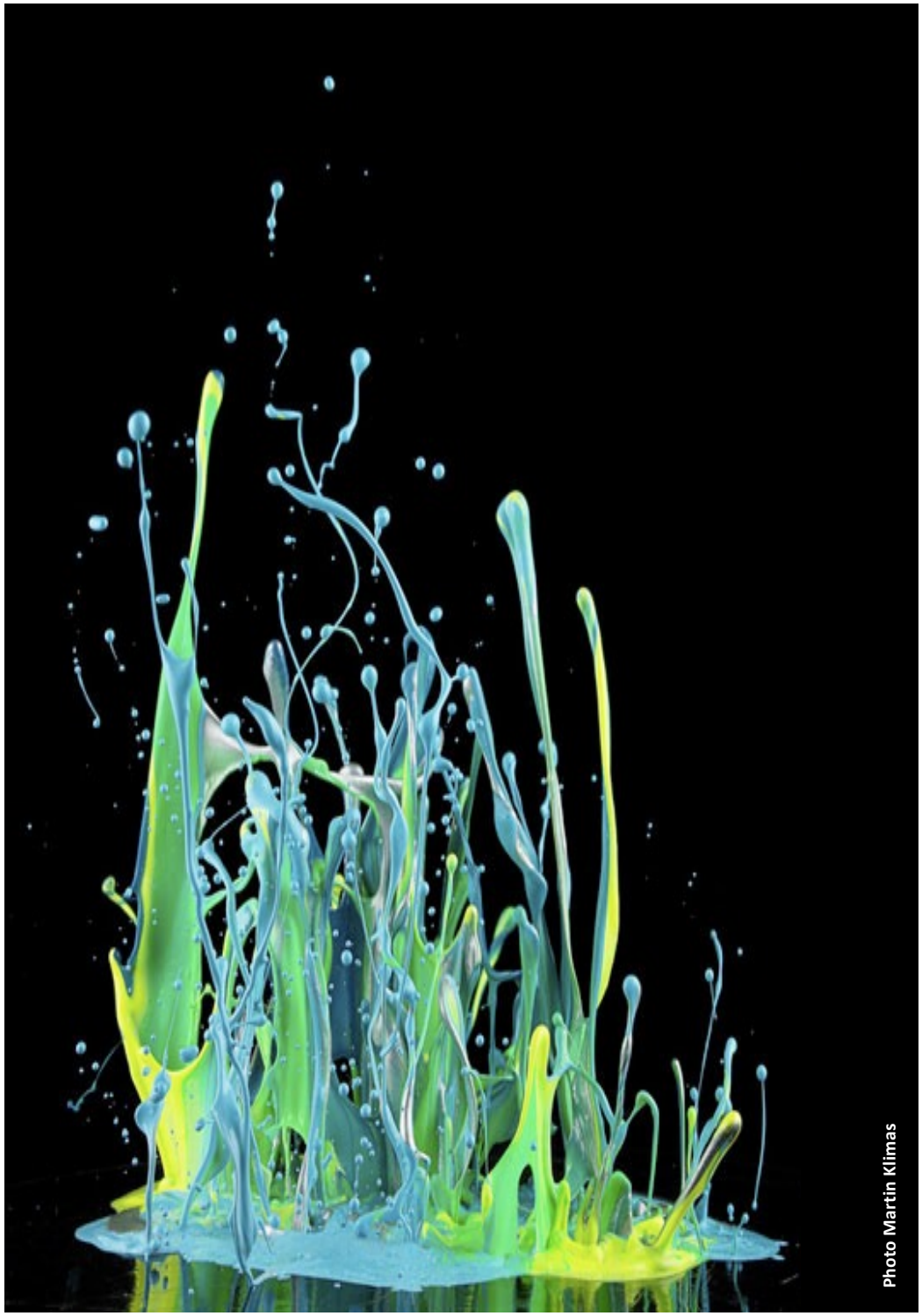


EPFL

Programmation pour Ingénieur

Récap. + Mem. alloc.

