

# Ch. 3 - Fonctions

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant  
École supérieure d'Informatique



6 octobre 2021

# Table des matières

1 Introduction

2 Passage d'argument

3 Fonctions inline

# Table des matières

1 Introduction

2 Passage d'argument

3 Fonctions inline

# Table des matières

- 1 Introduction
- 2 Passage d'argument
- 3 Fonctions inline

# Introduction

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
- Augmente la lisibilité

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
- Augmente la lisibilité

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité



# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité

# Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

## Avantages

- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)



# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)

# Caractéristiques

- 1 Possède des paramètres et un retour
  - `sqrt` prend en paramètre un flottant et retourne un flottant
- 2 En C, identifiées par leur nom uniquement
  - Pas de surdéfinition possible
- 3 En C, toutes les fonctions sont dites « indépendantes »
  - Déclarées en dehors de toute classe
- 4 Plus qu'une fonction mathématique
  - Effectue un travail
  - Possibilité de modifier les paramètres
  - Peut ne rien retourner (`void`)



# Illustration

## ■ Fichier surdef.c

```
1 int f(int i)
2 {
3     printf("Integer_%d\n", i);
4     return 0;
5 }
6
7 int f(double d) //you can't overload
8 {
9     printf("Double_%f\n", d);
10    return 0;
11 }
12
13 int f(int i, int j) //you still can't, even with more params
14 {
15     printf("Integers_%d_and_%d\n", i, j);
16     return 0;
17 }
```

# Déclaration et définition

- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire
- Seul les types des paramètres sont nécessaires dans le prototype
- Pour rappel, les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc

# Déclaration et définition

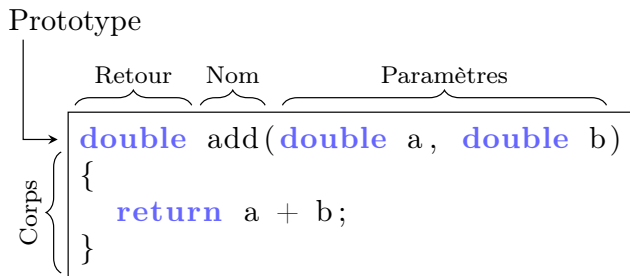
- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire
- Seul les types des paramètres sont nécessaires dans le prototype
- Pour rappel, les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc

# Déclaration et définition

- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire
- Seul les types des paramètres sont nécessaires dans le prototype
- Pour rappel, les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc

# Déclaration et définition

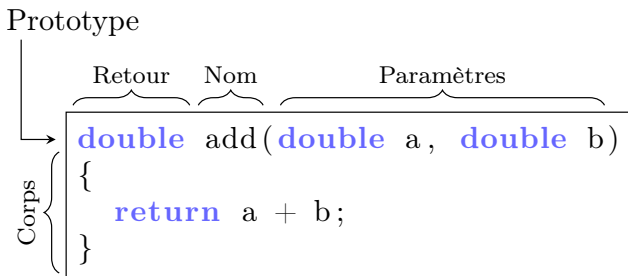
- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire



- Seul les types des paramètres sont nécessaires dans le prototype
- Pour rappel, les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc

# Déclaration et définition

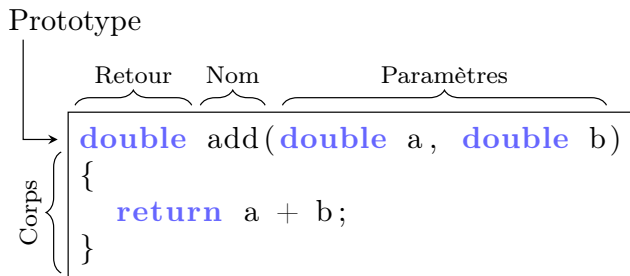
- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire



- Seul les types des paramètres sont nécessaires dans le prototype
- Pour rappel, les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc

# Déclaration et définition

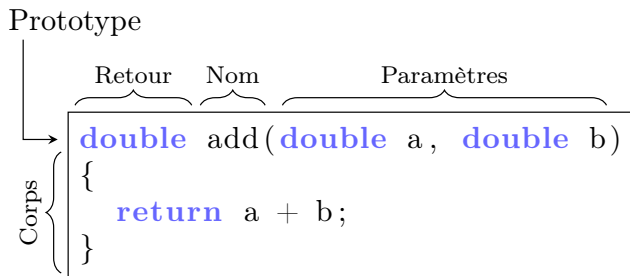
- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire



- Seul les types des paramètres sont nécessaires dans le prototype
- Pour rappel, les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc

# Déclaration et définition

- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire



- Seul les types des paramètres sont nécessaires dans le prototype
- Pour rappel, les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc



