de noeud en relation, il y a un nombre positif qui représente une distance pour aller d'un noeud à l'autre. Dans ce graphe il y a un noeud `depart` et un noeud `arrive`. L'objectif est de trouver le chemin le plus court entre le noeud `depart` et le noeud `arrive` au sens où la somme des valeurs de chacun des liens est la plus faible possible.

L'algorithme s'appuie sur une fonction `f` qui pour chaque noeud détermine l'ensemble de noeuds qui sont en lien direct avec ce noeud. Les résultats intermédiaires de l'algorithme sont stockés dans un tableau qui pour chaque noeud indique quand c'est possible, la plus courte distance avec le noeud `depart` et l'avant dernier noeud du chemin permettant d'aller du noeud `depart` à ce noeud. Lorsque sur ce tableau un noeud `i` est indiqué avec un chiffre, il est possible de retrouver le chemin permettant d'aller du noeud `depart` à `i` en retrouvant les noeuds dans l'ordre inverse : au noeud `i` est indiqué le noeud précédent et au noeud précédent est encore indiqué le noeud précédent et ainsi de suite. L'algorithme utilise aussi un deuxième fonction `g` qui en fonction d'une paire de noeuds `(i,j)` actualise

le tableau en déterminant s'il est préférable de conserver le chemin indiqué sur le tableau pour aller de `depart` à `j` où s'il vaut mieux adjoindre au chemin indiqué sur le tableau pour aller de `depart` à `i`, le chemin de `i` à `j`. L'algorithme utilise aussi une pile qui sert juste à se souvenir des noeuds à traiter. L'algorithme est alors constitué des étapes suivantes :

1. On met le noeud i=depart dans un pile.
2. Lire et retirer le dernier élément de la pile, qu'on appelle `i`.
3. Pour tous les noeuds `j`$\in$`f(i)`,
   (a) marquer que le noeud `j` a été considéré,
   (b) appliquer `g(i,j)` qui actualise le tableau,
   (c) on rajoute `j` dans la pile,
4. Tant que l'ensemble des noeuds n'ont pas été marqués, répéter l'étape 2.
5. A partir du tableau retrouver la distance la plus courte et l'ensemble des noeuds permettant de relier `depart` à `fin` avec la plus courte distance.

Exercice 43 *On considère un jeu d'échec composé de* 8 *lignes et chaque ligne est composé de* 8 *cases, au total il y a* 64 *cases. Dans le jeu d'échec le cavalier peut se déplacer à chaque tour de 2 cases dans une direction et d'une case dans*

l'autre direction (les directions étant soit verticales, soit horizontales), ainsi un des déplacement forme un L. On se donne une case de départ et une case d'arrivée, montrez comment l'algorithme de Dijkstra permet de calculer le nombre minimal de déplacements nécessaire pour que le cavalier passe de la case de départ à la case d'arrivée. L'idée est que chaque case du tableau est considérée comme un noeud d'un graphe et deux noeuds sont reliés entre eux quand le cavalier peut passer de la case associée au premier noeud à la case associée au deuxième noeud. Pour appliquer l'algorithme de Dijkstra, on construit un tableau regroupant tous les noeuds et dont les valeurs correspondent au nombre minimal de déplacements pour atteindre cette case. Pour se faire vous pourriez utiliser les définitions suivantes

- `typedef enum Boul {FALSE=0, TRUE=1} Boul;` *définit le type Boul.*
- `void setTab(int tab[],const int i, const int j,const int val);` *Cette fonction assigne* `val` *à la case* `i,j` *du tableau* `tab`*.*
- `int getTab(const int tab[],const int i,const int j);` *Cette fonction lit la valeur assignée à la case* `i,j` *du tableau* `tab`*.*
- `void initTab(int tab[]);` *Cette fonction initialise le tableau* `tab` *avec des valeurs* -1 *dont la signification est qu'ils n'ont pas encore été atteints par le cavalier.*

- `void ajustTab(int tab[],const int i,const int j,const int valNv);` *cette fonction actualise la valeur à la case* `i,j` *du tableau* `tab` *avec* `valNv`.

- `Boul estDansJeu(const int i, const int j);` *Cette fonction renvoie* `TRUE` *si les coordonnées* `i,j` *correspondent é une case du tableau.*

- `void marquer(int tab[],const int i,const int j);` *Cette fonction parcourt toutes les cases que le cavalier peut atteindre en partant de la case* `i,j` *en un déplacement et pour chacune de ces déplacements, les valeurs correspondantes des cases sont actualisées.*

- `void explorer1Dep(int tab[]);` *Cette fonction considère toutes les noeuds déjà rencontrès et explore les conséquences d'un déplacement supplémentaire.*

- `Boul explorer(const int iD,const int jD,const int iA,const int jA,const int cptMax,int * nbDep);` *Cette fonction indique s'il est possible de joindre la case* `iD,jD` *à la case* `iA,jA` *en moins de* `cptMax` *déplacements et dans ce cas elle retourne* `TRUE` *et indique ce nombre minimal de déplacements à l'adresse mémoire* `nbDep`.

