Travaux pratiques de langage C
2015-2016

Gabriel Dauphin

Ce document s'appuie sur un polycopié de C ainsi qu'un complément sur les listes chaînées. Ces travaux pratiques sont organisés en six séances de 2h de TP 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.

Table des matières

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

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. Cette partie 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. Les accents ne sont pas gérés en C, une solution possible est de les gérer avec `wchar_t`, il faut aussi adapter le reste (inclusion de bibliothèque, utilisation d'autres fonctions que `printf` et `scanf`, utilisation d'autre format que `%s` et `%c`, place mémoire plus grande).

L'adéquation entre le format et le type de la variable à afficher n'est pas vérifié par le compilateur. A 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.

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

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

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. 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'arrét 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. 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. 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. En fonction de ces choix, comment traiter la premiére itération. Si la premiére itération n'est pas identique aux autres, il est préférable d'implémenter cette différence avec une expression conditionnelle plutét que d'avoir du code similaire à deux endroits différents.

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

1.3 Algorithme avec un parcours séquentiel avec un nombre 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 aprés l'itération. Les questions à se poser sont les suivantes :

1. La condition qui entraéne la répétition ne doit en général pas se limiter au fait de constater que le probléme 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.
2. La tâche à itérer doit à terme pouvoir modifier la condition qui entraéne la répétition.
3. Quelles sont les quantités à calculer ? Ces quantités doivent être initialisées.
4. 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 ?
5. En fonction de ces choix, 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.
6. Est-ce que la dernière condition est correcte ? Est-ce qu'elle ne modifie pas les quantités à calculer d'une fois en trop ? Est-ce que les traitements faits à la dernière itération sont autorisés ?

Exercice 3 *On considére une suite définie par* $u_{n+1} = u_n^2 + u_n + 1$ *et* $u_0 = 0$. *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 6 (p. 12) ou de l'exercice 17 (p. 32).

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 6 (p. 12) et 15 (p. 31).

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 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. 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 pére.

```
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. 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. 26).

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 ?*
- *Ecrire le programme correspondant.*

On cherche maintenant à mettre des tests en commun.

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

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

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

Ecrivez un programme permettant de faire la multiplication entre deux entiers. Pour cela vous pourrez utiliser la fonction modulo qui s'implémente pour les entiers positifs avec un pourcentage, x *est paire si* $x\%2 == 0$ *et* x *est impaire si* $x\%2 == 1$.

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

2 Tableaux, tableaux de tableaux, pointeurs, fonctions

Dans cette partie, la fonction `main` génére les valeurs et les tableaux. On suppose que l'on connaét 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.

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 à 2.34 (p. 20 à 24).

L'utilisation des chaînes de caractères est décrite sur le polycopié de C en section 2.34 (p. 24-25). A ceci près, 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";
```

1. La raison pour 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.

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 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 à 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

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`.

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.

- 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.

- On considére ici le cas où l'argument correspond à une valeur de la variable `a`, qu'il s'agisse d'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 $\star$ et suivi du nom du pointeur de la variable, c'est ce pointeur qui est alors utilisé dans la fonction.
- Pour transmettre un tableau, il faut aussi transmettre la taille de ce tableau qui identifient les cases que la fonction pourra utiliser.
  - 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.
- Une chaîne de caractère est similaire à un tableau, à ceci prés que la taille peut se déduire du fait que la derniére case est occupée par `\0`, il n'y a pas de taille é 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 `[]`.
- Il existe aussi le cas où l'on transmet 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 23), c'est aussi le cas des tableaux de chaînes de caractères (voir la section 2.5 et 27). C'est aussi le cas pour permettre qu'une fonction manipule des valeurs d'un tableau sans que la déclaration de cette fonction n'ait à préciser le type de ces valeurs (voir la section 4.3 et 44).

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 é 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.

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 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'ambiguété.

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. 5)*

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. 6 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[4]. 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

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

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

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

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. 32). 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. 5 et 6). 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. 8). On peut aussi être amené à rajouter des variables temporaires dans les différentes tâches, mais si cette va-

riable temporaire est un tableau, il est préférable d'utiliser des fonctions pour agir dessus, ce qui est décrit dans la section 2.4 (p. 26).

