Cours 2. Les structures de contrôle

Dimitri Galayko

1 Les cycles

Les cycles sont utilisés pour répéter un code plusieurs fois. La possibilité de faire des cycles, ou des boucles, est la principale force de l'informatique. On pourra, ainsi, *programmer* une action fastidieuse et la faire exécuter par un ordinateur qui ne connaîtra pas la fatigue, même si l'action est répétée des millions de fois.

En langage C, il y a 3 possibilités de définir une boucle: à l'aide des mécanismes *for*, *wnile* et *do...while*. On passera en revue les trois mécanismes. On montrera l'utilisation des trois boucles sur l'exemple du calcul d'une factorielle d'un nombre.

1.1 Boucle *while*

Une boucle *while* est une boucle la plus générale, qui s'emploie par défaut. A elle seule, elle suffit à tous les besoins de la programmation. L'utilisation des deux autres boucles permet surtout d'améliorer la lisibilité des programmes.

Une boucle *while* peut se lire ainsi: *"Répéter le corps tant que la condition de poursuite est vraie"* (cf. l'organigramme, fig. 1). En langage C, cela s'écrit:

```
while(<condition de poursuite>)
        bloc représentant le corps de boucle
```

Le bloc appelé "corps de boucle" est le code qui est répété tant que la condition de poursuite est vraie. La condition de poursuite est vérifiée *avant* chaque exécution du corps de boucle; ce dernier est exécuté si et seulement si la condition est vraie. Si la condition est fausse, le programme *sort* de la boucle, et exécute la première commande qui suit le corps de la boucle.

Il y a deux cas extrêmes: 1) si dès la première vérification la condition de poursuite est fausse, le corps de boucle ne s'exécute jamais, 2) si la condition de poursuite n'est jamais invalidée, la boucle s'exécute sans jamais s'arrêter, sauf présence du mot clé *break*, cf. le paragraphe 1.5.

Cet exemple montre l'utilisation de la boucle *while* pour le calcul de la factorielle d'un nombre entier.

```
int i,n;

// on veut calculer la valeur de la factorielle de n (n!), n>0
n=6; // on donne à n une valeur quelconque

i=1; // on initialise la variable i que l'on appellera compteur

f=1; // on initialise la variable dans laquelle on écrira le résultat

// la boucle: on répète tant que la condition i<=n est vraie,
// le corps de boucle contient deux instructions, d'où la nécessité des accolades
while(i<=n){
        f=f*i;
        i=i+1;
}

// Lorsque l'on sort de la boucle, la variable f contient la valeur de la factorielle
   .
```

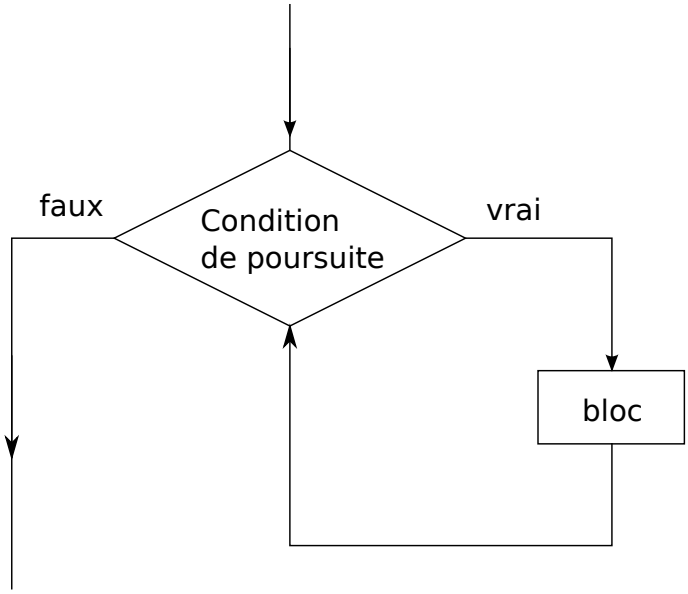

Figure 1: Organigramme de la boucle *while*

1.2 Boucle *do...while*

La boucle *do...while* fonctionne exactement comme la boucle *while*, sauf que la condition de poursuite se vérifie *après* l'exécution du corps de boucle (cf. fig. 2). Ainsi, le corps de boucle est toujours exécuté *au moins une fois*. En langage C, cela s'écrit ainsi:

```
do
        bloc représentant le corps de boucle
while(<condition de poursuite>);
```

Notez le point-virgule obligatoire après la condition de poursuite, et jamais après *do*.