# Exemple

## ■ Fichier `before.c`

```
1  int main()  
2  {  
3      print("Hello");  
4  }  
5  
6  void print(const char* s)  
7  {  
8      printf("%s\n", s);  
9  }
```

## ■ Déclaration anticipée

```
1  void print(const char*);  
2  
3  int main()  
4  {  
5      print("Hello");  
6  }  
7  
8  void print(const char* s)  
9  {  
10     printf("%s\n", s);  
11 }
```

## ■ Même principe en C++

# Les fonctions sans arguments

- En C uniquement, si on veut qu'une fonction n'accepte aucun argument, il faut écrire `void` dans la liste des paramètres
- Si on déclare `void f() ;`, la fonction `f` accepte un nombre arbitraire d'arguments (et les ignore)

```
1 void f() {}  
2 void g(void) {}  
3  
4 int main()  
5 {  
6     f();  
7     f(1); //ok  
8     f(1,2); //ok  
9  
10    g();  
11    //g(1); //ko  
12 }
```

# Passage d'argument

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de copie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites



# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

- Performances réduites

# Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. ?)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

## Avantages

- Pas d'effet de bord

## Inconvénients

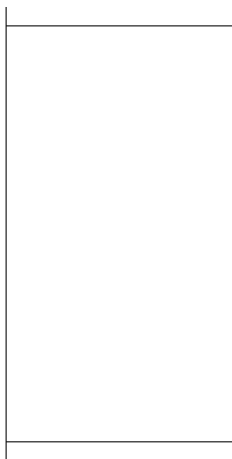
- Performances réduites

# Mécanisme

```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Pile



# Mécanisme

```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Pile

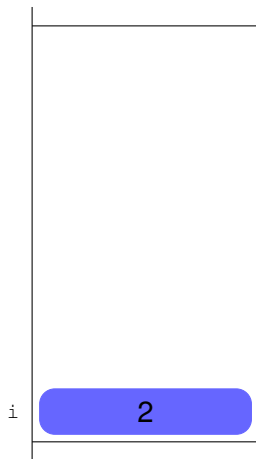


# Mécanisme

```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Pile



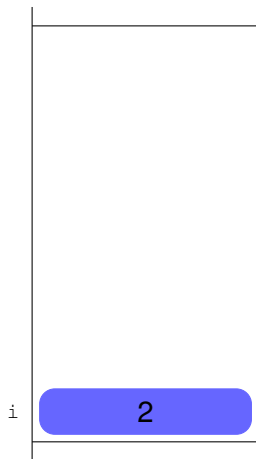


# Mécanisme

```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Pile



# Mécanisme

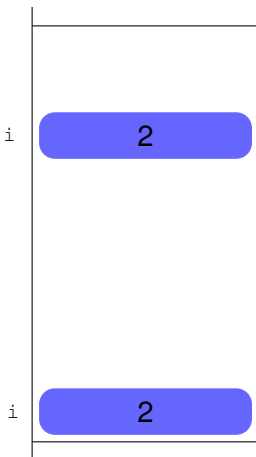


```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Copie de i

Pile



# Mécanisme

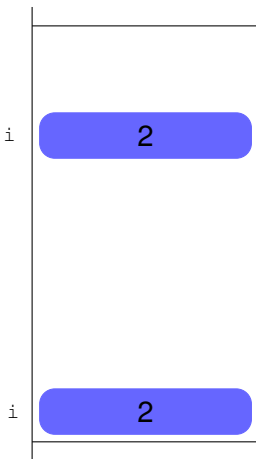


```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Copie de i

Pile



# Mécanisme

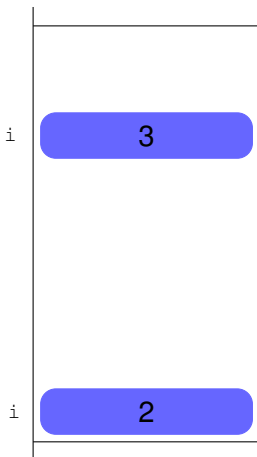


```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Copie de i

Pile



# Mécanisme

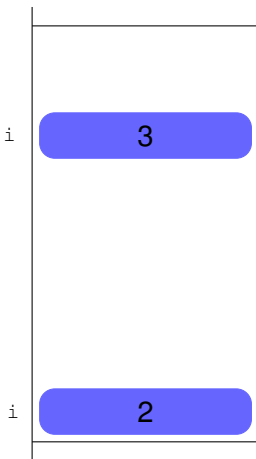


