

Introduction à la Programmation

Benoit Donnet
Année Académique 2021 - 2022



Agenda

- Introduction
- Chapitre 1: Bloc, Variable, Instruction Simple
- Chapitre 2: Structures de Contrôle
- Chapitre 3: Méthodologie de Développement
- Chapitre 4: Introduction à la complexité
- Chapitre 5: Structures de Données
- Chapitre 6: Modularité du Code
- **Chapitre 7: Pointeurs**
- Chapitre 8: Allocation Dynamique

Agenda

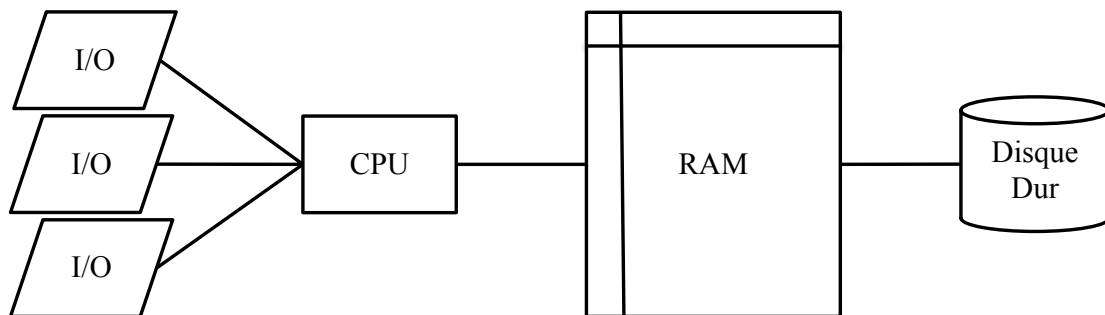
- Chapitre 7: Pointeurs
 - Introduction
 - Passage de Paramètres
 - Pointeurs et Enregistrements
 - Pointeurs et Tableaux

Agenda

- Chapitre 7: Pointeurs
 - Introduction
 - ✓ Mémoire Centrale
 - ✓ Opérations sur les Pointeurs
 - Passage de Paramètres
 - Pointeurs et Enregistrements
 - Pointeurs et Tableaux

Mémoire Centrale

- Architecture classique d'un ordinateur
 - l'ordinateur est composé d'un CPU, d'appareils d'ES, d'une mémoire rapide et volatile (RAM) et d'une mémoire à long terme (HD)