Notre code calculant la factorielle s'écrit maintenant ainsi:

```
int i,n;

// on veut calculer la valeur de la factorielle de n (n!), n>0
n=6; // on donne à n une valeur quelconque

i=1; // on initialise la variable i que l'on appellera compteur

f=1; // on initialise la variable dans laquelle on écrira le résultat

// la boucle: on répète tant que la condition i<=n est vraie,
// la première fois, f=1, i=1, on multiplie 1 par 1. Cela rend valide la première exé
    cution du corps de la boucle dans tous les cas.
// le corps de boucle contient deux instructions, d'où la nécessité des accolades
do{
        f=f*i;
        i=i+1;
while(i<=n)

// Lorsque l'on sort de la boucle, la variable f contient la valeur de la factorielle
    .
```

Notez que dans ce cas particulier, il n'y a pas de différence avec la boucle *while*, essentiellement parce que la première exécution du corps de la boucle multiplie 1 par 1, ce qui donne toujours un résultat correct.

Considérons l'exemple suivant. Soit on souhaite demander à l'utilisateur de saisir une valeur soumise aux conditions, par ex., se trouvant dans un intervalle donné. On souhaite redemander tant que l'utilisateur n'a pas saisi une bonne valeur. Il est évident, qu'il faut demander au moins une fois, et éventuellement,

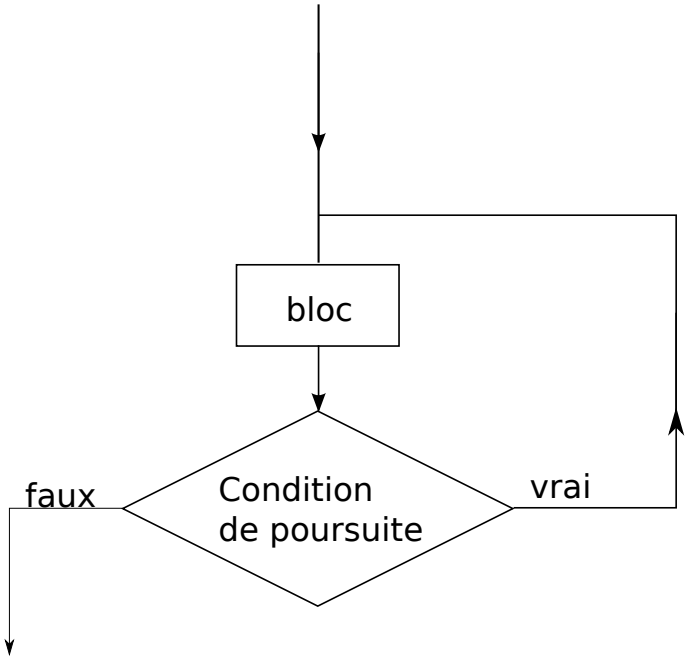

Figure 2: Organigramme de la boucle *do...while*

redemander si la valeur ne remplit pas la condition. L'utilisation de la boucle *do...while* est alors la plus logique:

```
int x;
do{
        printf("Saisissez une valeur entière se trouvant entre 1 et 10:");
        scanf("%d",x)
}while((x>=1) && (x<=10));
```

Ici, tant qu'une bonne valeur n'est pas saisie, l'ordinateur refait la même demande à l'utilisateur. On peut imaginer une interface plus conviviale (annoncer à l'utilisateur qu'il a fait une erreur, etc.). On verra cet exemple dans la suite.

1.3 Boucle *for*

La boucle *for* est surtout utilisée lorsque le nombre d'itérations (de répétitions du corps de la boucle) est connu en avance. On introduit alors un compteur, que l'on initialise, que l'on incrémente après chaque exécution, et que l'on compare avec le nombre d'itérations à faire. En langage C, cela s'écrit comme:

```
for(<instruction d'initialisation>;<condition de poursuite>;<instruction d'ité
    ration>)
 corps de la boucle
```

L'organigramme correspondant est donné fig. 3.

Notez l'utilisation obligatoire des paranthèses après le mot clé *for*, et les points virgules séparant les champs entre les paranthèses.