```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Copie de i

Pile



# Mécanisme

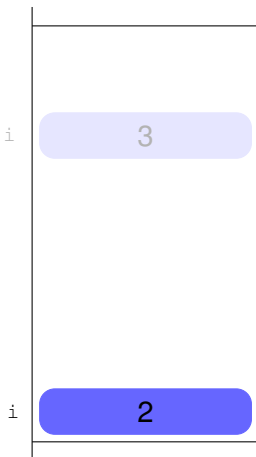


```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```

Copie de i

Pile



# Mécanisme

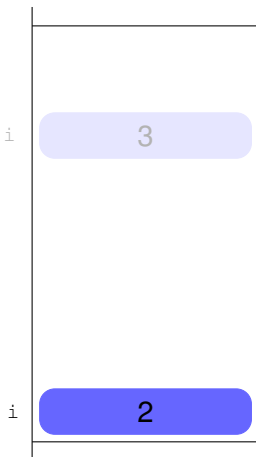
```
void f(int i)
{
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main()
{
    int i = 2;
    f(i);
    printf("%d\n", i);
}
```



Copie de i

Pile



# Mauvais swap

## ■ Fichier swap-value.c

```
1 void swap(int x, int y)
2 {
3     printf("Entering_swap_:_%d_%d\n", x, y);
4
5     int tmp = y;
6     y = x;
7     x = tmp;
8
9     printf("Exiting_swap_:_%d_%d\n", x, y);
10 }
11
12 int main()
13 {
14     int x = 2;
15     int y = 3;
16
17     printf("Before_call_:_%d_%d\n", x, y);
18     swap(x, y);
19     printf("After_call_:_%d_%d\n", x, y);
20 }
```



# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déferencé est passé par adresse
  - Le pointeur est passé par valeur

# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déferencé est passé par adresse
  - Le pointeur est passé par valeur

# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déferencé est passé par adresse
  - Le pointeur est passé par valeur

# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déferencé est passé par adresse
  - Le pointeur est passé par valeur

# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déferencé est passé par adresse
  - Le pointeur est passé par valeur

# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déferencé est passé par adresse
  - Le pointeur est passé par valeur

# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déferencé est passé par adresse
  - Le pointeur est passé par valeur

# Passage par adresse

- On ne transmet pas une du paramètre, mais son adresse
  - « Comme en Java » pour les objets
- Permet d'émuler un passage par référence du C++
  - Utiliser des pointeurs a des inconvénients, comparé aux références
  - En C pur, pas d'autre solution
- On fournit un pointeur à la fonction
  - Le paramètre déférencé est passé par adresse
  - Le pointeur est passé par valeur

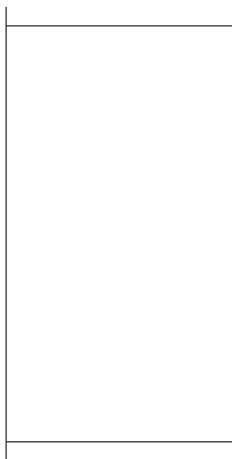


# Mécanisme

```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```

Pile

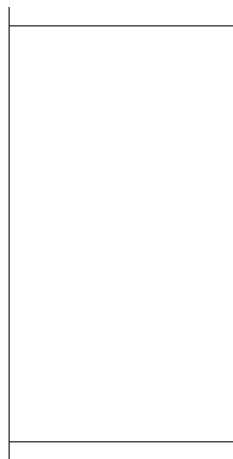


# Mécanisme

```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```

Pile



# Mécanisme

```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```



0xCAFE : i

Pile

2

# Mécanisme

```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```



0xCAF6 : &i

0xCAFE : i

Pile

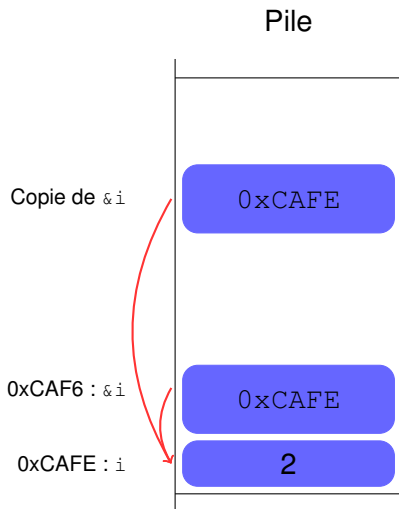


# Mécanisme




```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```

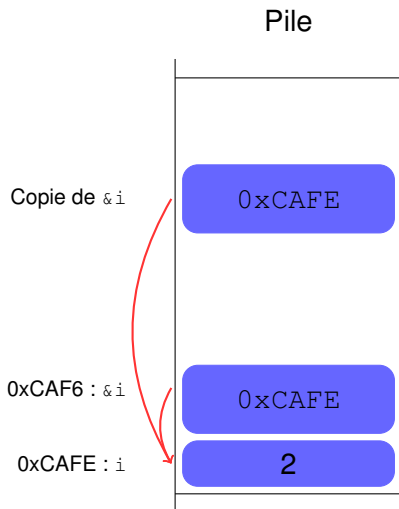


# Mécanisme



```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```



# Mécanisme



```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```

Copie de &i

0xCAFE6 : &i

0xCAFE : i

Pile

0xCAFE

0xCAFE

3

# Mécanisme



```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```

Copie de &i

0xCAF6 : &i

0xCAFE : i

Pile


0xCAFE

0xCAFE

3



# Mécanisme