ME 3e semestre

rev. 2025.1

Christophe Salzmann

Laboratoire d'Automatique

modifications

- 22.08.2025, r1 – intial

Quiz

Combien d'erreurs trouvez-vous dans les prochains slides ?

Exercices 1 (C)

```
if (x!=0); {
        y=1/x;
   }
```

```
const int sizeA = 8;

double myArray[sizeA];

for (int i=sizeA; i>0; i--)

          myArray[i]=i*i;
```

```
double x;

for(x=0.0; x != 10.0; x += 0.1)

        printf("c2:%f\n",c);
```

Exercices 2 (C)

```
 int main() {


     int Moyenne(int a, int* b) {

        return a+b/2;
       }


    int m=Moyenne(4,3);

}
```

```
void loop(int x){

    for (int i=0;i<x;i++) {
        continue;
        int y= y+x;
       }

    printf("%d\n",y);;

    return x;
}
```

Exercices 3 (C)

```c
#include <stdio.h>

int i=2;

int add1(int x) {

    return 1 + i;
}

int main() {

    int a=0;

    a=add1(a);

    printf("%d",a);

}
```

Qu'affiche le programme ?

Exercices 4 (C)

```c
#include <stdio.h>


void initArray(int A[], int sz){
   for (int i=0; i<=sz; i++) {
        A[i]=0;
    }
}


int main() {
    int p=123;
    int A[1];

    initArray(A,1);

    printf(”%d\n”,p);

    return 0;
}
```

Qu'affiche le programme ?

Exercices 5 (C)

```c
#include <stdio.h>


int Double2(int x) {
    return 2*x;

}

int Divise2(int x) {
    return x/2;

}


int main() {

    int x=3;

    if (x=0) ;
        x= x+3;

    printf("x= %d\n", Double2(Divise2(x)));

}
```

Qu'affiche le programme ?

Exercices 6 (C)

```c
int x=-1,y=3;

x= (1==2) && (y=3+y);

printf("%d%d\n",x,y);
```

Qu'affiche le programme ?

Exercices 7 (C)

```c
int i = 9;

int main() {
    printf("i    %d\n",i);

    int i = 8;
    {
      int i = 7 ;
      printf("{{i}}  %d\n",i);
    }
    printf("{i}   %d\n",i);
}
```

Qu'affiche le programme ?

Exercices 8 (C)

Qu'affiche la console après le code ci-dessous ?

```
int i=0;
printf("%d, %d",i, i++);
```

