Algorithmique et programmation – Cours 3. Boucle while, expressions booléennes. Algorithmes élémentaires.

Pierre Boudes

12 septembre 2012

semestre

L'instruction de contrôle while

- Syntaxe
- Trace
- For ou while ?

Expressions booléennes

- Syntaxe
- Constantes

Algorithmique élémentaire

- Recherche d'un diviseur (test de primalité)
- Boucle événementielle
- Attente active

Démos

Premier partiel

printf/scanf (1)

- Pour afficher un texte à l'écran, nous utilisons la fonction printf (*print formatted*).
- Chaque % dans le texte à afficher est substitué par la valeur formatée d'un paramètre supplémentaire de la fonction. Le caractère suivant le symbole % signale le format à utiliser. Un %d met une valeur au format entier décimal.
- Exemples :
  - `printf("Bonjour\n")` affiche `Bonjour` et un saut de ligne
  - `printf("i vaut %d\n", i)` affiche `i vaut` suivi de la valeur décimale de i (et d'un saut de ligne)
  - `printf("(%d, %d)\n", 31, -4)` affiche `(31, -4)` et un saut de ligne.
- Réciproquement pour faire entrer dans le programme une donnée saisie par l'utilisateur, nous utiliserons `scanf`.
- Exemple : `scanf("%d", &x)`

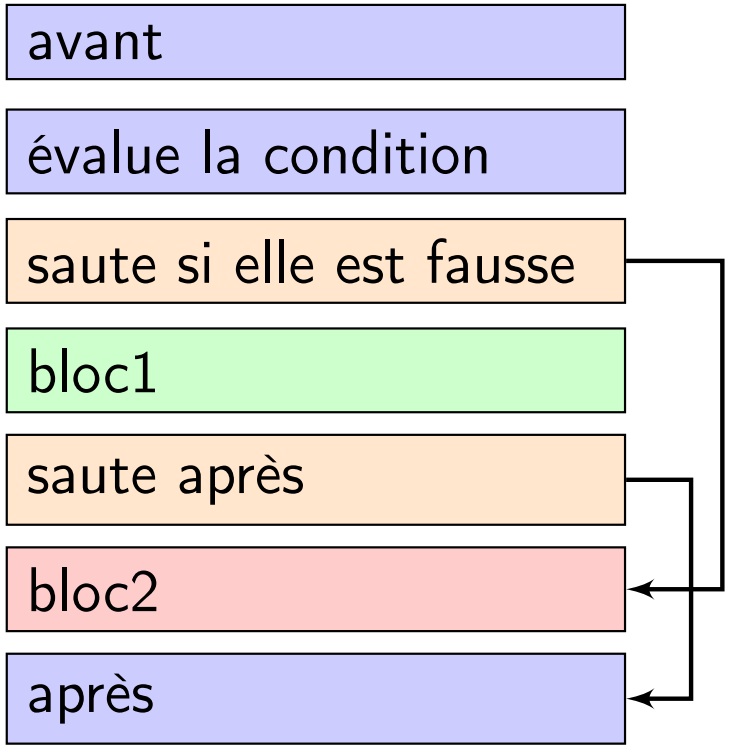
Rappels sur l'instruction de contrôle if

Syntaxe : `if (condition) { bloc1 } else { bloc2 }`.

Code source

```
/* avant */
if (age < 18)
{
  permis = 0;
}
else
{
  permis = 1;
}
/* après */
```

Schéma de traduction



Rappels sur l'instruction de contrôle for

Syntaxe :

