Boucles

Lycée Thiers

Crédits

- Openclassrooms

Crédits

- Openclassrooms
- ≪ Langage C ≫ -Claude Delannoy- Ed. Eyrolles

Crédits

- Openclassrooms
- ≪ Langage C ≫ -Claude Delannoy- Ed. Eyrolles
- ≪ Informatique - MP2I/MPI - CPGE 1re et 2e années ≫ -Balabonski Thibaut, Conchon Sylvain, Filliâtre Jean-Christophe, Nguyen Kim, Sartre Laurent- Ed. Ellipse

Boucles while

Pour mémoire on rappelle le fonctionnement de la boucle `while` :

```
while (Condition )
{
    // Instructions à répéter
}
```

Voici une boucle qui demande d'entrer un nombre jusqu'à ce que l'utilisateur saisisse 10 :

```
# include <stdio.h>

int main()
{
  int nb =0 ;
  while (nb != 10)
    {
      printf("entrer un nb entier :");
      scanf ("%d", &nb);
    }
}
```

Boucle do ... while

Dans une boucle `while`, si la condition n'est pas réalisée alors le corps de la boucle n'est jamais exécuté.
Une façon d'éviter cela est de mettre le test de sortie de boucle à la fin :

```
1  int compteur = 0;// le compteur
2
3  do
4    {
5      printf("le compteur vaut %i.\n", compteur);
6      compteur++; // incrémentation de 1
7    }while (compteur < 0);// ne pas oublier le point virgule
```

Après compilation et exécution :

```
le compteur vaut 0.
```

Boucle `for`

- Si on souhaite parcourir tous les entiers de 0 à $n-1$, on peut le faire avec une boucle while (et gérer soi-même le compteur) :

```
{int i = 0; while (i<n){...; i = i+1; }}
```

- Une boucle `for(declaration; test; mise à jour)` permet de faire la même chose et gère la mise à jour :

```
for (int i = 0; i<n; i = i+1) {...;}
```

- Incrémentation : On peut écrire `i+=1` à la place de `i=i+1`, ou même `i++`.
  Décrémentation : `i--`

Boucle `for`

Compléments

La boucle `for` a une forme non limitée à l'exemple du transparent précédent.

- Pour une variable `x` déclarée hors de la boucle, on peut écrire :

```
1 int x = 3; ...
2 for (; x!=1; x=f(x)){ ...}
3
```

Boucle `for`

Compléments

La boucle `for` a une forme non limitée à l'exemple du transparent précédent.

- Pour une variable `x` déclarée hors de la boucle, on peut écrire :

```
1 int x = 3; ...
2 for (; x!=1; x=f(x)){ ...}
3
```

- Et si `start`, `stop`, `step` sont 3 fonctions définies par ailleurs

```
1 for (start(); tests(); step()) {...}
2
```

Opérateurs d'incrémentation

- `i++` est une expression qui renvoie la valeur de la variable *i* avant son incrémentation.

Opérateurs d'incrémentation

- `i++` est une expression qui renvoie la valeur de la variable *i* avant son incrémentation.
- `++i` est une expression qui incrémente *i* puis renvoie la valeur de la variable *i*.

Opérateurs d'incrémentation

- `i++` est une expression qui renvoie la valeur de la variable *i* avant son incrémentation.
- `++i` est une expression qui incrémente *i* puis renvoie la valeur de la variable *i*.
- Par exemple

```
1 int n = 3;
2 while(n -->0) printf("n=%d",n);
```

affiche `n=2,n=1,n=0`. On compare en effet successivement 3,2,1, 0 avec 0 car `n-->0` décrémente **après** la comparaison.

Opérateurs d'incrémentation

- `i++` est une expression qui renvoie la valeur de la variable *i* avant son incrémentation.
- `++i` est une expression qui incrémente *i* puis renvoie la valeur de la variable *i*.
- Par exemple

```
1 int n = 3;
2 while(n-->0) printf("n=%d",n);
```

affiche `n=2,n=1,n=0`. On compare en effet successivement 3,2,1, 0 avec 0 car `n-->0` décrémente **après** la comparaison.

- Mais

```
1 int n = 3;
2 while(--n>0) printf("n=%d",n);
```

affiche `n=2,n=1`. On compare en effet 2,1,0 avec 0 car `--n>0` décrémente **avant** la comparaison.