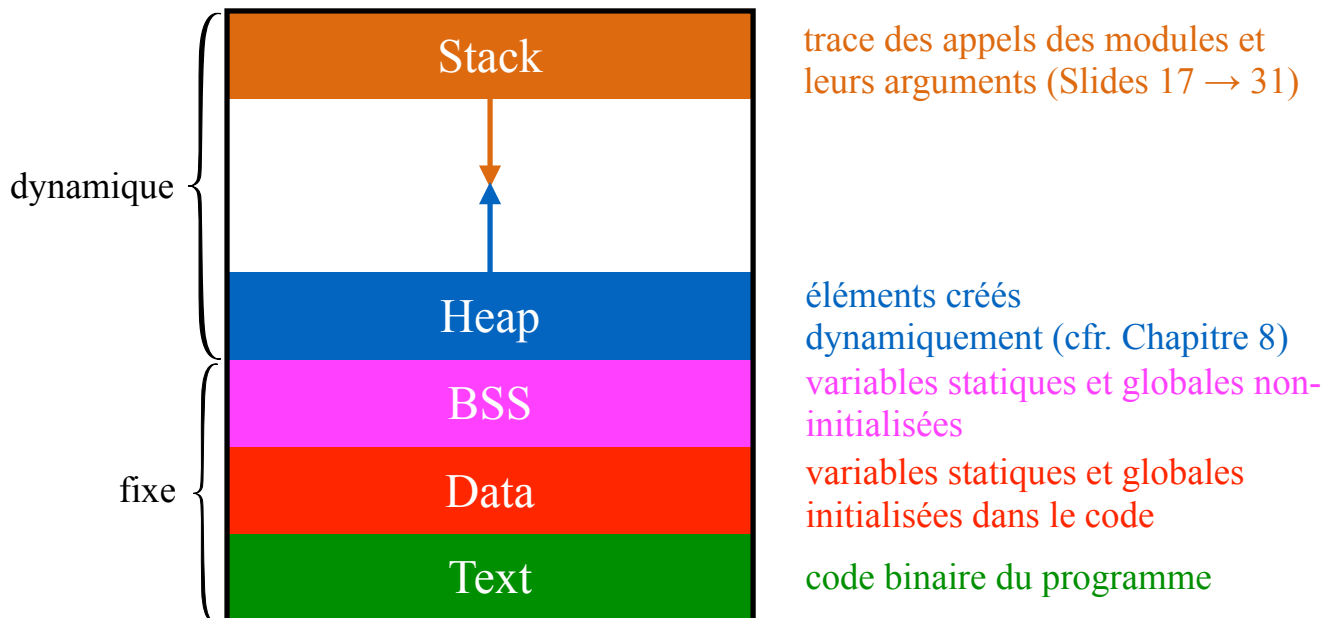


Mémoire Centrale (2)

- Les programmes sont stockés sur le disque dur
- Quand on exécute un programme
 - le système d'exploitation le charge en mémoire RAM
 - le programme devient un **processus**
- A chaque processus est associée une zone mémoire composée de 5 parties distinctes, chacune ayant un rôle bien défini

Mémoire Centrale (3)

- Schéma (simplifié) de la mémoire d'un processus



Mémoire Centrale (4)

- Chaque variable dans la mémoire occupe des octets continus
 - exemple: un `float` occupe 4 octets qui se suivent
- L'opérateur `&` permet de connaître l'adresse d'une variable
 - exemple: `int x; &x;`

Adresses
(en hexa)

Mémoire Centrale

N				
N-4				
...				
i				
...				
8				
4				
0				

8 bits

Opérations Pointeurs

- Il est possible de mémoriser, dans un emplacement mémoire, l'adresse d'une variable
- **Pointeur**
 - variable dont la valeur est une adresse
- Si T est un type quelconque
 - T* désigne le type d'un pointeur vers une variable de type T
 - void* désigne un pointeur vers des variables de n'importe quel type
- On a déjà rencontré des pointeurs dans le cours (cfr. Chapitre 1 et 5)
 - FILE* fp

Opérations Pointeurs (2)

- Exemple

```
#include <stdio.h>

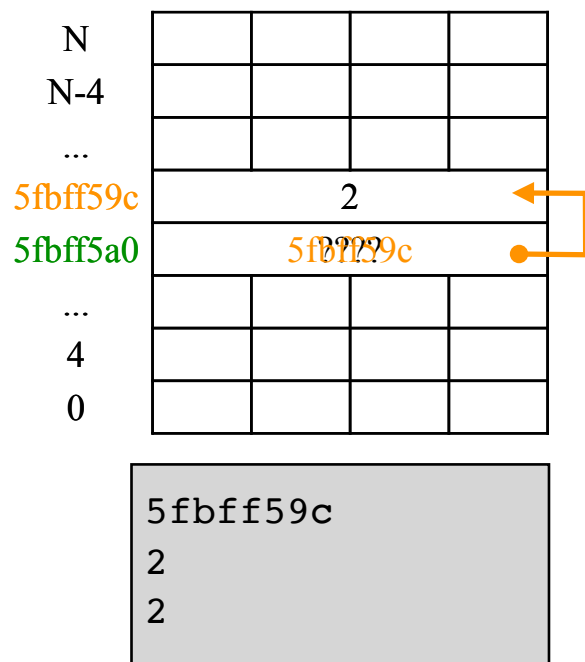
int main(){
    int x = 2;
    int *p;

    p = &x;

    printf("%x\n", p);
    printf("%d\n", x);
    printf("%d\n", *p);

    return 0;
} //fin programme
```

Mémoire Centrale



Opérations Pointeurs (3)

