

Chapter 4

Structures de contrôle

Nous avons vu jusqu'à maintenant des **ordres** (ou **instructions**) **simples** (ou **élémentaires**), indiquant d'effectuer une action très élémentaire. La programmation prend tout son intérêt avec les **structures de contrôle** qui sont de nouveaux types d'ordres permettant d'effectuer des **ordres composés**.

Il y a deux types apparents de structures de contrôle : les *répétitions* (ou *itérations*, ou *boucles*) et les *tests* (ou *alternatives*). Avant d'étudier les structures de contrôle à proprement parler, nous allons voir une première façon de regrouper les instructions (simples ou non), à savoir le *séquençement*, ainsi que les *conditions* qui interviennent dans les structures de contrôle.

4.1 Le séquençement

Introduction.- La première méthode permettant de composer des ordres élémentaires est d'effectuer un premier tel ordre, puis un second, puis un troisième, etc. Ceci est tellement évident qu'il ne nous apparaît pas, bien souvent, comme une composition d'ordres simples. Ceci en est pourtant une, certainement la plus élémentaire de toutes, appelée le **séquençement**.

Syntaxe du séquençement en langage C.- Si on veut un séquençement des instructions `INS_1`, `INS_2` et `INS_3`, par exemple, on écrit un **bloc** :

```
{
INS_1 ;
INS_2 ;
INS_3 ;
}
```

en indiquant les instructions dans l'ordre dans lequel on désire les voir effectuer et en les entourant par une accolade ouvrante '{' pour indiquer le début du séquençement et une accolade fermante '}' pour indiquer la fin du séquençement.

Remarques.- 1°) Dans une première étape on fait des séquençements d'instructions élémentaires telles que les affectations et les ordres de lecture et d'écriture. Mais on peut, bien sûr, faire des séquençements de toutes sortes d'instructions.

2°) Les accolades sont des marqueurs qui sont indispensables lorsqu'un séquençement sera placé dans certaines instructions ci-dessous. Mais ceci n'est pas toujours le cas, en particulier si le corps du programme ne comprend qu'un séquençement. Il est alors inutile d'indiquer explicitement le séquençement, en particulier la structure :

```
{{ ... }}
```

serait ridicule.

Instruction vide.- Comme on a pris l'habitude de placer des points virgules là où il ne devrait pas y en avoir, le langage C possède la notion d'**instruction vide** qui ne fait rien. Ainsi :

```
{
INS_1 ;
INS_2 ;
INS_3 ; ;
}
```

sera considéré comme une instruction valable : c'est un séquençement de quatre instructions, la dernière instruction étant l'instruction vide.

4.2 Conditions

Les structures de contrôle suivantes vont faire appel à la notion de *condition*. Commençons donc par étudier celle-ci.

4.2.1 Expressions booléennes

4.2.1.1 Booléen

Notion.- Vous avez étudié, en Mathématiques, la *logique propositionnelle* avec la notion fondamentale de *proposition* ne pouvant prendre que l'une des *valeurs de vérité vrai* ou *faux* (tout au

moins en logique classique). Vous avez également vu que la logique propositionnelle est liée à l'*algèbre de Boole*. Cette logique est implémentée en langage C avec un vocabulaire légèrement différent.

Booléens.- Outre les nombres et les textes, on veut en général manipuler dans un langage de programmation les valeurs de vérité, appelées dans ce contexte **valeurs booléennes** ou **booléens**.

Cas du langage C.- Il n'existe pas de type booléen explicite en langage C. Les entiers de type `int` en tiennent lieu : la valeur 0 correspond à la valeur de vérité *faux* et les autres entiers à la valeur de vérité *vrai*.

