




*Éléments d'informatique – Cours 4.
Compilation. Instruction de contrôle for.*

Pierre Boudes

7 octobre 2011





- *Éléments d'architecture des ordinateurs (+mini-assembleur)* 
- *Éléments de systèmes d'exploitation*
- Programmation structurée impérative (éléments de langage C)
 - *Structure d'un programme C*
 - *Variables : déclaration (et initialisation), affectation*
 - **Évaluation d'expressions**
 - **Instructions de contrôle** : *if, for, while*
 - Types de données : entiers, caractères, réels, tableaux, enregistrements
 - **Fonctions d'entrées/sorties (scanf/printf)**
 - Écriture et appel de fonctions
 - Débogage
- **Notions de compilation**
 - **Analyse lexicale, analyse syntaxique, analyse sémantique**
 - **préprocesseur du compilateur C (include, define)**
 - **Édition de lien**
- Algorithmes élémentaires
- Méthodologie de résolution, manipulation sous linux



Compilation

Analyse lexicale

Analyse syntaxique

Analyse sémantique

Génération du code

Édition de liens

La compilation en pratique (gcc)

L'instruction de contrôle for du langage C

Rappel sur la programmation structurée

Rappels sur l'instruction de contrôle if

L'instruction de contrôle for

Démos

Trace

printf/scanf (1)



Liens utiles

- ma page : <http://www-lipn.univ-paris13.fr/~boudes/>
- Un livre de la BU : *Le livre du C, premier langage (pour les vrais débutants en programmation)*, Claude Delannoy.
- <http://www.siteduzero.com/> (chercher langage C)
- <http://www.developpez.com/> (chercher langage C)
- le cours de Anne Canteaut :
http://www-roc.inria.fr/secret/Anne.Canteaut/COURS_C/
- le cours de Bernard Cassagne :
http://clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html
- le cours de Henri Garreta :
<http://www.dil.univ-mrs.fr/~garreta/generique/>
- codeblocks : <http://www.codeblocks.org/>
- ubuntu : <http://www.ubuntu-fr.org/>
- virtualbox : <http://www.virtualbox.org/>



Compilation

- *Compiler* un programme c'est traduire un texte (*code source*) d'un langage de haut niveau (langage C) en code de bas niveau (*code machine*), de manière à ce que le système d'exploitation puisse, au besoin, déclencher l'exécution de ce programme.



Compilation

- *Compiler* un programme c'est traduire un texte (*code source*) d'un langage de haut niveau (langage C) en code de bas niveau (*code machine*), de manière à ce que le système d'exploitation puisse, au besoin, déclencher l'exécution de ce programme. **Dans un langage compilé, l'étape de traduction a lieu une fois pour toutes.**



Compilation

- *Compiler* un programme c'est traduire un texte (*code source*) d'un langage de haut niveau (langage C) en code de bas niveau (*code machine*), de manière à ce que le système d'exploitation puisse, au besoin, déclencher l'exécution de ce programme. **Dans un langage compilé, l'étape de traduction a lieu une fois pour toutes.**
- *Interpréter*, c'est faire en même temps la traduction et l'exécution du texte d'un langage de haut niveau (un *script*). Un interprète simule ainsi un processeur capable d'exécuter le langage de haut niveau.



Compilation

- *Compiler* un programme c'est traduire un texte (*code source*) d'un langage de haut niveau (langage C) en code de bas niveau (*code machine*), de manière à ce que le système d'exploitation puisse, au besoin, déclencher l'exécution de ce programme. **Dans un langage compilé, l'étape de traduction a lieu une fois pour toutes.**
- *Interpréter*, c'est faire en même temps la traduction et l'exécution du texte d'un langage de haut niveau (un *script*). Un interprète simule ainsi un processeur capable d'exécuter le langage de haut niveau. **Dans un langage interprété, l'étape de traduction a lieu à chaque exécution.**



Compilation

- *Compiler* un programme c'est traduire un texte (*code source*) d'un langage de haut niveau (langage C) en code de bas niveau (*code machine*), de manière à ce que le système d'exploitation puisse, au besoin, déclencher l'exécution de ce programme. **Dans un langage compilé, l'étape de traduction a lieu une fois pour toutes.**
- *Interpréter*, c'est faire en même temps la traduction et l'exécution du texte d'un langage de haut niveau (un *script*). Un interprète simule ainsi un processeur capable d'exécuter le langage de haut niveau. **Dans un langage interprété, l'étape de traduction a lieu à chaque exécution.**
- Le langage C est traditionnellement un langage compilé.