Xcode -> **0, 0**

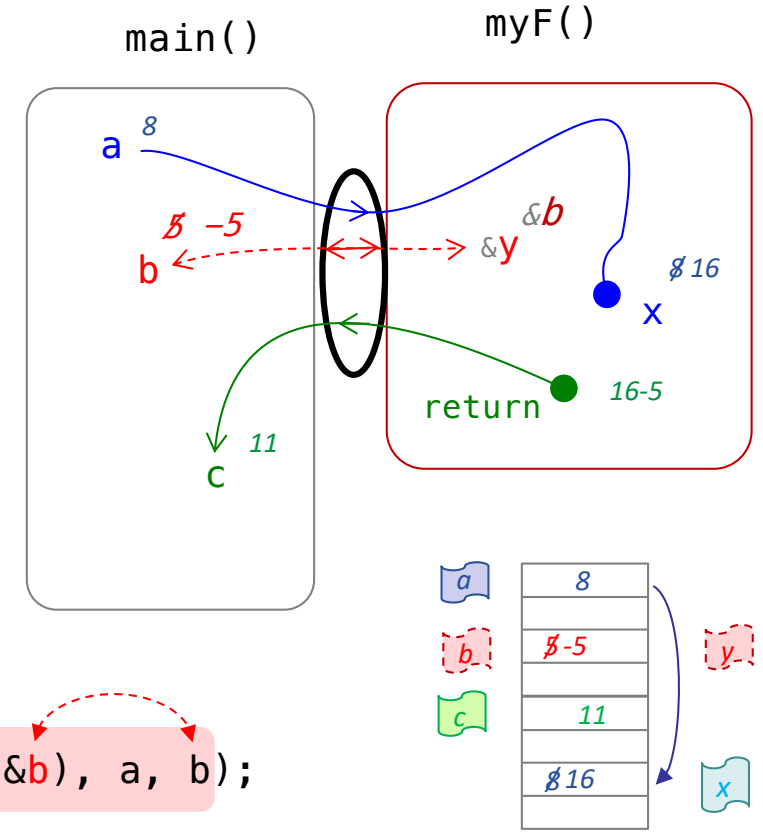
Visual Studio -> **1, 0**

Why different ?

Rappel: passage des paramètres (C)

par valeur — par référence/pointeur

```
int myF( int x, int *y ) {
  x = 2 * x;
  *y = -*y;
  return x+*y;
}


main() {

  int a = 8, b = 5;
  int c = myF(a,&b);
  printf("a: %d b: %d c: %d\n",a,b,c);
  printf("myF= %d, a: %d, b: %d\n", myF(a,&b), a, b);
}
```



Δ odre des operations indéfini, i.e. compilateur dépendant

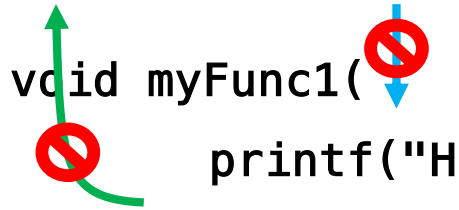
Affiche: **a: 8 b: -5 c: 11**

Affiche OSX: **myF=21, a: 8, b: 5**

Affiche VS: **myF=21, a: 8, b: -5**

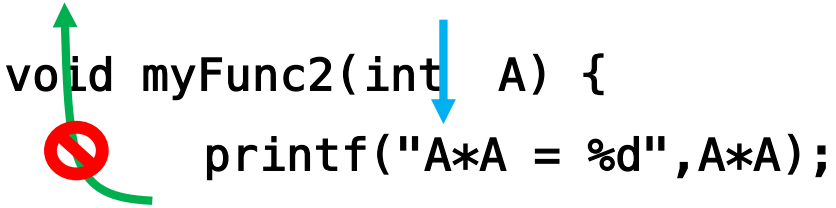
Passage par **valeur**: le paramètre est copié, les modifications faites sur la copie ne sont pas reportées
Passage par **référence**: une référence sur le paramètre est transmise, les modifications sont alors directement effectives

Rappel: fonction – paramètres (C)



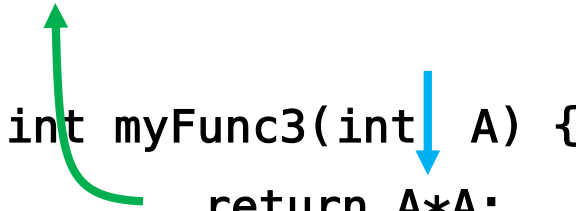
```
void myFunc1(  ) {
    printf("Hello");
    }
```

Affiche **Hello** dans la console



```
void myFunc2(int  A) {
    printf("A*A = %d",A*A);
    }
```

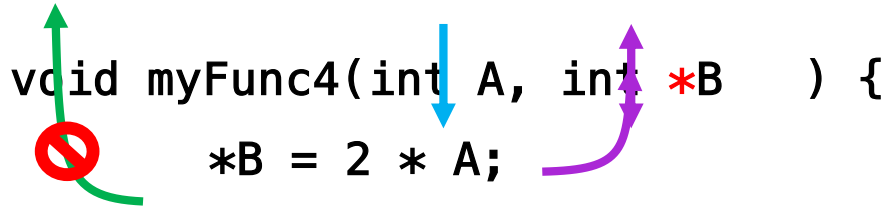
Si **A** vaut 2
Affiche **A*A = 4** dans la console



```
int myFunc3(int  A) {
    return A*A;
    }
```