4.2.1.2 Connecteur logique

Notion.- Vous avez également vu en logique propositionnelle la notion de *connecteur logique* et, en particulier, les connecteurs logiques naturels de *négation* (le *non*), de *conjonction* (le *et*), de *disjonction* (le *ou* inclusif), d'*implication* (le *si ... alors ...*) et d'*équivalence* (le *si, et seulement si*). Vous avez vu aussi que tous les connecteurs logiques peuvent être engendrés à partir de deux d'entre eux bien choisis (par exemple la négation et la conjonction, ou la négation et la disjonction) ou même d'un seul (par exemple le *symbole de Sheffer*).

Cas du langage C.- On n'a retenu que les connecteurs de négation, de conjonction et de disjonction. Si P et Q sont des **expressions booléennes**, c'est-à-dire des expressions dont la valeur est un booléen (donc un `int` en langage C), on obtient la négation de P, la conjonction de P et de Q, et enfin la disjonction de P et de Q en écrivant respectivement :

```
! P
P && Q
P || Q
```

où `||` est formé de deux symboles `|` (on le trouve facilement sur le clavier).

Exercice.- Considérer une expression booléenne assez compliquée, par exemple :

(P et Q et R) ou (P et non Q et non R) ou (non P et non Q).

Écrire un programme C qui demande les valeurs de vérité des variables propositionnelles intervenant dans cette expression et qui affiche la valeur de vérité de cette expression pour ces valeurs-là.

[Les valeurs de vérité `true` et `false` seront lues et affichées comme les entiers respectifs 1 et 0. On a, par exemple, le programme `logique.c` suivant :

```
#include <stdio.h>

void main(void)
{
    int P, Q, R, E;
    printf("P = ");
    scanf("%d",&P);
    printf("Q = ");
    scanf("%d",&Q);
    printf("R = ");
```

```

scanf("%d",&R);
E = (P && Q && R) || (P && !Q && !R) || (!P && !Q);
printf("E = %d", E);
}
]

```

4.2.2 Relations définies en langage C

4.2.2.1 Notions générales

Introduction.- Nous avons rencontré jusqu'à maintenant un premier type d'expressions booléennes correspondant aux formules propositionnelles, c'est-à-dire formées à partir des variables booléennes et des connecteurs logiques. Mais vous avez vu en *logique élémentaire*, plus explicitement en *logique des prédicats*, que l'on peut obtenir de nouvelles expressions booléennes en analysant les propositions elles-mêmes. Les notions de fonctions, de relations et de quantificateurs apparaissent.

Cette notion plus générale d'expression booléenne n'est pas manipulable dans un langage de programmation. Mais l'introduction des relations nous permet d'obtenir des expressions plus complexes.

Problème.- La notion générale d'expression booléenne (plus techniquement de *formule de la logique du premier ordre*) peut-elle être manipulable dans un langage de programmation ?

La réponse est non. On le démontre en *théorie de la calculabilité*, qui n'est pas du programme des toutes premières années des universités.

Relations élémentaires du langage C.- Vous avez vu la notion générale de *relation en théorie (élémentaire) des ensembles*. Quelques relations particulières sont définies en langage C. Toutes ces relations sont des relations binaires. Ces relations sont notées sous **forme infixé**, c'est-à-dire que si a est en relation R avec b alors on note $a R b$, et non sous la **forme suffixé** $R(a, b)$.

4.2.2.2 Relations prédéfinies

Liste.- Les relations prédéfinies en langage C sont la relation d'égalité, la relation de **diséquation** (c'est le terme informatique pour "différent de") et les relations d'inégalité, notées de la façon suivante :

```

==  pour la relation d'égalité,
!=  pour la relation de diséquation,
<   pour strictement plus petit,
<=  pour plus petit ou égal,
>   pour strictement plus grand,
>=  pour plus grand ou égal.

```

Les types permis pour la comparaison sont les types entiers, réels et pointeurs. Les deux opérandes n'ont pas besoin d'être du même type.

Le résultat d'une opération de comparaison est 1 ou 0 suivant que la relation est réalisée ou non.