Compilation

Les cinq grandes étapes de la compilation :

1. Analyse lexicale
2. Analyse syntaxique
3. Analyse sémantique
4. Génération du code
5. Édition de liens



Analyse lexicale

Analyse lexicale

Identifie les *lexèmes* (unités lexicales du langage). Les espaces sont inutiles ($3*x+1$ ou $3 * x + 1$), sauf comme séparateurs (`int x, intx`).



Analyse lexicale

Analyse lexicale

Identifie les *lexèmes* (unités lexicales du langage). Les espaces sont inutiles ($3*x+1$ ou $3 * x + 1$), sauf comme séparateurs (`int x, intx`).

Erreur lexicale :

code source `int x = @;`

compilation error: stray '@' in program



Analyse lexicale

Analyse lexicale

Identifie les *lexèmes* (unités lexicales du langage). Les espaces sont inutiles ($3*x+1$ ou $3 * x + 1$), sauf comme séparateurs (`int x, intx`).

Erreur lexicale :

code source `int x = @;`

compilation error: stray '@' in program

Erreur détectée uniquement au moment de l'analyse sémantique :

code source `intx = 0;`

compilation error: 'intx' undeclared (first use in this function)



Analyse syntaxique

Analyse syntaxique

trouve la structure syntaxique, (arbre syntaxique), et teste l'appartenance au langage.



Analyse syntaxique

Analyse syntaxique

trouve la structure syntaxique, (arbre syntaxique), et teste l'appartenance au langage.

Exemple : dans l'expression $x = 3 * x + 1$, est-ce que la sous-suite $x + 1$ correspond à une structure syntaxique ?

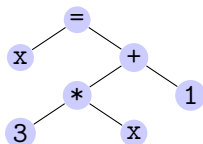


Analyse syntaxique

Analyse syntaxique

trouve la structure syntaxique, (arbre syntaxique), et teste l'appartenance au langage.

Exemple : dans l'expression $x = 3 * x + 1$, est-ce que la sous-suite $x + 1$ correspond à une structure syntaxique ?



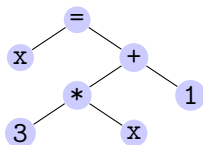


Analyse syntaxique

Analyse syntaxique

trouve la structure syntaxique, (arbre syntaxique), et teste l'appartenance au langage.

Exemple : dans l'expression $x = 3 * x + 1$, est-ce que la sous-suite $x + 1$ correspond à une structure syntaxique ?



code source Un else sans if le précédant immédiatement
(point-virgule mal placé?)

compilation error: expected expression before 'else'



Analyse sémantique

Analyse sémantique

trouver le sens des différentes actions voulues par le programmeur.

- Quels sont les objets manipulés par le programme,
- quelles sont les propriétés de ces objets,
- quelles sont les actions du programme sur ces objets.



Analyse sémantique

Analyse sémantique

trouver le sens des différentes actions voulues par le programmeur.

- Quels sont les objets manipulés par le programme,
- quelles sont les propriétés de ces objets,
- quelles sont les actions du programme sur ces objets.

Beaucoup d'erreurs peuvent apparaître durant cette phase :
identificateur utilisé mais non déclaré (la réciproque génère un *warning* avec l'option `-Wall`), opération n'ayant aucun sens, etc.

code source variable `x` utilisée mais non déclarée

compilation error: 'x' undeclared (first use in this function)



Génération du code

Génération du code

encodage en assembleur, optimisations et allocations des registres, traduction en code objet.



Édition de liens

Édition de liens

le code objet des fonctions externes (bibliothèques) est ajouté à l'exécutable. Le point d'entrée dans le programme est choisi (`main`). Insertion de données de débogage (option `-g`).



