

Les itérations en C

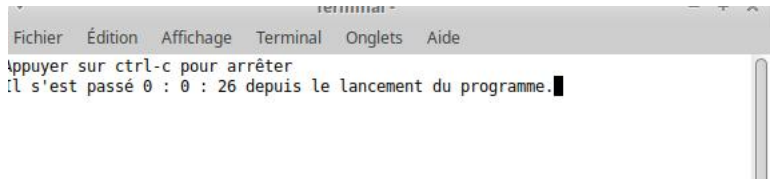
Jean-Gabriel Luque

Département d'informatique de l'université de Rouen - Normandie

Qu'est-ce que c'est et à quoi ça sert?

On veut: affichage dynamique du temps écoulé depuis lancement du prg.

Fin: c'est l'utilisateur qui décide.



```
Terminal
Fichier  Édition  Affichage  Terminal  Onglets  Aide
Appuyer sur ctrl-c pour arrêter
il s'est passé 0 : 0 : 26 depuis le lancement du programme.
```

La commande `printf("...%d : %d : %d..",...)` doit s'exécuter un nombre indéterminé de fois.

Trace de programme:

... -> 25 -> ... -> 25 ... -> 25 -> ...

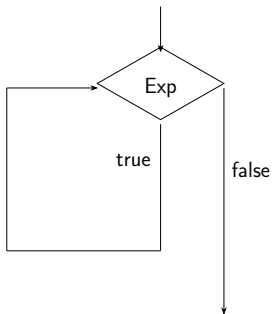
arbitrairement longue!

Impossible de faire cela uniquement avec des tests.

⇒ Instructions d'itérations.

Qu'est-ce que c'est et à quoi ça sert?

Représentation graphique

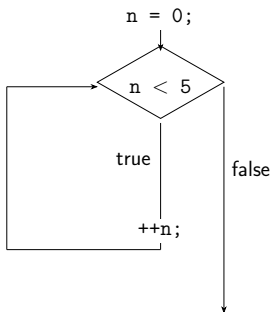


Exp: une expression booléenne.

Le bloc est exécuté si Exp est évaluée à true, sinon le programme "continue son chemin".

Qu'est-ce que c'est et à quoi ça sert?

Représentation graphique

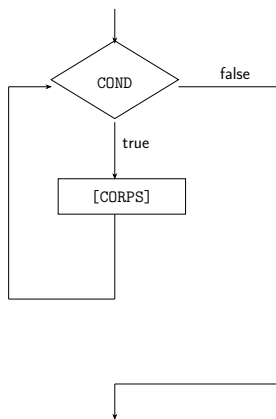


Exp: une expression booléenne.

Le bloc est exécuté si Exp est évaluée à `true`, sinon le programme “continue son chemin”.

Trois instructions

Boucles while



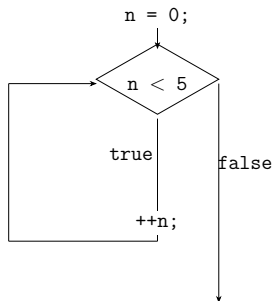
```
while (COND) {  
    [CORPS]  
}
```

COND: Expression booléenne.
Condition de continuité.

CORPS: Bloc d'instructions.
Corps de la boucle.

Trois instructions

Boucles while

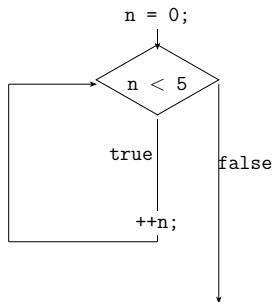


```
1 n = 0;  
2 while (n < 5) {  
3   ++n;  
4 }  
printf(n);
```

Après la boucle `n == ?`
nb executions ligne 3: ?
nb de tests `n < 5` : ?