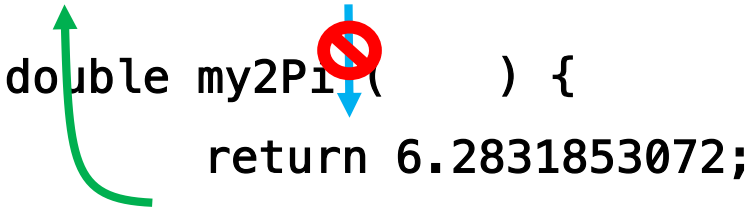
Pas d'affichage dans la console,
mais retourne A^2

Rappel: fonction – paramètres (C)



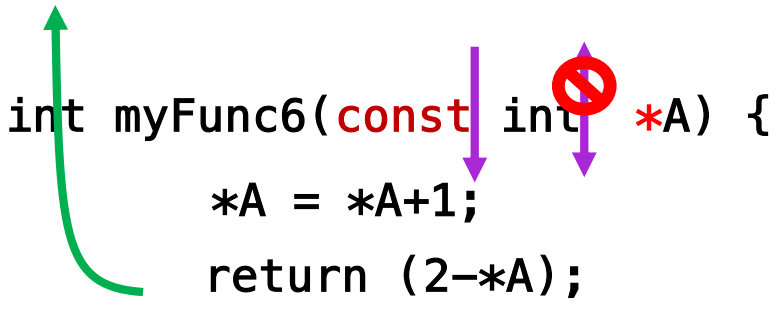
```
void myFunc4(int A, int *B   ) {
        *B = 2 * A;
        }
```

Pas d'affichage dans la console



```
double my2Pi(    ) {
        return 6.2831853072;
        }
```

Pas d'affichage dans la console



```
int myFunc6(const int *A) {
        *A = *A+1;
        return (2-*A);
               }
```

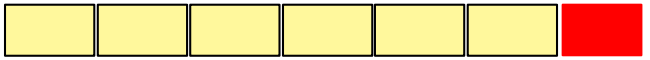
<-- Erreur de compilation

Tableau

Arrangement de plusieurs données de même type: **-> tableau**

- Ex: Liste de valeurs, chaine de caractères

```
int tab[6] = {1,2,3,4,5,6};
tab[0] = -1;
```



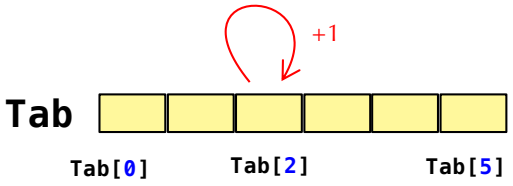
- L'accès à un élément du tableau se fait via **Tab[index]**
- Les indices d'un tableau de taille *n* varient entre 0 et *n*-1
- Lors de l'appel de fonctions les tableaux sont toujours passés par **référence**
- Les tableaux ne connaissent **PAS** leur taille, il faut la stocker dans une variable supplémentaire
- Il n'y a **PAS** de contrôle de débordement
- Accéder à un indice qui n'est pas dans les bornes 0 et *n*-1 entraine des résultats aléatoires (=> crash)

Tableau

L'accès à un élément du tableau se fait via **`Tab[index]`**

Les indices d'un tableau de taille *n* varient entre 0 et *n*-1.

```
int Tab[6]; //definition

Tab[2] = Tab[2] + 1; //accès
```

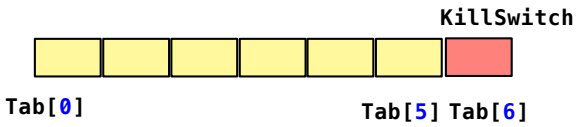


Attention! Les indices d'un tableau de taille *n* varient entre 0 et *n*-1. Il n'y a PAS de contrôle de débordement. Accéder à un indices qui n'est pas dans les bornes 0 et *n*-1 entraine des résultats aléatoires (=> crash).