```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}
```

```
int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```

Copie de &i

0xCAF6 : &i

0xCAFE : i

Pile

0xCAFE

0xCAFE

3

# Mécanisme

```
void f(int * i)
{
    printf("%d\n", *i);
    (*i)++;
    printf("%d\n", *i);
}

int main()
{
    int i = 2;
    f(&i);
    printf("%d\n", i);
}
```



Copie de &i

0xCAF6 : &i

0xCAFE : i

Pile

0xCAFE

0xCAFE

3

# Exemple

## ■ Fichier swap-addr-wrong.c

```
1 void swap(int * ptx, int * pty)
2 {
3     printf("Entering_swap: %d %d\n", *ptx, *pty);
4
5     int* tmp = pty;
6     pty = ptx;
7     ptx = tmp;
8
9     printf("Exiting_swap: %d %d\n", *ptx, *pty);
10 }
11
12 int main()
13 {
14     int x = 2;
15     int y = 3;
16
17     printf("Before_call: %d %d\n", x, y);
18     swap(&x, &y);
19     printf("After_call: %d %d\n", x, y);
20 }
```

# Exemple

## ■ Fichier swap-addr.c

```
1 void swap(int * ptx, int * pty)
2 {
3     printf("Entering_swap: %d %d\n", *ptx, *pty);
4
5     int tmp = *pty;
6     *pty = *ptx;
7     *ptx = tmp;
8
9     printf("Exiting_swap: %d %d\n", *ptx, *pty);
10 }
11
12 int main()
13 {
14     int x = 2;
15     int y = 3;
16
17     printf("Before_call: %d %d\n", x, y);
18     swap(&x, &y);
19     printf("After_call: %d %d\n", x, y);
20 }
```

# Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f()` ;
  - par adresse : `int* f()` ;

## Attention

- Ne créez pas de pointeurs vers des temporaires
- Ils vont « pendouiller » (dangling)

# Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f () ;`
  - par adresse : `int * f () ;`

## Attention

- Ne créez pas de pointeurs vers des temporaires
- Ils vont « pendouiller » (dangling)

# Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f () ;`
  - par adresse : `int * f () ;`

## Attention

- Ne créez pas de pointeurs vers des temporaires
- Ils vont « pendouiller » (dangling)

# Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f () ;`
  - par adresse : `int * f () ;`

## Attention

- Ne créez pas de pointeurs vers des temporaires
- Ils vont « pendouiller » (dangling)



# Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f () ;`
  - par adresse : `int * f () ;`

## Attention

- Ne créez pas de pointeurs vers des temporaires
  - Ils vont « pendouiller » (dangling)

# Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f () ;`
  - par adresse : `int * f () ;`

## Attention

- Ne créez pas de pointeurs vers des temporaires
- Ils vont « pendouiller » (dangling)

# Illustration

## ■ Fichier return.c

```
1  int f() //ok
2  {
3      int i = 42;
4      return i;
5  }
6
7  int* g() //wrong
8  {
9      int i = 42;
10     int * pti = &i;
11     return pti;
12 }
13
14 void h(int * pt) //wrong
15 {
16     int i = 42;
17     pt = &i;
18 }
19
20 void hh(int * pt) //also wrong
21 {
22     int i = 42;
23     *pt = i;
24 }
```

# Fonctions inline

# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`

# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`

# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`

# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`



# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`

# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`

# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`

# Fonctions inline

- Fonction dont le corps est substitué à l'appel

## Avantages

- Gain de temps (pas de `call`)

## Inconvénients

- Exécutable grossit (copier / coller)
- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse
- Déclarée avec le mot-clé `inline`

# Exemple

## ■ Fichier inline.c

```
1 void for_each(int* array, unsigned n, int (*f)(int))
2 {
3     for(unsigned i = 0; i < n; i++)
4         array[i] = f(array[i]);
5 }
6
7 int plus_one(int i)
8 {
9     return i + 1;
10 }
11
12 inline int plus_two(int i)
13 {
14     return i + 2;
15 }
16
17 int main()
18 {
19     int array[] = {1,2,3,4,5};
20     for_each(array, 5, plus_one);
21     for(unsigned i = 0; i < 5; i++)
22         printf("%d_", array[i]);
23     printf("\n");
24
25     for_each(array, 5, plus_two); //wrong
26 }
```