L'utilisation de la boucle *for* est immédiate et logique pour le calcul de la factorielle, car on connait précisement le nombre des fois à répéter le corps de la boucle:

```
int i,n;

// on veut calculer la valeur de la factorielle de n (n!), n>0
n=6; // on donne à n une valeur quelconque


f=1; // on initialise la variable qui va recevoir le resultat
```

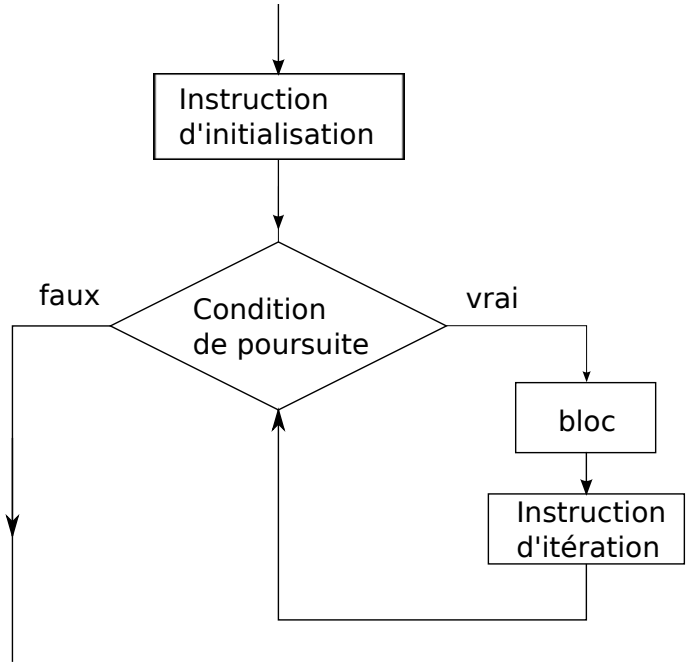

Figure 3: Organigramme de la boucle *for*

```
// Avec la boucle for, le corps de boucle contient une seule instruction, d'où l'
    absence des accolades
for(i=1; i<=n; i=i+1)
        f=f*i;
```

Notez que dans ce programme, i est un compteur: il compte le nombre d'exécution du cycle, et il évolue de 1 à n. Il est initialisé ($i = 1$), il est comparé à n, et il est incrémenté ($i = i + 1$). Ainsi, l'instruction d'itération (celle qui est exécutée après chaque corps de boucle) fait partie de la structure de la boucle, et non plus du corps de boucle. Ce dernier ne contient qu'une seule instruction ($f = f * i$), et les accolades ne sont plus nécessaires.

1.4 Les boucles: le résumé

Lorsque l'on a besoin d'itérer (de répéter) un bout de code, laquelle des trois boucles va-t-on utiliser ?

Pour cela, il faut répondre à la séquence des questions suivantes:

1) Est-ce que l'on connaît le nombre d'itération (le nombre de répétition du corps de la boucle)? Si la réponse est oui, il faut utiliser la boucle *for*.

2) Si la réponse à la question (1) est "non": est-ce que le corps de la boucle doit s'exécuter au moins une fois ? Si la réponse est "oui", alors on utilise la boucle *"do...while"*.

3) Si les réponses aux questions (1) et (2) sont "non", on utilise la boucle *while*.

1.5 Les interruption des boucles

Parfois, il faut modifier l'exécution d'une boucle pendant l'exécution du corps de boucle. Pour cela, il y a deux commandes, *break* et *continue*.

La commande *continue* permet d'interrompre l'exécution du corps de la boucle, et de passer directement à l'évaluation de la condition de poursuite, dont le résultat peut laisser la boucle continuer. Ainsi, la commande *continue* met juste fin à l'exécution de l'itération (corps de la boucle) courante, mais pas de toute la boucle.

La commande *break* permet d'interrompre la boucle, et de passer au code suivant. La commande *break* est souvent utilisée pour mettre fin aux boucles infinies, comme le montre l'exemple suivant. Ici, on demande l'utilisateur de saisir un nombre entier entre 1 et 10, et s'il se trompe, on lui indique son erreur et on redemande:

```
int x;
        printf("Saisissez une valeur entière se trouvant entre 1 et 10:");
while(1){ // la condition de poursuite est toujours vraie (1)
        scanf("%d", &x);

        if ((x>=1) && (x<=10)) // la condition qui vérifie la validité de la valeur
            saisie
                break; // si la valeur est bonne, on sort de la boucle
        else
                // sinon, on dit à l'utilisateur qu'il a fait une erreur
                // et on re-exécute la boucle
                printf("La valeur saisie doit être entre 1 et 10. Essayez de nouveau:
                    ");
}
printf("La valeur saisie est %d\n", x);
```

1.6 Boucles multiples

Il est important de savoir et de comprendre que les structures *while*, *do...while* et *for* sont des instructions. Par conséquent, à elles seules elles forment un bloc, qui peut être inclus dans n'importe quelle structure admettant un bloc.