Exercice 1.- Faire quelques tests, par exemple grâce à :

```
printf("%d", 2 < 3);
printf("\t %d", 23 <= 54);
```

Exercice 2.- Essayer de trouver les limites de l'implémentation de ces relations (par exemple relation entre grands entiers ou entre réels très proches).

4.2.2.3 *Relations définissables

Introduction.- Ces relations prédéfinies permettent d'en définir d'autres, en utilisant les connecteurs logiques d'une part, les fonctions d'autre part.

Exemple.- L'ensemble des valeurs x dont l'image par le polynôme $x^2 - 5x + 6$ est négative peut s'écrire soit :

$$X * X - 5 * X + 6 <= 0$$

soit :

$$(2 <= X) \&\& (X <= 3)$$

(l'équivalence est facile si vous savez résoudre les équations du second degré).

Syntaxe.- Nous ne définirons pas formellement la syntaxe de formation de telles relations pour l'instant. Une utilisation au niveau intuitif devrait être suffisant.

Problème.- Toutes les relations sur les entiers (ou sur les réels) sont-elles définissables de la façon précédente ? Sinon quelles sont les relations définissables ?

Ces questions, et d'autres analogues, font l'objet d'un très beau chapitre d'informatique théorique (la *théorie de la calculabilité*) qui n'est pas abordé dans les toutes premières années de l'université.

4.3 La répétition

Reprenons, pour motiver les structures de contrôle, le problème initial du chapitre précédent, à savoir celui qui consiste à calculer un certain nombre de valeurs de la fonction :

$$f(x) = \frac{\sin x + \ln x}{e^x + 2}.$$

Il peut sembler curieux d'avoir à relancer le programme pour chaque valeur désirée, ce que nous avons fait jusqu'à maintenant. L'ordinateur ne pourrait-il pas faire cela automatiquement ? Bien sûr que si, et pas seulement pour ce problème. C'est ce qu'on appelle la **répétition**, ou l'**itération**, qui s'effectue grâce à une **boucle**. En fait il y a trois façons de mettre en place la répétition en langage C.

4.3.1 Boucle tant que faire

Syntaxe.- La forme de cette instruction est :

```
while (condition) instruction
```

où *instruction* est une instruction (*sic*), appelée le **corps de la boucle**, et où *condition* est une expression booléenne. La partie **while (condition)** est appelée l'**en-tête de la boucle**.

Remarques.- 1°) Rappelons que nous avons déjà vu plusieurs instructions, mais que nous n'avons pas donné de définition générale de ce qu'est une instruction. Nous continuerons pour l'instant à en parler au sens intuitif.

2°) Lorsque *instruction* est une instruction complexe de séquençement, il est important de la commencer par { et de la terminer par }.

Sémantique.- La condition est évaluée. Si elle est fausse alors l'exécution de la boucle est achevée et on passe à l'instruction suivante. Si elle est vraie alors le corps de la boucle est exécutée une fois puis on revient au début de cette instruction (on évalue donc la condition...).

Remarque.- Il est essentiel, du point de vue de la signification mais non de la syntaxe, que le corps de la boucle puisse changer la valeur de la condition, sinon la boucle continuera indéfiniment.

Exemple.- Améliorons le programme concernant notre problème pris en exemple. Décidons, par exemple, qu'une valeur ne nous intéresse pas (disons $x = 1000$) ; nous nous arrêterons si nous introduisons cette valeur comme donnée. Ceci donne lieu au programme `fonct_3.c` suivant :

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

void main(void)
{
    float x, y;
    printf("x = ");
    scanf("%f",&X);
    while (x != 1000.0)
    {
        y = (sin(x) + log(x))/(exp(x) + 2);
        printf("f(%f) = %f", x, y);
        printf("\nx = ");
        scanf("%f",&X);
    }
}
```

Commentaires.- 1°) Remarquons l'obligation d'écrire deux fois les instructions permettant l'entrée des données. La première, précédant l'écriture de la boucle, s'appelle l'**initialisation** de la boucle.

2°) La valeur 1000 est ce qu'on appelle une **valeur signal** ou **sentinelle**.

3°) Une variable telle que *x* dont la valeur conditionne l'exécution des instructions du corps de la boucle, donc contrôle le déroulement d'une structure répétitive, s'appelle **variable de boucle** ou **variable de contrôle de boucle**.

Exercice.- Faites effectuer ce programme pour quelques valeurs en terminant par la valeur signal.

4.3.2 Boucle répéter tant que

Syntaxe.- La forme de cette instruction est :

