Eléments d'Informatique

Cours 5 – Instructions de contrôle : boucles et branchements

Catherine Recanati

UNIVERSITÉ PARIS 13 NORD

Plan général

- **Représentation des nombres. Notion de variable.**
- **Programme. Expressions.**
- **Architecture des ordinateurs: langage machine, langage assembleur, AMIL.**
- **Systèmes d'exploitation : fichiers, processus, compilation.**
- **Instructions de contrôle: boucles et branchements.**
- Programme, définition de fonction, appel fonctionnel.
- Tableaux de variables et fonctions d'arguments de type tableau.
- Sens d'un programme, pile d'exécution, compilation.
- Pointeurs et tableaux.
- Chaines de caractères, bibliothèque <string.h>.
- Allocation dynamique, liste chaînées.
- Révisions.

- Cours 5 -
Instructions de contrôle : boucles et branchements

- Instructions de contrôle
- Boucles while
- Boucle for
- Branchements

Plan

Instructions de contrôle

Nous avons introduit le `if` en cours, et le `while` en TD. Le `if` et le `while` sont des instructions appelées « instructions de contrôle ».

Il y a plusieurs types d'instructions. Les expressions sont des instructions qui retournent une **valeur**, et les instructions de contrôle sont des instructions qui ne retournent pas de valeur, mais qui **contrôlent l'ordre d'exécution** des instructions (par défaut séquentiel).

Boucles

Une boucle est une instruction qui **permet** de **répéter les mêmes instructions plusieurs fois** de suite.

Il y a trois types de boucles en C: le `while` (tant que), le `do` (faire) et le `for` (pour).

Le schéma suivi est toujours le même:

Les instructions de la boucle sont exécutées successivement les unes à la suite des autres, et, à la fin, on repart au début de la boucle.

Boucle while

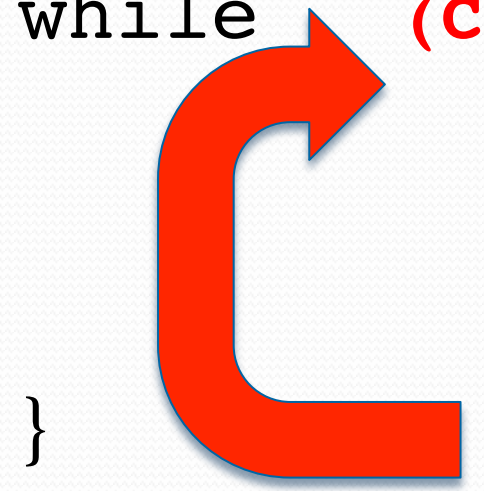
La boucle `while` (tant que) ne peut être exécutée que si une certaine condition est réalisée. La condition s'écrit sous la forme d'une *expression logique*, (ou expression booléenne) qui retourne une valeur.

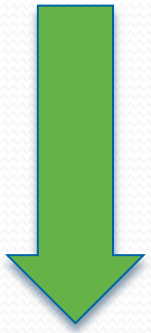
Cette valeur doit être non nulle (=VRAI) pour que l'on puisse exécuter séquentiellement les instructions de la boucle .

Quand la dernière instruction est exécutée, on repart du début du `while` pour tester à nouveau la condition.

Boucle `while`

```
while (Condition VRAIE ?) {
        instruction ;
        instruction ;
        instruction ;
}
```





```
/*  ici, la condition est FAUSSE */
/* et on est sorti du while */
```

« Tant que la condition est VRAIE, exécuter le bloc (une nouvelle fois) ».

Remarques: le bloc d'instruction peut n'avoir qu'une seule instruction. Dans ce cas, les accolades sont facultatives, mais, si on les supprime, on doit mettre un point-virgule pour marquer la fin du `while`.

```
while (Condition)
        instruction ;
/*  ici, on est sorti du while: Condition == FAUX  */
```

Le bloc peut même être vide, et dans ce cas, on aura :

```
while (Condition)
        ;
```

Boucle while

Si la condition est fausse (`==0`), on ignore les instructions du bloc et on sort du `while`.

Et si la condition est fausse dès le début, on ne pénètre jamais dans le bloc, et le bloc n'est jamais exécuté.

Mais une autre instruction, la boucle `do`, permet d'exécuter toujours au moins une fois le bloc des instructions à répéter.