Édition de liens

Édition de liens

le code objet des fonctions externes (bibliothèques) est ajouté à l'exécutable. Le point d'entrée dans le programme est choisi (`main`). Insertion de données de débogage (option `-g`).

code source Oublie de `stdio.h` et utilisation de `printf`.

compilation `warning: incompatible implicit declaration of built-in function 'printf'`



Édition de liens

Édition de liens

le code objet des fonctions externes (bibliothèques) est ajouté à l'exécutable. Le point d'entrée dans le programme est choisi (`main`). Insertion de données de débogage (option `-g`).

code source Oublie de `stdio.h` et utilisation de `printf`.

compilation `warning: incompatible implicit declaration of built-in function 'printf'`

code source Pas de fonction principale (`main`)

compilation `Undefined symbols: "_main", ...`



Édition de liens

Édition de liens

le code objet des fonctions externes (bibliothèques) est ajouté à l'exécutable. Le point d'entrée dans le programme est choisi (`main`). Insertion de données de débogage (option `-g`).

code source Oublie de `stdio.h` et utilisation de `printf`.

compilation warning: incompatible implicit declaration of built-in function 'printf'

code source Pas de fonction principale (`main`)

compilation Undefined symbols: "_main", ...

À votre avis : Undefined symbols: "_printf" ?



La compilation en pratique (gcc)

source.c



exécutable

gcc

source.c -o nom



La compilation en pratique (gcc)

source.c

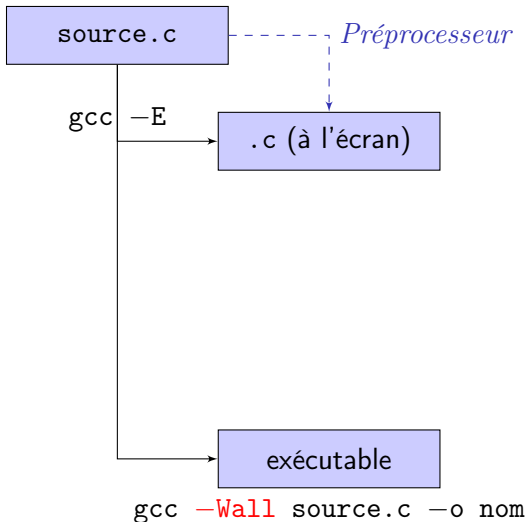
```
graph TD; A[source.c] --> B[exécutable];
```

exécutable

```
gcc -Wall source.c -o nom
```



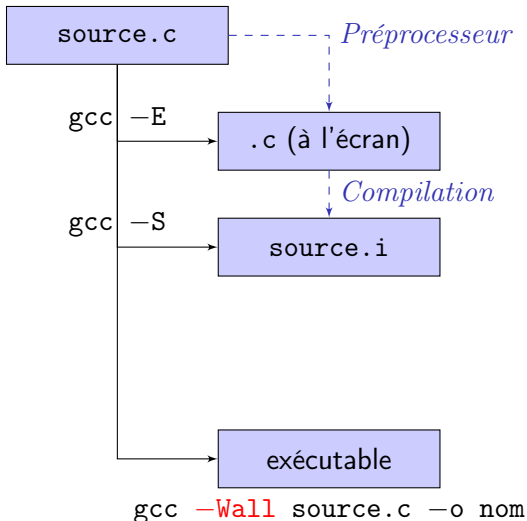
La compilation en pratique (gcc)



Le préprocesseur enlève les commentaires et exécute les instructions commençant par un dièse : `#define` (rechercher-remplacer) et `#include` (insertion de fichier).



