



Algorithmique et programmation – Cours 2.

Compilation. Instruction de contrôle `for`.

Pierre Boudes

10 septembre 2012



Langages et programmes (reloaded)

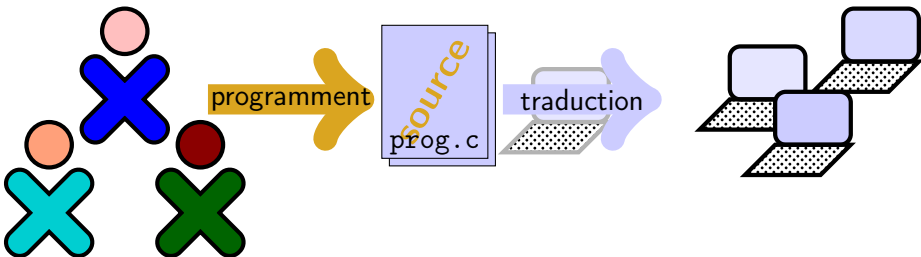
Humains

langage naturel

Langage de
programmation

Ordinateurs

langage machine



Pour le moment nous nous concentrons sur les rudiments de langage C.

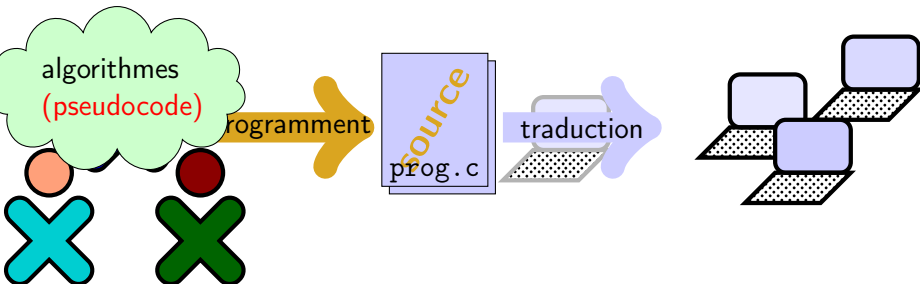
Langages et programmes (reloaded)

Humains

langage naturel

Langage de
programmation

Ordinateurs
langage machine



Pour le moment nous nous concentrons sur les rudiments de langage C. Mais ne perdons pas de vue que si savoir s'exprimer est nécessaire, il faut aussi savoir réfléchir à ce que nous faisons faire à nos programmes. **Écrivez vos algorithmes!**



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

Trace

Algorithmes élémentaires

Demos

printf/scanf (1)



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 ?

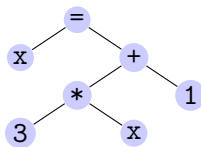


Analyse syntaxique

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 ?



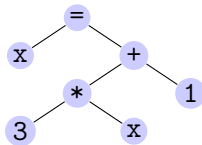


Analyse syntaxique

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* (du code machine non exécutable en l'état).



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`.

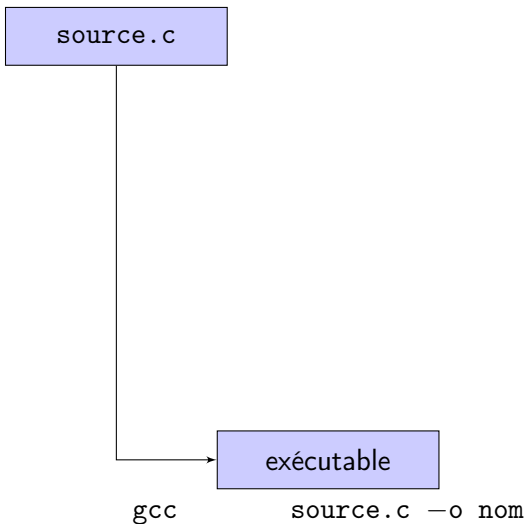
compilation warning: incompatible implicit declaration of built-in function 'printf'

code source Pas de fonction principale (main)

compilation Undefined symbols: "_main", ...

À votre avis : Undefined symbols: "_prinft" ?