```
do instruction while (condition) ;
```

où `instruction` est une instruction, encore appelée **corps de la boucle**, et `condition` est une expression booléenne.

Signification.- Le corps de la boucle est effectué au moins une fois puis la condition est évaluée. Si elle est fausse alors l'exécution de la boucle est achevée et on va à l'instruction suivante. Si elle est vraie alors on repart de `do`.

Remarque.- Comme pour le type de boucle précédent, il est essentiel que le corps de la boucle puisse changer la valeur de la condition, sinon la boucle continuera indéfiniment.

Exemple.- Toujours en utilisant la valeur signal $x = 1000$, on peut utiliser le programme `fonct_4.c` suivant pour résoudre notre problème :

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

void main(void)
{
    float x, y;
    printf("x = ");
    scanf("%f",&x);
    do
    {
        y = (sin(x) + log(x))/(exp(x) + 2);
        printf("f(%f) = %f \n", x, y);
        printf("x = ");
        scanf("%f",&x);
    }
    while (x != 1000.0);
}
```

Exercice 1.- *Faites effectuer ce programme pour quelques valeurs en terminant par la valeur signal.*

Exercice 2.- *Ce programme fait-il bien ce que l'on veut si on commence par la valeur signal $x = 1000$?*

4.3.3 Boucle pour

Syntaxe.- La forme de cette instruction est :

```
for (expression_I ; expression_C ; expression_R ) instruction
```

où :

- `expression_I` est une instruction initialisant la (ou les) **variable(s) de contrôle** ;
- `expression_C` est une expression booléenne, appelée **condition de bouclage** ;
- `expression_R` est une expression permettant de réinitialiser la (ou les) variable(s) de contrôle après le passage dans le **corps de la boucle instruction**.

La partie qui précède le corps de la boucle est l'**en-tête de la boucle**.

Exemple.- On a :

```
for (i = 1; i <= 100; i = i+1) sum = sum + i;
```

Sémantique.- La première phase consiste à initialiser les variables de contrôle (et autres variables) grâce à **expression_I**. On teste alors la condition de bouclage. Si elle n'est pas réalisée l'instruction est terminée. Si elle est réalisée on exécute une fois le corps de la boucle, on ré-initialise les variables grâce à **expression_R** puis on recommence en testant la condition de bouclage.

Exemple.- Nous pouvons utiliser ce type de boucle pour résoudre notre problème si nous connaissons *a priori* le nombre de valeurs de x que nous voulons évaluer. Par exemple pour calculer dix valeurs de $f(x)$ nous pouvons utiliser le programme `fonct_5.c` suivant :

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

void main(void)
{
    float x, y;
    int i;
    for (i=1 ; i <= 10 ; i = i+1)
    {
        printf("x = ");
        scanf("%f",&x);
        y = (sin(x) + log(x))/(exp(x) + 2);
        printf("f(%f) = %f \n", x, y);
    }
}
```

Exercice 1.- *Faites exécuter ce programme.*

Commentaire.- Dans le programme précédent la variable de contrôle i n'intervient pas dans le corps de la boucle, ce qui ne sera pas le cas en général, comme le montre l'exercice suivant.

Exercice 2.- *Écrire un programme C qui demande un entier naturel N comme donnée et retourne la somme des entiers naturels de 0 à cet entier, chacun à la puissance cinq, c'est-à-dire $0^5 + 1^5 + \dots + N^5$.*

[On n'utilisera pas la formule donnant immédiatement le résultat, à supposer que vous la connaissiez. Ceci donne lieu au programme `somme.c` suivant :

