# Lecture écriture dans des fichiers : langage C

1. Présentation
2. Écriture
3. Lire
4. Ligne de commandes

# Crédits

- Un cours du site OpenClassRoom

# Crédits

- Un cours du site OpenClassRoom
- Un cours de Anne Canteaut

1 Présentation

2 Écriture

3 Lire

4 Ligne de commandes

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).
- Pour manipuler un fichier, il faut :

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).
- Pour manipuler un fichier, il faut :
  - l'adresse où il est stocké dans la mémoire tampon,

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).
- Pour manipuler un fichier, il faut :
  - l'adresse où il est stocké dans la mémoire tampon,
  - la position de la tête de lecture dans le fichier,

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).
- Pour manipuler un fichier, il faut :
  - l'adresse où il est stocké dans la mémoire tampon,
  - la position de la tête de lecture dans le fichier,
  - le mode d'accès (lecture ou écriture) etc.

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).
- Pour manipuler un fichier, il faut :
  - l'adresse où il est stocké dans la mémoire tampon,
  - la position de la tête de lecture dans le fichier,
  - le mode d'accès (lecture ou écriture) etc.
- Ces informations sont contenues dans une structure `FILE` définie dans `stdio.h`. Cependant, on ne manipule pas directement les objets de ce type, seulement un pointeur sur ces objets.

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).
- Pour manipuler un fichier, il faut :
  - l'adresse où il est stocké dans la mémoire tampon,
  - la position de la tête de lecture dans le fichier,
  - le mode d'accès (lecture ou écriture) etc.
- Ces informations sont contenues dans une structure `FILE` définie dans `stdio.h`. Cependant, on ne manipule pas directement les objets de ce type, seulement un pointeur sur ces objets.
- Un objet de type `FILE *` est appelé un *flot de données* (ou *stream* en anglais).

# Manipuler un fichier

- Les accès à un fichier du disque dur se font par l'intermédiaire d'un *buffer* (ou *mémoire tampon*). Cela permet de limiter le nombre d'accès aux périphériques (disque dur, clé USB etc).
- Pour manipuler un fichier, il faut :
  - l'adresse où il est stocké dans la mémoire tampon,
  - la position de la tête de lecture dans le fichier,
  - le mode d'accès (lecture ou écriture) etc.
- Ces informations sont contenues dans une structure `FILE` définie dans `stdio.h`. Cependant, on ne manipule pas directement les objets de ce type, seulement un pointeur sur ces objets.
- Un objet de type `FILE *` est appelé un *flot de données* (ou *stream* en anglais).
- La fonction qui permet l'accès à un fichier est `fopen`. Elle est décrite dans `stdio.h`.

# Séquence des opérations

1. Ouverture du fichier dans le mode désiré (**read, write**..) avec `fopen`. On récupère un flot de données.

# Séquence des opérations

1. Ouverture du fichier dans le mode désiré (**read, write**..) avec `fopen`. On récupère un flot de données.
2. Vérification du succès de l'ouverture : le flot doit être différent de `NULL`.

# Séquence des opérations

1. Ouverture du fichier dans le mode désiré (**read, write**..) avec `fopen`. On récupère un flot de données.
2. Vérification du succès de l'ouverture : le flot doit être différent de `NULL`.
3. Manipulation proprement dite en cas de succès d'ouverture. On utilise des fonctions décrites plus loin.

# Séquence des opérations

1. Ouverture du fichier dans le mode désiré (**read, write**..) avec `fopen`. On récupère un flot de données.
2. Vérification du succès de l'ouverture : le flot doit être différent de `NULL`.
3. Manipulation proprement dite en cas de succès d'ouverture. On utilise des fonctions décrites plus loin.
4. Annulation de la liaison entre le flot et le fichier par la fonction `fclose`.

# Prototype

```
1 FILE * fopen(const char *  restrict  filename,
2              const char *  restrict  accessMode);
```

On indique un nom de fichier et un type d'accès (lecture/écriture et texte/binaire). On récupère un flot.

# Prototype

```
1 FILE * fopen(const char *  restrict  filename,
2               const char *  restrict  accessMode);
```

On indique un nom de fichier et un type d'accès (lecture/écriture et texte/binaire). On récupère un flot.

- Arguments :

# Prototype

```
1 FILE * fopen(const char * restrict filename,
2               const char * restrict accessMode);
```

On indique un nom de fichier et un type d'accès (lecture/écriture et texte/binaire). On récupère un flot.