- On accède à la donnée pointée par un pointeur via l'étoile

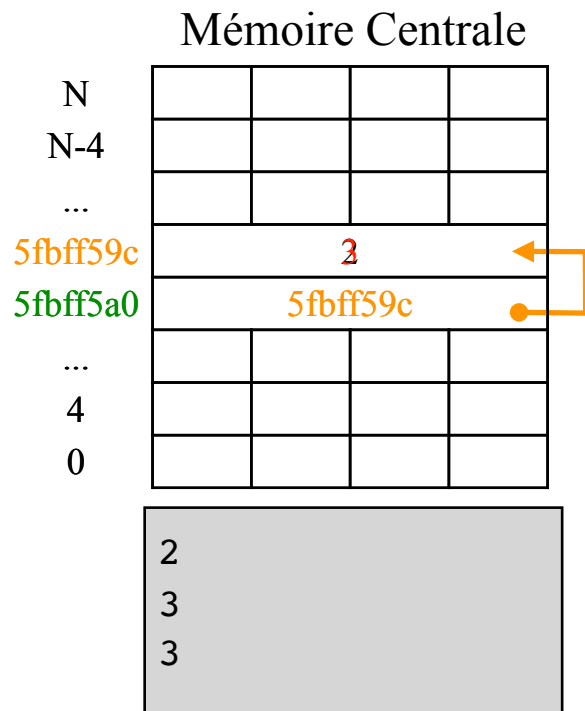
```
#include <stdio.h>

int main(){
    int x = 2;
    int *p = &x;

    printf("%d\n", x);
    *p = 3;

    printf("%d\n", *p);
    printf("%d\n", x);

    return 0;
} //fin programme
```



Opérations Pointeurs (4)

- Les pointeurs peuvent
 - subir des affectations par le biais de l'opérateur =
 - être comparés à l'aide de l'opérateur ==
 - ✓ `int *p; int *q`
 - ✦ `p==q`
 - ✦ `*p==*q`
- Il existe un pointeur spécial défini dans `stdlib.h`
 - NULL
 - représente le pointeur vide
 - ✓ il ne pointe sur rien
 - ✓ utile pour récupérer certains cas particuliers

Opérations Pointeurs (5)

- Un peu de vocabulaire
 - **référencement**
 - ✓ obtenir l'adresse d'une variable
 - ✓ opérateur: &
 - **déréférencement**
 - ✓ obtenir la valeur vers laquelle on pointe
 - ✓ opérateur: *

```
int v = 10;  
int *ptr = &v;
```

```
*ptr = 20;
```

pointent sur la même
zone mémoire
v vaut 20

Opérations Pointeurs (6)

- Attention
 - l'opérateur d'incrémentaion a priorité sur celui de déréférencement

```
int i;  
int *entier;
```

//du code

```
i = *++entier;
```

incrémente d'abord le pointeur
déréférence la nouvelle adresse pointée

```
i = ++*entier;
```

incrémente la valeur pointée
affecte le résultat à i

```
i = (*entier)++;
```

affecte la valeur pointée par entier à i
incrémente cette valeur

Opérations Pointeurs (7)

programme	a	b	c	p1, *p1	p2, *p2
int a, b, c, *p1, *p2;	Ø	Ø	Ø	Ø	Ø
a = 1, b = 2, c = 3;					
p1 = &a, p2 = &c;					
*p1 = (*p2)++;					
p1 = p2;					
p2 = &b;					
*p1 -= *p2;					
++*p2;					
*p1 *= *p2;					
a = ++*p2 * *p1;					
p1 = &a;					
*p2 = *p1 /= *p2;					

Opérations Pointeurs (8)

- En résumé
 - lorsque p pointe sur x
 - ✓ la valeur de p est l'adresse de x en mémoire
 - ✓ toute modification de *p modifie x
 - ✓ toute modification de x modifie *p
 - raison?
 - ✓ *p et x sont sur le même emplacement mémoire dans la mémoire centrale