5.5 Utilisation d'instructions provenant du C++ et de la bibliothèque STL

C++ est connu pour être un langage de programmation orienté objet. Il est aussi un langage contenant des instructions qui peuvent aider à programmer et ce sans connaître la programmation orienté objet. Cette utilisation d'instructions provenant du C++ et de la librairie STL (Standard Template Library) ne pose pas de problème à la compilation car quand on compile un programme en C, très souvent on utilise un compilateur C++.

5.5.1 Utilisation de `bool`

`bool` est un type dont les objets ne peuvent prendre que deux valeurs `false` ou `true`. Le résultat d'un test est de type `bool`, il n'est donc plus nécessaire de faire un conversion.

5.5.2 Utilisation de vecteurs permettant de faire de l'allocation dynamique et connaître la taille du tableau

Il est très délicat de passer d'un tableau classique à un `vector` et vice-versa.

Les bibliothèques à utiliser sont `iostream` et `vector`. Il est nécessaire de mettre sur chaque fichier l'instruction

```
using namespace std;
```

La déclaration se fait d'un vecteur de 7 éléments initialisés à 0 et dont les éléments sont de type `int`

```
vector<int> v(7,0);
```

On peut lui assigner des valeurs de la façon suivante

```
int tab[]={1,0,2,0,3};
v.assign(tab,tab+5);
```

Remarquez qu'alors la longueur du vecteur est modifiée.

Comme pour les tableaux l'accès aux éléments se fait avec `v[i]` où `i` est une valeur de type `int` entre 0 et 6, aucune vérification n'est faite du respect de la taille du vecteur.

On peut connaître la taille de ce vecteur

```
printf("longueur du vecteur %d\n",v.size());
```

Si on veut trier les éléments du vecteur, il faut rajouter la bibliothèque `algorithm`.

```
std::sort(v.begin(),v.end());
```

5.5.3 Utilisation de vecteurs pour réaliser une pile

La création d'une pile de `int`

```
vector<int> v;
```

On peut vérifier que la pile est vide :

```
if (v.empty()) printf("La pile est vide\n");
else printf("La pile n'est pas vide\n");
```

On peut empiler des éléments :

```
v.push_back(10);
v.push_back(20);
```

On peut lire le dernier élément rajouté :

```
printf("Le dernier element est %d\n",v.back());
```

On peut supprimer le dernier élément rajouté :

```
v.pop_back();
```

Un `vector` peut être défini sur n'importe quel type, cependant si on le définit avec une structure ou une chaîne de caractère, il est alors plus compliqué d'utiliser `sort`.

Appendix A

Autre cours conseillés et disponibles en ligne

Le polycopié de C donne déjà des références bibliographiques sur des cours. Le cours suivant est un cours par l'exemple, ce qui lui permet d'être à la fois simple, concis et précis :

`http://www-inf.enst.fr/~charon/CFacile/exemples/index.html`

Le cours suivant est bien expliqué et complet

`http://www.ltam.lu/Tutoriel_Ansi_C/`

Un extrait d'une discussion sur les raisons pour lesquelles `gcc` n'implémentent pas les fonctions `_s` de Microsoft et qui sont quasiment passées dans le standard `C11`.

https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1106.txt

Appendix B

Divers

B.1 Exemple de programmes

B.1.1 Le programme suivant illustre le fonctionnement de strtok_s.

```
#include <string.h>
#include <stdio.h>
int main(void)
{
  char phrase[] =    "Cette phrase est composee de mots separees d'espac
  char separateurs[]   = " ,\t\n";
  char *mot, *memoire;
```

```
  mot = strtok_s(phrase, separateurs, &memoire);
  while (1)
  {
    if (NULL==mot) break;
    printf(" %s\n", mot);
    mot = strtok_s(NULL, separateurs, &memoire);
  }
}
```

B.1.2 Utilisation d'une union dans une structure

On considère ici un tableau d'objets qui peuvent être soit des chaînes de caractères de 3 lettres soit des entiers. On pourrait définir une structure `Elemt` qui contient d'une part un tableau de trois lettres et d'autre part des entiers.

```
#include <stdio.h>

typedef struct {
  char str[4];
  int nb;
} Elemt;

int main(void) {
```

```
  Elemt tab[]={{"abc",0},{"",5},{"bac",0}};
  printf("%s %d %s\n",tab[0].str,tab[1].nb,tab[2].str);
  return 0;
}
```

Cette implémentation pose un problème celui de savoir si un élément est un nombre ou une chaîne de caractère. On code généralement cette information, ici dans un type énuméré.

```
#include <stdio.h>

typedef enum Type {
  STR=0,NB=1,
} Type;

typedef struct {
  Type type;
  char str[4];
  int nb;
} Elemt;

int main(void) {
  int i;
```

```
  Elemt tab[]={{STR,"abc",0},{NB,"",5},{STR,"bac",0}};
  for(i=0; i<sizeof(tab)/sizeof(tab[0]); i++) {
    if (STR==tab[i].type) printf("%s ",tab[0].str);
    else printf("%d ",tab[i].nb);
  }
  printf("\n");
  return 0;
}
```