- Arguments :

  filename : définit le nom du fichier à ouvrir. On peut spécifier un chemin absolu ou relatif. Pour plus de portabilité, préférer les chemins relatifs.
  En fonction du système d'exploitation considéré, le nom du fichier sera sensible à la casse (différences entre minuscules et majuscules) ou non.

# Prototype

```
1 FILE * fopen(const char * restrict filename,
2               const char * restrict accessMode);
```

On indique un nom de fichier et un type d'accès (lecture/écriture et texte/binaire). On récupère un flot.

- Arguments :

  filename : définit le nom du fichier à ouvrir. On peut spécifier un chemin absolu ou relatif. Pour plus de portabilité, préférer les chemins relatifs.
  En fonction du système d'exploitation considéré, le nom du fichier sera sensible à la casse (différences entre minuscules et majuscules) ou non.

  accessMode : mode d'ouverture du fichier.

# Note

On précise le sens du mot clé `restrict` utilisé dans le prototype de la fonction `fopen`.

- A partir du standard C99, le mot clé `restrict` est une déclaration d'intention faite par le programmeur pour le compilateur au moment de la déclaration d'un pointeur.

# Note

On précise le sens du mot clé `restrict` utilisé dans le prototype de la fonction `fopen`.

- A partir du standard C99, le mot clé `restrict` est une déclaration d'intention faite par le programmeur pour le compilateur au moment de la déclaration d'un pointeur.
- Il indique que, pour la durée de vie du programme, seul le pointeur (ou une valeur qui en est issue directement comme `pointeur+1`) sera utilisé pour accéder à l'objet sur lequel il pointe.

# Note

On précise le sens du mot clé `restrict` utilisé dans le prototype de la fonction `fopen`.

- A partir du standard C99, le mot clé `restrict` est une déclaration d'intention faite par le programmeur pour le compilateur au moment de la déclaration d'un pointeur.
- Il indique que, pour la durée de vie du programme, seul le pointeur (ou une valeur qui en est issue directement comme `pointeur+1`) sera utilisé pour accéder à l'objet sur lequel il pointe.
- Une telle déclaration peut permettre au compilateur d'optimiser le code.

# Note

On précise le sens du mot clé `restrict` utilisé dans le prototype de la fonction `fopen`.

- A partir du standard C99, le mot clé `restrict` est une déclaration d'intention faite par le programmeur pour le compilateur au moment de la déclaration d'un pointeur.
- Il indique que, pour la durée de vie du programme, seul le pointeur (ou une valeur qui en est issue directement comme `pointeur+1`) sera utilisé pour accéder à l'objet sur lequel il pointe.
- Une telle déclaration peut permettre au compilateur d'optimiser le code.
- Si la déclaration d'intention n'est pas respectée, le comportement du compilateur est *Undefined Behaviour*.

# Note

On précise le sens du mot clé `restrict` utilisé dans le prototype de la fonction `fopen`.

- A partir du standard C99, le mot clé `restrict` est une déclaration d'intention faite par le programmeur pour le compilateur au moment de la déclaration d'un pointeur.
- Il indique que, pour la durée de vie du programme, seul le pointeur (ou une valeur qui en est issue directement comme `pointeur+1`) sera utilisé pour accéder à l'objet sur lequel il pointe.
- Une telle déclaration peut permettre au compilateur d'optimiser le code.
- Si la déclaration d'intention n'est pas respectée, le comportement du compilateur est *Undefined Behaviour*.
- C'est au programmeur de s'assurer que la déclaration d'intention est bien respectée, pas au compilateur.

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"`).

r : ouverture d'un fichier texte en lecture,

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"`).

r : ouverture d'un fichier texte en lecture,

w : ouverture d'un fichier texte en écriture avec écrasement,

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"`).

- r : ouverture d'un fichier texte en lecture,
- w : ouverture d'un fichier texte en écriture avec écrasement,
- a : ouverture d'un fichier texte en écriture à la fin,

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"`).

- r : ouverture d'un fichier texte en lecture,
- w : ouverture d'un fichier texte en écriture avec écrasement,
- a : ouverture d'un fichier texte en écriture à la fin,
- rb,wb,ab : ouverture d'un fichier binaire en lecture, écriture ou ajout

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"`).

- r : ouverture d'un fichier texte en lecture,
- w : ouverture d'un fichier texte en écriture avec écrasement,
- a : ouverture d'un fichier texte en écriture à la fin,
- rb,wb,ab : ouverture d'un fichier binaire en lecture, écriture ou ajout
- r+ : ouverture d'un fichier texte en lecture/écriture (équivalent à **w+**),

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"`).