Un algorithme de recherche dichotomique peut se voir comme un cas particulier de ceux définis dans la section 1.3 (p. 6). Une itération est caractèrisée 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 répéte 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 particuliére. 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 maniére fixe soit en utilisant un tableau comportant plusieurs lignes soit en utilisant de structures définies dans la section 3.3 (p. 40). 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. 21) : 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. 26)

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 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 premiére 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'accés 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 premiére 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]`.[5] Il est conseillé de réaliser des fonctions `getM` et `setM` pour accéder à ces données, par exemple avec la deuxième convention :

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

5. 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.

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. [6]

Exercice 11 *On considère un tableau à deux dimensions. Calculez la somme des éléments du tableau. Proposez deux solutions, la première utilisant une implémentation 1D et la deuxième utilisant une implémentation 2D.*

6. Ceci n'est plus vrai pour une allocation dynamique.

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. 20, p. 21, p. 22). 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 déjé existentes notamment celles qui existent pour les chaînes de caractères et qui sont rappelés briévement en section 3.1 (p. 32). Plus généralement, l'idée est qu'il est préférable de chercher à constituer des fonctions qui servent à manipuler des données particuliéres sans nécessiter la connaissance de l'ensembles des données du probléme, cette façon de choisir les fonctions amènent à raisonner de maniére 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 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. 23), à ceci près 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 [7]
  ```
  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
  ```
  const char tab[][30]={"maison","arbre","voiture"};
  const int taille=3;
  ```

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

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`.

2.6 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'occurence 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
```

2.7 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 ");
  }
}
```

2.8 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. 6).*

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.*

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

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

- `char * strcpy(char *,const char *)` : copie du deuxième argument sur le premier argument.
- `char * strncpy ( char * destination, const char * source, size_t num )` : copie des `num` premiers caractères du deuxième argument sur le premier argument.
- `char * strcat(char *,const char *)` : 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(char * cs,const char * ct)` 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).

- `double 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 à 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 sur la même ligne la fonction qui va donner l'adresse mémoire qui va être affecté à ce pointeur.

Exercice 18 *On considére un tableau de mots. Comptez le nombre de mots finissant par* tion. *Vous utiliserez la fonction* `strstr` *et* `strlen`. *On suppose ici que les mots ne sont pas suivis d'espaces.*

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`[8].

Il n'est pas possible d'allouer de la place mémoire à une adresse donnée. 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. 32) et au sein des fonctions `comparer` de la section 4.3 (p. 44).

8. Cela entraîne la non-désallocation de la mémoire et donc ce qu'on appelle une fuite de mémorie

Sauf pour faire de la réallocation de mémoire (voir section 3.2.1 p. 36), 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 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`. La désallocation de la mémoire se fait avec `free`.

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 [9]. En effet le 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.*

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. 36).

9. 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.

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*));
for(int i=0; i<L; i++)
  tab[i]=(int *)malloc(C*sizeof(int));
```

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 ne modifie pas 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. [10]

Exercice 20 *Utilisez une allocation dynamique pour allouer la matrice et le vecteur suivant. Faites le produit matriciel du premier par le deuxième*

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

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. 37), 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 :

```
char ** tab = (char **) malloc(L*sizeof(char *));
```

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

```
for(int i=0; i<L; i++)
  tab[i]=(char *)malloc((1+C)*sizeof(char));
```

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 à 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 à 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. On ne mettra pas non plus de tableaux ou de pointeurs dans une structure sauf dans le cadre spécifique d'un arbre ou d'une liste chaînée [11]