La structure **Elemt** mesure 12 octets, il est possible de la raccourcir en remarquant qu'on utilise jamais la partie **nb** en même temps que la partie **str**. Cela se fait avec une **union** au lieu d'un **struct**. Ici le nom de la structure associé à cette union est anonyme, ce qui permet de ne pas précéder les champs **STR** ou **NB** du nom de l'union. La structure **Elemt** avec la nouvelle implémentation mesure maintenant 8 octets au lieu de 12 ($8 = 4 + 4\times1$ au lieu de $12 = 4 + 4\times1 + 4$).

```
#include <stdio.h>

typedef enum Type {
  STR=0,NB=1,
} Type;

typedef struct {
```

```
  Type type;
  union {
    char str[4];
    int nb;
  };
} Elemt;

int main(void) {
  int i;
  Elemt tab[]={
    {STR,{.str="abc"}},
    {NB,{.nb=5}},
    {STR,{.str="bac"}}
  };
  printf("%ld\n",sizeof(Elemt));
  for(i=0; i<sizeof(tab)/sizeof(tab[0]); i++) {
    if (STR==tab[i].type) printf("%s ",tab[0].str);
    else printf("%d ",tab[i].nb);
  }
  printf("\n");
  return 0;
}
```

B.2 Conseil de programmation

À titre personnel, j'estime qu'il est précieux de pouvoir exécuter régulièrement le programme que l'on écrit. Plus précisément lorsque j'écrit le programme, je l'écris dans un certain ordre de façon à toujours pouvoir tester ce que je suis en train d'écrire.

Il est parfois utile de vérifier le bon fonctionnement, une manière est d'afficher le contenu d'une variable ou de faire un test qui conduit à un affichage en cas d'échec, une autre façon, plus rapide à programmer provoque l'arrêt du programme chaque fois que le test a échoué. Cela peut se faire en incluant `#include <assert.h>` et faisant précéder l'expression test à évaluer de `assert`. Ainsi le programme suivant s'assure qu'il n'y a pas de caractères placés à droite du nom du programme lorsque celui est exécuté en ligne de commande.

```
#include <assert.h>
int main(int argc, char *argv[])
{
  assert(1==argc);
  return 0;
}
```

B.3 À propos de `sizeof`

À propos de l'utilisation de **`sizeof`**, cet exemple montre qu'il ne faut pas utiliser cette instruction dans une fonction ne disposant que de la copie du pointeur pointant sur le premier élément du tableau.

```
#include <stdio.h>
void fun(int *A);
int main(void) {
  int A[3];
  printf("%i\n",(int)(sizeof(A)/sizeof(A[0])));
  fun(A);
  return 0;
}

void fun(int *A){
  printf("%i\n",(int)(sizeof(A)/sizeof(A[0])));
}
```

Ce programme affiche

```
3
2
```

On peut noter que si dans la fonction `fun`, on remplace `int *A` par `int A[]`, on obtient un message d'avertissement

```
warning: 'sizeof' on array function parameter 'A' will return size of 'i
```

B.4 Exemple montrant que le qualificatif `const` ne protége pas tant que cela

```
#include <stdio.h>
void modifier(const int * a);
int main(void) {
  const int a=2;
  modifier(&a);
  printf("a=%i\n",a);
  return 0;
}
void modifier(const int * a) {
  int *b =(int*) a;
  (*b)++;
}
```

Ce programme compile et l'exécution affiche `3` au lieu de `2`.

Voici une façon de vérifier l'impact de la fonction `modifier`.

```
#include <stdio.h>
//#define NDEBUG
#include <assert.h>
void modifier(const int * a);
int main(void) {
  const int a=2;
  modifier(&a);
  printf("a=%i\n",a);
  #ifndef NDEBUG
    const int b=3; modifier(&b); assert(3==b);
  #endif
  return 0;
}
void modifier(const int * a) {
  int *b =(int*) a;
  (*b)++;
}
```

Lorsqu'on décommente la deuxième, c'est-à-dire l'instruction `#define NDEBUG`, le programme montre qu'il y a une erreur. Voici ce qu'on observe.

```
a=3
ex10: ex10.c:10: main: Assertion `3==b' failed.
Aborted (core dumped)
```

B.5 Constantes et variables intermédiares

Lorsqu'on utilise des chaînes de caractères, on peut souhaiter avoir une variable intermédiaire constante.

```
#include<stdio.h>
void aff(const char liste[][30]);
int main(void) {
  char liste[][30]={"bonjour","mot","demain"};
  aff(liste);
  return 0;
}

void aff(const char liste[][30]){
  int i;
  for(i=0; i<3; i++){
    const char * mot=liste[i];
    printf("%s ",liste[i]);
```

```
  }
  printf("\n");
}
```