Ex.

```
int Tab[6];
for (int i=0; i <= 6; i++) {
                    Tab[i]=i;
        }
```

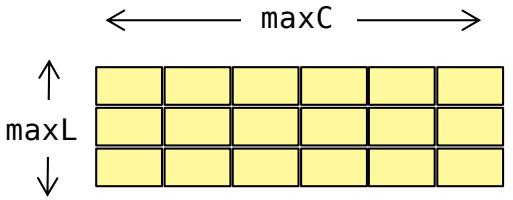
Dernière itération, i=6



Tableaux à plusieurs dimensions

Il est possible de définir des tableaux à plus d'une dimension, il suffit d'ajouter une dimension avec`[]`. Un tableau à deux dimensions pour être vu comme un tableau de tableaux.

Ex:




```
#define maxC 6
#define maxL 3

unsigned int Matrice[maxL][maxC];
```

Initialisation d'un tableau à 2 dimensions

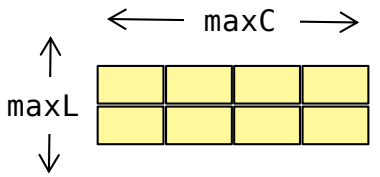
```
unsigned int Matrice[maxL][maxC]= { {1, 2, 3, 4, 5, 6},
                                    {2, 4, 6, 8, 10,12},
                                    {3, 6, 9, 12,15,18} };
```

Les {} imbriquées sont optionnelles.

```
int Matrice[2][3]= {1,2,3,4,5,6};
```

Tableaux à plusieurs dimensions (C)

Ex:




```
#define maxC 4
#define maxL 2

unsigned int Mat[maxL][maxC]= { {1, 2, 3, 4},
                                {5, 6, 7, 8} };
for (int L = 0; L< maxL; L++)
    for (int C = 0; C< maxC; C++){
        printf("Mat[%d][%d]: %d\n",L,C,Mat[L][C]);
}
```

Affiche:

Mat[0][0]: 1
Mat[0][1]: 2
Mat[0][2]: 3
Mat[0][3]: 4
Mat[1][0]: 5
Mat[1][1]: 6
Mat[1][2]: 7
Mat[1][3]: 8

Le passage d'un tableau en paramètre d'une fonction (C)

Un tableau statique n'a pas connaissance de sa propre taille. Il faut donc veillez à ce que la fonction connaisse le nombre d'éléments du tableau, soit en utilisant la constante contenant la taille du tableau, soit en passant un paramètre supplémentaire indiquant la taille. La taille des tableaux à une dimension peut être omise lors du passage de paramètres.

Ex.

```
int myF3 (int Tab[], int SizeTab) {
   for (int i(0); i< SizeTab; i++)
         Tab[i] = i;
   }
```

Dans l'exemple ci-dessus, il faut s'assurer que `SizeTab` est plus petit ou égale à la taille de `Tab[]` lors de l'appel à `myF3()`

Le passage d'un tableau en paramètre d'une fonction

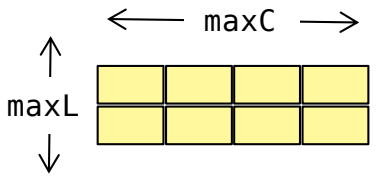
Le passage d'un tableau de taille fixe en argument d'une fonction se fait toujours **par référence**, bien que ce ne soit pas explicitement marqué par le signe **&**. Il pourra donc être **modifié**! Il faut employer `const` si l'on veut interdire toutes modifications au tableau.

Note: il n'est pas possible de retourner un tableau, par contre il est possible de retourner un pointeur sur un tableau.