- r : ouverture d'un fichier texte en lecture,
- w : ouverture d'un fichier texte en écriture avec écrasement,
- a : ouverture d'un fichier texte en écriture à la fin,
- rb,wb,ab : ouverture d'un fichier binaire en lecture, écriture ou ajout
- r+ : ouverture d'un fichier texte en lecture/écriture (équivalent à **w+**),
- a+ ouverture d'un fichier texte en lecture/ et écriture à la fin,

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"` ).

- r : ouverture d'un fichier texte en lecture,
- w : ouverture d'un fichier texte en écriture avec écrasement,
- a : ouverture d'un fichier texte en écriture à la fin,
- rb,wb,ab : ouverture d'un fichier binaire en lecture, écriture ou ajout
- r+ : ouverture d'un fichier texte en lecture/écriture (équivalent à **w+**),
- a+ ouverture d'un fichier texte en lecture/ et écriture à la fin,
- r+b ouverture d'un fichier binaire en lecture/écriture (équivalent à **w+b**).

# Modes d'ouverture

Il faut faire figurer ces modes entre guillements (par exemple `"r"`).

- r : ouverture d'un fichier texte en lecture,
- w : ouverture d'un fichier texte en écriture avec écrasement,
- a : ouverture d'un fichier texte en écriture à la fin,
- rb,wb,ab : ouverture d'un fichier binaire en lecture, écriture ou ajout
- r+ : ouverture d'un fichier texte en lecture/écriture (équivalent à **w+**),
- a+ ouverture d'un fichier texte en lecture/ et écriture à la fin,
- r+b ouverture d'un fichier binaire en lecture/écriture (équivalent à **w+b**).
- a+b : à votre avis ?

# Modes d'ouverture (suite)

Si le mode contient :

- la lettre `r`, le fichier doit exister.

# Modes d'ouverture (suite)

Si le mode contient :

- la lettre `r`, le fichier doit exister.
- la lettre `w`, le fichier peut ne pas exister. Dans ce cas, il est créé. Si le fichier existe déjà, son ancien contenu sera perdu.

# Modes d'ouverture (suite)

Si le mode contient :

- la lettre `r`, le fichier doit exister.
- la lettre `w`, le fichier peut ne pas exister. Dans ce cas, il est créé. Si le fichier existe déjà, son ancien contenu sera perdu.
- la lettre `a`, le fichier peut ne pas exister. Dans ce cas, il est créé. Si le fichier existe déjà, les nouvelles données sont ajoutées à la fin du fichier.

# Flots standard

Trois flots standard peuvent être utilisés en **C** sans qu'il soit nécessaire de les ouvrir ou de les fermer :

stdin : (standard input) : unité d'entrée (par défaut, le clavier),

stdout : unité de sortie (par défaut, l'écran),

stderr : unité d'affichage des messages d'erreur (par défaut, l'écran).

**Remarque**

- On peut rédéfinir ces flots.

# Flots standard

Trois flots standard peuvent être utilisés en **C** sans qu'il soit nécessaire de les ouvrir ou de les fermer :

stdin : (standard input) : unité d'entrée (par défaut, le clavier),

stdout : unité de sortie (par défaut, l'écran),

stderr : unité d'affichage des messages d'erreur (par défaut, l'écran).

**Remarque**

- On peut rédéfinir ces flots.
- Le plus souvent, on ne modifie pas `stderr` pour pouvoir lire à l'écran les messages d'erreurs et les mises en garde (WARNING).

# Fermeture

- La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1        int  fclose ( FILE * stream );
2
```

# Fermeture

- La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1        int  fclose ( FILE * stream );
2
```

- Lors de la fermeture d'un flot :

# Fermeture

- La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1    int  fclose ( FILE * stream );
2
```

- Lors de la fermeture d'un flot :
  - le buffer en mémoire associé est automatiquement synchronisé (**fflush** si le flot est ouvert en écriture), et est libéré automatiquement (ce n'est pas le programmeur qui le libère).

# Fermeture

- La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1    int  fclose ( FILE * stream );
2
```

- Lors de la fermeture d'un flot :
  - le buffer en mémoire associé est automatiquement synchronisé (**fflush** si le flot est ouvert en écriture), et est libéré automatiquement (ce n'est pas le programmeur qui le libère).
  - Les opérations d'écritures sont souvent enregistrées en mémoire-tampon puis toutes effectuées dans le fichier cible à la fermeture du flot

