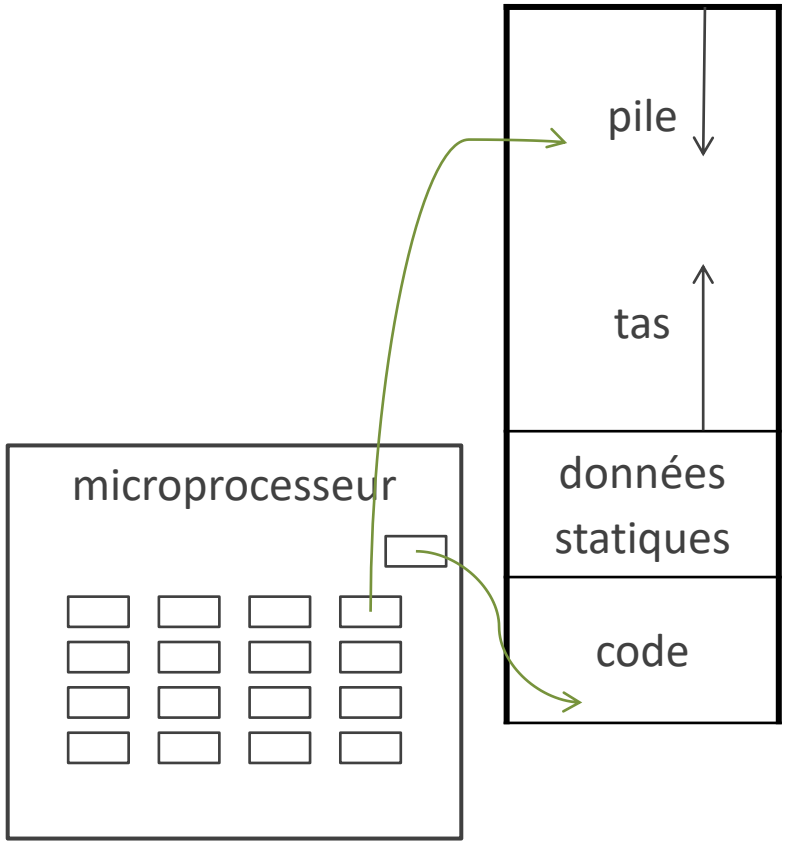
Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Moteurs d'exécution

Moteurs d'exécution (*runtime systems*)



Ce que fait le système d'exploitation pour exécuter un programme en binaire

Charger le programme

Copier en mémoire - l'exécutable
- les arguments en ligne de commande

Initialiser les registres : - registre de pile
- compteur ordinal

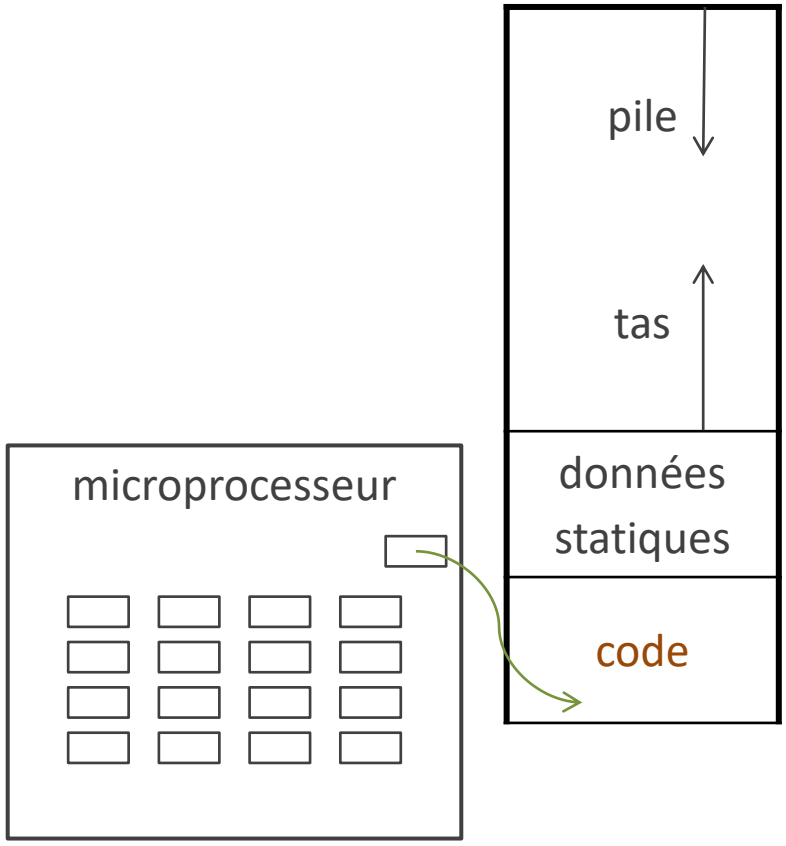
Passer d'une instruction à une autre

Incrémenter le compteur ordinal

Interface avec le système d'exploitation

Lire, écrire dans des fichiers

Statique ou dynamique



Statique

Connu à la compilation

Dynamique

Dépend de chaque exécution

Le code du programme

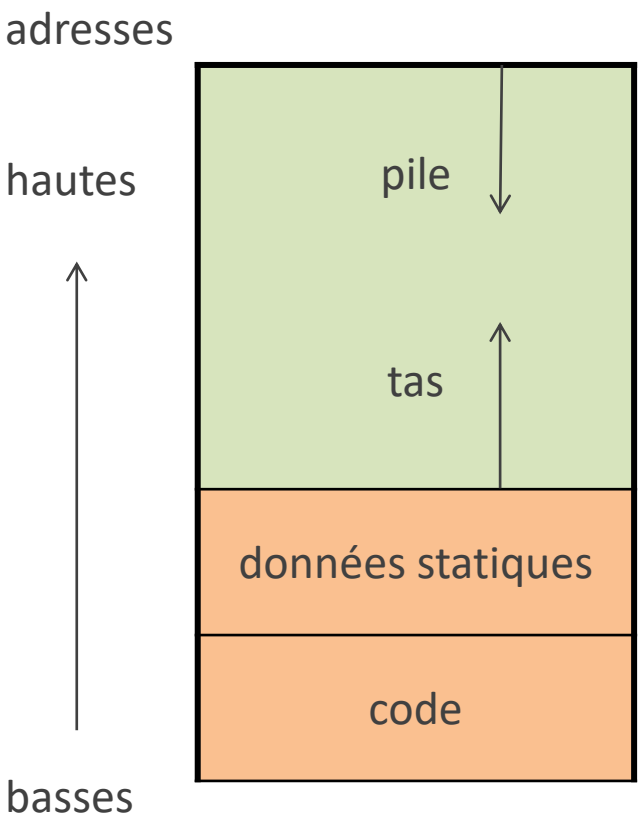
Liste d'instructions

Ne change pas pendant l'exécution

Le compteur ordinal

Passe d'une instruction à une autre

Organisation de la mémoire



Partie statique

Durée de vie : toute l'exécution du programme

Données statiques : dont les adresses sont en dur dans le code

Partie dynamique

Durée de vie : une partie de l'exécution du programme

Pile d'exécution

Le tas est l'emplacement de toutes les autres informations : en C, la mémoire allouée dynamiquement