Ex.

```
int myF(int Tab[maxTab]) {
   for (int i(0); i<maxTab; i++)
         Tab[i] = i;
   }

int myF2(const int Tab[maxTab]) {
   for (int i(0); i<maxTab; i++)
         Tab[i] = i;  <<-- ERROR
   }
```

Tableaux à plusieurs dimensions en paramètre (C)




```c
#define maxC 4
#define maxL 2

unsigned int Mat[maxL][maxC]= { {1, 2, 3, 4}, {5, 6, 7, 8} };


void initArray1(int Mat[maxL][maxC], int nL, int nC) {  // fixed size
    for (int L = 0; L< nL; L++)
        for (int C = 0; C< nC; C++)
             Mat[L][C]= 11;   // the compiler compute the array cell for you
}

void initArray2(int Mat[][maxC], int nL, int nC) {  // var size, only first [sz] can be omitted
    for (int L = 0; L< nL; L++)
        for (int C = 0; C< nC; C++)
            Mat[L][C]= 22;    // the compiler compute the array cell for you
}

void initArray3(int Mat[], int nL, int nC) {  // array, var size
    for (int L = 0; L< nL; L++)
        for (int C = 0; C< nC; C++)
            Mat[L * nC + C]= 33;  // you have to compute the array cell
}

void initArray4(int *Mat, int nL, int nC) { // pointer, var size
    for (int L = 0; L< nL; L++)
        for (int C = 0; C< nC; C++)
            *(Mat + L*nC + C)= 44; // you have to compute the pointer value
 }
```

Tableau de caractères

En C, les chaines de caractères sont définies comme des tableaux de caractères avec la convention que la fin de la chaine de caractère est définie par le caractère '**\0**', (caractère invisible).

Ex:

```
const int maxSrt = 32 + 1;
char maChaine1[maxSrt];
```

Il est possible d'assigner une valeur lors de la **déclaration** de la chaine de caractères, `'\0'` y est automatiquement ajouté:

```
char maChaine2[maxSrt]= "Une chaine de caracteres";
```

`""` défini une chaine de caractères, `''` -> défini un caractère

La fonction **`strlen()`** retourne la taille utile d'une chaine de caractères (i.e. regarde où se trouve le caractère '**\0**' dans la chaine)

```
printf("l: %d, t: %d\n", strlen(maChaine2), sizeof(maChaine2));
```

affiche **l: 24, t: 33**

Tableau de caractères

Exemples:

```
char maChaine1[32]= "Une chaine de caracteres";
char maChaine2[]= "Une chaine de caracteres";
char* maChaine3 = "Une chaine de caracteres";


printf("%d %d %d\n", strlen(maChaine1),strlen(maChaine2),strlen(maChaine3));
```

24 24 24

```
printf("%d %d %d\n", sizeof(maChaine1),sizeof(maChaine2),sizeof(maChaine3));
```

32 25 8

```
maChaine1[0] ='x';
maChaine2[0] ='x';
maChaine3[0] ='x';
```

Ok
Ok
Bad access `maChaine3` est *readonly*

```
char maChaine4[24]= "Une chaine de caracteres1234";
```

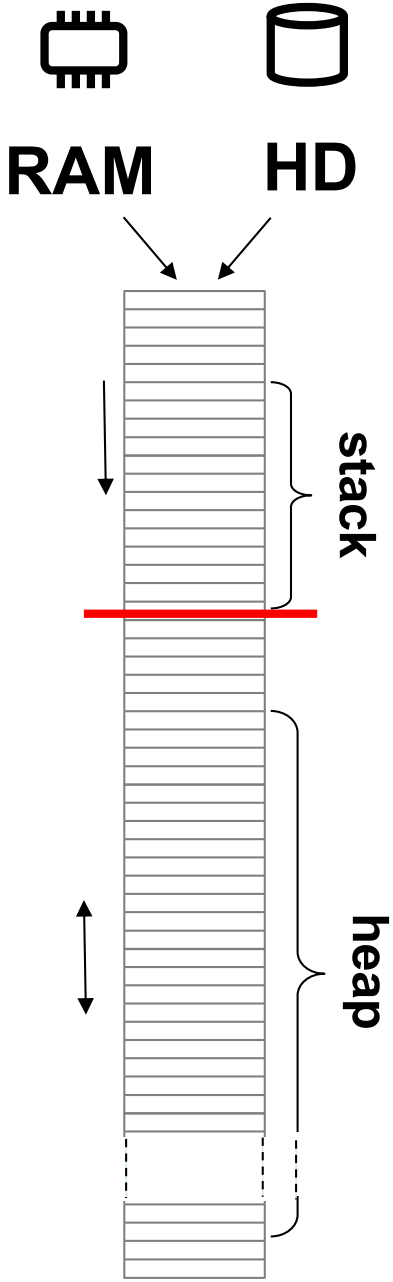
⚠ Initializer-string for char array is too long

```
printf("%s %d %d\n", maChaine4, sizeof(maChaine4),strlen(maChaine4));
```