```
#include <stdio.h>

void main(void)
{
    int i, N, S;
    printf("N = ");
    scanf("%d",&N);
    S = 0;
```

```

    for (i = 1; i <= N; i = i + 1)
        S = S + i*i*i*i*i;
    printf("S = %d.", S);
}
]

```

Commentaire.- Pour une bonne utilisation d'une boucle POUR, la valeur de I ne doit pas être changée dans le corps de la boucle (par exemple I ne doit pas apparaître dans le membre de gauche d'une affectation).

Exercice 3.- Concevoir un programme utilisant une boucle POUR dont le corps de la boucle contient une affectation avec I comme membre gauche. En donner le comportement a priori. Le faire exécuter et observer son comportement réel.

4.3.4 Choix du type de boucle

Si on veut être sûr que le programme termine il faut essayer d'utiliser le troisième type de boucle, la boucle Pour. Mais ceci n'est pas toujours possible. Entre les deux premiers types c'est souvent une question de goût.

En fait, dans tous les langages de *programmation structurée*, il suffit de l'un des deux premiers types de boucles pour obtenir les deux autres (si on dispose de l'*alternative* que nous verrons dans la section suivante). En langage C la sémantique de la boucle Pour est telle qu'il suffit même de ce seul type de boucle.

4.4 Le test et l'alternative

4.4.1 Le test

Introduction.- Si nous reprenons l'un de nos programmes de calcul de la fonction habituelle et que, à l'appel de x , nous répondions, par exemple, 0 alors il y aura un problème à cause de l'ensemble de définition du logarithme. Nous pouvons améliorer notre programme en ne permettant plus cette évaluation grâce à un **test**, c'est-à-dire que nous ne ferons rien si l'argument est négatif.

Syntaxe.- Cette instruction a la forme suivante :

```
if (condition) instruction
```

où **condition** est une expression booléenne et **instruction** une instruction.

Sémantique.- La condition est évaluée.

Si elle est vraie alors l'instruction **instruction** est effectuée puis on passe à l'instruction suivante ; si elle est fautive on passe directement à l'instruction suivante.

Exemple.- Avec notre problème de calcul de la valeur de notre fonction habituelle, nous pouvons utiliser le programme `fonct_6.c` suivant (en utilisant la boucle tant que) :

```

#include <stdio.h>
#include <math.h>

void main(void)
{

```

```

float x, y;
printf("x = ");
scanf("%f",&x);
while (x != 1000.0)
{
    if (x > 0)
    {
        y = (sin(x) + log(x))/(exp(x) + 2);
        printf("f(%f) = %f\n", x, y);
    }
    printf("x = ");
    scanf("%f",&x);
}

```

Exercice.- Faites exécuter ce programme en introduisant plusieurs valeurs de x , de façon à ce que les deux cas soient couverts.

4.4.2 L'alternative

Introduction.- Nous venons de voir comment éviter une erreur d'exécution si nous reprenons l'un de nos programmes de calcul de la fonction habituelle et que, à l'appel de x , nous répondions, par exemple, 0. Nous pouvons encore améliorer ce programme en indiquant explicitement qu'il y a un problème lors d'un argument négatif (tout en donnant la valeur lors d'un argument strictement positif). Ceci est réalisé grâce à l'**alternative**.

Syntaxe.- Cette instruction a la forme suivante :

```
if (condition) instruction1 else instruction2
```

où **condition** est une expression booléenne et **instruction1** et **instruction2** des instructions (préfixées et suffixées éventuellement par { et }, respectivement, pour bien les délimiter si c'est un séquençement).

Sémantique.- La condition est évaluée.

Si elle est vraie alors l'instruction **instruction1** est effectuée puis on passe à l'instruction suivante ; si elle est fautive alors l'instruction **instruction2** est effectuée puis on passe à l'instruction suivante.

Exemple.- Avec notre programme de calcul de la valeur de notre fonction habituelle, nous pouvons utiliser le programme `fonct_7.c` suivant (en utilisant la boucle tant que) :

