Table des matières

3 Simulation de programme

Simuler l'exécution d'un programme, c'est indiquer chaque étape de son exécution (y compris les évaluations). On écrira les simulations de programme en réécrivant à chaque étape importante de l'exécution d'une instruction, des expressions entre accolades qui permettront de suivre l'évolution des valeurs des expressions et des variables au cours de l'exécution de l'instruction.

Une autre manière de procéder serait d'écrire des tableaux fournissant les valeurs des variables après exécution de chaque instruction (comme on l'a vu en TD). Mais dans ce cas, on détaille moins l'exécution des instructions complexes, puisqu'il manque le résultat de l'évaluation des expressions figurant dans ces instructions.

Les expressions de simulation que nous utiliserons dans la suite du polycopié figureront généralement entre acccolades. Ainsi par exemple, on utilisera :

3.1 Simulation des évaluations

- Pour une expression figurant dans une instruction on écrira :
  {évaluation des variables intervenant dans l'expression s'il y a lieu ;
  <expression> = <valeur>}

Dans ce contexte, le signe = signifie « a la valeur» ou « vaut ». Attention à ne pas le confondre avec de signe d'affectation du langage C.
exemple : Pour l'évaluation d'une expression booléenne (la <condition> entre parenthèses dans les instructions conditionnelles) on écrira ainsi

{évaluation des variables intervenant dans l'expression booléenne s'il y a lieu ;
(<expression booléenne>) = valeur}

3.2 Simulation des instructions d'affectation

On écrira simplement{<variable> = <valeur>} après avoir calculé la <valeur> de l'expression figurant en partie droite de l'affectation dans le programme.

3.3 Simulation des instructions d'entrées/sorties

- Ecriture de <expression1>, <expression2>…

On écrira simplement
écriture de{<expression1> = <valeur1>, <expression2> = <valeur2>, …, etc.}

- Lecture de <variable1>,<variable2>…;

On écrira alors des suites de *lecture de*{<variable1>=<valeur>, ..., }, récapitulant les valeurs des variables obtenues après lecture.

3.4 Simulation des instructions conditionnelles

Si des instructions sont exécutées dans une instruction conditionnelle, on simulera ces instructions en écrivant les valeurs nécessaires pour la simulation de l'exécution de chaque instruction, et en effectuant les indentations nécessaires pour la compréhension des cas de branchements d'instructions.

On procèdera de même pour les structures de contrôle qui vont être introduites maintenant. Nous verrons plus loin des exemples de simulation de programme.

4 Structures de contrôle

Les structures de contrôle sont les instructions prédéfinies d'un langage de programmation qui permettent d'organiser et de structurer l'ensemble des instructions qui composent un programme. Ces structures « contrôlent » donc l'enchaînement des instructions du programme. Les instructions d'un programme sont normalement organisées en séquence. Les structures de contrôle permettent de modifier cet ordre en introduisant des branchements conditionnels ou des boucles, ce qui permet d'éviter les répétitions.

Les structures conditionnelles *si-alors* et *si-alors-sinon* sont donc des structures de contrôle. L'exemple qui suit montre cependant combien il peut être laborieux d'écrire, de maintenir, et même de lire, un programme qui ne comporterait que des structures conditionnelles.

Exemple : moyenne

```
Algorithme moyenne
        10 variables d'entrées entières : terme1, terme2, terme3, terme4, terme5, terme6,
        terme7, terme8, terme9, terme10
        1 variable de sortie réelle : moyenne
        1 variable auxiliaire entière : somme
début
        écrire(" Entrez les 10 termes de la moyenne à calculer ")
        lire(terme1, terme2, terme3, terme4, terme5,
                terme6, terme7, terme8, terme9, terme10)
        somme <- la somme des 10 termes
        moyenne <- somme/10
        écrire(" La moyenne de ces 10 termes est ",moyenne)
fin algorithme
```

Malgré sa simplicité, cet algorithme qui calcule la moyenne de 10 entiers est peu satisfaisant : il faut manipuler 10 variables différentes, écrire de longues listes d'arguments... et surtout ce programme ne calcule la moyenne que pour 10 entiers exactement!

Il serait évidemment plus intéressant de réécrire le programme pour qu'il calcule les moyennes d'un nombre quelconque d'entiers. Cela peut être fait en introduisant une variable représentant le nombre de termes participant dans la moyenne, et en utilisant comme structure de contrôle une boucle *tant que* :

```
Algorithme moyenne2
        2 variables d'entrée : nbTermes et terme
        1 variable de sortie réelle : moyenne
        2 variables auxiliaires entières : somme et i
```

```
début
        écrire(" Combien de termes entrent dans la moyenne à calculer ? ")
        lire(nbTermes)
        somme<-0
        i<-1
        tant que (i ≤ nbTermes) faire
                écrire(" Entrez le terme ",i)
                lire(terme)
                somme <- somme + terme
                i<-i+1
        fin tant que
        moyenne <- somme/nbTermes
        écrire(" La moyenne de ces 10 termes est ",moyenne)
fin algorithme
```

De façon générale, si on n'utilisait pas de structures de contrôle, les programmes comporteraient toujours des lignes de code répétées. Il est bien entendu intéressant de pouvoir factoriser ce code et de rendre ainsi le texte du programme plus concis. Cela diminue le temps d'écriture du programme, facilite sa mise à jour (une modification peut se faire en un point au lieu de devoir être faite en plusieurs), et augmente la lisibilité du code produit.

Les structures de contrôle permettent donc d'organiser les séquences d'instructions d'un programme de manière plus économique et plus rationnelle. Il existe deux grands types de structures de contrôle : les structures conditionnelles, que nous avons déjà rencontrées, et les boucles.

4.1 Les boucles

Les boucles permettent de répéter une instruction ou une séquence d'instructions plusieurs fois. En pratique, on contrôle la répétition en évaluant une expression conditionnelle.