Lire au clavier
Afficher

Lire au clavier un entier décimal

initialiser [number] à 0
lire un caractère dans [digit]
tant que [digit] est un chiffre :
    soustraire 48 de [digit]
    multiplier [number] par 10
    ajouter [digit] à [number]

Afficher un entier décimal

initialiser i à imax
faire :
    diviser [number] par 10
    copier le quotient dans [number]
    ajouter 48 au reste
    copier le résultat dans [digits+i]
    décrémenter i
tant que [number] est > 0
écrire [digits+i+1] ... [digits+imax]
écrire 10

Lire ou afficher des caractères

Appel système de `nasm` avec 0 (lire) ou 1 (écrire)
    dans `rax` et `rdi`

Lire ou afficher un entier décimal

Décomposer chiffre par chiffre

Gérer `+` et `-`

Attention, `syscall` change `rcx` et `r11`

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Activations

Fonction

A un nom (identificateur) et un corps (instruction)

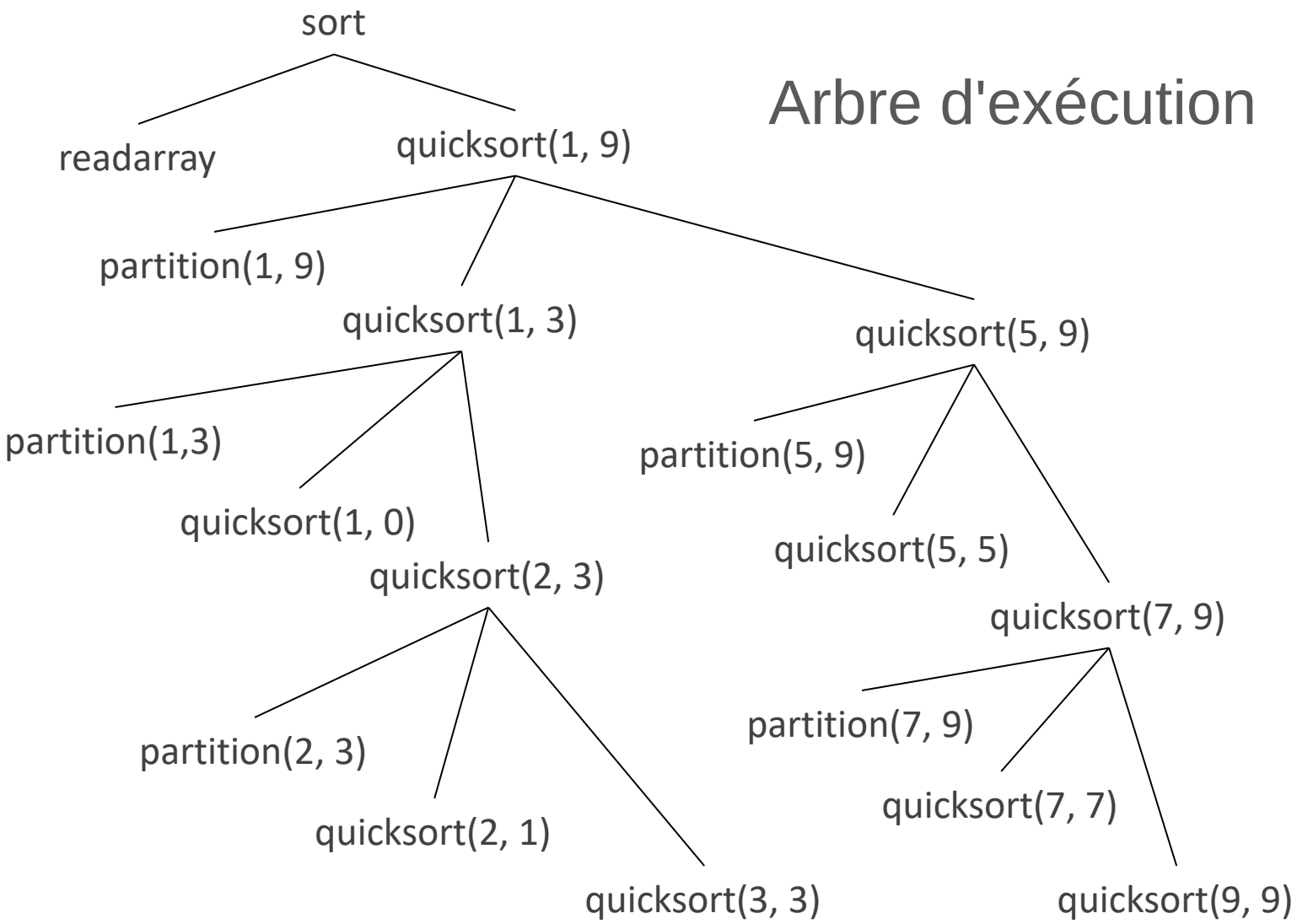
Activation d'une fonction

Période qui va de l'appel d'une fonction (avec éventuellement des paramètres) à son retour