Ainsi, une structure du type

```
while(x>1)
        while(a!=4)
                bloc
```

est parfaitement admise. Le bloc est exécuté tant que a n'est pas égale à 4, ensuite, si x est plus grand que 1, le bloc de nouveau sera exécuté tant que a n'est pas égale à 4, etc.

La structure suivante est utilisée très souvent lorsqu'il s'agit de balayer les coordonnées d'un plan à deux dimensions, par exemple, les coordonnées des pixels d'une zone d'affichage, délimitées par les coordonnées x allant de $x0$ à $x1$, et des coordonnées y allant de $y0$ à $y1$. Par exemple, on veut dessiner un point sur tous les pixels dans la zone.

```
for (x=x0; x<=x1; x=x+1)
        for (y=y0; y<=y1; y=y+1)
                afficher_point(x,y); // la commande qui affiche un point à (x,y)
```

Notez qu'ici pour chaque valeur de x de $x0$ à $x1$, tous les y de $y0$ à $y1$ seront balayés. On affiche, ainsi, ligne horisontale par ligne horisontale. Si on permutait les lignes *for*, on dessinerait alors des lignes verticales une après l'autre. Notez l'absence d'accolades dans ce bout de code.

2 Bifurcations : *if...else*

L'instruction *if...else* permet de réaliser une bifurcation de l'exécution d'un programme: selon la validité d'une condition, l'un ou l'autre code est exécuté.

Le syntaxe est le suivant:

```
if (condition)
        bloc1
else
        bloc2
```

Ainsi, si *condition* est vraie, *bloc1* est exécuté, sinon, *bloc2* est exécuté. Notez que la partie *else...* est optionnelle: une structure de ce type est également possible:

```
if (condition)
        bloc1
```

Ici, *bloc1* est exécuté si la condition est vraie. Sinon, le programme exécute l'instruction suivante (*bloc1* est ignoré).

Voici un exemple d'utilisation de l'instruction if. Soit on souhaite calculer la valeur de la fonction

$$f(x) = \begin{cases} 0, & x \leq 0 \\ x^2, & sinon \end{cases} \quad (1)$$

```
double x,f;
x=5; // x a une valeur quelconque préalablement définie
if (x<=0)
        f=0;
else
        f=x*x;
```

Notez que dans cette configuration, les blocs de code vers lesquels on bifurque sont composés d'une seule instruction ($f = 0$ ou $f = x * x$). Ainsi, on n'a pas besoin d'utiliser les accolades.

La construction *if...else* est une instruction: cela veut dire qu'elle est également un bloc élémentaire. Ainsi, elle peut s'insérer partout où un *bloc* est admis, y compris dans une structure de *if...else* ou dans les boucles. On autorise ainsi des *aiguillages*: des structures où un programme peut prendre plusieurs chemins, en fonction de la validité de plusieurs conditions.

Exemple: afficher en *caractères* la valeur de la variable entière x qui peut prendre des valeurs de 1 à 5.

On écrit:

```
if (x==1)
        printf("x vaut un\n");
if (x==2)
        printf("x vaut deux\n");
if (x==3)
        printf("x vaut trois\n");
if (x==4)
        printf("x vaut quatre\n");
if (x==5)
        printf("x vaut cinq\n");
```

On peut également écrire d'une manière plus complexe:

```
if (x==1)
        printf("x vaut un\n");
else
        if (x==2)
                printf("x vaut deux\n");
        else
                if (x==3)
                        printf("x vaut trois\n");
                else
                        if (x==4)
                                printf("x vaut quatre\n");
                        else
                                if (x==5)
                                        printf("x vaut cinq\n");
```

Paradoxalement, le dernier code ne représente qu'une seule instruction: *if...else* qui inclut dans son corps tout le reste. C'est le bloc de *else* qui est le plus long ici. Notez l'absence des accolades, et notez la manière d'indenter (insérer des espaces avant le code) qui rend le programme lisible. Les lignes décalées à droite font partie des blocs qui entrent dans les instructions commençant sur les positions plus à gauche. Ainsi, le décalage à droite montre le niveau d'inclusion. Notez également que cette indentation n'est pas obligatoire (on peut justifier à gauche toutes les lignes), mais un tel programme serait illisible.