4.1.1 La boucle `tant que`

a. Algorithme et utilisation

Les boucles *tant que* permettent de répéter une séquence d'instructions tant qu'une condition donnée est vérifiée (i.e. l'expression booléenne correpondante est évaluée à vrai).

La syntaxe est la suivante :

```
tant que (<condition>) faire
   <suite d'instructions >
fin tant que
```

Ce qui signifie : " tant que la <condition> est vérifiée, exécuter la <suite d'instructions> ".

Si la condition, qui est une expression booléenne, est évaluée à vrai, le programme entre dans la boucle et exécute la suite d'instructions. Une fois la suite d'instructions exécutée, la condition est réévaluée. Si elle est évaluée à vrai, le programme entre de nouveau dans la boucle et exécute une nouvelle fois la suite d'instructions. Si elle est évaluée à faux, le programme n'exécute pas la suite d'instructions, on dit qu'il « sort de la boucle », et l'instruction suivante (celle qui suit l'instruction *tant que*) est exécutée.

Il est possible que dès la première évaluation de la condition, la valeur retournée soit fausse. Dans ce cas, le programme « n'entrera » jamais dans la boucle et n'exécutera jamais la suite d'instructions du *tant que*. Le contrôle du programme enchaîne alors directement sur l'instruction suivante qui suit l'instruction tant que.

L'exemple qui suit montre l'économie que permet de faire cette boucle.

Exemple

L'algorithme suivant est une variante de l'algorithme moyenne2 ci-dessus. Cette version est adaptée au cas où l'utilisateur ne sait pas à l'avance combien de termes il veut entrer.

```
Algorithme moyenne3
        1 variable d'entrée réelle : terme
        1 variable d'entrée caractère : reponse
        1 variable de sortie réelle : moyenne
        2 variables auxiliaires entières : somme, cmpt
début
            /* Initialisations des variables  reponse, somme et cmpt */
        écrire(" Voulez-vous calculer une moyenne (o/n) ? ")
        lire(réponse)
        somme<-0
        cmpt<-0
            /* Lecture des termes et calcul de leur somme */
        tant que (reponse='o') faire
           écrire(" Entrez un terme ")
           lire(terme)
           somme<-somme+terme
           cmpt<-cmpt+1
           écrire(" Voulez-vous continuer (o/n) ? ")
           lire(reponse)
        fin tant que
            /* calcul et affichage du résultat */
        moyenne <- somme/cmpt
        écrire(" La moyenne de ces " ,cmpt, " termes est ",moyenne)
fin algorithme
```

Il faut souligner l'importance de la condition (appelée souvent *condition d'arrêt*). Si cette condition ne peut pas fonctionner comme condition d'arrêt (c'est-à-dire si elle ne prend jamais la valeur *faux*), le programme exécute alors une boucle infinie (on dit qu'il « boucle »).

Exemple

```
Algorithme boucle_infinie
        1 variable d'entrée entière : n
début
           /* Initialisation du compteur n*/
        n <- 1
           /* Affichage */
        tant que (n ≠ 100) faire
           écrire (" Ligne " ,n )
        fin tant que
        écrire(" Fin de l'impression ");
fin algorithme
```

Exemple

```
Algorithme multiplication_par_deux
        1 variable d'entrée entière : n
début
            /*Initialisation de n*/
        écrire(" Entrez  un entier positif :  ")
        lire(n)
        tant que (n>0) faire
            n<-n*2;
            écrire(n)
        fin tant que
fin algorithme
```

Dans le premier cas, on a oublié – et c'est une erreur fréquente – d'incrémenter le « compteur », c'est-à-dire la variable n, dans le corps de la boucle. La variable n garde donc toujours la valeur 1 et le programme affiche « ligne 1 » indéfiniment. La condition (n ≠ 100) ne peut en effet jamais être invalidée. Dans le deuxième cas, l'entier n est multiplié par deux à chaque pas, et il ne prendra jamais la valeur 0. (Il n'est pas absurde cependant d'écrire des boucles de programme infinie : beaucoup de programmes destinés à la gestion système sont en effet organisés sous forme de boucle infinie. Par exemple, attendre une entrée, la traiter, puis recommencer).

Une autre erreur de programmation classique consiste à donner une condition d'arrêt qui est vérifiée à l'origine (à cause d'une mauvaise initialisation de variables), et qui empêche dès le départ de rentrer dans la boucle.

b. Simulation de l'exécution d'une boucle tant que

La simulation d'une boucle *tant que* est semblable à la simulation d'une série d'instructions conditionnelles.

Exemple : Simulation de moyenne2

```
Ecriture de " Combien de termes entrent dans la moyenne à calculer ? "
Lecture (nbTermes) {nbtermes = 4}
{somme = 0}
{i=1}
{(i ≤  nbTermes)=vrai} début tant que
                écriture de " Entrez le terme 1 "
                lecture(terme) {terme=3}
                {somme=3}
                {i=2}
{(i ≤  nbTermes)=vrai} suite tant que
                écriture de " Entrez le terme 2 "
                lecture(terme) {terme=5}
                {somme=8}
```

{i=3}
{(i ≤ nbTermes)=vrai} suite tant que
écriture de " Entrez le terme 3 "
lecture(terme) {terme=4}
{somme=12}
{i=4}
{(i ≤ nbTermes)=vrai } suite tant que
écriture de " Entrez le terme 4 "
lecture(terme) {terme=0}
{somme=12}
{i=5}
{(i ≤ nbTermes)=faux} fin tant que
{somme=12 ; nbTermes=4 ; moyenne=3}
écriture de " La moyenne de ces 4 termes est 3 "

c. Les deux boucles `while` *du C*

Comme beaucoup de langages de programmation, le C dispose de deux types de boucles de type " tant que " : des boucles *tant que condition... faire* et des boucles *faire... tant que condition*.