Agenda

- Chapitre 7: Pointeurs
 - Introduction
 - Passage de Paramètres
 - ✓ Principe
 - ✓ Passage par Valeur
 - ✓ Passage par Adresse
 - ✓ Paramètres d'un Programme
 - ✓ Application
 - Pointeurs et Enregistrements
 - Pointeurs et Tableaux

Principe

- Rappels
 - *paramètre(s) formel(s)*
 - ✓ argument(s) d'un module défini(s) lors de la déclaration du module
 - ✓ un paramètre possède un type et un identificateur
 - ✓ exemples
 - `void affiche_tableau(int tab[], int n);`
 - `int factorielle(int n);`
 - *paramètre(s) effectif(s)*
 - ✓ expression(s) passée(s) à un module lors de l'invocation
 - ✓ exemples
 - `affiche_tableau(t, 10);`
 - `printf("%d\n", factorielle(6));`

Principe (2)

- **Passage de paramètres**
 - mécanisme permettant de faire le lien entre le(s) paramètre(s) effectif(s) et le(s) paramètre(s) formel(s) lors de l'invocation d'un module
- Ce lien se fait en créant sur la pile un espace mémoire particulier
 - **frame**
 - **contexte**
- A chaque invocation de module correspond un contexte
 - le contexte est détruit à la fin de l'exécution du module

Principe (3)

- Un contexte contient (entre autres)
 - des espaces mémoires pour les (éventuels) paramètres formels du module
 - ✓ dans l'ordre inverse
 - des espaces mémoires pour les (éventuelles) variables locales du module
 - l'adresse de retour
 - ✓ pour que le code appelant puisse reprendre la main à la fin de l'exécution du module
- La fonction `int main()` dispose d'un contexte
 - c'est le 1^{er} contexte créé sur la pile

Principe (4)

- En C, il existe deux modes de passage de paramètres
 - par **valeur**
 - ✓ la valeur du paramètre effectif est recopiée sur la pile, dans le contexte associé à l'invocation
 - ✓ le module travaille sur une copie locale du paramètre effectif
 - ✓ le paramètre formel contient la copie locale
 - ✓ à la fin de l'exécution du module toute modification du paramètre formel est perdue car
 - ✓ le contexte est détruit
 - ✓ le module travaillait sur une copie locale
 - par **adresse**
 - ✓ on copie sur la pile l'adresse du paramètre effectif, dans le contexte associé à l'invocation
 - ✓ le paramètre formel contient donc l'adresse du paramètre effectif
 - ✓ à la fin de l'exécution du module, toute modification du paramètre formel est conservée par le code appelant

Passage par Valeur

```
#include <stdio.h>
```

```
float calcul(float a, float b){  
    float res;  
    res = -b/a;  
    return res;  
} //fin calcul()
```

```
int main(){  
    float coef1 = 0.5;  
    float coef2 = 128.2;  
    float sol;  
    sol = calcul(coef1, coef2);  
  
    printf("%f\n", sol);  
  
    return 0;  
} //fin programme
```

Pile

main()	
coef1	0.5
coef2	128.2
sol	-256.39
calcul()	
b	128.2
a	0.5
res	-256.39
adresse retour	

Passage par Valeur (2)

```
#include <stdio.h>
```

```
void incremente(int x){  
    x++;  
} //fin incremente()
```

```
int main(){  
    int var = 10;  
  
    printf("%d\n", var);  
    incremente(var);  
    printf("%d\n", var);  
  
    return 0;  
} //fin programme
```

Pile

main()	
var	10
incremente()	
x	10
adresse retour	

Passage par Adresse

```
#include <stdio.h>
```

```
void incremente(int *x){  
    (*x)++;  
} //fin incremente()
```

```
int main(){  
    int var = 10;  
  
    printf("%d\n", var);  
    incremente(&var);  
    printf("%d\n", var);  
  
    return 0;  
} //fin programme
```

Pile

main()	
var	10
incremente()	
x	• 5fbff59c
adresse retour	

Paramètres d'un Prog.

