

Algorithmique et Programmation, IMA 3

Cours 6 : Variables Modifiables, Pointeurs

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/>
 Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

d'après A. Miné (ÉNS Ulm)



- 1 Schémas d'exécution - variables modifiables
- 2 Les pointeurs
- 3 Quelques exemples
- 4 Validité des pointeurs
- 5 Et encore ...

Passage de paramètres par valeur

Variables D en langage algorithmique.

Fonction *ajoute_un(a) : Entier*

D: a : Entier

Retourner a+1

FFonction

Appel :

Programme Main

L: x,y :Entiers

x ← 12

y ← ajoute_un(x)

Imprimer(y)

Retourner 0

FProgramme

► Que se passe-t-il lors de l'appel de *ajoute_un* ?

Passage de paramètres par adresse

Variables R ou D/R en langage algorithmique.

Exemple :

Action *inc(x)*

D/R: x : Entier

x ← x+1

FAction

{Donnée/Résultat}

Que fait la suite d'instructions suivante :

y : Entier ;

y ← 100 ;

inc(y) ; inc(y) ;

Que vaut y à la fin ?

► Que se passe-t-il lors de l'appel de *inc* ?

Modèle mémoire simplifié

Pour expliquer : **Mémoire** \simeq tableau d'octets.

Chaque octet a une **adresse** en mémoire.

► Chaque variable **déclarée** a une adresse (début du placement mémoire).

type	int	int	char	int
nom	x	y	c	z
adresse	3A00	3A04	3A08	3A40
valeur	10	42	'a'

► La place en mémoire dépend du **type** de la variable (1 octet pour un char, 4 pour un int (ou 8), ...)

Schéma d'exécution d'une fonction

Fonction *ajoute_un(a) : Entier*

D: a : Entier

Retourner a+1

FFonction

Appel :

Programme *Main*

... y \leftarrow ajoute_un(x)

...

FProgramme

► Dessin !

Schéma d'exécution d'action

Action *inc(x)*

D/R: x : Entier

x \leftarrow x+1

FAction

Appel :

y : Entier ;

y \leftarrow 100 ;

inc(y) ;

► Dessin !

Résumé

- un morceau de mémoire indépendant par fonction/action
- les variables locales **n'existent plus** après la fin de l'appel de fonction.
- lors d'un passage par valeur, il y a une copie des **valeurs**
- lors d'un passage par adresse, il y a une copie des **adresses**

En C

- 1 Schémas d'exécution - variables modifiables
- 2 Les pointeurs
- 3 Quelques exemples
- 4 Validité des pointeurs
- 5 Et encore ...

A SAVOIR :

le passage par valeur est le passage standard en C SAUF pour les tableaux qui sont passés par adresse.

► Pour passer un paramètre par adresse, on utilise **les pointeurs**

Les adresses et le C

En C, on peut :

- obtenir l'adresse d'une variable **existante** (&),
 - accéder au contenu stocké à une adresse **valide** (*),
 - passer des adresses en argument, les retourner, les copier (=),
 - effectuer des opérations **limitées** sur les adresses (+, ==, ...).
- C'est l'objet des transparents suivants.

L'opérateur d'adresse &

Obtenir l'adresse d'un objet en mémoire :

&expr

expr doit être une lvalue (i.e., modifiable) **existante !**

- variable scalaire,
- case d'un tableau.

Exemple :

```
int i, a[2];
&i           /* adresse de i */
&(a[1])     &a[1] /* adresse de a[1] */
&a          &(i+1) /* invalides */
scanf("%d",&i) ; /* acces a l'adresse*/
```

Déréférencement *

Accéder au contenu stocké à une adresse **valide** (*)

Si p est une adresse valide, alors $*p$ donne le contenu de la case située à l'adresse p .

```
void inc (int* px)
{
    *px=*px+1
}
```

```
(main)
int y=100;
inc(&y);
```

► De quel type est la variable **px** ?