On peut faire cela aussi avec une variable allouée statiquement une seule fois.

```
#include<stdio.h>
void aff(const char liste[][30]);
int main(void) {
  char liste[][30]={"bonjour","mot","demain"};
  aff(liste);
  return 0;
}

void aff(const char liste[][30]){
  int i;
  const char * mot[1];
  for(i=0; i<3; i++){
    *mot=liste[i];
    printf("%s ",liste[i]);
  }
  printf("\n");
}
```

B.6 Peut-on utiliser une fonction qui considère un tableau comme une constante sur un tableau qui n'est pas une constante

On peut effectivement utiliser une fonction qui prend en argument un tableau de type constant alors que le tableau passé n'est pas constant, **lorsque** le tableau est alloué statiquement ou lorsqu'il est 1D et alloué dynamiquement, mais pas lorsqu'il est 2D et alloué dynamiquement.

Le premier exemple concerne un tableau 1D alloué statiquement.

```
#include <stdio.h>
void aff(const int a[], int taille);
int main(void) {
  int a[]={1,2};
  aff(a,2);
  return 0;
}
void aff(const int a[], int taille) {
  int i;
  for(i=0; i<taille; i++) printf("a[%i]=%i\n",i,a[i]);
}
```

Le deuxième exemple concerne un tableau 2D alloué statiquement.

```
#include <stdio.h>
int somme(const int a[][2]);
int main(void) {
  int a[][2]={{1,2},{3,4}};
  printf("%i\n",somme(a));
  a[0][0]=5;
  printf("%i\n",somme(a));
  return 0;
}
int somme(const int a[][2]) {
  int i,j;
  int res=0;
  for(i=0; i<2; i++)
    for(j=0; j<2; j++)
      res+=a[i][j];
  return res;
}
```

Le troisième exemple concerne un tableau 1D alloué statiquement.

```
#include <stdio.h>
#include <stdlib.h>
void aff(const int a[], int taille);
```

```
int main(void) {
  int *a=(int *) malloc(2*sizeof(int));
  if (NULL==a) {printf("echec allocation\n"); getchar(); exit(-1);}
  a[0]=1; a[1]=2;
  aff(a,2);
  a[0]=3; a[1]=5;
  aff(a,2);
  free(a);
  return 0;
}
void aff(const int a[], int taille) {
  int i;
  for(i=0; i<taille; i++) printf("a[%i]=%i\n",i,a[i]);
}
```

Le dernier exemple concerne un tableau 2D alloué dynamiquement. Lorsqu'on déclare la fonction `somme` comme indiquée dans la ligne en commentaire, cela provoque un Warning.

```
ex6.c:14:23: warning: passing argument 1 of 'somme' from incompatible po
   14 |   printf("%i\n",somme(a));
      |                       ^
      |                       |
```

```
      |                           int **
ex6.c:3:29: note: expected 'const int * const*' but argument is of type
    3 | int somme(const int *const *a);
```

```
#include <stdio.h>
#include <stdlib.h>
//int somme(const int *const *a);
int somme( int *const *a);
int main(void) {
  int ** a=(int **)malloc(2*sizeof(int *));
  if (NULL==a) {printf("echec allocation\n"); getchar(); exit(-1);}
  int i;
  for(i=0; i<2; i++){
    a[i]=(int *) malloc(2*sizeof(int));
    if (NULL==a[i]) {printf("echec allocation\n"); getchar(); exit(-1);}
  }
  a[0][0]=1; a[0][1]=2;
  a[1][0]=3; a[1][1]=4;
  printf("%i\n",somme(a));
  for(i=0; i<2; i++){
    free(a[i]);
  }
  free(a);
```

```
  return 0;
}
//int somme(const int *const *a) {
int somme(int *const *a) {
  int i,j;
  int res=0;
  for(i=0; i<2; i++)
    for(j=0; j<2; j++)
      res+=a[i][j];
  return res;
}
```

Appendix C

Questions relatives à l'utilisation sous Windows

C.1 Installer `gcc` sur Windows 10 ou 11

Il est possible d'avoir accès à un noyau ayant des points de type Linux via Windows en installant `wsl` par exemple avec une configuration `Ubuntu`. Et avec `wsl` , il est possible d'installer `gcc`.

```
sudo apt-get update && sudo apt-get upgrade -y
sudo apt install gcc
```

Enfin une compilation très simple et une exécution se font en ligne de commande avec

```
gcc -o ex2 ex2.c
./ex2
```

Attention comme sur tous les systèmes linux, lorsqu'on utilise la bibliothèque `math.h`, il faut rajouter l'option `-lm` dans la commande `gcc`.