Boucle do

```
do
{
        instruction ;
        instruction ;
        instruction ;
} while (Condition VRAIE ?)
```

« Répéter les instructions du bloc entre accolades tant que la condition est VRAIE ».

Boucle do

Remarque 1: Une boucle do sera exécutée toujours au moins une fois, car la condition d'entrée n'est testée qu'à la fin du premier tour de boucle.

Remarque 2: La boucle do n'est en réalité que peu utilisée et vous pouvez l'ignorer.

On peut en effet se passer de l'instruction do, car on peut toujours réécrire le code d'un do avec un while :

```
do
{
   bloc d'instr;
} while (condition);
```

```
{
    bloc d'instr;
}
while (condition)
{
    bloc d'instr;
}
```

Exemples de `while`

- Lecture d'un caractère c de type char

```
do {
     printf("Entrez une lettre:\n");
     scanf( "%c", &c);
} while ((c < 'a') || (c > 'z'));
```

- Affichage d'un compteur de tour

```
while (i < 100) {
    printf( "compteur = %i", i);
    i = i + 1;
}
```

Ecritures condensées

- ```
  while (i < 100) {
      printf( "compteur = %d", i);
      i++ ; /*  i++ signifie i = i+1  */
  }
  ```
- ```
  while (i++ < 100)
      printf("compteur = %d", i);
  ```
- ```
  do {
      printf("Entrez une lettre:\n");
      scanf("%c", &c);
  } while ((c < 'a') || (c > 'z'));
  ```
  ```

Boucle for

Le `for` permet d'exécuter « Pour i variant de 1 à `N`, répéter l'instruction » :

```
for (int i = 1; i <= N; i++)
    instruction;
```

De même, « Pour i décroissant de `N` à 1, répéter l'instruction »

```
for (int i = N; i > 0 ; i--)
    instruction;
```

En réalité, la sémantique du `for` est plus générale et permet d'autres traitements.

Boucle for

La définition syntaxique, en notation BNF:

```
for ([for-init]; [expr-log] ; [expr])
        instr
```

`for-init` peut être une déclaration de variable, une expression, une déclaration initialisation comme `int i=1`. Elle n'est exécuté qu'une fois, au début.

`expr-log` est une expression logique exprimant la condition d'entrée dans la boucle. Sa valeur doit être non nulle (= VRAI) pour entrer dans la boucle.

Enfin, la dernière expression `expr` du `for` sera exécutée en fin de tour de boucle.

```
for ([for-init]; [expr-log] ;[expr])
```

0 1 3

```
instr
```

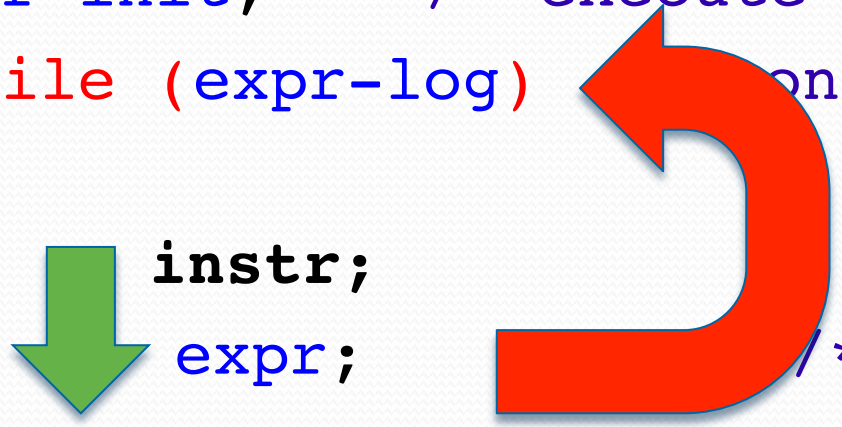
2

Remarque: l'instruction `instr` de la boucle peut être remplacée par un bloc (entre accolades) d'instructions successives séparées par des points-virgules.

Le `for` est très proche du `while`. On peut en effet ré-écrire un `for` avec un `while` :

```
for ([for-init]; [expr-log] ;[expr])
        instr;
```

```
for-init;     /* exécuté une fois */
while (expr-log)      ondition */
{
    instr;
    expr;             /* rajouté */
}
```



Exemples de for

On peut écrire les exemples de `while` rencontrés précédemment avec un for :

```
for (int i=0; (i < 100) ; i++)
    printf( "compteur = %i", i);
```

```
for ( ;((c >= 'a') && (c <= 'z')); )
  scanf("Entrez une lettre:%c", &c);
```

On peut aussi écrire une boucle infinie :

```
for ( ; ; )
  instruction;
```

Instruction `break`

L'instruction `break` est une instruction qui permet de sortir d'une d'une boucle `for`, `while` ou `do` en cours d'exécution.