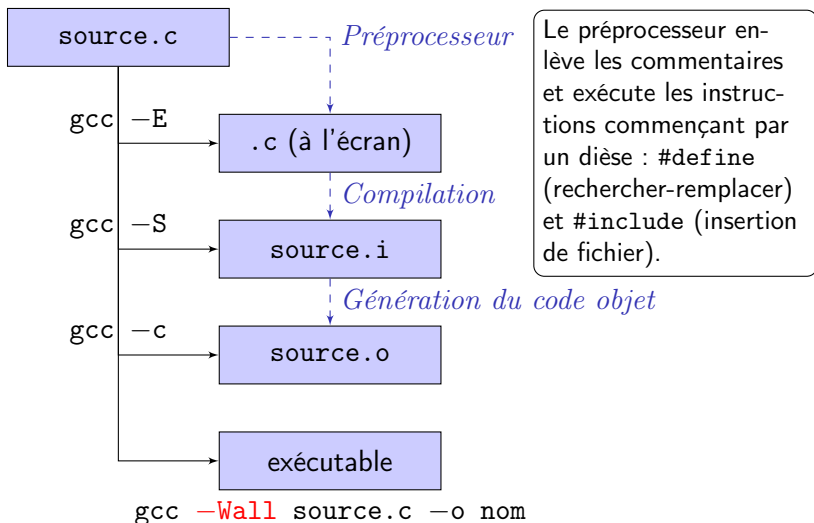
La compilation en pratique (gcc)



Le préprocesseur enlève les commentaires et exécute les instructions commençant par un dièse : `#define` (rechercher-remplacer) et `#include` (insertion de fichier).

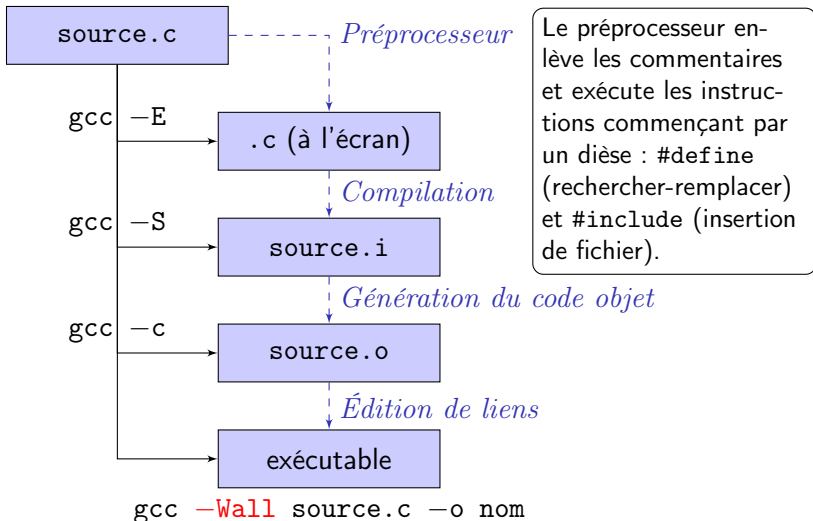


La compilation en pratique (gcc)

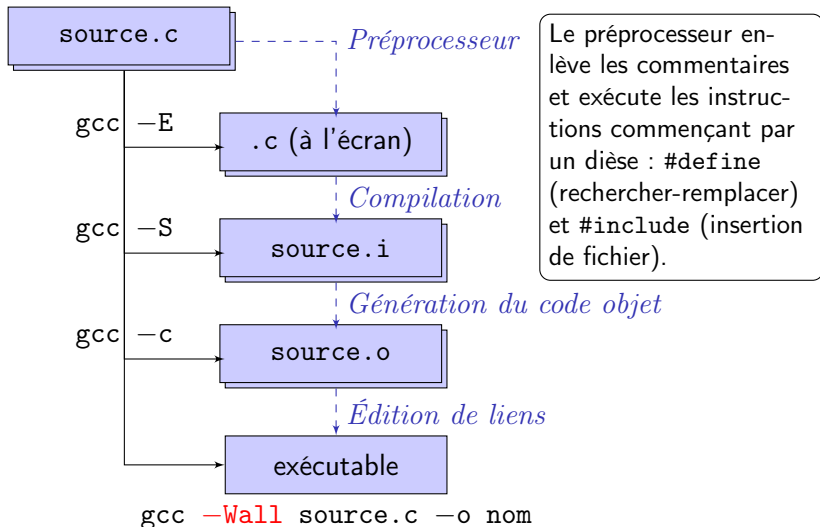




La compilation en pratique (gcc)



La compilation en pratique (gcc)





Rappel sur la programmation structurée

Definition (Programmation structurée)

Programmer par *blocs* d'instructions en combinant ces blocs de trois manières :

1. exécuter les blocs les uns à la suite des autres (*séquence*)
2. si une certaine condition est vraie, exécuter un bloc sinon en exécuter un autre (*sélection*)
3. recommencer l'exécution d'un bloc tant qu'une certaine condition est vraie (*répétition*).