La boucle **tant que** de notre langage algorithmique s'implémente en C au moyen de la structure de contrôle (ou instruction) `while`. Traduisons l'exemple précédent (moyenne3) en C.

```
#include <stdio.h>

int main() {
      int terme, somme , cmpt ;
      char reponse ;
      float moyenne ;
                  /* Initialisations des variables */
      printf("Voulez-vous calculer une moyenne (o/n) ? ") ;
      scanf("%c", &reponse) ;
      somme = 0 ;
      cmpt = 0 ;
            /* Lecture des termes et calcul de leur somme */
      while (reponse =='o') {
          printf("\nEntrez un terme : ") ;
          scanf("%d",&terme);
          somme = somme + terme ;
          cmpt = cmpt + 1 ;
          printf("\nVoulez-vous continuer (o/n) ? ") ;
          scanf("%c",&reponse) ;
      }
            /* calcul et affichage du résultat */
      moyenne = (float) somme/ cmpt ;
      printf("La moyenne de ces %d termes est %f ",
                  cmpt, moyenne);
      return 0 ;
}
```

Comme on le voit sur cet exemple, la syntaxe d'une boucle `while` est la suivante :

```
while (<expression booléenne>) {
     <suite d'instructions>
}
```

En contraste, la syntaxe de la boucle `do-while` est la suivante :

```
do {
     <suite d'instructions>
} while (<expression booléenne>) ;
```

(On peut aussi dans ces deux formes de `while`, remplacer les blocs d'instructions entre accolades par une instruction simple finissant par un point-virgule, comme dans le cas des instructions conditionnelles).

Prenons un exemple pour illustrer le contraste entre ces deux structures :

```
#include <stdio.h>

int main() {
        int terme , somme , cmpt ;
        char reponse ;
        float moyenne ;
                    /* Initialisations des variables */
        printf("Voulez-vous calculer une moyenne (o/n) ? ") ;
        scanf("%c", &reponse) ;
        somme = 0 ;
        cmpt = 0 ;
              /* Lecture des termes et calcul de leur somme */
        if (reponse == 'o') {
            do {
                printf("Entrez un terme : ") ;
                scanf("%d", &terme) ;
                somme = somme + terme ;
                cmpt = cmpt + 1 ;
                printf("Voulez-vous continuer (o/n) ? ") ;
                scanf("\n%c", &reponse) ;
            }
            while (reponse != 'n');
        }
              /* calcul et affichage du résultat */
        moyenne= (float) somme / cmpt ;
        printf("La moyenne de ces %d termes est %f \n",
                cmpt, moyenne);
        return 0 ;
}
```

La différence entre ces deux structures de contrôle tient à la place de la condition d'arrêt. Dans le premier cas (instruction *while*), la condition est évaluée avant d'entrer dans la boucle. Si donc cette condition n'est pas vérifiée initialement, la boucle n'est pas exécutée du tout ; En revanche, dans une instruction *do-while*, la condition est testée à la fin de la boucle. Les instructions constituant la partie *action* de la boucle sont donc toujours exécutées au moins une fois, (la première fois), *même si la condition* est fausse dès le départ.

Dans l'exemple ci-dessus, il faudrait insérer la boucle *do-while* dans une structure conditionnelle pour empêcher le programme d'entrer dans la boucle, au cas où l'utilisateur ne souhaiterait entrer aucun entier. Sinon, le programme devrait lire un terme au moins une fois.

Une boucle *while* est donc plus indiquée ici, car elle bloque l'entrée dans la boucle si l'utilisateur ne veut pas calculer de moyenne.

4.1.2 Les boucles `pour`

a. Algorithme et utilisation

Les boucles *pour* permettent de répéter une séquence d'instructions un nombre de fois déterminé, et fixé à l'avance. La syntaxe d'une instruction *pour* est la suivante :

pour <*compteur*> ← <*valeur de début*> *à* <*valeur de fin*> *faire*
    <*suite d'instructions* >
fin pour

<*compteur*> , <*valeur de début*> et <*valeur de fin*> sont des entiers. Les valeurs de début et de fin forment un intervalle, et le compteur parcourt cet intervalle de manière ascendante ou descendante. Pour chaque valeur prise par le compteur dans cet intervalle, la suite d'instructions est exécutée une seule fois.

En pratique, la variable <*compteur*> est initialisé à <*valeur de début*> à l'entrée dans la boucle. Cette initialisation est faite automatiquement.

Si on a <*valeur de début*> ≤ <*valeur de fin*> au départ, on effectue un premier passage dans la boucle avec compteur égal à <*valeur de début*>. A la fin du passage, le compteur est incrémenté de 1 puis comparé à <*valeur de fin*>. S'il est inférieur ou égal à <*valeur de fin*>, on entre de nouveau dans la boucle. On en sortira finalement quand le compteur se trouvera supérieur à <*valeur de fin*>.

A l'inverse, si on a <*valeur de fin*> ≤ <*valeur de début*>, on effectue un premier passage dans la boucle avec compteur égal à <*valeur de début*>, mais à chaque passage, le compteur est décrémenté de 1 en fin de boucle, puis comparé à <*valeur de fin*>. S'il est supérieur ou égal à <*valeur de fin*>, on entre de nouveau dans la boucle. On en sort quand le compteur est inférieur à <*valeur de fin*>.

Les boucles *pour* sont utilisées lorsqu'on connaît par avance le nombre d'itérations à effectuer.