Trois instructions

Boucles while



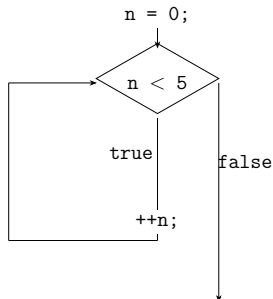
```
1 n = 0;
2 while (n < 5) {
3     ++n;
4 }
5 printf(n);
```

Après la boucle `n == ?`
nb executions ligne 3: ?
nb de tests `n<5` : ?

| | |
|----------------|----------|
| 1 -> 2 -> 3 | : n == 1 |
| -> 2 -> 3 | : n == 2 |
| -> 2 -> 3 | : n == 3 |
| -> 2 -> 3 | : n == 4 |
| -> 2 -> 3 | : n == 5 |
| -> 2 -> 4 -> 5 | : n == 5 |

Trois instructions

Boucle while



```
1 n = 0;
2 while (n < 5) {
3     ++n;
4 }
5 printf(n);
```

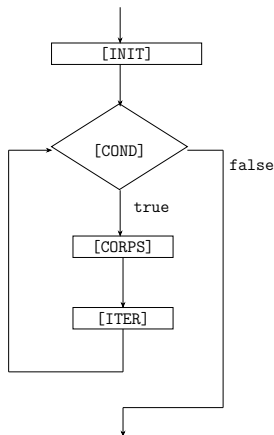
Après la boucle $n == 5$
nb executions ligne 3: 5
nb de tests $n < 5$: 6

Affichage: 5

| | |
|----------------|------------|
| 1 -> 2 -> 3 | : $n == 1$ |
| -> 2 -> 3 | : $n == 2$ |
| -> 2 -> 3 | : $n == 3$ |
| -> 2 -> 3 | : $n == 4$ |
| -> 2 -> 3 | : $n == 5$ |
| -> 2 -> 4 -> 5 | : $n == 5$ |

Trois instructions

Boucles for



```
for( [INIT] ; [COND] ; [ITER] ) {  
    [CORPS]  
}
```

[INIT] : Séquence d'instructions.
Initialisation

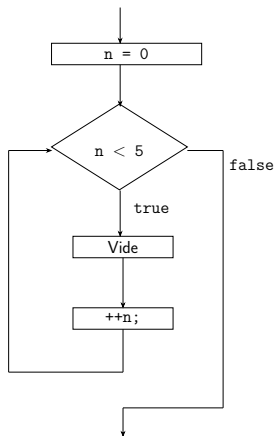
[COND] : Expression booléenne.
Condition de continuité

[CORPS] : bloc d'instructions.
Corps de la boucle

[ITER] : Séquence d'instructions.
Itération.

Trois instructions

Boucles for



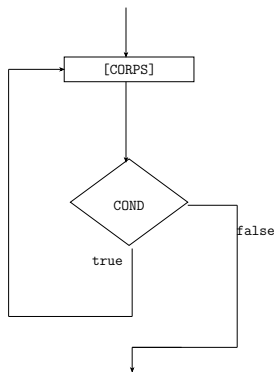
```
1 int n;  
2 for(n = 0; n < 5; ++n) {  
3 }  
4 printf(n);
```

Après la boucle `n == 5`
nb executions instruction `++n`: 5
nb de tests `n < 5` : 6

Affichage 5.

Trois instructions

Boucles `do ... while`



```
do {  
    [CORPS]  
} while (COND)
```

`COND`: Expression booléenne.
Condition de continuité.

`CORPS`: Bloc d'instructions.
Corps de la boucle.

Le corps de la boucle est exécuté au moins une fois.

Trois instructions

Boucles `do ... while`

Exemple typique:

```
int n = 0;
double s = 0;
char c;

do {
    double x;
    printf("une valeur, svp:");
    assert (scanf("%lf",&x) == 1);

    s += x;
    ++n;

    printf("Voulez-vous ajouter une valeur? [o/n]");
    assert(scanf(" %c",&c) == 1);
} while (c == 'o');

printf("La moyenne des valeurs est %lf", s / n);
```

Trois instructions

Équivalence entre les trois types de boucles : `while` \rightarrow `for`, `do`

| <code>while</code> | | <code>for</code> | | <code>do</code> |
|-----------------------------|--|---------------------------------|--|------------------------------|
| <code>/* *****</code> | | <code>*****</code> | | <code>***** */</code> |
| | | | | |
| <code>while (COND) {</code> | | <code>for (; COND ;) {</code> | | <code>do {</code> |
| <code>[CORPS]</code> | | <code>[CORPS]</code> | | <code>if (COND) {</code> |
| <code>}</code> | | <code>}</code> | | <code>[CORPS]</code> |
| | | | | <code>}</code> |
| | | | | <code>} while (COND);</code> |
| | | | | |

Trois instructions

Équivalence entre les trois types de boucles : `for` \rightarrow `while`, `do`.

| for | | while | | do |
|------------------------------|--|-----------------|--|------------------|
| /* ***** | | ***** | | ***** */ |
| | | | | |
| for ([INIT];[COND];[ITER]) { | | {<- | | {<- |
| [CORPS] | | [INIT;] | | [INIT;] |
| } | | while (COND#) { | | do { |
| | | [CORPS] | | if (COND#) { |
| | | [ITER;] | | [CORPS] |
| | | } | | [ITER;] |
| | | }<- | | } |
| | | | | } while (COND*); |
| | | | | }<- |

COND#: Si COND est vide alors écrire true.