- La fonction `main()`
 - peut prendre des arguments
 - intérêt?
 - ✓ passer des paramètres en ligne de commande au programme
- Les arguments
 - `int argc`
 - ✓ nombre d'arguments de la ligne de commande
 - `char** argv`
 - ✓ tableau de pointeurs contenant la liste des arguments
 - ✓ chaque pointeur pointe sur une chaîne de caractères se terminant par `'\0'`
 - ✓ `argv[0]` donne le nom du programme

Paramètres d'un Prog. (2)

- Exemple

```
#include <stdio.h>

int main(int argc, char** argv){
    int i;
    printf("nom du programme: %s\n", argv[0]);

    for(i=1; i<argc; i++)
        printf("argument %d : %s\n", i, argv[i]);

    return 0;
} //fin programme
```

```
$ ./test 1 2 3 4 5 6
nom du programme: test
argument 1 : 1
argument 2 : 2
argument 3 : 3
argument 4 : 4
argument 5 : 5
argument 6 : 6
```

Application

- Les pointeurs, en paramètre d'un module, permettent (dans un sens) de renvoyer plusieurs résultats en une fois.
- Exemple: fonction qui renvoie le minimum et le maximum de deux nombres
 - fichier `minmax.h`

```
/*  
 * @pre: min!=NULL, max!=NULL;  
 * @post: *min = MINIMUM{a, b}, *max = MAXIMUM{a, b}  
 */  
void minmax(int a, int b, int *min, int *max);  
           passage par valeur      passage par adresse
```

Application (2)

- Fichier `minmax.c`

```
#include <assert.h>  
#include <stdlib.h>  
#include "minmax.h"  
  
void minmax(int a, int b, int *min, int *max){  
    assert(min!=NULL && max!=NULL);  
  
    if(a<b){  
        *min = a;  
        *max = b;  
    }else{  
        *min = b;  
        *max = a;  
    }  
}  
} //fin minmax()
```

Application (3)

- Fichier main-minmax.c

```
#include <stdio.h>
#include <stdlib.h>
#include "minmax.h"

int main(int argc, char** argv){
    if(argc!=3){
        printf("Usage: ./main a b\n");
        return -1;
    }

    int min, max;

    minmax(atoi(argv[1]), atoi(argv[2]), &min, &max);

    printf("le minimum est %d\n", min);
    printf("le maximum est %d\n", max);

    return 0;
} //fin programme()
```

transforme un char en int
int atoi(char*)

Application (4)

- Quel est le résultat de ce code?

```
#include<stdio.h>

void affiche(int a, int b){
    printf("%d, %d\n", a, b);
}

void incr1(int x){ x = x + 1; }
void incr2(int* x){ *x = *x + 1; }
void decr1(int* x){ x = x - 1; }
void decr2(int* x){ *x = *x - 1; }

int main(){
    int i = 1;
    int j = 1;
    affiche(i, j);
    incr2(&i);
    affiche(i, j);
    decr1(&j);
    affiche(i, j);
    decr2(&j);
    affiche(i, j);
    while(i != j){
        incr1(j);
        decr2(&i);
    }
    affiche(i, j);
} //fin programme
```

Exercices

- Ecrire une fonction qui initialise deux entiers et un réel à -5
 - écrire le programme qui appelle cette fonction et affiche les variables avant et après l'initialisation
- Ecrire une fonction qui retourne le quotient et le reste de la division d'un entier p par un entier q
 - Ecrire le programme qui appelle cette fonction et affiche le résultat à l'écran
- Ecrire une fonction qui échange le contenu de 2 entiers
 - écrire le programme qui appelle cette fonction et qui affiche les valeurs des variables avant et après l'appel