Les types pointeur - 1

Exemple :

```
int i=4;           /* i est un entier */
int* p;           /* p pointeur d'entier */
p = &i;           /* adresse de i */
```

La **variable pointeur** p contient l'adresse de la variable entière i . On dit souvent :

"p pointe sur i"

Contenu de la mémoire : dessin !

Les types pointeur - 2

Type d'un pointeur

- t^* est un pointeur sur un objet de type t .
- si $expr$ a pour type t , alors $\&expr$ a pour type t^* .

Attention ! Le type pointé est important ! Si $t1 \neq t2$, alors $t1^*$ et $t2^*$ sont incompatibles.

Les variables pointeurs

Maintenant que t^* est un nouveau type, on peut déclarer des variables de type t^* :

$t^* \text{ var}$

var peut contenir l'adresse de tout objet de type t .

var ne peut pas contenir l'adresse d'un objet de type différent !

```
int i;
float f;
int* p ; /* p << pointeur d'entier >> */
p = &i; /* p pointe sur i */
p = &f; /* non defini */
p = &(i+1); /* erreur de syntaxe */
```

Quelques exemples de & et *

- 1 Schémas d'exécution - variables modifiables
- 2 Les pointeurs
- 3 Quelques exemples
- 4 Validité des pointeurs
- 5 Et encore ...

Rappel

- ***p** : contenu de la mémoire à l'adresse p.
- **&x** : adresse de la variable x.

```
int x, y;
int* p = &x;
*p = 2;      /* place 2 dans x */
p = &y;
*p = *p+1;  /* incremente y */
```

Ex : Permutation

```
void permuter (int* px, int* py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf ("%d_ %d", a, b);
```

- Réalisons le **schéma d'exécution** de ce programme.

Ex : Division

```
void divise(int a, int b, int* div, int* rem){
    *div = a / b;
    *rem = a % b;
}
```

```
void f(){
    int x, y;
    divise( 100, 10, &x, &y );
}
```

Attention ! C'est à l'appelant d'allouer la mémoire pour les valeurs de "retour".

Ex : Copies de pointeurs

```
Soit : int x = 1, y = 2;
      int* p = &x; int* q = &y;
```

Que valent x, y, p et q après :

- p = q; *p = -1;
- *p = *q; *p = -1;

?

Ex : Copies de pointeurs

```
Soit : int x = 1, y = 2;
      int* p = &x; int* q = &y;
```

Que valent x, y, p et q après :

- p = q; *p = -1;
p et q pointent sur y, (alias !)
- *p = *q; *p = -1;

?

Ex : Copies de pointeurs

```
Soit : int x = 1, y = 2;
      int* p = &x; int* q = &y;
```

Que valent x, y, p et q après :

- p = q; *p = -1;
p et q pointent sur y, (alias !)
y = -1. x est inchangé.
- *p = *q; *p = -1;

?

Ex : Copies de pointeurs

```
Soit : int x = 1, y = 2;
      int* p = &x; int* q = &y;
```

Que valent x, y, p et q après :

- p = q; *p = -1;
p et q pointent sur y, (alias !)
y = -1. x est inchangé.
- *p = *q; *p = -1;
*q = y = 2 est placé dans *p = x,

?

Ex : Copies de pointeurs

```
Soit : int x = 1, y = 2;
      int* p = &x; int* q = &y;
```

Que valent x, y, p et q après :

- p = q; *p = -1;
p et q pointent sur y, (**alias !**)
y = -1. x est inchangé.
- *p = *q; *p = -1;
*q = y = 2 est placé dans *p = x,
puis -1 est placé dans *p = x. y est inchangé.

!

Et les tableaux

En C, les tableaux sont passés par adresse :

Dans une expression, tout tableau unidimensionnel est remplacé par **un pointeur vers son premier élément**.