# Fermeture

- La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1    int  fclose ( FILE * stream );
2
```

- Lors de la fermeture d'un flot :
  - le buffer en mémoire associé est automatiquement synchronisé (**fflush** si le flot est ouvert en écriture), et est libéré automatiquement (ce n'est pas le programmeur qui le libère).
  - Les opérations d'écritures sont souvent enregistrées en mémoire-tampon puis toutes effectuées dans le fichier cible à la fermeture du flot
- Si la fermeture se déroule

# Fermeture

- La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1    int  fclose ( FILE * stream );
2
```

- Lors de la fermeture d'un flot :
  - le buffer en mémoire associé est automatiquement synchronisé (**fflush** si le flot est ouvert en écriture), et est libéré automatiquement (ce n'est pas le programmeur qui le libère).
  - Les opérations d'écritures sont souvent enregistrées en mémoire-tampon puis toutes effectuées dans le fichier cible à la fermeture du flot
- Si la fermeture se déroule
  - sans erreur : la valeur **0** est retournée.

# Fermeture

- La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1    int  fclose ( FILE * stream );
2
```

- Lors de la fermeture d'un flot :
  - le buffer en mémoire associé est automatiquement synchronisé (**fflush** si le flot est ouvert en écriture), et est libéré automatiquement (ce n'est pas le programmeur qui le libère).
  - Les opérations d'écritures sont souvent enregistrées en mémoire-tampon puis toutes effectuées dans le fichier cible à la fermeture du flot
- Si la fermeture se déroule
  - sans erreur : la valeur **0** est retournée.
  - avec erreur : la constante **EOF** de **stdlib.h** est retournée.
    Du fait de l'utilisation de **EOF**, il est utile d'importer **stdlib.h**.

1 Présentation

2 Écriture

3 Lire

4 Ligne de commandes

# 3 fonctions d'écriture

**fputc** : écrit un caractère dans le fichier (UN SEUL caractère à la fois),

**fputs** : écrit une chaîne dans le fichier ;

**fprintf** : écrit une chaîne ≪ formatée ≫ dans le fichier, fonctionnement quasi-identique à `printf`.

Seule `fprintf` est explicitement au programme.

# Écriture par chaîne formatée

```
int  fprintf (FILE * restrict stream,
              const char * restrict format, ... );
```

Écrit une chaîne de caractères formatée sur un flot. La chaîne passée en argument n'est pas modifiée. Les arguments de **printf** sont d'abord le flot, puis la chaîne à formater et enfin ses arguments.

```
int main(void)
{
    FILE* flot = fopen("test3.txt", "w");
    if ( flot != NULL)
    {
      int a=1, b=2, c=5;
       fprintf ( flot ,"Est-ce que %d+%d=%d?\n",a,b,c);
       fclose ( flot );    }
    else  printf ("PB\n"); }

```

Après compilation puis exécution du pgm ; la lecture de **test3.txt** donne :

```
Est-ce que 1+2=5?
```

# Sortie standard et **printf**

- `printf(...)` écrit sur le flux de sortie standard **stdout**.

# Sortie standard et **printf**

- `printf(...)` écrit sur le flux de sortie standard **stdout**.
- `printf(...)` est équivalent à `fprintf(stdout,...)`

1 Présentation

2 Écriture

3 Lire

4 Ligne de commandes

# Lecture par chaîne formatée

- Prototype :

```
1   int fscanf( FILE * restrict stream,
2   const char * restrict format, ... );
3
```

# Lecture par chaîne formatée

- Prototype :

```
1  int fscanf( FILE * restrict stream,
2  const char * restrict format, ... );
3
```

stream : représente le flot sur lequel les extractions de valeurs devront être réalisées.

# Lecture par chaîne formatée

- Prototype :

```
1  int fscanf( FILE * restrict stream,
2  const char * restrict format, ... );
3
```

stream : représente le flot sur lequel les extractions de valeurs devront être réalisées.

format : représente le format à utiliser pour décoder la chaîne de caractères.

# Lecture par chaîne formatée

- Prototype :

```
1  int fscanf( FILE * restrict stream,
2  const char * restrict format, ... );
3
```

stream : représente le flot sur lequel les extractions de valeurs devront être réalisées.

format : représente le format à utiliser pour décoder la chaîne de caractères.

- Les descripteurs de format sont communs à **scanf** ou **fscanf**.

# Lecture par chaîne formatée

- Prototype :

```
1  int fscanf( FILE * restrict stream,
2  const char * restrict format, ... );
3
```

stream : représente le flot sur lequel les extractions de valeurs devront être réalisées.

format : représente le format à utiliser pour décoder la chaîne de caractères.

- Les descripteurs de format sont communs à **scanf** ou **fscanf**.
- Cette fonction `fscanf` s'emploie souvent lorsque le fichier à lire est écrit selon des règles précises (voir plus loin).

# Lecture par chaîne formatée

- Prototype :