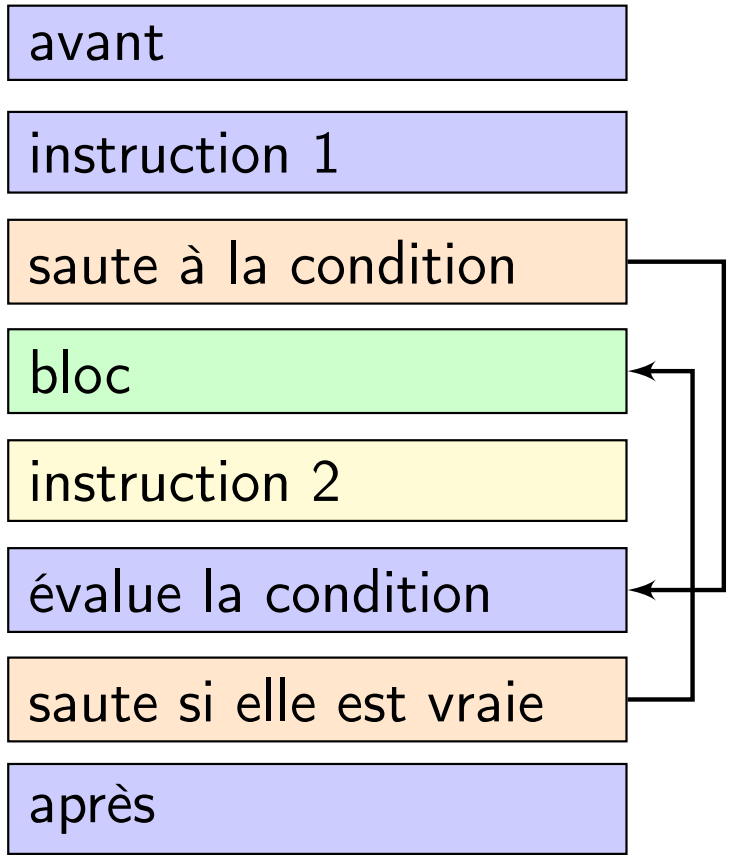
`for (instruct1; condition; instruct2) { bloc }.`

Code source

```
/* avant */
for (i = 0; i < 5; i = i + 1)
{
  printf("%d\n", i);
  ...
}
/* après */
```

La variable `i` est appelée variable de boucle, elle doit être préalablement déclarée comme toute autre variable.

Schéma de traduction



L'instruction de contrôle while

Syntaxe :

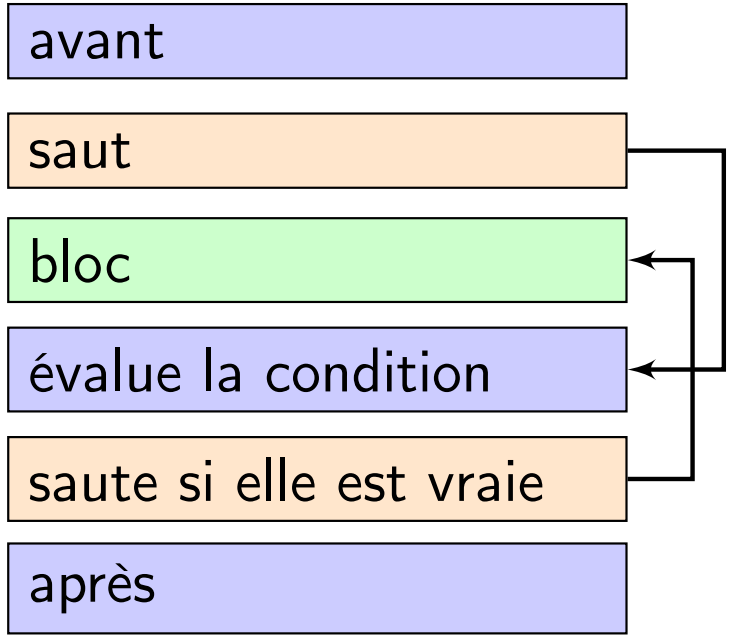
`while (condition) { bloc }.`

Code source

```
/* avant */
while ( ... )
{
  ...
}
/* après */
```

Pour assurer la terminaison, le bloc du while doit modifier la condition.

Schéma de traduction



Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre p
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que
11     {
12         x = x / 10; /* enlever
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	
12	60		
13		3	
12	6		
13		4	
15			ce nombre a 4...
18	Renvoie EXIT_SUCCESS		

For ou while ?

- Un `for` peut toujours être simulé par un `while` et le code machine sera identique. Il suffit d'introduire un compteur de boucle (la variable de boucle du `for`).
- Par convention, les programmeurs préfèrent utiliser un `for` lorsque le nombre d'itérations est connu à l'avance. Par exemple, pour parcourir un ensemble de cas. Dans le cas contraire, les programmeurs utilisent un `while`. Par exemple, pour chercher un cas particulier.
- Maintenant que nous avons le `while`, il est possible qu'un programme ne termine jamais (`Ctrl-C`).
- *L'arrêt des programmes est indécidable*.