Agenda

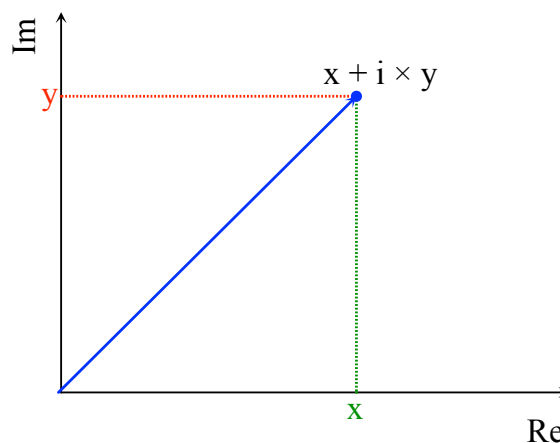
- Chapitre 7: Pointeurs
 - Introduction
 - Passage de Paramètres
 - Pointeurs et Enregistrements
 - ✓ Principe
 - ✓ Exemple
 - Pointeurs et Tableaux

Principe

- En C, il est très fréquent de manipuler des données structurées via des pointeurs
- Si p est un pointeur vers une structure possédant un champ c , alors la notation
 - $p->c$
 - est un synonyme pour $(*p) . c$
 - ✓ on utilise jamais cette notation (peu claire)
 - on accède donc au champ c de la donnée pointée par p

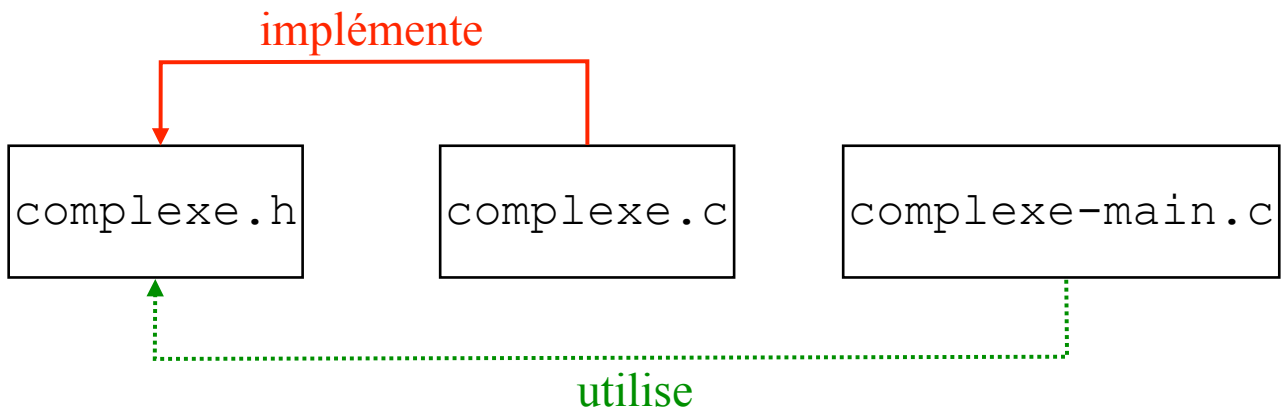
Exemple

- Les nombres complexes
 - représentation cartésienne: $z = x + i \times y$
 - ✓ x : partie réelle
 - ✓ y : partie imaginaire



Exemple (2)

- Architecture du code



Exemple (3)

- Fichier `complexe.h`

```
typedef struct{
    double reel; //partie réelle
    double imag; //partie imaginaire
}Complexe;

/*
 * @pre: c est valide
 * @post: phase vaut la phase du complexe
 */
double phase(Complexe *c);

/*
 * @pre: c est valide
 * @post: module vaut le module du complexe
 */
double module(Complexe *c);
```

Exemple (4)

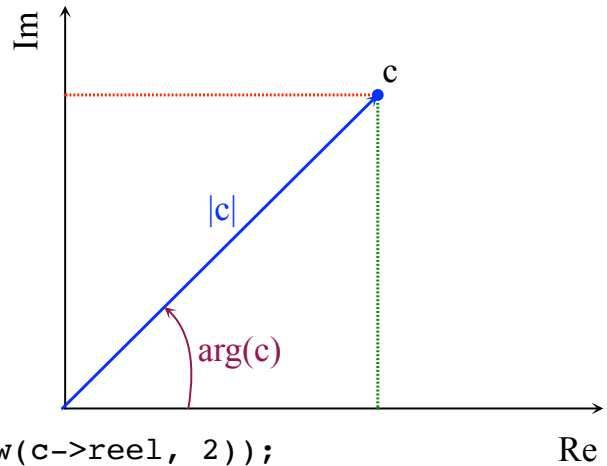
- Fichier `complexe.c`

```
#include <math.h>

#include "complexe.h"

double phase(Complexe *c){
    return atan(c->imag/c->reel);
} //fin phase()

double module(Complexe *c){
    return sqrt(pow(c->imag, 2)+pow(c->reel, 2));
} //fin module()
```