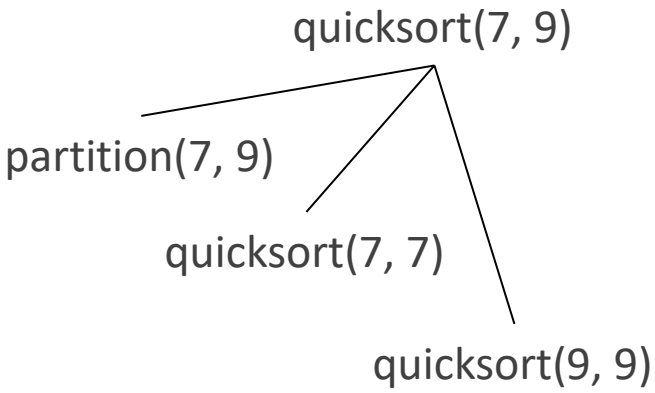
Fonction récursive

Fonction dont une activation peut commencer alors qu'une autre est en cours

Arbre d'exécution



Pile d'exécution



Arbre d'exécution

Les nœuds sont les activations des fonctions avec leurs paramètres

Chaque fils d'un nœud correspond à un appel

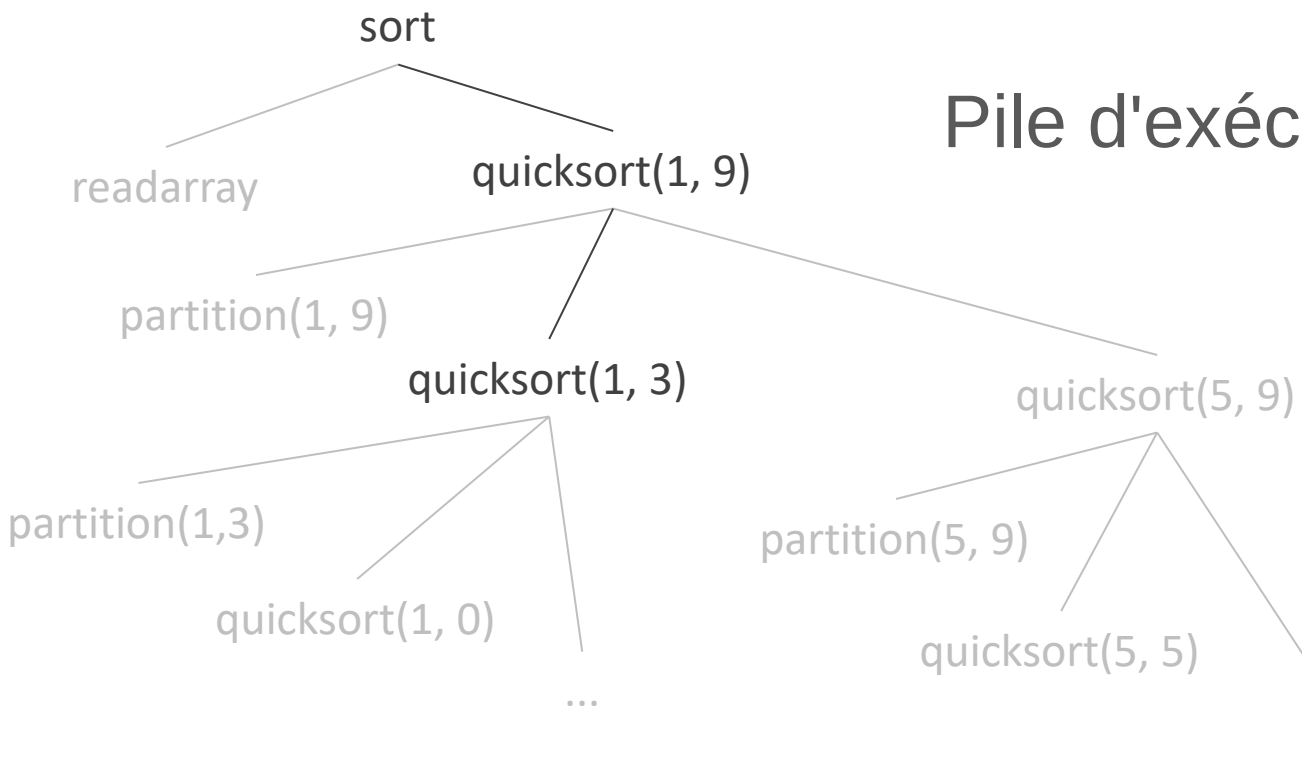
Pile d'exécution

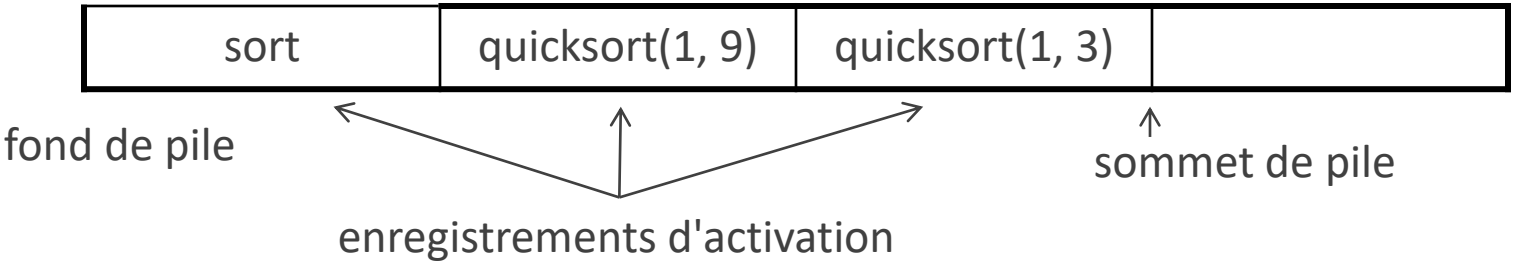
La pile des activations en cours à un instant donné

L'instruction en cours d'exécution fait partie de la fonction dont l'**enregistrement d'activation** est en sommet de pile

Les autres activations en cours sont suspendues

Pile d'exécution





Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Différences entre langages

Langages

C, Java, C++, Ada, Python, Perl

Pascal, Lisp, Fortran, Algol, APL, Snobol...