C.2 Compilation avec des fichiers séparés

Les fichiers `.c` rajoutés ne doivent pas contenir de seconde fonction `main`. Ils contiennent les définitions des fonctions mais pas les déclarations ni les définitions nouveaux types utilisés. Les fichier `.h` rajoutés contiennent les déclarations des fonctions ainsi que les définitions des nouveaux types utilisés.

L'inclusion des fichiers `.h` est faite dans chaque fichier `.c` susceptible d'en avoir besoin. Elle est faite avec une protection

```
#ifndef __nom1_h__
  #def __nom1__h__
#include "nom1.h"
#endif
```

L'instruction de compilation ne doit pas mentionner de fichiers `.h`, elle mentionne tous les fichiers .c utiles dans un ordre quelconque.

C.3 Microsoft Visual C++

Le logiciel Microsoft Visual C++ est plus qu'un simple éditeur de texte combiné avec un compilateur. Il permet de faire le lien entre différents fonctions appartenant au même programme (partageant une même fonction `main`), il permet aussi de regrouper différents programmes. On appelle Projet un ensembles de fichiers permettant de générer un programme avec une seule fonction `main`. On appelle solution un ensemble de projets pouvant donc avoir plusieurs fonctions `main`. Pour l'utiliser, voici les étapes :

- Créer une nouvelle solution s'obtient en cliquant l'onglet `Fichier` puis l'onglet `Nouveau` puis l'onglet `Projet` et en sélectionnant dans le menu déroulant l'élément `Autre type de projet` puis l'élément `Nouvelle solution`. Le bas de la fenêtre qui s'ouvre permet de choisir un nom et un emplacement dans le disque dur (pour l'emplacement, il est nécessaire de travailler sur un disque dur de l'ordinateur et non sur un répertoire contenu dans un serveur distant). La solution apparaît dans la case explorateur de solution.

- Créer un nouveau projet contenu dans cette solution s'obtient en cliquant sur cette solution avec le clic droit et en sélectionnant l'élément `Ajouter` puis l'élément `Nouveau projet` puis la rubrique `Application Console Win32`. Le bas de la fenêtre permet de donner un nom au projet et de choisir l'emplacement, (il est logique de laisser un emplacement à l'intérieur de la solution). Il apparaît un fenêtre `Bienvenu dans l'assistant` , il est important de créer un projet vide et pour

se faire de sélectionner l'onglet **Suivant** qui permet ensuite de cocher sur la case **Projet vide**.

- Créer un fichier .c s'obtient en sélectionnant dans l'explorateur de solutions la rubrique **fichiers sources** puis l'élément **Ajouter** puis l'élément **Nouvel élément**. Dans la nouvelle fenêtre apparue, il faut choisir **fichier c++**. Le bas de cette fenêtre permet de donner un nom et de préciser l'emplacement. Ce nom doit avoir l'extension **.c** et non **.cpp**.

- Quand il y a plusieurs projets, il est nécessaire de choisir le projet sur lequel vous travaillez. C'est ce que permet un clic droit sur le projet puis le choix de l'élément **Définir comme projet de démarrage**.

- Pour compiler, l'onglet Générer offre deux possibilités, soit compiler l'ensemble de la solution soit compiler le projet uniquement. Ce dernier choix évite de croire qu'il y a une erreur sur le projet sur lequel on travaille quand en réalité l'erreur est sur un autre projet.

- Pour exécuter, l'onglet Déboguer donne le choix entre **Démarrer le débogage** et **Exécuter sans débogage**. Il est possible aussi d'éxecuter le programme dans une fenêtre de type Dos. Cette fenêtre apparaît en écrivant dans le cercle Windows en bas à gauche de l'écran la commande cmd. L'exécutable généré lors de la compilation est à l'intérieur de l'emplacement de la solution dans le disque dur plus précisément dans le répertoire **Debug** à l'extérieur des projets. Les commandes **cd**

suivi d'un nom de répertoire ou un lettre suivi de : permettent de se positionner sur cet emplacement. Avec la souris et un clic droit il est possible de copier cet emplacement (l'indication correspondante est `copier l'adresse en tant que texte`). Un deuxième clic droit dans la fenêtre de type Dos permet de coller coller cet emplacement. L'exécution est obtenu en tapant le nom du fichier exécutable.

C.3.1 Évolution de l'interface

Dans la version 2015, `scanf` n'est plus autorisée et il faut utiliser `scanf_s`.