La compilation en pratique (gcc)



La compilation en pratique (gcc)

source.c

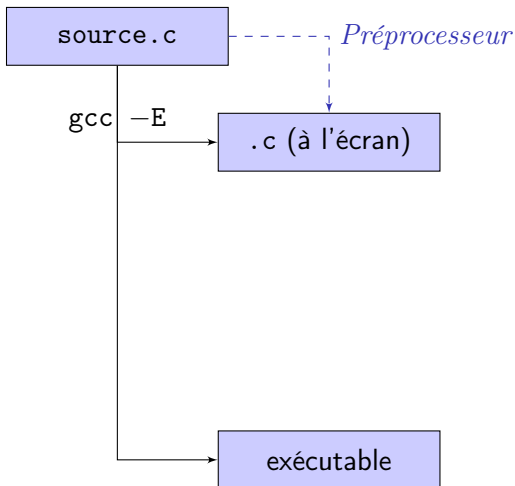
```
graph TD; A[source.c] --> B[exécutable];
```


A diagram illustrating the compilation process. A light blue rectangular box labeled 'source.c' is at the top left. A vertical line descends from the bottom of this box, and then a horizontal arrow points to the right, ending at a second light blue rectangular box labeled 'exécutable'.

exécutable

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

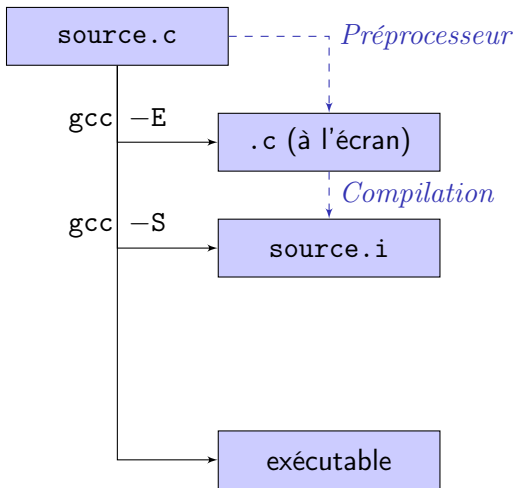
La compilation en pratique (gcc)




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

`gcc -Wall source.c -o nom`

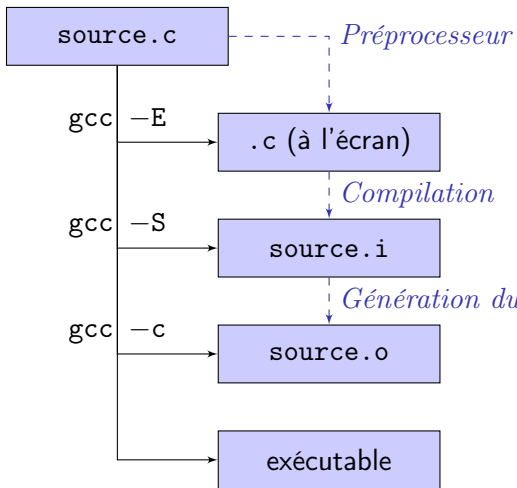
La compilation en pratique (gcc)




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

`gcc -Wall source.c -o nom`

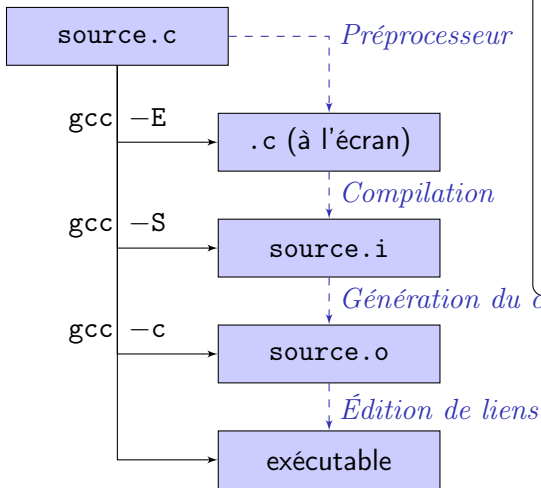
La compilation en pratique (gcc)