```

#include <stdio.h>
#include <math.h>

void main(void)
{
    float x, y;
    printf("x = ");
    scanf("%f",&x);
    while (x != 1000.0)
    {

```

```

if (x <= 0)
    printf("f(%f) non definie\n", x);
else
    {
        y = (sin(x) + log(x))/(exp(x) + 2);
        printf("f(%f) = %f\n", x, y);
    }
printf("x = ");
scanf("%f",&x);
}
}

```

Exercice.- *Faites exécuter ce programme en introduisant plusieurs valeurs de x, de façon à ce que les deux cas soient couverts.*

4.4.3 Réduction du nombre d'instructions primitives

Introduction.- En fait il suffit du test ou de l'alternative comme instructions primitives, l'autre instruction pouvant s'obtenir à partir de celle qui est retenue.

Exercice 1.- *Émuler l'alternative avec le test.*

[L'instruction :

```
if (condition) instruction1 else instruction2
```

est obtenue par :

```
if (condition) instruction1
if !(condition) instruction2
```

c'est-à-dire par un séquençement de deux tests.]

Exercice 2.- *Émuler le test avec l'alternative.*

[L'instruction :

```
if (condition) instruction
```

est obtenue par :

```
if (condition) instruction else;
```

autrement dit *instruction2* est l'instruction vide.]

Exercice 3.- *Imaginons que nous ne disposions que d'un seul type de boucle parmi les deux premiers rencontrés ci-dessus. Montrer comment on peut simuler les deux autres types de boucles.*

4.5 Retour sur l'affectation

Introduction.- Pour rendre les expressions de réinitialisation des boucles POUR plus concises on utilise un jeu d'abréviations pour les affectations en langage C. Par exemple l'affectation souvent rencontrée $i = i + 1$ se notera plus simplement $i++$.

Opérateurs d'affectation.- En fait on n'a pas une seule affectation mais plusieurs *opérateurs d'affectation*. Les **opérateurs d'affectation** sont des opérateurs binaires écrits sous forme infixe qui mettent dans leur opérande de gauche la valeur de leur opérande de droite. L'opérande de gauche doit être une **Lvalue** (pour *valeur à gauche du signe d'affectation* évidemment) : il s'agit d'une expression désignant une adresse dans la mémoire de l'ordinateur, par exemple le nom d'une variable (mais pas une constante). L'opérande de droite peut être n'importe quelle expression. En fait certains opérateurs d'affectation seront unaires mais représentent en fait des notations abrégées d'opérations binaires.

Affectations en chaîne.- On peut écrire en C des affectations en chaîne telle que l'affectation suivante :

```
a = b = c = 0;
```

qui affecte la valeur 0 aux trois variables a, b et c.

Affectations combinées.- Une **affectation combinée** mélange une opération d'affectation avec une opération arithmétique (ou de bits). Dans le tableau suivant les expressions de la colonne de gauche équivalent aux expressions de la colonne de droite :

```
x += y   x = x + y
x -= y   x = x - y
x *= y   x = x*y
x /= y   x = x/y
x %= y   x = x%y
```

Incrémement et décrémentation.- Il est fréquent qu'on doive augmenter ou diminuer de 1 une variable (entière), en particulier à propos des boucles POUR. On peut le faire grâce à $x = x+1$ ou à $x += 1$, mais on peut encore faire plus bref. On écrit $x++$ ou $++x$ (resp. $x--$ ou $--x$) pour augmenter (resp. diminuer) la valeur de x de 1.

Remarque.- Il y a une différence entre $x++$ et $++x$. Lorsque l'une de ces expressions est utilisée dans une expression complexe, on incrémente x, suivant le cas, avant ou après en avoir tenu compte dans l'expression complexe. Ce genre de raccourci est à éviter, aussi n'insisterons-nous pas dessus.