<-: Attention à la portée des variables, il s'agit bien de blocs.

Trois instructions

Équivalence entre les trois types de boucles : `do` \rightarrow `while`, `for`

| <code>do</code> | <code>while</code> | <code>for</code> |
|------------------------------|-----------------------------|---------------------------------|
| <code>/* *****</code> | <code>*****</code> | <code>***** */</code> |
| | <code>{<-</code> | <code>{<-</code> |
| <code>do {</code> | <code>[CORPS]</code> | <code>[CORPS]</code> |
| <code> [CORPS]</code> | <code>while (COND) {</code> | <code>for (; COND ;) {</code> |
| <code>} while (COND);</code> | <code> [CORPS#]</code> | <code> [CORPS#]</code> |
| | <code>}</code> | <code>}</code> |
| | <code>}<-</code> | <code>}<-</code> |

Remarquez la répétition de `[CORPS]` pour les type `while` et `for`.

`<-`: Attention à la portée des variables, il s'agit bien de blocs.

`[CORPS#]` : Ne pas répéter les déclarations mais bien les affectations.

Par exemple: `int k = 0; \rightarrow k = 0;`

Trois instructions

Rôles

Utilisez de préférence

- `for`: lorsqu'une variable parcourt un intervalle (compteur de boucle).
- `do...while`: lorsque le corps de la boucle doit être exécuté au moins une fois.
- `while`: dans les autres cas.

Quelques pièges

Boucles infinies

```
int n;  
assert(scanf("%d",&n) == 1);  
  
int i=1, fact=1;  
  
while (i<n) {  
    fact = fact * i;  
}
```

Les variables de la condition ne sont pas modifiées par le corps de la fonction.

```
int n;  
assert(scanf("%d",&n) == 1);  
assert( n > 0 );  
int  lg=1;  
  
while (n >= 0) {  
    ++ lg;  
    n /= 2;  
}
```

La condition `n >= 0` est toujours vraie.

Quelques pièges

Boucles inutiles

```
int n, base;
assert(scanf("%d%d",&n,&base) == 2);
assert( n > 0 && base > 1);
int lg=1;

while (n <= 0) {
    ++ lg;
    n /= base;
}
```

La condition $n \leq 0$ est fausse juste avant la boucle, celle-ci n'est jamais exécutée.

Quelques pièges

Comportements indéterminés

```
int n, k;
assert(scanf("%d",&n) == 1);
// essayer avec echo "$"2147483647" | ./etrange
// essayer en enlevant l'option de compilation -O2
for (k = 1; k <= n; ++k) {
    // essayer en enlevant les commentaires
    /*if (k % STEP == 0) {
        printf("%d\n",k);
    }
    */
}
printf("\n k == %d\n",k);
```

La boucle s'arrête ou ne s'arrête pas suivant les options de compilation, ce que fait le corps de la boucle, si les variables sont saisies ou si elle sont fixées “en dur” par le programme etc.

Quelques pièges

Conseils

- Bien vérifier la condition de continuité.
- Vérifier que les variable de la condition de continuité sont bien modifiées dans le corps du programme.
- Préférez les inégalités strictes aux inégalités larges.
- Prendre en compte le vrai type de donnée. Par exemple, pour bien gérer les dépassements de capacités.
- Respectez la norme.

Méthodologie de construction d'une boucle

Invariants de boucle

- Proposition logique (vaut VRAI ou FAUX)
- Décrit ce qu'est une étape de calcul.
- Doit valoir VRAI:
 - ▶ Juste avant la boucle
 - ▶ à la fin de chaque tour de boucle,
 - ▶ juste après la boucle.
- Se décrit comme les assertions d'entrée et de sortie:
 - ▶ Utilisez le plus possible la syntaxe du C
 - ▶ Utilisez les connecteurs logiques `&&`, `||`, `!` et `^^`.
 - ▶ Utilisez `init_val` pour désigner la valeur d'une variable avant le début de la boucle.
 - ▶ Utiliser `somme(f(i), i=a..b)` pour désigner $\sum_{i=a}^b f(i)$.
Convention : vaut 0 si `b < a`.
 - ▶ Utiliser `produit(f(i), i=a..b)` pour désigner $\prod_{i=a}^b f(i)$.
Convention : vaut 1 si `b < a`.
 - ▶ Possibilité d'écrire de façon littérale ou d'introduire d'autres notations.