Exemple

```
Algorithme moyenne4
        2 variables d'entrée : nbTermes et terme
        1 variable de sortie réelle : moyenne
        2 variables auxiliaires entières : somme et i
début
        écrire(" Indiquez le nombre de termes dont vous boulez calculer la moyenne : ")
        lire(nbTermes)
        somme← 0
        pour i ← 1 à nbTermes faire
                écriture(" Entrez le terme ",i)
                lire(terme)
                somme = somme + terme
        fin pour
        moyenne = somme/nbTermes
        écrire(" La moyenne de ces " ,nbtermes, " termes est ",moyenne)
fin algorithme
```

Si l'on compare les algorithmes moyenne 2 et moyenne 4, on voit ce qui distingue les boucles `while` des boucles *pour* : lorsqu'on utilise une boucle `while`, il faut initialiser la variable qui sert à contrôler la boucle (ici i) avant la boucle (instruction i←1), puis la mettre à jour à la fin de la boucle (instruction i←i+1). Ces deux opérations d'initialisation (i←1), et d'incrémentation (i← i+1) sont implicites dès lors qu'on utilise une structure *pour*.

Exemple : Algorithme factorielle avec une boucle `pour`

```
Algorithme
        1 variable d'entrée entière : n
        1 variable de sortie entière : res
        1 variable auxiliaire entière : i
début
        écrire (" Quelle factorielle voulez-vous calculer ? ");
        lire(n);
        res <- 1
        pour i← 2 à n faire
                res <- res * i
        fin pour
        écrire(" La factorielle de ", n, " est " , res);
fin algorithme
```

b. Simulation

La simulation d'une boucle `pour` fait apparaître l'initialisation et l'incrémentation implicites du compteur avant sa comparaison avec la borne d'arrêt (supérieure ou inférieure).

Simulation factorielle_pour

Ecriture de " Quelle factorielle voulez-vous calculer ? "
Lecture de n {n = 4}
{res = 1}
{i = 2 ; (i <= n) = vrai} Début pour
{res = 1*2=2}
{i=3 ; (i<=n)=vrai} Suite pour
{res = 2*3 = 6}

{i=4 ; (i<= n) = vrai} Suite pour
{res = 6*4 = 24}
{i=5 ; (i<= n) = faux} Fin pour
{n=4 ; res=24} Ecriture de " La factorielle de 4 est 24. "

c. Comparaison entre les boucles pour et tant que

On peut généralement traduire une boucle *pour* en boucle *tant que* assez facilement. Il faut simplement initialiser le compteur au préalable et en gérer explicitement l'incrémentation. L'inverse n'est pas toujours vrai : certaines boucles *tant que* ne peuvent pas être transformées en boucle *pour*. On notera sur l'exemple qui suit que lorsqu'une boucle `pour` peut être utilisée, elle améliore la lisibilité du programme. Par ailleurs, le risque d'avoir des boucles infinies est moins grand (à moins que la valeur du compteur ne soit systématiquement modifiée dans le corps de la boucle *pour* : une pratique à proscrire !!).

Exemple : Algorithme factorielle-tant-que

```
Algorithme
        1 variable d'entrée entière : n
        1 variable de sortie entière :res = n !
        1 variable auxiliaire entière : i
début
        écrire (" Quelle factorielle voulez-vous calculer ? ");
        lire(n);
        i<- 1
        res <- 1
        tant que (i≤n) faire
                res <- res * i
                i<-i+1
        fin tant que
        écrire(" La factorielle de ", n, " est " , res)
fin algorithme
```

d. Un moyen d'implanter la boucle `pour` en C : l'instruction `for`

Syntaxe

Instruction `for` en C

```
for ( <init>; <condition> ; <pas>)
<instruction> ;
[ ou <bloc d'instructions> ]
```

où `<init>` est une instruction évaluée avant de pénétrer dans la boucle. En général, il s'agit de l'initialisation de la variable de contrôle de la boucle, et init a donc la forme : <compteur> = <valeur de début>. Cette instruction n'est exécutée qu'une seule fois.

`<condition>` est la condition d'entrée dans la boucle (en général <compteur> ≤ <valeur de fin>). C'est une expression booléenne. Si elle est évaluée à vraie, l'instruction est exécutée, suivie de l'exécution du <pas>, puis la condition est réévaluée afin de déterminer s'il y a une

itération suivante. Dans le cas où la condition est évaluée à faux, le <pas> n'est pas exécuté, et on sort de la boucle (on enchaîne sur la continuation du calcul après le for).

<pas> est donc une instruction exécutée en fin de boucle, avant que la condition soit à nouveau testée. Cette instruction permet en général de mettre à jour la variable qui contrôle la condition (c'est-à-dire en général , régler le pas d'incrémentation avec <compteur> <- <compteur>+1).

La boucle `for` est donc en réalité équivalente à :

```
<init> ;
while(<condition>) {
    <instruction> ;
    <pas>;

}
```

Pour écrire un C « discipliné », on restreindra les instructions <init> et <pas> à des affectations de variables (ou éventuellement, des séquences d'affectations, séparées par des virgules).

Exemple

```
#include <stdio.h>
int main() {
        int n, res = 1, i ;

        printf(" Quelle factorielle voulez-vous calculer ? ") ;
        scanf("%d",&n) ;
        for(i=1 ;i<=n ; i=i+1)
            res = res*i ;
        printf(" La factorielle de %d est %d.", n, res) ;
        return 0 ;
}
```

Notez que la boucle `for` du C est beaucoup plus générale que la boucle *pour* que nous avons introduite dans le langage algorithmique. La `<condition>` peut être n'importe quelle expression booléenne ; le `<pas>` peut également se traduire par une instruction quelconque, effectuée en fin de boucle. Le pas d'incrémentation le plus classique est l'incrémentation de 1 mais on peut utiliser d'autres instructions. Dans l'exemple ci-dessous qui affiche les nombres pairs, et on utilise une incrémentation de 2. On peut également remplacer l'incrémentation par une instruction de décrémentation (`<compteur>` <- `<compteur>` - 1) pour traduire les boucles *pour* descendantes (celles où *<valeur de début>* est supérieure ou égale à *<valeur de fin>*).