C.3.2 Utilisation de l'interface pour exécuter un programme en ligne de commande

Pour utiliser un programme en ligne de commande, il faut tout d'abord que le programme soit compilé (par exemple en avec l'onglet *Générer*). Ensuite il faut ouvrir le répertoire où se trouve l'exécutable. En se positionnant dans la fenêtre explorateur sur le projet (et non la solution) et avec un clic droit, il apparaît l'onglet *Ouvrir le dossier dans l'explorateur Windows*, cet onglet créé une fenêtre de cet explorateur dans un répertoire cousin de celui que l'on cherche, il faut donc remonter d'un niveau puis sélectionner le répertoire `Debug` et vérifier qu'il contient bien l'exécutable recherché. Attention, il y a deux répertoires Debug, celui qui nous intéresse ici n'est pas celui qu'on le voit initialement dans cette fenêtre. Avec l'onglet Windows en bas à gauche de l'écran où figure écrit

Taper ici pour rechercher, en écrivant `cmd`, on fait apparaître un terminal. Dans ce terminal, il faut se positionner sur le répertoire contenant l'exécutable. Cela se fait en deux temps, d'une part en tapant une lettre suivi de `:` pour choisir le disque souhaité, celui qui correspond à la fenêtre de l'explorateur. Puis dans un deuxième temps, avec la souris il est possible de copier le répertoire souhaité dans l'onglet au milieu ou en haut de la fenêtre contenant le chemin exact du répertoire et coller ce texte dans le terminal et précéder ce texte collé de l'instruction `cd`, ceci permet de se positionner dans le répertoire souhaité. L'instruction `dir` permet d'avoir la liste des fichiers et ainsi vérifier qu'on a bien l'exécutable (en principe un fichier qui se termine par `.exe`.

Appendix D

Idées pour trouver les erreurs dans un programme

Le compilateur détecte un certain nombre d'erreurs mais en laisse passer beaucoup aussi. Les warning peuvent servir à détecter des erreurs.

D.1 Erreurs d'algorithme

Les erreurs à l'exécution peuvent provenir d'une erreur d'algorithme :

D.1.0.1 Il ne faut pas utiliser utiliser les instructions suivantes même si syntaxiquement elles ne posent pas de problème

```
for(i=0;i<taille;i++) {
  if (pos==i) tab[i]=3;
}
```

D.1.0.2 Utilisation d'une double boucle, là où une simple boucle aurait permis de résoudre le problème.

D.1.0.3 Mauvaise utilisation d'une fonction qui renvoie un bouléen, en oubliant le point d'exclamation

```
if (test(...)) ...
```

au lieu de

```
if (!test(...)) ...
```

D.1.0.4 Lorsqu'on cherche à calculer un tableau à partir de deux tableaux, il est important de réfléchir à la taille du nouveau tableau qui ne dépend peut-être pas seulement des tailles de chacun des deux tableaux.

D.2 Erreurs ayant trait à l'utilisation particulière d'un compilateur particulier

»D.2.0.1 Dans certains cas, le compilateur exige une conformité avec la syntaxe C. Cette syntaxe oblige à ce que toutes les variables soient déclarées avant qu'il y ait exécution d'une commande.

»D.2.0.2 # define _CRT_SECURE_NO_WARNINGS n'est pas localisé avant l'inclusion des bibliothèques

D.3 Faux amis

»D.3.0.1 Il n'y a pas de `;` après

```
#include <...>
```

»D.3.0.2 Utilisation de `abs` au lieu de `fabs`

»D.3.0.3 L'instruction suivante affiche zéro

```
int k=5; printf("%d\n",1/k);
```

La raison est que l'opérateur `/` entre deux entiers donne un entier et non un double. La solution est par exemple

```
int k=5; printf("%lf\n",1.0/k);
```

D.4 Définition des fonctions

»D.4.0.1 Dans l'exemple suivant,

```
void produit_matriciel(double X[],const double ** A, const * B,ir
{
  ...
}
```

Le programme compile mais donne des résultats absurde. Il manque le fait de précise le type pour `B`.

D.5 Utilisation des pointeurs

»D.5.0.1 Quand on alloue dynamiquement un tableau à deux dimensions noté `A`, il faut désallouer d'abord `A[i]` puis désallouer `A`.

»D.5.0.2 Dans l'instruction suivante :

```
char c;scanf("%c",c);
```

il faudrait que le deuxième argument de `scanf`, `c` soit une adresse, et probablement il convient de mettre `&c`.

»D.5.0.3 Dans l'instruction suivante :

```
char mot[30];scanf("%s",&mot);
```

cela fonctionne mais on écrit jamais cela, on écrit plutôt :

```
char mot[30];scanf("%s",mot);
```

»D.5.0.4 La fonction `echanger` ne fonctionne pas :

```
void echanger(int * a,int * b)
{
  int * tmp;
  tmp=a;
  a=b;
  b=tmp;
}
```

Cette fonction échange des adresses qui sont les adresses des copies des adressses des variables à échanger. Il faut échanger les valeurs référencées par les copies des adresses des variables à échanger. Le corps de la fonction est donc plutôt :

```
  int tmp;
  tmp=*a;
  *a=*b;
  *b=tmp;
```

»D.5.0.5 Quand on veut dans une fonction mettre une valeur dans une structure allouée hors de la fonction et définie ainsi

```
typedef struct Valeur {
  double * valeur;
} Valeur;
```

On ne peut fonctionner ainsi

```
void set_valeur(Valeur * v,double val)
{
  v->valeur=&val;
}
```