Un langage sans fonctions récursives : Fortran 77

Connaissant ce que fait le moteur d'exécution, on peut traduire un langage de haut niveau en langage de bas niveau

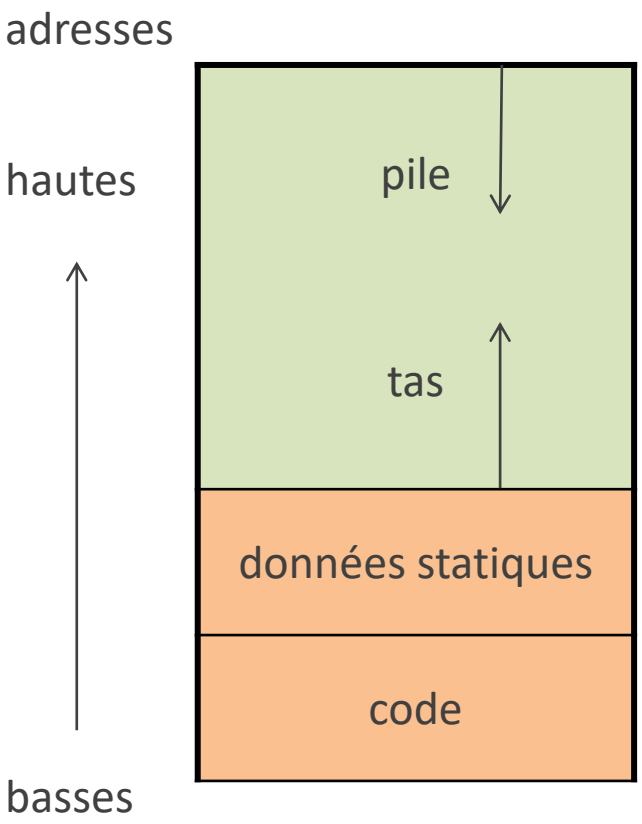
L'organisation d'un compilateur dépend des caractéristiques du langage source

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Organisation de la mémoire



Allocation statique

Allocation dynamique

Sur demande explicite dans le source

Allocation automatique

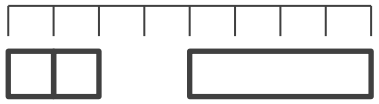
À l'entrée des blocs

Les adresses dans la pile sont des adresses relatives (*offsets*) par rapport à une certaine position dans la pile (base)

Alignement en mémoire

Avec gcc sous Linux

machine	type	long	long long	float	double	long double
32 bits	taille (octets)	4	8	4	8	12
	alignement	4	4	4	4	4
64 bits	taille	8	8	4	8	16
	alignement	8	8	4	8	16

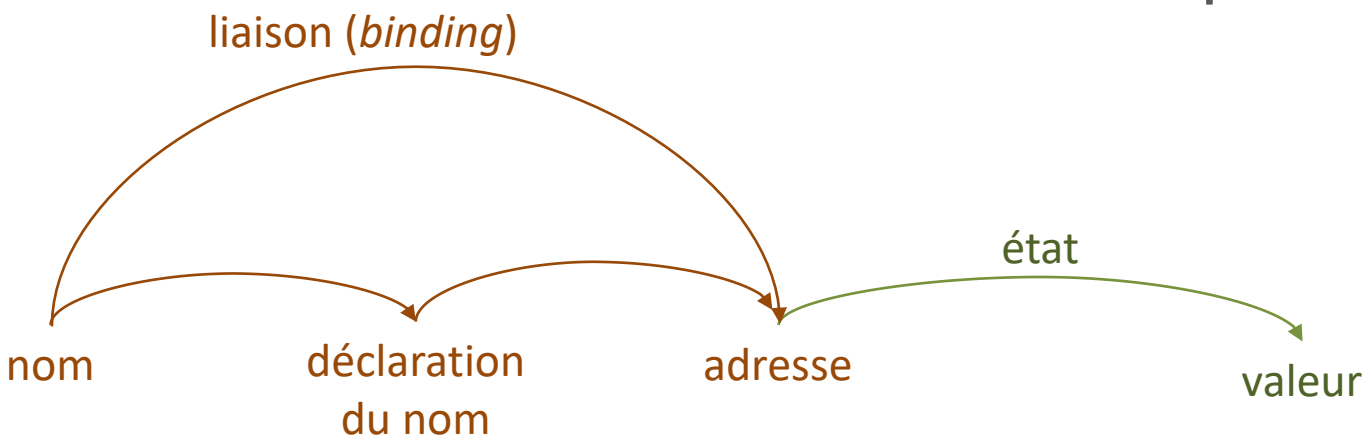


Alignement : l'adresse d'une donnée doit être divisible par un certain entier

Cela peut obliger le compilateur à laisser du remplissage (*padding*)

L'alignement dépend aussi du compilateur

Allocation statique



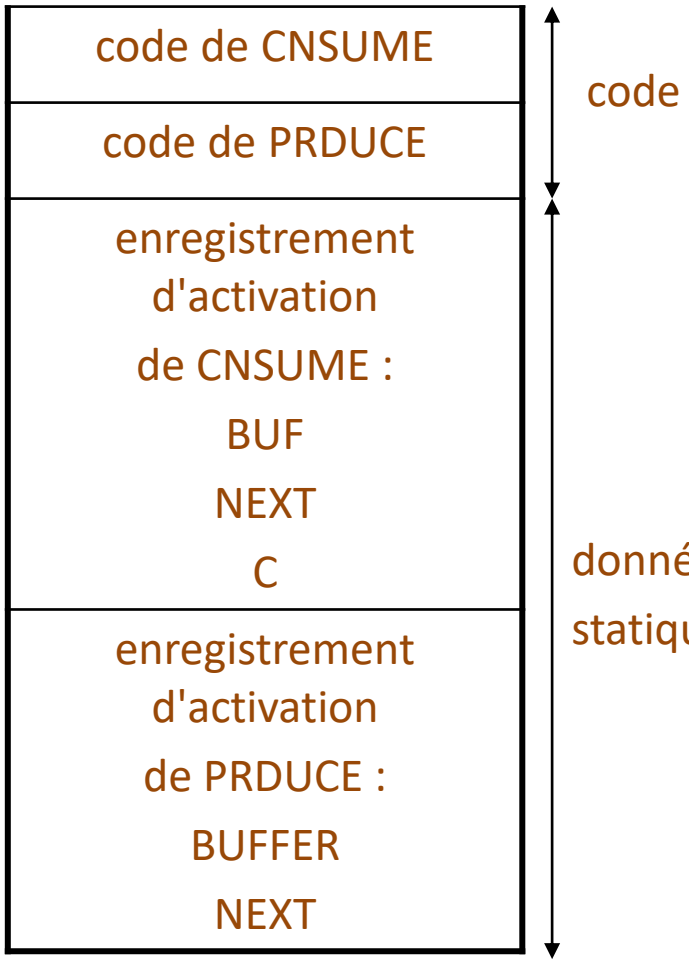
L'adresse est connue à la compilation

Si l'environnement associe une occurrence du nom *id* à l'adresse *a*, on dit que *id* est lié à *a*