Exemple

```
#include <stdio.h>

int main() {
                    /*affiche les entiers pairs compris dans un
                      intervalle fixé par l'utilisateur */
      int borneInf, borneSup, i ;

      printf("Fixez un intervalle\n") ;
```

```
    printf("Donnez la borne inférieure : ") ;
    scanf("%d",&borneInf) ;
    printf("Donnez la borne supérieure : ") ;
    scanf("%d",&borneSup) ;
    if (borneInf <= borneSup) {
      if (borneInf%2 !=0)
          borneInf = borneInf + 1 ;
      for (i=borneInf ; i<=borneSup; i=i+2)
          printf("%d",i) ;
    };
    return 0;

}
```

D'une manière générale cependant, il est préférable, d'utiliser une boucle `while` qui est plus explicite. On essayera dans ce cours de restreindre l'usage des boucles `for` du C aux cas où l'on connaît par avance le nombre de passage dans la boucle, afin de traduire uniquement avec un `for`, la boucle *pour* de notre langage algorithmique. Cependant, on peut noter qu'en C, la boucle `for` est très générale, et qu'elle permet d'écrire n'importe quelle boucle.

En outre, il faut noter que les instructions et expressions d'un for peuvent être vides. Si la condition est absente, elle agira comme une condition évaluée à vraie. Ainsi l'instruction

```
for ( ; ; )
    <instruction> ;
```

représente une boucle infinie qui exécute indéfiniement l'instruction.

4.2 Les structures conditionnelles

4.2.1 Structures `if-then` et `if-then-else`

Voir au paragraphe 1.6. la définition des structures conditionnelles.

4.2.2 L'instruction sélective ou instruction `switch`

Cette structure permet d'introduire un traitement différencié pour chacune des valeurs que peut prendre une expression de type `int` ou `char`. Il s'agit d'un branchement calculé sur la valeur de cette expression. On liste l'ensemble des valeurs constantes de type `int` ou `char` que peut prendre une expression à valeur entière ou caractère, et on associe une série d'instructions à chaque cas.

La syntaxe de cette instruction en C, est la suivante :

```
switch (<expression>) {
     case <constante 1> : <bloc d'instructions 1>;
     case <constante 2> : <bloc d'instructions 2>;
     ...
     case <constante N> : <bloc d'instructions N>;
     [default : <bloc d'instructions par défaut> ;]
}
```

En pratique, l'expression (de type `int` ou `char`) est évaluée et sa valeur est comparée aux constantes associées aux différents cas (introduit par `case`). Les cas sont ainsi parcouru jusqu'à ce que la bonne valeur soit trouvée. Le bloc d'instructions associé à ce cas est alors exécuté. Il est possible également d'avoir un cas par défaut qui sera exécuté si la valeur de l'expression ne correspond à aucune des constantes introduites par `case`.

Mais en réalité, si la valeur de l'expression correspond à la constante du cas i, tous les blocs d'instructions des cas suivant le cas i (ici, tous les cas de i à N plus le cas de défaut) sont alors exécutés successivement. Il s'agit donc en réalité d'un branchement en séquence au niveau du cas i.

Si on veut que seul le bloc d'instructions associé au cas i soit exécuté, il faut ajouter l'instruction `break` à la fin du bloc d'instructions i, pour forcer la sortie de la boucle du `switch`. En fait, l'usage systématique du `break` est recommandé à la fin de chaque bloc d'instructions. Mais on groupera aussi parfois des cas ayant un traitement analogues, en les plaçant successivement, avec des blocs d'instructions vides.

Exemple

On suppose que les procédures `sortir`, `continuer`, `recommencer` et `mémoriser` sont définies dans d'autres fichiers.

```
# include <stdio.h>
int main () {
        char rep ;

        printf("Pour sortir, tapez x\n");
        printf("Pour continuer le jeu, tapez c\n");
        printf("Pour recommencer le jeu, tapez r\n");
        printf("Pour mémoriser l''état du jeu,tapez m ou e\n");
        scanf("%c", &rep);
        switch (rep){
            case 'x' :  printf("Sortir du jeu... ");
                        sortir();
                        break;
            case 'e' :
            case 'm' :  printf("Mémoriser l'état du jeu... ");
                        memoriser();
                        break;
            case 'c' :  printf("Continuer à jouer... ");
                        continuer();
                        break;
            case 'r' :  printf("Recommencer à jouer... ");
                        recommencer();
```

```
                        break;
            default :   printf("Réponse incorrecte \n");
        }
}
```

Cette structure conditionnelle gagne à être utilisée lorsque l'on a plusieurs cas à distinguer. Combinée avec la définition de constantes aux noms judicieusement choisis (par #define …) elle permet d'augmenter beaucoup la compréhension du programme. Il est évident que dans l'exemple ci-dessus, l'utilisation de structures `if-then-else` enchâssées serait moins lisible.