Exemple (3)

- Fichier `complexe-main.c`

```
#include <stdio.h>

#include "complexe.h"

int main(){
    Complexe c1 = {2.0, 3.0};
    Complexe c2 = {4.6, -2.0};

    printf("phase c1: %lf\n", phase(&c1));
    printf("module c2: %lf\n", module(&c2));

    return 0;
} //fin programme
```

Exercice

- Soit une structure `Point` contenant deux champs `x` et `y` de type `float`
 - écrire une fonction qui échange deux structures `Point` passées par adresse
 - écrire le programme qui
 - ✓ saisit deux structures `Point` dans des variables
 - ✓ échange le contenu de ces variables en appelant la fonction
 - ✓ affiche le nouveau contenu des variables

Agenda

- Chapitre 7: Pointeurs
 - Introduction
 - Passage de Paramètres
 - Pointeurs et Enregistrements
 - Pointeurs et Tableaux
 - ✓ Principe
 - ✓ Exemples
 - ✓ Arithmétique des Pointeurs

Principe

- Dans le Chap. 5, nous avons vu la notion de tableau
- Si v est un tableau d'entiers
 - $v[0]$ renvoie le 1^{er} élément du tableau
 - $v[i]$ renvoie le $i+1^{\text{ème}}$ élément du tableau
- En fait, on peut reformuler comme suit
 - $v[0]$ pointe vers le 1^{er} élément du tableau
 - $v[i]$ pointe vers le $i+1^{\text{ème}}$ élément du tableau

Principe (2)

- On peut en déduire que
 - les expressions v et $\&v[0]$ sont équivalentes
 - $v+i$ et $\&v[i]$ sont équivalentes
 - ✓ *arithmétique des pointeurs*
 - $v[i]$ et $*(v+i)$ sont équivalentes

Principe (3)

- Exemple

- soit, `t`, un tableau d'entiers, de taille 10
- placer la valeur 1 dans les 10 cases du tableau

```
int t[10];  
  
int i;  
for(i=0; i<10; i++)  
    t[i] = 1;
```

```
int t[10];  
  
int i;  
for(i=0; i<10; i++)  
    *(t+i) = 1;
```

Principe (4)

- Lorsqu'un tableau est passé en paramètre d'un module
 - il est converti automatiquement en son adresse
 - ✓ pour être précis, l'adresse de son premier élément
 - un tableau n'est donc jamais recopié
 - ✓ raison?
 - ♦ rapidité
 - ♦ mémoire
- Les prototypes suivants sont identiques

```
void init_tab(int *ptr, int n);  
void init_tab(int ptr[], int n);
```

Exemples

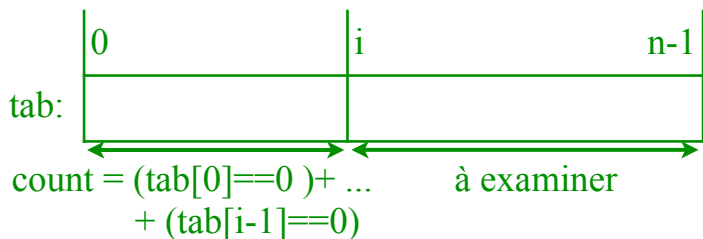
- Exemple 1

- écrire une fonction qui retourne le nombre de zéros d'un tableau `tab` de taille `n`

```
/*
 * @pre: tab est un tableau valide, n ≥ 0
 * @post: compte_zero vaut le nombre de zéro dans tab
 */
int compte_zero(int *tab, int n){
    int i, count=0;

    for(i=0; i<n; i++) //Inv: tab:
        if(tab[i]==0)
            count++;

    return count;
} //fin compte_zero()
```