Par contre, on peut faire cela

```
void set_valeur(Valeur * v,double val)
{
```

```
  v->valeur=(double *)malloc(sizeof(double));
  if (NULL==v->valeur) {printf("echec malloc\n"); exit(EXIT_FAILU
  *(v->valeur)=val;
}
```

»D.5.0.6 Quand il est demandé de réaliser une désallocation à propos d'une structure ainsi définie :

```
typedef struct Valeur {
  double * valeur;
} Valeur;
```

On ne peut fonctionner ainsi

```
Valeur v;
...
free(v);
```

Par contre, on peut faire cela

```
Valeur v;
...
free(v.valeur);
```

»D.5.0.7 Quand il est demandé de réaliser une allocation à propos d'une structure ainsi définie :

```
typedef struct Valeur {
  double * valeur;
} Valeur;
```

on ne peut faire cela

```
void allouer(Valeur v)
{
  v.valeur=(double *)malloc(sizeof(double));
if (NULL==v.valeur) {
    printf("echec malloc");
    getchar();
    exit(EXIT_FAILURE);
  }
}
```

par contre on peut faire cela

```
void allouer(Valeur *v)
{
```

```
  v->valeur=(double *)malloc(sizeof(double));
if (NULL==v->valeur) {
    printf("echec malloc");
    getchar();
    exit(EXIT_FAILURE);
  }
}
```

D.6 Mauvaise utilisation de const

»D.6.0.1 La fonction

```
void produit_matriciel(const double X[],const double ** A,\
 const double * B,int L,int C)
{
  int i,j;
  for(i=0;i<L;i++) {
    X[i]=0;
    for(j=0;j<C;j++) {
      X[i]+=A[i][j]*B[j];
    }
  }
```

```
}
```

provoque une erreur de compilation à cause de `const` sur le premier argument.

D.7 Fonction d'affichage

»D.7.0.1 L'instruction

```
printf("\n","arbre");
```

ne semble pas provoquer d'erreur à l'exécution, c'est quand même une erreur.

»D.7.0.2 L'instruction

```
printf("%s\n");
```

compile mais provoque une erreur à l'exécution.

»D.7.0.3 Le langage C n'a pas prédéfini l'affichage des types énumérés. Les instructions suivantes ne fonctionnent pas

```
#include <stdio.h>
typedef enum Jour {
```

```
LUNDI,MARDI,MERCREDI,JEUDI,VENDREDI,SAMEDI,DIMANCHE
} Jour;
int main(void)
{
  Jour j=LUNDI;
  printf("%s\n",j);
  return 0;
}
```

Par contre il est possible de prévoir une liste en vue de l'affichage

```
const char liste[][50]={"lundi","mardi","mercredi","jeudi",\
"vendredi","samedi","dimanche"};
Jour j=LUNDI;
printf("%s\n",liste[j]);
```

D.8 A propos de l'affichage des résultats

»D.8.0.1 L'affichage des résultats doit a priori se faire soit dans des fonctions pour l'affichage soit dans le `main`.

D.9 Non-utilisation des fonctions spéciales pour les chaînes de caractères

»D.9.0.1 Dans la fonction suivante

```
void remplir(char mot[],int nb)
{
  int i;
for(i=0;i<nb;i++) mot[i]='*';
}
```

il se peut qu'elle ne provoque aucune erreur, cependant il est très important de rajouter

```
mot[nb]='\0';
```

»D.9.0.2 L'affichage d'une chaîne de caractère qui ne se termine pas par \0 provoque un affichage problèmatique.

```
#include <stdio.h>
int main(void) {
  char mot[] = "bonjour";
  int i;
```

```
  mot[7] = ' ';
  for (i = 0; i < 7; i++) printf("%c", mot[i]);
  printf("\n");
  printf("%s\n", mot);
  return 0;
}
```

provoque par exemple l'affichage suivant

```
bonjour
bonjour ÌÌÌÌ!1°ü/ér
```

D.10 Expression à éviter

`n=n++;`

En réalité `n++` signifie déjà n=n+1;

D.11 Ne pas définir une fonction dans une fonction

Ceci est un exemple de programme ne compilant pas.

```
#include <stdio.h>

int main(void)
{
  int n=1;
  n=n++;
  void montrer(int n) {
    printf("n=%d\n",n);
  }
  montrer(n);
  return 0;
}
```

D.12 Ne pas utiliser comme nom de variable une instruction C

Voici un exemple de programme qui ne compile pas du tout.

```
void copie(const int tab1[], int tab2[], int taille)
{
  int case;
  for(case=0;case <taille; case++) {
```

```
    tab2[case]=tab1[case];
  }
}
```

D.13 Mauvaise implémentation d'un test d'appartenance à un intervalle

L'exécution de l'instruction suivante affiche `2<1<3 est vrai`.

```
if (2<1<3) printf("2<1<3 est vrai\n");
```

La raison est que `2<1<3` est équivalent à `(2<1)<3` qui est équivalent à `0<3` qui est effectivement vrai.