```
for (i = 0; i < N; i = i + 1)
{
   printf("i = %d", i);
   if (i%5 == 0)
         break;
}
```

Instruction `continue`

L'instruction `continue` est une interruption partielle dans une boucle `for`, `while` ou `do`. Le tour de boucle est interrompu, mais au lieu de sortir de l'instruction `for`, `while` ou `do` englobante, on continue d'enchaîner les répétitions en passant au tour de boucle suivant.

```
for (i = 0  ; i < 5;  i = i + 1)  {
   if ( i%2 == 0 )
         continue;
   printf( "i vaut %d\n", i);
}
```

Branchements

Les branchements sont des instructions de contrôle qui modifient l'ordre d'exécution des instructions. On a rencontré en C, le branchement conditionnel `if`.

Remarque: mentionnons ici au passage une expression réalisant une sorte de if, l'expression conditionnelle du C. En notation BNF :

```
expr-cond ::=   expr0 ?   expr1 :  expr2
```

Exemple: `( x < 0 ? -x : x )`. Cette expression retourne la valeur absolue de `x` car elle retourne la valeur de `expr1 (-x)` ou de `expr2 (x)`, selon la valeur booléenne de `expr0`.

Le switch

Le switch est un branchement un peu complexe. Exemple :

```
switch ( nb ) {
            case 1:   printf("un");
                      break;
            case 2:   printf("deux");
                      break;
            case 3:   printf("trois");
                      break;
            default:  printf("erreur");
}
```

Le switch

Le `switch` est un branchement **sur un bloc d'instructions comportant plusieurs points d'entrée**, appelés `case` (cas). L'expression sur laquelle porte le `switch` est évaluée, et sa valeur est comparée successivement aux différentes constantes introduites par les différents `case`. L'entrée dans le bloc s'effectue au niveau du premier cas d'égalité entre la valeur de l'expression du `switch` et la valeur de la constante du `case`.

S'il n'y a aucun cas d'égalité, on peut forcer l'entrée dans un cas par défaut noté `default`.

Le switch

Le `switch` n'est pas un branchement sur des cas bien séparés – bien qu'on l'utilise très souvent de cette manière (on termine alors chaque liste d'instructions correspondant aux différents `case` par `break`).

Le `switch` est un branchement dans un bloc sur une instruction particulière, introduite par `case`, mais cette instruction fait partie d'un bloc plus large, comportant les instructions de tous les cas successifs mises bout-à-bout.

Pour interrompre l'exécution du bloc, il faut rencontrer l'instruction `break` qui permet de sortir du `switch`.

```
switch ( nb ) {
   case 1:
   case 5:
   case 3:  instruction;
            instruction;
            break;
   case 4:
   case 6:  instruction;
   case 8:  instruction;
            break;
   default:  instruction;
             /* ici break inutile */
}
```

Un `switch` pour traiter des cas de manière identique :

```
switch ( nb ) {
        case 1:
        case 5:
        case 3:   printf("impair");
                  break;
        case 4:
        case 6:
        case 8:   printf("pair");
                  break;
        default:  printf("erreur");
}
```

```
switch (c) {
        case ‘a’:
        case ‘b’:
           ...
        case ‘z’: printf("lettre");
                  break;
        case ‘0’:
        case ‘1’:
           ...
        case ‘9’: printf("chiffre");
                  break;
        default:  printf("erreur");
}
```

Une source fréquente d'erreur, souvent difficile à déceler, est d'oublier d'écrire l'instruction `break` pour quitter le `switch` après le traitement d'un cas.

Nous n'allons pas tout de suite utiliser le `switch`, mais *retenez qu'un point essentiel sera de de ne pas oublier les* `break` *à la fin des différents cas.*

Plan

Merci pour votre attention !

Des questions ?