Un bloc peut lui-même contenir une combinaison de blocs.



Rappel sur la programmation structurée

Definition (Programmation structurée)

Programmer par *blocs* d'instructions en combinant ces blocs de trois manières :

1. exécuter les blocs les uns à la suite des autres (*séquence*)
2. si une certaine condition est vraie, exécuter un bloc sinon en exécuter un autre (*sélection*)
3. recommencer l'exécution d'un bloc tant qu'une certaine condition est vraie (*répétition*).

Un bloc peut lui-même contenir une combinaison de blocs.

Aujourd'hui nous allons voir une première forme de répétition en C, le `for`. Avant cela nous revenons sur la sélection (le `if` ou `if else`).



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 */
```



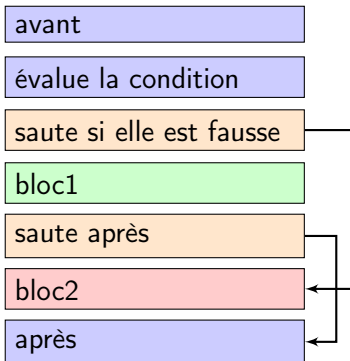
Rappels sur l'instruction de contrôle if

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

Code source

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

Schéma de traduction





L'instruction de contrôle for

Syntaxe :

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



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.



L'instruction de contrôle for

Syntaxe :

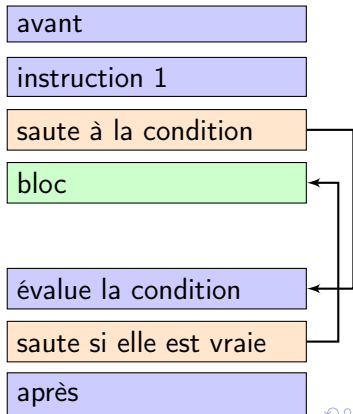
```
for (instruct1; condition; instruct2) { bloc }.
```

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 for

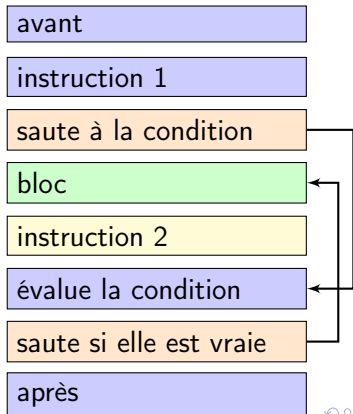
Syntaxe :

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

Code source

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

Schéma de traduction



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



Démos



Trace

```
1 int main()
2 {
3     /* Declaration et initialisation de variables */
4     int i; /* var. de boucle */
5
6     for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7     {
8         printf("etape_␣%d\n", i);
9     }
10    printf("i_␣vaut_␣%d\n", i);
11
12    return EXIT_SUCCESS;
13 }
```



Trace

```
1 int main()
2 {
3     /* Declaration et initialisation de variables */
4     int i; /* var. de boucle */
5
6     for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7     {
8         printf("etape_\u%d\n", i);
9     }
10    printf("i_\u vaut_\u%d\n", i);
11
12    return EXIT_SUCCESS;
13 }
```

ligne	i	sortie écran



Trace

```
1 int main()
2 {
3     /* Declaration et initialisation de variables */
4     int i; /* var. de boucle */
5
6     for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7     {
8         printf("etape_\u%d\n", i);
9     }
10    printf("i_\u vaut_\u%d\n", i);
11
12    return EXIT_SUCCESS;
13 }
```

ligne	i	sortie écran
initialisation	?	



Trace