Méthodologie de construction d'une boucle

Trouver l'invariant de boucle (sur un exemple)

Factorielle : $n! = \prod_{i=1}^n i$ si $n \geq 0$. Non définie pour $n < 0$.

Calcul de $n!$ de façon décroissante:

| f | | n |
|-------------|--|-------|
| /* ***** */ | | |
| 1 | | 5 |
| 5 | | 4 |
| 5*4 | | 3 |
| 5*4*3 | | 2 |
| 5*4*3*2 | | 1 |
| 5*4*3*2*1 | | 0 FIN |

Invariant:

`f == produit(i, i= n + 1 ... init_val(n)) && n >= 0`

Méthodologie de construction d'une boucle

Trouver l'invariant de boucle (sur un exemple)

| f | | n |
|-------------|--|----------|
| /* ***** */ | | |
| 1 | | 5 <- |
| 5 | | 4 |
| 5*4 | | 3 <- |
| 5*4*3 | | 2 |
| 5*4*3*2 | | 1 |
| 5*4*3*2*1 | | 0 FIN <- |

Invariant:

`f == produit(i, i= n + 1 ... init_val(n)) && n >= 0`

- Avant la boucle: `f == produit(i, i= 6 ... 5) && 5 >= 0`
- Une étape: `f == produit(i, i= 4 ... 5) && 3 >= 0`
- Fin: `f == produit(i, i= 1 ... 5) && 0 >= 0`

Méthodologie de construction d'une boucle

Condition d'arrêt

Proposition logique

- vaut VRAI lorsque le calcul est terminé
- Condition d'arrêt && Invariant de boucle \Rightarrow Résultat

Négation de la condition de continuité.

Conseils:

- L'exprimer dans la syntaxe du C dans la mesure du possible. Permet d'écrire la condition de continuité.
- Préférer des inégalités à des égalités car
 - ▶ Expression homogène avec l'invariant de boucle.
 - ▶ Permet d'éviter des erreurs (par exemple si un compteur progresse par pas de 2).
 - ▶ Permet de prendre en compte directement des cas particuliers dans la boucle.

Méthodologie de construction d'une boucle

Exemple de condition d'arrêt

| f | | n |
|-------------|--|-------|
| /* ***** */ | | |
| 1 | | 5 |
| 5 | | 4 |
| 5*4 | | 3 |
| 5*4*3 | | 2 |
| 5*4*3*2 | | 1 |
| 5*4*3*2*1 | | 0 FIN |

Condition d'arrêt: $n \leq 0$.

$(f == \text{produit}(i, i = n + 1 \dots \text{init_val}(n)) \ \&\& \ n \geq 0)$ IB
 $\&\& \ n \leq 0$ && CA

=>

$(f == \text{produit}(i, i = n + 1 \dots \text{init_val}(n)) \ \&\& \ n == 0$

=>

$(f == \text{produit}(i, i = 1 \dots \text{init_val}(n)))$ Résultat

Méthodologie de construction d'une boucle

Configuration initiale

Choisir des valeurs

- qui rendent vrai l'invariant de boucle
- qui sont facile à obtenir

| f | | n |
|------------------|--|------|
| /* ***** **** */ | | |
| 1 | | 5 <- |
| 5 | | 4 |
| ... | | |

f == 1 && n == 5

f == produit(i, i = n + 1 ... init_val(n)) && n >= 0)

1 == produit(i, i = 6 ... 5) && 5 >= 0)

true

Méthodologie de construction d'une boucle

Obtenir le programme

Config initiale

```
/* Description de l'invariant de boucle en commentaire */  
while ( ! Condition d'arrêt) {  
    [Comment passer à l'étape suivante]  
}
```

Exemple:

```
f = 1; // Configuration initiale
```

```
/* Invariant de Boucle:
```

```
 * f == produit(i, i = n + 1 ... init_val(n)) && n >= 0  
 */
```

```
while (n > 0) {  
    f *= n;  
    --n;  
}
```