Une bonne façon d'implémenter ce test est la suivante.

```
if ((2<1) && (1<3)) printf("2<1<3 est vrai\n");
```

D.14 Implémention simpliste d'un `qsort` pour deux éléments

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

```
#include <memory.h>

int compare(const void * a,const void * b)
{
  const int * i1 =(const int * ) a;
  const int * i2 =(const int * ) b;
  return *i1-*i2;
}
void memcpy_(void * dest, void * src, int taille)
{
  int i;
  for(i=0; i<taille; i++)
    ((char*)dest)[i]=((char*)src)[i];
}
void qsort_(void * tab,int taille, int (*compare)(const void *,const voi
{
  void * case1=tab;
  void * case2=(void *)((char *)tab+taille);
  void * tmp;
  if (0<compare(case1,case2)) {
    tmp=(void *) malloc(taille);
    if (NULL==tmp) {
```

```
      printf("allocation impossible\n"); getchar(); exit(-1);
    }
    memcpy_(tmp,case1,taille);
    memcpy_(case1,case2,taille);
    memcpy_(case2,tmp,taille);
  }
}

int main(void)
{
  int A[2]={50,0};
  //qsort(A,2,sizeof(int),compare);
  qsort_(A,sizeof(int),compare);
  printf("A=[%d,%d]\n",A[0],A[1]);
  getchar();
  return 0;
}
```

D.15 Exemple de fonctions de vérification

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

```
#include <assert.h>
#include <time.h>
#include <stdlib.h>

//Fonction a tester
void insererElement2(int tab[], int taille, int rang, int val);
//Fonction qui réalise le test
void v_insererElement(void (__cdecl *insererElement)(int tab[],int, int,
//Fonctions utilisées par la fonction qui réalise le test
void copie(const int tab1[], int tab2[], int taille);
void affecter_case(int tab[],int taille,int val);
void afficher(const char nom[], const int tab[],int taille);
void affecter_hasard(int tab[],int taille);

int main(void)
{
  srand(time(NULL));
  v_insererElement(insererElement2);
  return 0;
}

//Fonction a tester
```

```
void insererElement2(int tab[], int taille, int rang, int val)
{
    int tmp1, tmp2,i;
    for (i = rang; i < taille-2; i++)
    {
        tmp1 = tab[i];
        tmp2 = tab[i + 1];
        tab[i + 1] = tmp1;
        tab[i + 2] = tmp2;
    }
    tab[rang] = val;
}

//Fonction qui réalise le test
//void v_insererElement(void (__cdecl *insererElement)(int tab[],int tai
void v_insererElement(void (*insererElement)(int tab[],int taille, int i
{
  int tab_avant[50],tab_apres[50], taille,\
    val_en_dehors1, val_en_dehors2, val, ind_a_inserer, ind ;
  val_en_dehors1=rand()%RAND_MAX;
  val_en_dehors2=rand()%RAND_MAX;
  val=rand()%RAND_MAX;
```

```
taille=rand()%49;
ind_a_inserer=rand()%(taille-1);
affecter_case(tab_avant,50,val_en_dehors1);
affecter_case(tab_apres,50,val_en_dehors2);
affecter_hasard(tab_avant,taille);
copie(tab_avant,tab_apres,taille);
insererElement(tab_apres,taille,ind_a_inserer,val);
printf("val=%d ind_a_inserer=%d\n",val,ind_a_inserer);
afficher("tab_avant",tab_avant,taille);
afficher("tab_apres",tab_apres,taille);
for(ind=0; ind<50; ind++)
{
  if (ind<ind_a_inserer) {
    assert(tab_avant[ind]==tab_apres[ind]);
  }
  else if (ind==ind_a_inserer) {
    assert(tab_apres[ind]==val);
    assert(tab_avant[ind]==tab_apres[ind+1]);
  }
  else if (ind<taille-1) {
    assert(tab_avant[ind]==tab_apres[ind+1]);
  }
```

```
    else if (ind==taille-1) ;
    else if (ind<50) {
      assert(tab_avant[ind]==val_en_dehors1);
      assert(tab_apres[ind]==val_en_dehors2);
    }
  }
}

//Fonctions utilisées par la fonction qui réalise le test
void afficher(const char nom[], const int tab[],int taille)
{
  int ind;
  printf("%s=[",nom);
  for(ind=0; ind<taille; ind++) printf("%d ",tab[ind]);
  printf("]\n");
}

void affecter_case(int tab[],int taille,int val)
{
  int ind;
  for(ind=0; ind<taille; ind++){
    tab[ind]=val;
```

```
  }
}

void copie(const int tab1[], int tab2[], int taille)
{
  int index;
  for(index=0;index <taille; index++) {
    tab2[index]=tab1[index];
  }
}

void affecter_hasard(int tab[],int taille)
{
  int index;
  for(index=0;index <taille; index++) {
    tab[index]=rand()%RAND_MAX;
  }
}
```