```
1 int main()
2 {
3     /* Declaration et initialisation de variables */
4     int i; /* var. de boucle */
5
6     for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7     {
8         printf("etape_\u%d\n", i);
9     }
10    printf("i_\u vaut_\u%d\n", i);
11
12    return EXIT_SUCCESS;
13 }
```

ligne	i	sortie écran
initialisation	?	
6	0	



Trace

```

1  int main()
2  {
3      /* Declaration et initialisation de variables */
4      int i; /* var. de boucle */
5
6      for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7      {
8          printf("etape_\u%d\n", i);
9      }
10     printf("i_\u vaut_\u%d\n", i);
11
12     return EXIT_SUCCESS;
13 }

```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0



Trace

```

1  int main()
2  {
3      /* Declaration et initialisation de variables */
4      int i; /* var. de boucle */
5
6      for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7      {
8          printf("etape_\u%d\n", i);
9      }
10     printf("i_\u vaut_\u%d\n", i);
11
12     return EXIT_SUCCESS;
13 }

```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0
9	1	



Trace

```
1 int main()
2 {
3     /* Declaration et initialisation de variables */
4     int i; /* var. de boucle */
5
6     for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7     {
8         printf("etape_\u%d\n", i);
9     }
10    printf("i_\u vaut_\u%d\n", i);
11
12    return EXIT_SUCCESS;
13 }
```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0
9	1	
8		etape 1



Trace

```
1 int main()
2 {
3     /* Declaration et initialisation de variables */
4     int i; /* var. de boucle */
5
6     for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7     {
8         printf("etape_\%d\n", i);
9     }
10    printf("i_\%vaut_\%d\n", i);
11
12    return EXIT_SUCCESS;
13 }
```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0
9	1	
8		etape 1
9	2	



Trace

```
1 int main()
2 {
3     /* Declaration et initialisation de variables */
4     int i; /* var. de boucle */
5
6     for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7     {
8         printf("etape_\u%d\n", i);
9     }
10    printf("i_\u vaut_\u%d\n", i);
11
12    return EXIT_SUCCESS;
13 }
```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0
9	1	
8		etape 1
9	2	
8		etape 2



Trace

```

1  int main()
2  {
3      /* Declaration et initialisation de variables */
4      int i; /* var. de boucle */
5
6      for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7      {
8          printf("etape_\%d\n", i);
9      }
10     printf("i_\%d\n", i);
11
12     return EXIT_SUCCESS;
13 }

```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0
9	1	
8		etape 1
9	2	
8		etape 2
9	3	



Trace

```

1  int main()
2  {
3      /* Declaration et initialisation de variables */
4      int i; /* var. de boucle */
5
6      for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7      {
8          printf("etape_\%d\n", i);
9      }
10     printf("i_\%d\n", i);
11
12     return EXIT_SUCCESS;
13 }

```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0
9	1	
8		etape 1
9	2	
8		etape 2
9	3	
10		i vaut 3



Trace

```

1  int main()
2  {
3      /* Declaration et initialisation de variables */
4      int i; /* var. de boucle */
5
6      for (i = 0; i < 3; i = i + 1) /* pour chacune des 3 etapes */
7      {
8          printf("etape_\%d\n", i);
9      }
10     printf("i_\%d\n", i);
11
12     return EXIT_SUCCESS;
13 }

```

ligne	i	sortie écran
initialisation	?	
6	0	
8		etape 0
9	1	
8		etape 1
9	2	
8		etape 2
9	3	
10		i vaut 3
12		Renvoie EXIT_SUCCESS



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 % détaille la conversion à utiliser. La conversion %d met une valeur au format **entier décimal**.



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 % détaille la conversion à utiliser. La conversion %d met une valeur au format **entier décimal**.
- Exemples :
 - `printf("Bonjour\n")` affiche Bonjour et un saut de ligne



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 % détaille la conversion à utiliser. La conversion %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/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 % détaille la conversion à utiliser. La conversion %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.



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 % détaille la conversion à utiliser. La conversion %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)`



Démos