Utilité des pointeurs

Les pointeurs peuvent servir à

- passer des variables par adresse,
- simuler le retour de plusieurs valeurs,
- lire des données entrées au clavier **scanf**
- traverser des tableaux (non vu ici, voir TP)
- gérer des blocs de mémoire dynamique (**pas tout de suite**).
- implémenter des listes (chaînées) (cf Semestre 6)
- passer des fonctions en paramètres (cf Prog Avancée)

- 1 Schémas d'exécution - variables modifiables
- 2 Les pointeurs
- 3 Quelques exemples
- 4 Validité des pointeurs
- 5 Et encore ...

Le pointeur NULL

NULL : valeur pointeur spéciale "vide", souvent utilisé pour dire "non défini". C'est souvent 0 mais pas toujours.

Attention Son déréférencement est impossible !

Utilisations standard :

- utilisée comme valeur pour "non définie",
- renvoyée par une fonction pour indiquer une erreur,
- passée en argument pour indiquer qu'on n'est pas intéressé par une valeur de retour.

Le pointeur NULL - ex

Ex d'utilisation :

```
#include <stdlib.h>
void divide(int a, int b, int* div, int* rem)
{
    if (div!=NULL) *div = a / b;
    if (rem!=NULL) *rem = a % b;
}
```

Note, si p est un pointeur :

- if (p) équivaut à if (p!=NULL),
- if (!p) équivaut à if (p==NULL).

Pointeurs valides et invalides

Attention ! Avant de déréférencer un pointeur par *, assurez-vous qu'il pointe vers un objet valide !

Pointeurs **valides** :

- pointeur vers une variable globale,
- pointeur vers une variable locale existante.

Pointeurs **invalides** :

- pointeur NULL ou non initialisé,
- pointeur en dehors des bornes d'un tableau,
- pointeur vers une variable locale détruite,
⇒ ne **jamais** retourner un pointeur vers une variable locale !

Attention : la durée de vie d'une variable–pointeur peut dépasser celle de l'objet sur lequel elle pointe !

Exemples incorrects

```
void g(int* x) {
    *x = 2;
}
```

```
void main() {
    int* z;
    g(z); /* ERREUR: z non initialise */
    {
        int k;
        z = &k;
        g(z); /* equivalent a g(&k) : g modifiera k */
    }
    g(z); /* ERREUR: k n'existe plus, z est invalide */
}
```

Exemples incorrects

```
int* f(){
    int z = 12;
    return &z;
}

void main()
{
    int* x = f();
    *x = 13;      /* ERREUR: z n'existe plus */
}
```

Note : l'adresse d'une variable locale change entre deux appels d'une même fonction !

- 1 Schémas d'exécution - variables modifiables
- 2 Les pointeurs
- 3 Quelques exemples
- 4 Validité des pointeurs
- 5 Et encore ...

Comparaison de pointeurs

On peut comparer deux pointeurs pour :

- l'égalité == (pointent sur la même adresse ?)
- la différence != (pointent sur des adresses différentes ?).

Priorité des opérateurs

Du plus prioritaire au moins prioritaire.

[]	accès dans un tableau
++ --	incrémentations et décréments
*	déréférencement de pointeur
&	prise d'adresse
* / %	opérateurs multiplicatifs
+ -	opérateurs additifs
== < > ...	opérateurs de comparaison
&&	opérateurs booléens
= op =	opérateurs d'affectation

Exemple : *p++ signifie *(p++), pas (*p)++;

► dans le doute : mettre des parenthèses.

Priorité dans les déclarations

Attention à la priorité de * et , dans les déclarations.

- `int *a,b;`
b a pour type `int`, pas `int*`.
- `int *a,*b;`
a et b ont le type `int*`.

Pointeurs complexes

Exemples complexes :

- `int** x;` pointeur sur un pointeur sur un `int`,
`*x` : pointeur sur un `int`,
`**x` : `int`.
- `int *x[10];` tableau de 10 pointeurs sur des `int`,
`x[1]` : pointeur sur un `int`,
`*(x[1])` : `int`.
- `int (*x)[10];` pointeur sur un tableau de 10 `int`.
(inutile : on préférera un pointeur sur un élément du tableau)