Opérateurs d'incrémentation

- Moyen mnémotechnique : mettre le `--` le plus loin possible de l'opérateur de comparaison.

Opérateurs d'incrémentation

- Moyen mnémotechnique : mettre le `--` le plus loin possible de l'opérateur de comparaison.
- Avec `--n>0` on comprend que a) d'abord on décrémente et b) ensuite on compare.

Opérateurs d'incrémentation

- Moyen mnémotechnique : mettre le `--` le plus loin possible de l'opérateur de comparaison.
- Avec `--n>0` on comprend que a) d'abord on décrémente et b) ensuite on compare.
- Avec `0<n--` on comprend que a) d'abord on compare et b) ensuite on décrémente.

Opérateurs d'incrémentation

- Moyen mnémotechnique : mettre le `--` le plus loin possible de l'opérateur de comparaison.
- Avec `--n>0` on comprend que a) d'abord on décrémente et b) ensuite on compare.
- Avec `0<n--` on comprend que a) d'abord on compare et b) ensuite on décrémente.
- Mais de toute façon, je déconseille fortement d'utiliser ces opérateurs dans d'autres instructions.

Danger des opérateurs d'incrémentation

- L'ordre d'évaluation des opérandes (de gauche à droite ?) ou des arguments d'un appel de fonction n'est pas spécifié en C.

Danger des opérateurs d'incrémentation

- L'ordre d'évaluation des opérandes (de gauche à droite ?) ou des arguments d'un appel de fonction n'est pas spécifié en C.
  - Il faut donc se méfier des appels comme `f(x++,x)` car la norme ne dit pas si le second paramètre est affecté ou non par l'incrémentation (ça peut dépendre des implémentations !).

Danger des opérateurs d'incrémentation

- L'ordre d'évaluation des opérandes (de gauche à droite ?) ou des arguments d'un appel de fonction n'est pas spécifié en C.
  - Il faut donc se méfier des appels comme `f(x++,x)` car la norme ne dit pas si le second paramètre est affecté ou non par l'incrémentation (ça peut dépendre des implémentations !).
  - Si `x=3`, cet appel peut être compris `f(3,3)` ou `f(3,4)`.

Danger des opérateurs d'incrémentation

- L'ordre d'évaluation des opérandes (de gauche à droite ?) ou des arguments d'un appel de fonction n'est pas spécifié en C.
  - Il faut donc se méfier des appels comme `f(x++,x)` car la norme ne dit pas si le second paramètre est affecté ou non par l'incrémentation (ça peut dépendre des implémentations !).
  - Si `x=3`, cet appel peut être compris `f(3,3)` ou `f(3,4)`.
- Nous réservons alors sagement les expressions comme `i++` et `i--` à la partie « incrémentation » des boucles, sans les inclures dans d'autres expressions.

Les instructions `break` et `continue`

- L'instruction `break` permet de sortir d'une boucle ; `continue` de passer à l'itération suivante.

Les instructions `break` et `continue`

- L'instruction `break` permet de sortir d'une boucle ; `continue` de passer à l'itération suivante.
- Ces instructions ne concernent que le bloc d'instruction courante.

Les instructions `break` et `continue`

- L'instruction `break` permet de sortir d'une boucle ; `continue` de passer à l'itération suivante.
- Ces instructions ne concernent que le bloc d'instruction courante.
- Si, par exemple, `break` est utilisée dans une double boucle imbriquée, il ne permet de sortir que de la boucle interne.

break

```
1  int i,j;
2      for(i=0;i < 4;i++){
3          printf("\ni = %d, ",i);
4          for(j=0;j< 5;j++){
5              if( j == 2 ) break; // quitter la boucle interne
6              printf("j = %d,",j);
7          }//for j
8      }// for i
9      printf("\nbye\n");
10
```

Produit l'affichage

```
i = 0, j = 0,j = 1,
i = 1, j = 0,j = 1,
i = 2, j = 0,j = 1,
i = 3, j = 0,j = 1,
bye
```

`j` ne prend aucune valeur au-delà de 1.

continue

Le code suivant :

```
1     int i;
2     for(i=0;i < 5;i++){
3         if( i==3 ) continue; // passer à l'itération suivante
4         // cette instruction est ignorée lorsque i==3
5         printf("i = %d \n",i);
6     }
7     printf("bye\n");
8
```

produit l'affichage

```
i = 0
i = 1
i = 2
i = 4
bye
```