Exemple

```
# include <stdio.h>
int main () {

      char rep ;

      printf("Pour sortir, tapez x\n");
      printf("Pour continuer le jeu, tapez c\n");
      printf("Pour recommencer le jeu, tapez r\n");
      printf("Pour mémoriser l''état du jeu,tapez m\n");

      scanf("%c", &rep);
      if (rep=='x') {
          printf("Sortir du jeu... ");
          sortir();
      }
      else
          if ((rep=='e') || (rep=='m')) {
              printf("Mémoriser l'état du jeu... ");
              memoriser();
          }
          else
              if (rep=='c') {
                  printf("Continuer à jouer... ");
                  continuer();
              }
              else
                  if (rep=='r') {
                      printf("Recommencer à jouer... ");
                      recommencer();
                  }
                  else
                      printf("Réponse incorrecte.");

}
```

Remarque : Pour l'indentation de plusieurs `if–then-else` successifs, on peut garder sans ambiguïté le même niveau d'alinéa. Ainsi l'exemple précédent devient en réalité plus agréable s'il est indenté de la façon suivante :

```
      if (rep=='x') {
          printf("Sortir du jeu... ");
          sortir();
      }
      else if ((rep=='e') || (rep=='m')) {
```

```
        printf("Mémoriser l'état du jeu... ");
        memoriser();
    }
    else if (rep=='c') {
        printf("Continuer à jouer... ");
        continuer();
    }
    else if (rep=='r') {
        printf("Recommencer à jouer... ");
        recommencer();
    }
    else
        printf("Réponse incorrecte.");
```

4.3 Les imbrications

De même qu'il est possible d'imbriquer une structure `if-then-else` à l'intérieur d'un bloc d'instructions d'une structure `if-then-else`, toute structure de contrôle (boucle ou instruction conditionnelle) peut être imbriquée dans une autre.

5 Les tableaux

Tous les types que l'on a rencontrés jusqu'à présent étaient des types simples (`int`, `char`, `float`, `double`). Or en programmation, il arrive souvent que l'on ait à traiter des données qui forment un groupe homogène. Si on mémorise ces données dans des variables ayant toutes des noms différents, on se rendra vite compte que leur traitement est difficile voire impossible. Par exemple, s'il s'agit de lire une suite de 1000 réels pour les afficher dans l'ordre inverse, il faut déclarer 1000 variables de type `float` dans le programme. En outre, si on ne connaît pas à l'avance le nombre de données à lire, on ne pourra pas se contenter de déclarer une liste de variables distinctes. C'est pour ces raisons que les langages impératifs offrent la possibilité de définir des *types structurés* de données. Un type structuré est un assemblage de types simples, définis par le programmeur ou prédéfinis.

Nous nous intéressons dans ce chapitre à un type structuré particulier : le type *tableau*. Le type *tableau* désigne une collection indicée (c'est-à-dire ordonnée) de variables, appelées cellules, ayant toutes le même type.

5.1 Les tableaux à une dimension

Une variable de type tableau à une dimension est un *nombre fixé de variables ordonnées du même type, repérées par un indice simple.*

Pour définir une variable, dont l'identificateur est `tab`, de type tableau à une dimension, constitué de N cellules de type `<id type>`, on écrit :

```
<id type> tab[N];
```

N doit désigner une valeur entière connue du compilateur pour qu'il puisse réserver, pour la variable `tab`, une zone mémoire contiguë de taille égale à N fois la taille nécessaire au codage d'une valeur de type `<id type>`. `N` est donc une constante entière. `N` ne peut pas être une variable car la valeur d'une variable n'est connue qu'à l'exécution d'un programme et non en phase de compilation. Par contre N peut être une constante « nommée » précédemment définie en tête de fichier par une instruction du type `#define N 300.`

Pour accéder à une cellule du tableau `tab`, il faut préciser l'indice (ou position) de cette cellule dans le tableau. Cette position pourra être donnée sous la forme d'une expression dont la valeur est un entier :

```
tab[expression]
```

Mais, attention, **en C les cellules d'un tableau de taille N sont indicées de 0 à N-1**. L'évaluation de l'`expression` doit donc retourner un entier positif ou nul strictement inférieur à N. La cellule `tab[expression]` est considérée comme une variable et elle peut être utilisée dans une instruction comme n'importe quelle variable de type `<id type>.`

Exemple

Le programme ci-dessous lit une suite de réels et les affiche ensuite dans l'ordre inverse de celui dans lequel ces réels ont été lus (on suppose que le nombre de réels à lire est supérieur ou égale à 1).

```
#include <stdio.h>
#include <stdlib.h>
#define TAILLE_MAX 1000

int main() {

  char reponse;
  float suite[TAILLE_MAX];
  int i;
      int compt=0;

      do {
      printf("Entrez un réel :");
      scanf("%f", &suite[compt]);
      compt = compt + 1 ;
      printf(" Voulez-vous continuer (o/n) ? ") ;
      scanf("%c",&reponse) ;
  } while ((reponse == 'o') && (compt < TAILLE_MAX));
  for(i=compt-1; i>=0; i=i-1)
      printf("%f\t", suite[i]);
      printf("\n");
      return EXIT_SUCCESS ;
}
```

La variable `suite` est un tableau contenant 1000 cellules ou variables de type float. Les identificateurs de ces cellules sont `suite[indice]` où `indice` variie entre 0 et 999. Ces cellules peuvent figurer dans n'importe quelle expression. L'expression `&suite[compt]` désigne l'adresse de la cellule `suite[compt]`et permet de lire une valeur réelle avec `scanf` pour initialiser la cellule.

Simulation :
Début du do-while
Ecriture de « Entrez un nombre réel : »
Lecture de suite[0] ; {suite[0]=6.8 }
{compt=1}
Ecriture de « Voulez-vous continuer (o/n) ? »
Lecture de reponse ; {reponse='o'}
((reponse=='o')=vrai) Suite du do-while
Ecriture de « Entrez un nombre réel : »
Lecture de suite[1] ; {suite[1]=3.21}
{compt=2}
Ecriture de « Voulez-vous continuer (o/n) ? »
Lecture de reponse ; {reponse='n'}
((reponse=='o')=faux) Fin du do-while
{i=1 ;(i>=0)=vrai)} Début for
{suite[1]=3.21} Ecriture de 3.21
{i=0 ;(i>=0)=vrai)} Suite for
{suite[0]=6.8} Ecriture de 6.8
{i=-1 ; i>=0)=faux)} Fin for

Dans l'exécution du programme ci-dessus, les cellules de la variable suite sont initialisées au fur et à mesure des passages dans la boucle `do-while`. Seul le premier passage est garanti, et

les cellules initialisées ne le sont que pour les indices de 0 à `compt-1` en sortie de boucle. C'est pourquoi seuls les indices correspondants seront affichés avec la boucle `for` suivante.