Expressions booléennes

Les *conditions* employées dans les structures de contrôle (`if`, `for` ou `while`) sont des expressions booléennes, pouvant être *Vrai*, *Faux* ou :

- des inégalités entre expressions arithmétiques

$$\begin{aligned} \textit{inégalité} := {} & e_1 \texttt{ < } e_2 \mid e_1 \texttt{ > } e_2 \mid e_1 \texttt{ != } e_2 \\ & \mid e_1 \texttt{ <= } e_2 \mid e_1 \texttt{ >= } e_2 \mid e_1 \texttt{ == } e_2 \end{aligned}$$

- ou des combinaisons logiques d'expressions booléennes :

$$\begin{aligned} \textit{condition} := {} & (\textit{condition}) \texttt{ \&\& } (\textit{condition}) && \text{(et)} \\ & \mid (\textit{condition}) \texttt{ || } (\textit{condition}) && \text{(ou)} \\ & \mid \texttt{!}(\textit{condition}) && \text{(non)} \\ & \mid \textit{Vrai} \mid \textit{Faux} \mid \textit{inégalité} && \text{(cas de base)} \end{aligned}$$

Constantes booléennes

- Certains langages possédent un type booléen (admettant deux valeurs *true* et *false*) pour les expressions booléennes.
- En langage C, les expressions booléennes sont de type entier (`int`), l'entier *zéro* joue le rôle du Faux, l'entier *un* joue le rôle du Vrai et tout entier différent de zéro est évalué à vrai.
- On se donne deux constantes symboliques :

```
/* Declaration des constantes et types utilisateur */
#define TRUE 1
#define FALSE 0

int main()
{
    int continuer = TRUE; /* faut-il continuer ?*/

    while (continuer)
    {
        ...
```

Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques
   - `if else` (différencier)
   - `#define` constantes symboliques (nommer)
   - arbre de décision (organiser)
2. Parcourir/générer des cas
   - boucle `for` (rarement while)
   - tableaux *(plus tard)*
3. Composer des cas
   - boucles (parcourir/générer)
   - accumulateur (à initialiser)
4. Sélectionner des cas
   - boucles (parcourir/générer)
   - `if` (sélectionner/traiter)

3'. Dénombrer des cas
   - boucles (parcourir/générer)
   - compteur (à initialiser à 0)

5. Rechercher un cas
   - boucle `while`, conditions booléennes, `if`

2'. Parcourir/générer : une *ligne* mais aussi une *surface*, un *volume*...
   - imbriquer les boucles (var. $\neq$)

6. Boucle événementielle
   - boucle while

7. Attente active
   - boucle while

Recherche d'un diviseur (test de primalité)

Exemple

Le programme cherche un entier supérieur à 1 qui divise n, s'il n'en trouve pas de plus petit que n, alors n est premier (ou bien $n = 1$).

```
31          /* test de primalite */
32          d = 2;
33          while ( premier && (d < n) ) /* sans diviseur < d */
34          {
35              if (n % d == 0) /* d divise n */
36              {
37                  printf("divisible par %d\n", d);
38                  premier = FALSE;
39              }
40              d = d + 1; /* candidat diviseur suivant */
41          }
42
43          if (premier)
44          {
45              printf("%d est premier\n", n);
46          }
47          else
48          {
```

Boucle principale en programmation événementielle

Le programme lit les événements (les actions de l'utilisateur) et les traite, continuellement, dans une boucle.

Remarque

`scanf` est un appel système bloquant (processus en attente).

Exemple

Jeu de calcul mental (additionner deux nombres au hasard) 

```
16      int continuer = TRUE; /* TRUE s'il faut continuer a jouer */

22      /* initialisation du generateur de nombres pseudo-aleatoires *
23      srand(time(NULL)); /* a ne faire qu'une fois */

31      while(continuer) /* le joueur veut faire un nouvel essai */
32      {

34          /* tirage de x */
35          x = rand() % NBMAX + 1; /* entre 1 et NBMAX inclus */

53          printf("Continuer␣(1)␣ou␣arreter␣(0)␣?\n");
54          scanf("%d", &choix);
```

Boucle d'attente active

La boucle sert à attendre qu'une condition externe devienne vraie, le programme teste continuellement cette condition.

Exemple

Tic tac boum, est-ce que 10 secondes sont écoulées ? 

```
19      printf("Auto-destruction␣en␣cours,␣ctrl-C␣pour␣desamorcer␣!\n"
20      debut = time(NULL); /* date de debut (secondes depuis 1/1/1970
21      while(duree < 10) /* pas encore 10 secondes */
22      {
23          duree = time(NULL) - debut; /* duree depuis debut (seconde

42      } /* fin des 10 secondes */
43      printf("**␣BOUM␣!␣**\n");
```

Démos et utilisation de codeblocks