- 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

Exercice 21 *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;
```

11. La difficulté qui apparaît quand on met un pointeur ou un tableau est que si on passe la structure en argument d'une fonction, il y aura copie des données et donc copie des pointeurs ou de l'adresse du tableau en revanche les données pointées ne seront pas dupliquées, le résultat risque de ne pas être celui souhaité.

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

3.4 Exercices supplémentaires

Exercice 22 *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`. *Vous pourrez utiliser les fonctions sur les chaînes de caractères* `strlen`, `strcpy`, `strncpy`.

Exercice 23 *Il s'agit ici de compléter l'exercice 21 (p. 41). 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 21 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. 32).

Exercice 24 *On considére deux ensembles d'entiers A, B ne comptenant aucun doublon. Trouvez $A \backslash B$. 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.*

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`, voir la section 5.2.1 du polycopié de C. Dans le cadre de ce cours, nous n'utiliserons pas le format `"%c"`.

Exercice 25 *écrivez un programme qui calcule la moyenne des notes entrées itérativement au clavier et qui affiche la moyenne quand l'utilisateur entre -1*

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`.

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

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. 31).
- calcul d'histogramme : voir l'exercice 31 (p. 49).
- tirage aléatoire d'un ordonnancement : voir l'exercice 32 (p. 49).

La définition de cette fonction `qsort` est

```
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 [12]. 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 [13]. On remarque que le type utilisé pour faire la conversion explicite 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

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

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

```
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[0])-*(tab2[0]);
}
```

– 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);
}
```

Exercice 27 *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 évènement 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);
```

4.5 Exercices supplémentaires

Exercice 28 *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

```
******
caractere ? a
******
caractere ? e
*e**e*
caractere ? t
```

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

Exercice 29 *Complétez l'exercice 26 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 30 *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`.

Exercice 31 *On considére un tableau composé d'un certain nombre d'entiers. Donnez le nombre de valeurs distinctes et le nombre d'occurences de chaque valeurs distinctes. C'est ce qu'on appelle un histogramme. L'idée consiste à d'abord trier le tableau.*

Exercice 32 *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.*

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 33 *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.*

Exercice 34 *Utilisez la pile créée à l'exercice 33 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 33 pour implémenter cette calculatrice.

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).

Exercice 35 *Utilisez un algorithme récursif pour implémenter l'exercice 34 (p. 50). Vous pourrez par exemple construire une fonction dont la déclaration est :*

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

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.*

5.3 Algorithmes utilisant un automate fini

L'état est en général un chiffre plutét 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,samed
```

Dans une fonction qui se trouve aprés 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'occurence 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 répéte 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 répéte 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 36
```
typedef enum Etats {
  NOMBRE1,          //choix du premier nombre à mettre dans la s
  NOMBRE2,          //choix d'un nombre à mettre dans la séquenc
  OPERATEUR,        //choix d'un opérateur
  CALCUL_SEQUENCE,//calcul de la séquence et comparaison avec
```

```
} Etats;
```

La table de transition est donnée par

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 37 *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 à 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 rencontrés 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 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 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`.

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/`

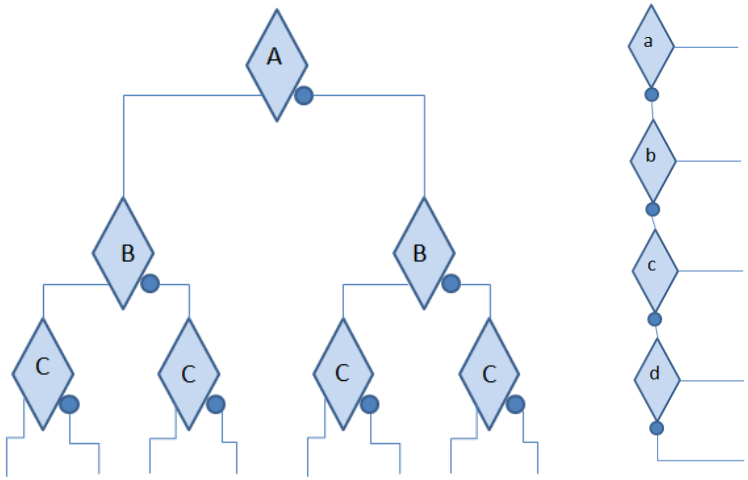

FIGURE 1 – Deux schémas de structures conditionnelles