La valeur stockée à une adresse dépend de chaque exécution

Une adresse peut contenir plusieurs valeurs successivement

Allocation exclusivement statique (Fortran 77)



La stratégie d'allocation mémoire la plus simple

Toutes les adresses sont statiques (constantes pendant toute l'exécution)

L'emplacement des enregistrements d'activation est statique

Les valeurs des noms locaux peuvent persister d'un appel au suivant (déclaration SAVE)

Limitations

La taille de toutes les données est statique

La récursivité est impossible

L'allocation dynamique est impossible

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Allocation dynamique

En langage C, indépendante de la compilation

Utilise le tas

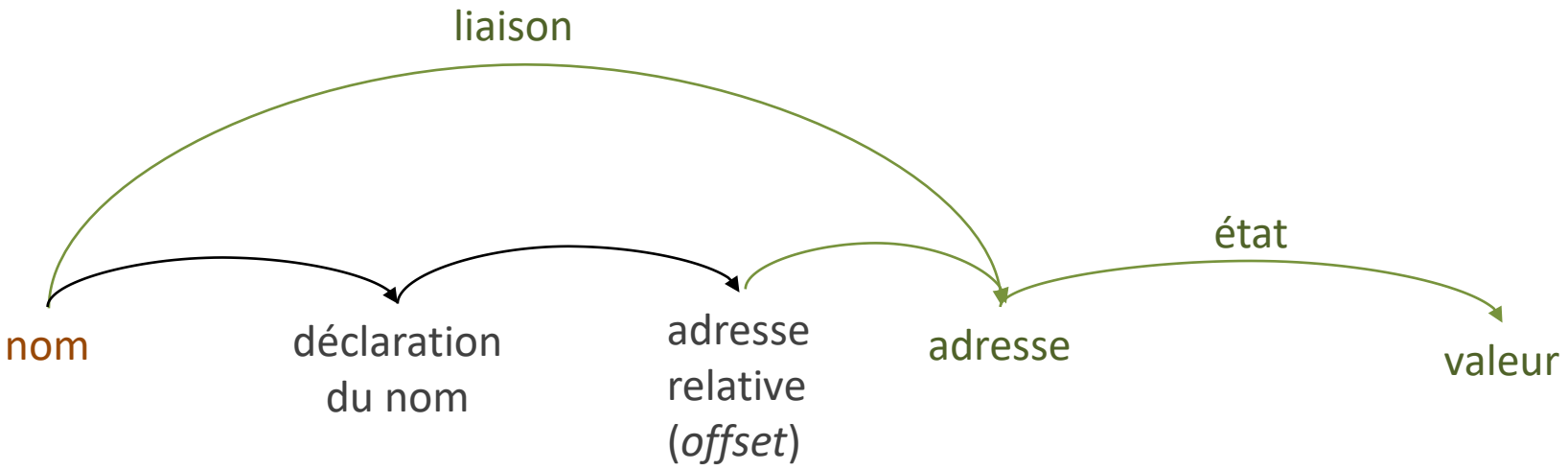
Allocation dynamique
Différences entre langages

Ada Lovelace par Margaret Sarah Carpenter
http://www.gac.culture.gov.uk/work.aspx?obj=28401
Public Domain, https://commons.wikimedia.org/w/index.php?curid=28993870

Libération de la mémoire allouée dynamiquement

- manuelle : C
- automatique (ramasse-miettes, *garbage collector*) : Java, Python, Perl
- manuelle ou automatique : Ada

Allocation automatique



Si l'environnement associe une occurrence du nom *id* à l'adresse *a*, on dit que *id* est lié à *a*

Un nom déclaré plusieurs fois peut être lié à plusieurs adresses relatives

Exemple : dans plusieurs fonctions

Une occurrence d'un nom peut être liée à plusieurs adresses

Exemple : fonction récursive

Différences entre langages

Déclaration globale

Valable par défaut dans tout le source et pendant toute l'exécution

Déclaration locale

Valable par défaut dans une fonction ou un bloc

Nom local

Déclaré dans le bloc où il apparait

La valeur d'un nom local peut-elle être conservée au retour de la fonction ?

En C et C++ : oui si le nom est local statique

En Java et Ada : non

Quels noms non locaux sont valables dans une fonction ?

En C : les noms globaux

En Ada : les noms valables dans la fonction englobante

En Lisp : les noms valables dans l'activation appelante

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Accès aux noms non locaux

Dépend des règles de **portée** (*scope*) des variables dans le langage source

Les règles qui relient les **occurrences** des variables à leurs déclarations

Portée lexicale ou statique

C, Java, Ada, Pascal

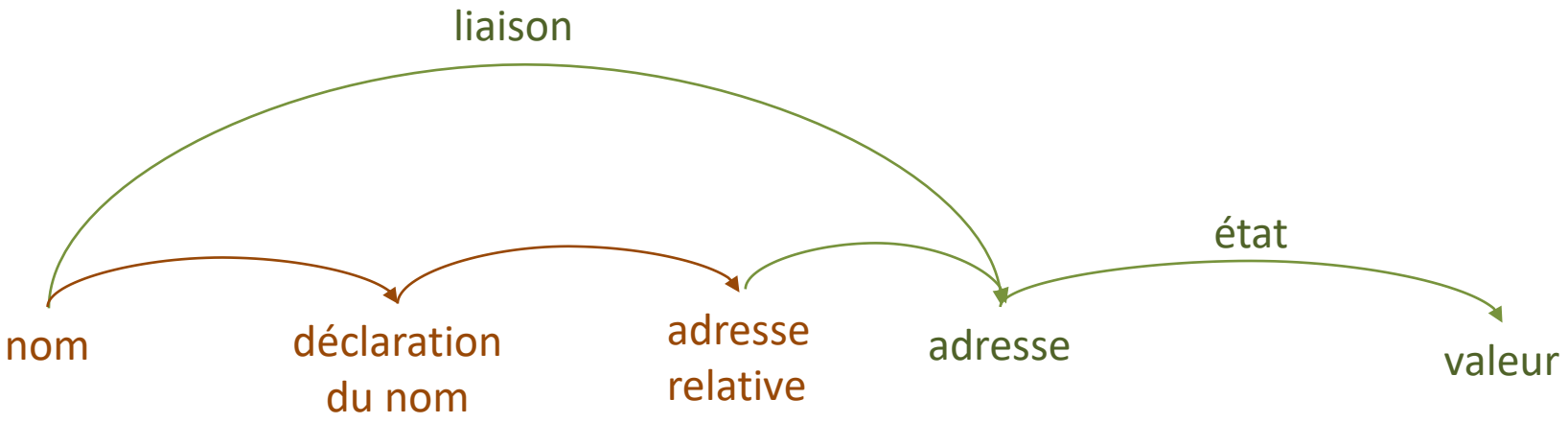
Portée déterminée par le code source du programme