Exemples (2)

- Exemple 2

- fonction qui retourne la longueur d'une chaîne de caractères `s`

```
unsigned strlen(char *s){
    unsigned l = 0;

    while(*s++)
        l++;

    return l;
} //fin strlen()
```

fonction similaire définie dans `string.h`

l'opérateur `++` est prioritaire sur l'opérateur `*`

Arithmétique Pointeurs

- Arithmétique des pointeurs

```
int a[10];  
int *p;
```

```
p = a;
```

p pointe sur a[0] (&a[0])

```
p++;
```

p pointe sur a[1] (&a[1])

```
n = 3;
```

p pointe sur a[1+n] (&a[1+n])

```
p += n;
```

```
p = a+9;
```

p pointe le dernier élément du tableau

```
p = a+11;
```

p sort des limites du tableau

Arithmétique Pointeurs (2)

- Exemple

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};  
int *P;  
P = A;
```

```
int i = *P+2;  
printf("%d\n", i);
```

```
i = *(P+2);  
printf("%d\n", i);
```

```
i = &P+1;  
printf("%d\n", i);
```

```
i = &A[4]-3;  
printf("%d\n", i);
```

```
i = A+3;  
printf("%d\n", i);
```

```
i = &A[7]-P;  
printf("%d\n", i);
```

```
i = P+(*P-10);  
printf("%d\n", i);
```

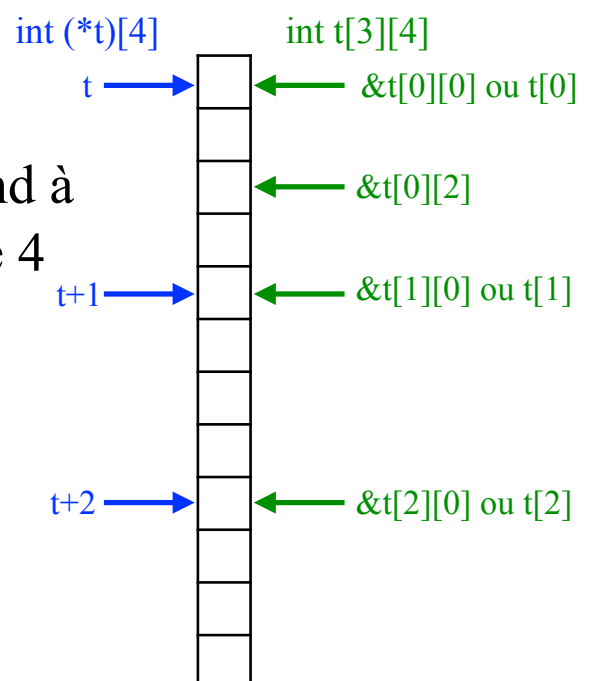
```
i = *(P+*(P+8)-A[7]);  
printf("%d\n", i);
```


Arithmétique Pointeurs (3)

- Soit la déclaration
 - `int t[3][4];`
 - `t` désigne un tableau de 3 éléments
 - ✓ chaque élément est un tableau de 4 entiers
- Si `t` représente bien l'adresse de début du tableau
 - il n'est plus de type `int *`
 - ✓ valable pour un tableau à 1 dimension
 - mais plutôt "pointeur sur des blocs de 4 entiers"

Arithmétique Pointeurs (4)

- L'expression `t+1` correspond à l'adresse de `t` augmentée de 4 entiers
- `t` et `&t[0][0]` sont équivalents



Exercices

- Ecrire un programme qui lit 10 entiers et les place dans un tableau avant d'en rechercher le minimum et le maximum
 - écrire une version qui utilise le formalisme tableau (cfr. Chapitre 4)
 - écrire une version qui utilise le formalisme pointeur
- Ecrire un programme qui ne renvoie aucune valeur mais qui détermine le minimum et maximum d'un tableau
 - la fonction doit avoir 4 arguments: le tableau, sa taille, le minimum, le maximum