```
1   int fscanf( FILE * restrict stream,
2   const char * restrict format, ... );
3
```

stream : représente le flot sur lequel les extractions de valeurs devront être réalisées.

format : représente le format à utiliser pour décoder la chaîne de caractères.

- Les descripteurs de format sont communs à **scanf** ou **fscanf**.
- Cette fonction `fscanf` s'emploie souvent lorsque le fichier à lire est écrit selon des règles précises (voir plus loin).
- Valeur de retour : le nombre de paramètres qui ont pu être extraits ou bien `EOF`.

# Exemple

Considérons un fichier **coordonnées.txt** dans le répertoire courant. Chaque ligne de ce fichier contient le nom d'un point du plan suivi de ses 2 coordonnées :

```
P1 10.3 2.1
P2 13.2 12.0
```

Le code ci-dessous boucle sur toutes les lignes du fichier et récupère le nom, la première puis la seconde coordonnée du point décrit. Ces valeurs sont mises en contenu d'une chaîne de caractères et de deux variables décimales.

# Exemple

```
1  int main(void){
2    char chaine[100];
3    float x, y;
4      FILE* flot = fopen("coordonnées.txt", "r");
5      if (flot != NULL){
6        while (fscanf(flot,"%s %g %g",chaine,&x,&y)!=EOF){
7          //%g pour optimiser l'affichage des float
8          printf("point %s, x=%g, y=%g\n",chaine,x,y);
9        }
10       fclose(flot);
11     }//fin if
12     else printf("PB\n");
13   }//fin main
14
```

Après compilation et exécution, on obtient

```
point P1, x=10.3, y=2.1
point P2, x=13.2, y=12
```

# Entrée standard et **scanf**

- `scanf(...)` lit sur le flux d'entrée standard **stdin**.

# Entrée standard et **scanf**

- `scanf(...)` lit sur le flux d'entrée standard **stdin**.
- `scanf(...)` est équivalent à `fscanf(stdin,...)`

1 Présentation

2 Écriture

3 Lire

4 Ligne de commandes

# Arguments d'un programme

- Les arguments de la ligne de commande d'un programme C sont passés à la fonction `main` sous forme de deux paramètres : un entier donnant le nombre d'éléments sur la ligne de commande ; un tableau de chaînes de caractères.

```
1 int main(int argc, char**argv) {...
2
```

# Arguments d'un programme

- Les arguments de la ligne de commande d'un programme C sont passés à la fonction `main` sous forme de deux paramètres : un entier donnant le nombre d'éléments sur la ligne de commande ; un tableau de chaînes de caractères.

```
1 int main(int argc, char**argv) {...
2
```

- Le nom du programme est le premier argument de la ligne de commande.

# Arguments d'un programme

- Les arguments de la ligne de commande d'un programme C sont passés à la fonction `main` sous forme de deux paramètres : un entier donnant le nombre d'éléments sur la ligne de commande ; un tableau de chaînes de caractères.

```
1 int main(int argc, char**argv) {...
2
```

- Le nom du programme est le premier argument de la ligne de commande.
- Les arguments sont exclusivement des chaînes de caractères. Si on prévoit de prendre des arguments numériques, il faut utiliser une fonction comme `atoi` (**ASCII to integer**) :

```
1 #include <stdlib.h>
2 int atoi( const char * theString );
```

Transforme une chaîne de caractères, représentant une valeur entière, en une valeur numérique de type `int`.

# Exemple

Le programme suivant lit deux entiers en ligne de commande, affiche son propre nom, les entiers saisis et enfin leur produit.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char **argv){
  assert(argc == 3);
  printf("programme %s\n",argv[0]);//nom du pgm
  int a = atoi(argv[1]), b= atoi(argv[2]);//entiers saisis
  int p = a*b;
  printf("%d * %d = %d\n", a, b, p);
  return EXIT_SUCCESS;
}
```

**Attention** : la fonction `atoi` retourne 0 si la chaîne de caractères ne contient pas une représentation de valeur numérique. Il n'est donc pas possible de distinguer la chaîne "0" d'une chaîne ne contenant pas un nombre entier. Préférer `strtol` dans ce cas.

# Exemple

## Compilation puis exécution

- On compile avec `gcc -Wall commande.c -o produit`

# Exemple

## Compilation puis exécution

- On compile avec `gcc -Wall commande.c -o produit`
- Exécution :

```
$ ./produit 10 3
programme ./produit
10 * 3 = 30
```