Portée dynamique

Lisp, APL, Snobol

Portée déterminée par les activations en cours

Portée lexicale ou statique



L'adresse relative est connue à la compilation

Structure de blocs

```
main()
{
    int a = 0 ;
    int b = 0 ;
    {
        int b = 1 ;
        {
            int a = 2 ;
            printf("%d %d", a, b) ;
        }
        {
            int b = 3 ;
            printf("%d %d", a, b) ;
        }
        printf("%d %d", a, b) ;
    }
    printf("%d %d", a, b) ;
}
```

B_0 B_1 B_2 B_3

Bloc --> **{** *Déclarations Instructions* **}**

Les instructions peuvent contenir des blocs

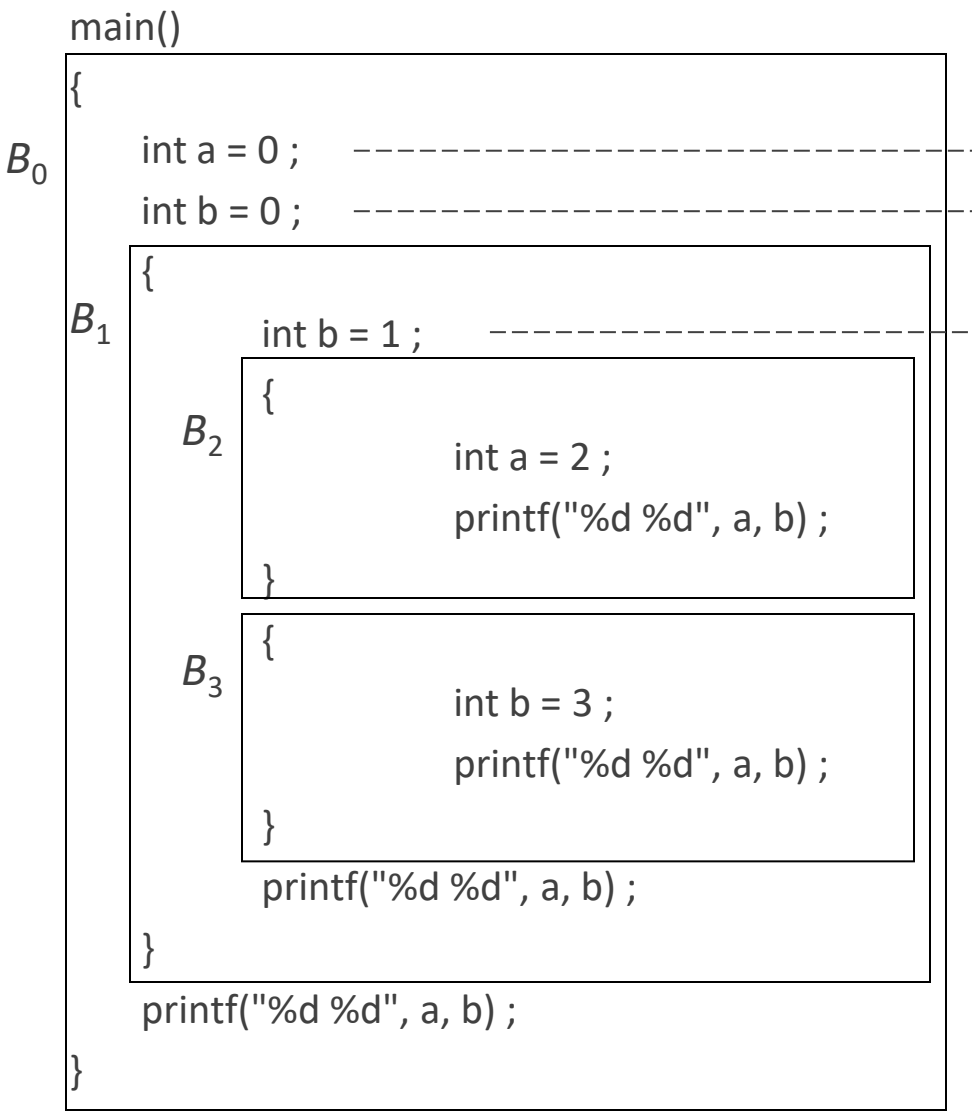
Une fonction peut contenir plusieurs blocs

Portée lexicale ou statique

La portée d'une déclaration faite au début de *B* est incluse dans *B*

Si un nom *x* dans *B* n'est pas déclaré au début de *B* (non local à *B*), une occurrence de *x* dans *B* est liée à la déclaration de *x* au début du plus petit bloc *B'* tel que

- *B'* contient *B*
- *x* est déclaré dans *B'*



Portée

B_0 sauf B_2

B_0 sauf B_1

B_1 sauf B_3

Résultat

2 1

0 3

0 1

0 0

Université Gustave Eiffel — INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Portée lexicale ou statique

```
main()
{
    int a = 0 ;
    int b = 0 ;
    {
        int b = 1 ;
        {
            int a = 2 ;
            printf("%d %d", a, b) ;
        }
        {
            int b = 3 ;
            printf("%d %d", a, b) ;
        }
        printf("%d %d", a, b) ;
    }
    printf("%d %d", a, b) ;
}
```

B_0 : int a = 0 ; int b = 0 ;

B_1 : int b = 1 ;

B_2 : int a = 2 ; printf("%d %d", a, b) ;

B_3 : int b = 3 ; printf("%d %d", a, b) ;

Implémentation facile quand toutes les déclarations locales sont au début d'une fonction

Sinon : deux méthodes

Allocation par bloc

On traite chaque bloc comme une fonction sans paramètres ni valeur de retour

Allocation par fonction

On regroupe les variables déclarées dans toute la fonction

On peut lier à une même adresse deux variables dont les portées sont disjointes (a de B_2 et b de B_3)

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Portée lexicale sans fonctions emboîtées

Exemple : le langage C

Les noms sont de deux sortes :