Il est également possible d'initialiser une variable de tableau lors de sa définition comme suit

```
<id type> tab[N]={val1,val2,...,valN};
```

pourvu que le nombre des valeurs introduites n'excède pas la taille N du tableau. A la suite de cette définition, on a :

tab[0]=val1, tab[1]=val2, …, tab[N-1]=valN

Remarque: Si le nombre des valeurs introduites est plus petit que N, les dernières cellules seront initialisées à 0.

En outre, on peut ne pas spécifier la taille N initiale du tableau et l'initialiser lors de sa définition comme suit:

```
<id type> tab[]={val1,val2,..., valP};
```

Dans ce cas, le tableau aura exactement exactement la taille P correspondant au nombre de valeurs introduites dans sa définition (on pourra donc accéder aux cellules `tab[j]` avec `j` variant entre 0 et P-1).

5.2 Les tableaux à deux dimensions

Une variable de type tableau à deux dimensions est *un nombre fixé de variables ordonnées du même type, repérées par deux indices.*

Pour définir une variable `mat` (comme matrice), de type tableau à deux dimensions, constitué de M×N cellules d'un type nommé <id_type>, on écrit :

```
<id type> mat[M][N];
```

M et N doivent être des constantes entières. On peut se représenter cette variable `mat` comme une matrice :

mat	0	1	…	N-1
0	mat[0][0]	mat[0][1]		mat[0][N-1]
1	mat[1][0]	mat[1][1]		mat[1][N-1]
…				
M-1	mat[M-1][0]	mat[M-1][1]		mat[M-1][N-1]

ou comme une suite de M tableaux (à une dimension) de taille N :

dim 1	0				1				M-1			
dim 2	0	1	…	N-1	0	…	N-1	…	0	1	…	N-1
mat	[0][0]	[0][1]	…	[0][N-1]	[1][0]	…	[1][N-1]	…	[M-1][0]	[M-1][1]	…	[M-1][N-1]

Cette dernière représentation correspond mieux à la représentation effective d'un tableau en mémoire centrale.

La cellule située en jème position du ième tableau de taille N de la variable `mat` est une variable de type <`id type`> dont l'identificateur est : `mat[i][j]` (et on a : i et j positifs et inférieurs respectivement à M-1 et N-1).

Exemple 1 :

La variable `mat` est utilisée ici pour représenter la matrice carrée Identité de taille 10×10.

```
#include <stdio.h>
#include <stdlib.h>
#define TAILLE 10

int main() {
  int mat[TAILLE][TAILLE];
  int dim1,dim2;

  for(dim1=0; dim1<TAILLE; dim1 = dim1+1) {
      for(dim2=0; dim2<TAILLE; dim2 = dim2+1)
          if(dim1==dim2)
                mat[dim1][dim2]=1;
          else
                mat[dim1][dim2]=0;
  }
  for(dim1=0; dim1<TAILLE; dim1 = dim1+1) {
      for(dim2=0; dim2<TAILLE; dim2 = dim2+1)
          printf("%d\t",mat[dim1][dim2]);
      printf("\n");
  }
      return EXIT_SUCCESS ;
}
```

Exemple 2 :

Le produit `matProd` de deux matrices, `mat1` et `mat2`, de taille respectives m1×n1 et m2×n2, s'obtient avec le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
#define MAXDIM 10

int main() {

      int mat1 [MAXDIM] [MAXDIM];
      int mat2 [MAXDIM] [MAXDIM];
      int matProd [MAXDIM] [MAXDIM];
      int i,j,k,m1,n1,m2,n2;

/*initialisation des matrices mat1 et mat2
      après lecture de m1, m2, n1 et n2 <= MAXDIM */
      printf("entrez m1 n1 (dim. de la première matrice) : ") ;
      scanf("%d%d", &m1, &n1) ;
      if ((m1 > MAXDIM) || (n1 > MAXDIM)) {
        printf("taille maximum = %d", MAXDIM);
        return EXIT_ERROR ;
      }
      ...
```

```
/* calcul du produit dans la matrice matProd */
      if(n1 == m2)
        for (i=0; i <m1 ; i=i+1)
        for(j=0; j<n2 ; j=j+1) {
        matProd[i][j]=0;
        for(k=0; k<n1; k=k+1)
          matProd[i][j] = matProd[i][j] +
                                mat1[i][k]*mat2[k][j];
      }
      else
        printf("multiplication impossible\n") ;
      return EXIT_SUCCESS ;
}
```

Remarque : On peut initialiser un tableau d'entiers à deux dimensions lors de sa définition en introduisant les éléments du tableau par lignes séparées par des accolades, par exemple avec

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

ou

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

Il faut cependant prendre garde à ne pas dépasser les bornes des dimensions déclarées du tableau. Par contre, si des lignes ou colonnes sont incomplètes, les termes manquants seront initialisés à 0. Ainsi,

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

initialise la première ligne du tableau à {1, 0, 0} et la seconde à {4, 5, 0}.

5.3 Les tableaux multidimensionnels

On définit une variable de type tableau à `k` dimensions de la façon suivante :

```
<id type> <id variable> [N1][N2]..[Nk];
```

`<id variable>` est une variable de type tableau contenant N1×N2×...×Nk cellules ou variables, les valeurs Ni étant des constantes entières. Chacune des cellules du tableau est une variable de type `<id type>` et peut donc être utilisée comme telle dans n'importe quelle expression. Pour accéder à une cellule du tableau on écrira :