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

`gcc -Wall source.c -o nom`

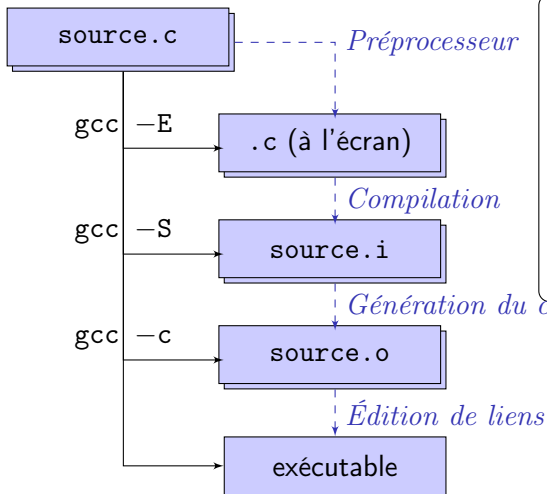
La compilation en pratique (gcc)




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

`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).

`gcc -Wall source.c -o nom`




Rappel sur la programmation structurée

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

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` qui sert pour exprimer : 

- la répétition un nombre fixé de fois **répéter n fois**
- l'itération sur un ensemble de cas (d'autres langages plus modernes ont le `for each` pour itérer sur un ensemble d'éléments). **Pour chaque entier de 1 à n faire ...**



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 */
```

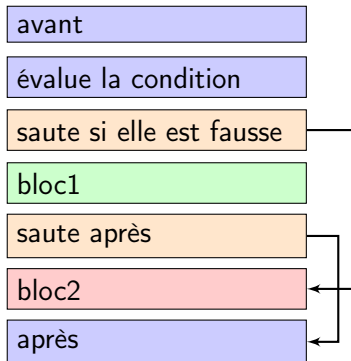
Rappels sur l'instruction de contrôle *if*

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

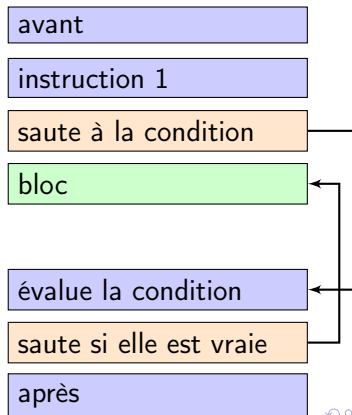
Syntaxe :

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

Code source

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

Schéma de traduction



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

L'instruction de contrôle for

Syntaxe :

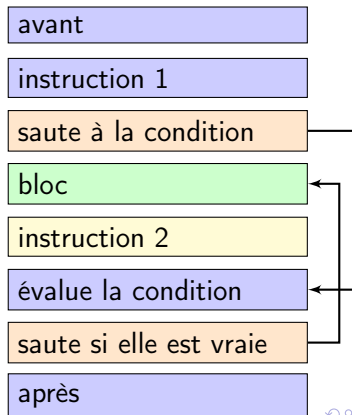
for (instruct1; condition; instruct2) { bloc }.

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



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_␣%d\n", i);
9      }
10     printf("i_␣vaut_␣%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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\%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

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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\td\n", i);
9      }
10     printf("i_\tvaut_\td\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_\td\n", i);
9      }
10     printf("i_vaut_\td\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



Algorithmes : quels outils pour quels problèmes 



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

2. Parcourir/générer des cas



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

2. Parcourir/générer des cas

- **boucle for** (rarement while)



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

2. Parcourir/générer des cas

- **boucle for** (rarement while)

3. Composer des cas

- boucles (parcourir/générer)



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

2. Parcourir/générer des cas

- **boucle for** (rarement while)

3. Composer des cas

- boucles (parcourir/générer)
- **accumulateur** (à initialiser)



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

2. Parcourir/générer des cas

- **boucle for** (rarement while)

3. Composer des cas

- boucles (parcourir/générer)
- **accumulateur** (à initialiser)



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

4. Sélectionner des cas

- boucles (parcourir/générer)
- **if** (sélectionner/traiter)

2. Parcourir/générer des cas

- **boucle for** (rarement while)

3. Composer des cas

- boucles (parcourir/générer)
- **accumulateur** (à initialiser)



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

2. Parcourir/générer des cas

- **boucle for** (rarement while)

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)



Algorithmes : quels outils pour quels problèmes

1. Traiter des cas spécifiques

- **if else** (différencier)
- **#define** constantes symboliques (nommer)

2. Parcourir/générer des cas

- **boucle for** (rarement while)

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)



Démos



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**.



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/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/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.



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)`