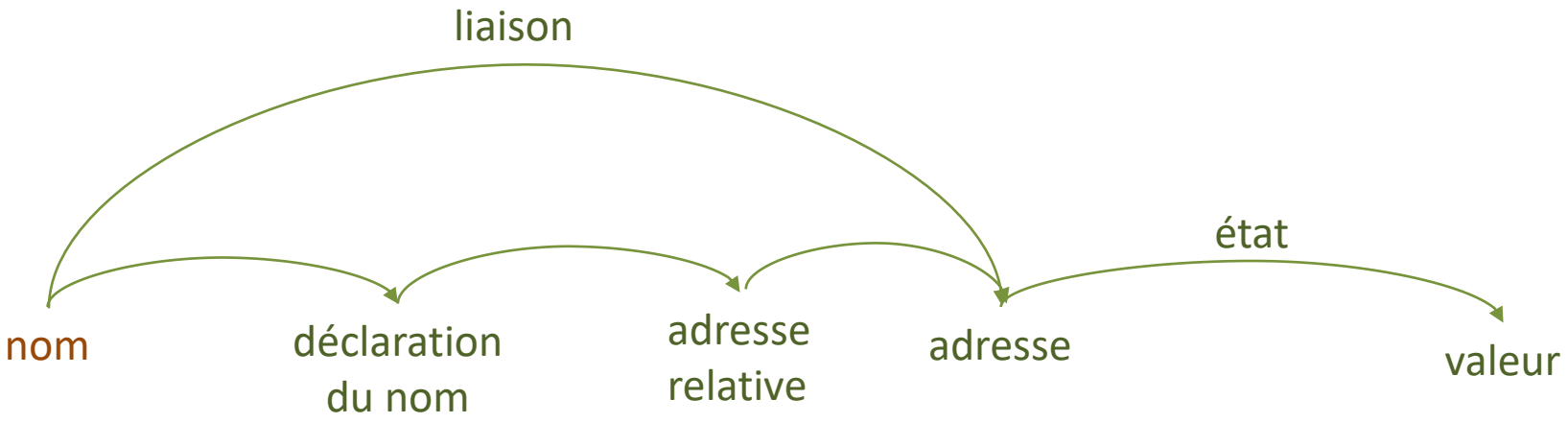
- locaux à une fonction ou à un bloc
- globaux, déclarés hors des fonctions

Les noms globaux sont liés à des adresses statiques

Les noms locaux sont liés à des adresses en pile, accessibles à partir du pointeur de base...

sauf les noms locaux statiques

Portée dynamique



L'adresse relative est connue seulement à l'exécution

Une liaison a la même durée de vie que l'activation correspondante

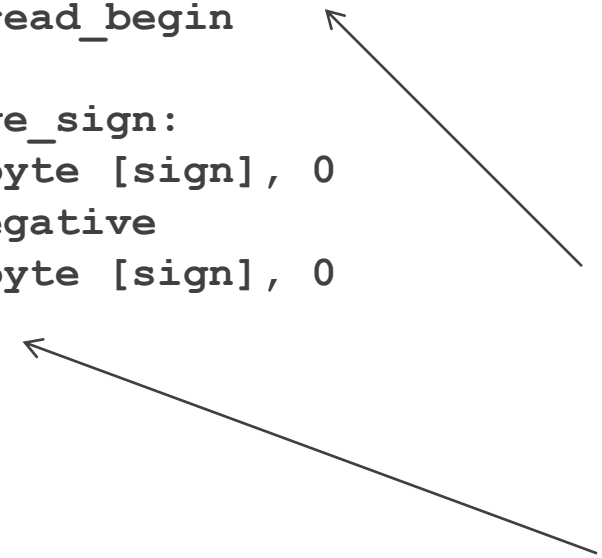
Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

```
cmp cl, 45          ; code for minus sign
jne read_begin      ; not a number, try again
call change_sign
jmp read_begin

change_sign:
cmp byte [sign], 0
je negative
mov byte [sign], 0
ret
```



Appel de fonction

Le moteur d'exécution

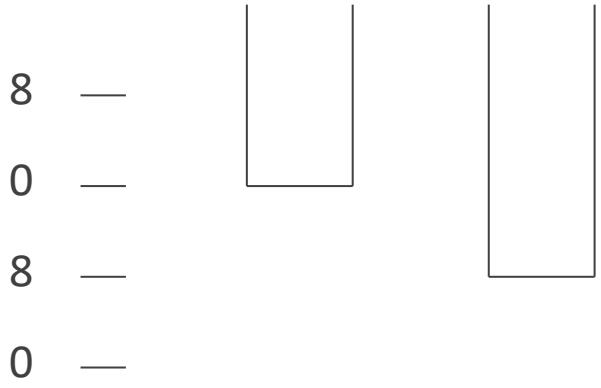
- empile l'adresse de retour
- saute à l'adresse de la fonction appelée