```
 <id variable> [i][j]..[l].
```

5.4 Application : les chaînes de caractères

Une ***chaîne de caractères*** est une suite ordonnée de caractères. Voici un exemple de chaîne de caractères :

Voulez-vous continuer ?

Si une valeur constante de type chaîne de caractères est utilisée dans un programme C, elle est entourée de guillemets :

```
"Voulez-vous continuer ?"
```

Les caractères d'une chaîne ne représentent qu'eux-mêmes et ne sont pas interprétés par le compilateur, à l'inverse des autres caractères d'un programme qui représentent : des nombres, des identificateurs, des mots réservés ou d'autres symboles. C'est ainsi que

La chaîne `"main"` n'a aucun rapport avec le mot-clé `main`

La chaîne `"123"` n'est pas un `nombre`

La chaîne `"+"` n'est pas un `opérateur`

Une chaîne de caractères est représentée en machine par un *tableau de type char* à une dimension. Ainsi, la constante "bonjour", de type chaîne de caractères, est représentée en machine par un tableau unidimensionnel contenant les caractères ci-dessous :

Le caractère nul, noté en C par '\0' est un caractère de contrôle, de code ASCII 0, qui marque la fin de toute chaîne de caractères.

En conséquence, on fera bien attention de distinguer `'c' et "c"` car :

`"c"` est un tableau unidimensionnel de type `char` contenant deux cellules valant respectivement `'c'` et `'\0'`

alors que `'c'` est simplement une valeur de type `char.`

Pour définir une variable de type chaîne de caractères, on définit une variable de type tableau à une dimension de type char. Par exemple, dans la définition ci-dessous,

```
char mot[80];
```

La variable `mot` est une variable de type tableau contenant 80 cellules de type `char`.

Il est possible d'initialiser lors de sa définition un tableau de type `char` comme suit :

```
    char mot[80]={'b','o','n','j','o','u','r','\0'};
⇔
    char mot[80]="bonjour";
```

ou encore

```
    char mot[]={'b','o','n','j','o','u','r','\0'};
⇔
    char mot[]="bonjour";
```

Cependant, le compilateur aura alloué 80 cellules de type `char` avec les deux premières définitions (le tableau mot sera donc de taille 80), alors qu'avec les deux suivantes, il n'aura réservé que l'espace mémoire nécessaire au stockage de la chaîne "bonjour", c'est-à-dire, un tableau de 8 cellules seulement, et la taille de mot sera alors 8.

Il est également possible, une fois la variable `mot` définie, d'accéder et d'initialiser ses différentes cellules comme suit :

```
   mot[0]='b'; mot[1]='o'; mot[2]='n'; mot[3]='j'; mot[4]='o';
   mot[5]='u'; mot[6]='r'; mot[7]='\0';
```

On sait déjà écrire une constante de type chaîne de caractères avec `printf` :

```
printf("Voulez-vous continuer ?");
```

Si l''on souhaite afficher le contenu d'une variable de type chaîne de caractère, on pourra également écrire :
```
printf("%s", mot);
```

où %s désigne une chaîne de caractère (s pour *string* en anglais), c'est-à-dire un tableau de caractères finissant par le caractère nul '\0'.

De même, pour lire une valeur de type chaîne de caractères dans une variable, on pourra utiliser :
```
scanf("%s", mot);
```

Remarques :

1. Pour lire une chaîne de caractères dans une variable de type tableau à une dimension de type char, on donne à la fonction `scanf` directement l'identificateur de la variable et non pas son adresse. Nous verrons plus loin pour quelle raison dans la partie du cours sur les pointeurs.

2. Dans la chaîne indiquant le format, %s permet de lire une chaîne de caractères (tapée sans guillemets et ne comportant aucun caractère d'espacement. Les caractères d'espacement servent au contraire ici à séparer la chaîne de caractères d'autres données entrées.

Il faut en outre s'assurer que la valeur lue (le mot constituant la chaîne de caractères) comporte un nombre de caractères strictement inférieur au nombre de cellules de la variable `mot`, qui doit être, rappelons-le, un tableau de `char` dont la taille aura été déclarée. Si la chaîne de caractères lue a une taille supérieure au tableau `mot`, le programme sortira en erreur. Par contre, si la taille de la chaîne lue est inférieure à la taille du tableau mot, le caractère '\0' sera automatiquement inséré dans le tableau `mot` pour marquer la fin des caractères significatifs de la chaîne.

Signalons également la fonction de la librairie standard `sscanf` qui est équivalente à la fonction `scanf(…)` excepté que les caractères entrés sont lus à partir d'une chaîne de caractères donnée en premier argument à la fonction (au lieu d'être lus à partir du clavier). On pourra s'en servir pour modifier une variable de type tableau :

```
char mot[80]= "bonjour";

sscanf("hello", "%s", mot);
```

Il existe également deux autres fonctions permettant de lire ou d'écrire des caractères un à un. Il s'agit des fonctions :

`getchar()` qui lit un caractère saisi au clavier et en retourne la valeur.

`putchar(c)` qui affiche un caractère à l'écran et en retourne la valeur.

Ces fonctions permettent de saisir des lignes de caractères comportant des caractères d'espacement.

Exemple:

Lire une phrase saisie au clavier par un utilisateur pour l'enregistrer dans une variable de type chaîne de caractères, en supprimant des caractères d'espacement (le caractère espace et le caractère tabulation) :

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

int main() {

        char c;
        int i=0;
        char phrase[100];

        printf("Entrez une phrase et \n");
        printf("terminez par une fin de ligne :\n");

        c = getchar();
        while ((c!='\n') && (i<100)){
          if(c!='\t' && c!=' ') {
            phrase[i]=c;
          i=i+1;
          }
          c = getchar();
        }
        phrase[i]='\0' ;
        printf("Les caractères lus (et collés) sont : %s\n",
                 phrase);
        return EXIT_SUCCESS ;
}
```