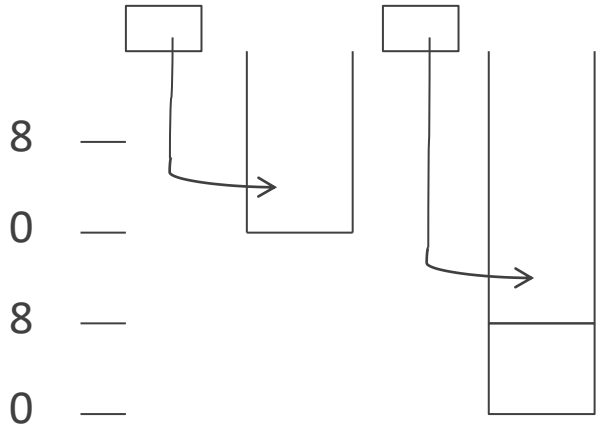
Retour d'une fonction

Le moteur d'exécution

- dépile l'adresse de retour
- saute à l'adresse de retour

Aligner la pile
Sauvegarder le pointeur de pile dans un registre



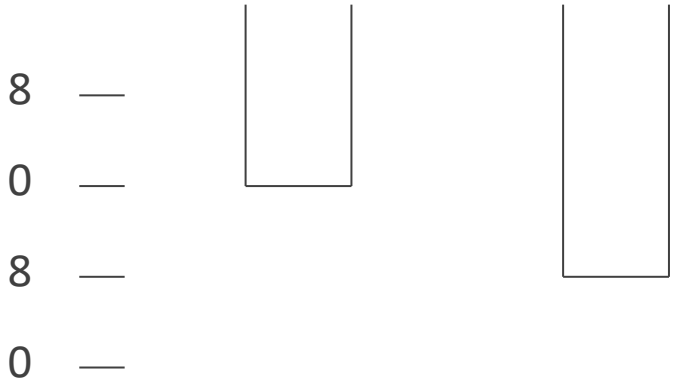


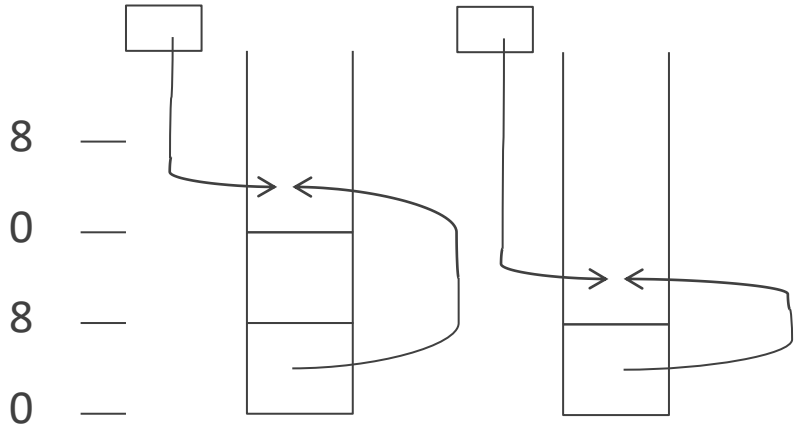
L'adresse du sommet de pile doit être un multiple de 16 avant un appel de fonction

On doit pouvoir rétablir le sommet après le retour

- Sauvegarder `rsp` dans un registre
- Aligner la pile
- Appeler la fonction
- Restaurer `rsp`

Aligner la pile
Sauvegarder le pointeur de pile dans la pile





- Sauvegarder **`rsp`** dans un registre
- Réserver dans la pile de quoi sauvegarder **`rsp`**
- Aligner la pile
- Sauvegarder l'ancien **`rsp`** dans la pile
- Appeler la fonction
- Restaurer **`rsp`**

Paramètres

```
swap(int *x, int *y) {
      int temp ;
      temp = * x ;
      * x = * y ;
      * y = temp ;
}
main() {
      int a = 1, b = 2 ;
      swap(& a, &b) ;
 }
```

Paramètres formels
Dans la définition de la fonction

Paramètres effectifs ou arguments
Dans un appel de la fonction

Passage des paramètres
Dépend des langages
Par valeur : C, Java, Ada
Par référence : Java, Ada
Par copie/restauration : Fortran
Par nom : Macros en C

Passage par valeur

```
swap(int i, int j) {
      int temp ;
      temp = a[i] ;
      a[i] = a[j] ;
      a[j] = temp ;
}
main() {
      swap(1, 2) ;
 }
```

Les valeurs des paramètres effectifs sont passés à la procédure appelée

Exemple : C, C++, Java

Ne modifie pas les valeurs dans l'enregistrement d'activation de l'appelant, sauf à travers des noms non locaux ou des pointeurs passés par valeur

Réalisation

Les paramètres formels sont traités comme des noms locaux

Les paramètres effectifs sont évalués par l'appelant

Passage par valeur

Pascal

```
procedure swap(i, j : integer) ;
var x : integer ;
    begin
    x := a[i] ;
    a[i] := a[j] ;
    a[j] := x
    end
```

C

```
swap(int i, int j) {
     int temp ;
     temp = a[i] ;
     a[i] = a[j] ;
     a[j] = temp ;
}
main() {
     swap(a, 1, 2) ;
 }
```

Passage par référence

```
program reference(input, output) ;
    var a, b : integer ;
    procedure swap(var x, y : integer) ;
            var temp : integer ;
            begin
                    temp := x ;
                    x := y ;
                    y := temp
            end ;
    begin
            a := 1 ;
            b := 2 ;
            swap(a, b)
    end .
```

Les adresses des paramètres effectifs sont passées à la procédure appelée

C++, Java, Ada, Pascal

Exemple en Pascal

Passage par copie et restauration (Fortran)

```
int a;
int main() {
  a=1;
  foo(a);
  printf("%d\n", a);
}
int foo(int x) {
  x=2;
  a=3;
}
```

- Les valeurs des paramètres effectifs sont passées à la procédure appelée
- Au retour, les nouvelles valeurs des paramètres sont copiées à leur adresse

Résultats

par référence	par copie et restauration
3	2

Passage par nom

```
#define intswap(a, b) { int x = a ; a = b ; b = x ; }

intswap(i, tab[i])
```

substitution :

```
{ int x = i ; i = tab[i] ; tab[i] = x ; }
```

équivalent à :

```
y=tab[i];
tab[tab[i]]=i;
i=y;
```

Passage par nom

Le corps de la fonction est substitué à l'appel

Les paramètres sont substitués littéralement

Exemples

Macros en C

Algol

Fonctions passées en paramètre Différences entre langages

Un langage où on ne peut pas

- passer une fonction en paramètre d'une autre
- renvoyer une fonction comme valeur d'une autre :

APL

```
m ← +/ (3 + ⍳ 4)
m
```

Fonctions passées en paramètre en **nasm**

```
#include <stdio.h>
int m ;
int f(int n) { return m + n ; }
int g(int n) { return m * n ; }
int b(int (*h)(int)) { printf("%\n", h(2)) ; }
int main(void) {
      m = 0 ;
      b(f) ; b(g) ; }
/* ou b(&f); b(&g); */
```

Comment passer une fonction en paramètre ou la renvoyer comme résultat

Copier l'adresse du code (l'étiquette)

Comment le moteur d'exécution appelle une fonction passée en paramètre

- il empile l'adresse de retour
- il saute à l'adresse du code (l'étiquette)

Fonctions passées en paramètre en C

```
#include <stdio.h>
int m ;
int f(int n) { return m + n ; }
int g(int n) { return m * n ; }
int b(int (*h)(int)) { printf("%\n", h(2)) ; }
int main(void) {
      m = 0 ;
      b(f) ; b(g) ; }
/* ou b(&f); b(&g); */
```

Portée lexicale sans fonctions emboîtées

Comment traduire en code intermédiaire le passage d'une fonction en paramètre ou le renvoi d'une fonction comme valeur de retour

On transmet l'adresse du code