Une chaine de caracteres\250aA 24 27

Dépends du contenu de la mémoire

Mémoire – stack vs heap



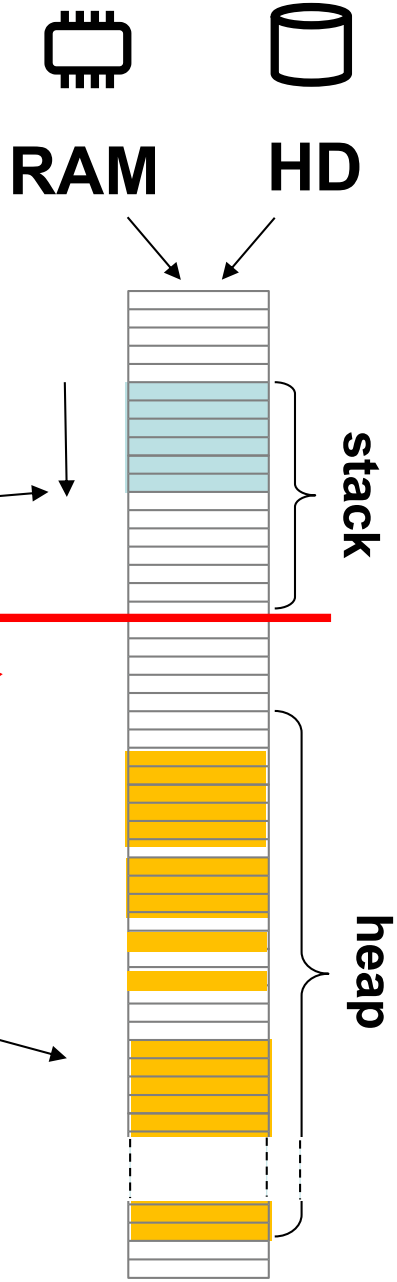
La ***stack*** (pile) est employée pour stocker des variables de manière **automatique**

lors de la déclaration des variables *et pour passer des paramètres lors d'appel de fonctions*. Cet espace mémoire est (très) limité, sa taille dépends de l'OS et/ou compilateur. Les variables y sont ordonnées comme sur une pile (d'assiettes) LIFO. La durée de vie d'une variable sur la ***stack*** est fonction de sa portée, généralement courte.

La ***heap*** (tas) est employée pour stocker des variables de manière dynamique.

L'allocation doit se faire de manière **explicite** (`malloc(), calloc()`).
L'espace alloué doit être explicitement libéré (`free()`) sous peine de ne plus avoir accès à cet espace par la suite. La durée de vie d'une variable sur la ***heap*** est généralement longue.
La taille d'une variable peut croître ou diminuer (`realloc()`) dynamiquement en fonction des besoins du programme. La ***heap*** peut stocker des variables de taille beaucoup plus grandes que celles stockées sur la ***stack***.

Mémoire – stack overflow




```
int main(int argc, const char * argv[]) {
    int A[6];

    int B = 3;
    char C = 'a';

    A[3]=3;

    int* D=malloc(1000000);

        if (D==NULL) {
            /* gestion error */}

    D[0]=0;
    ...

    free(D);

    return 0;
}
```

OSX stack size ~8 MB, heap size ~50 GB (very specific)

Tableau de taille variable avec pointeurs

En C, il est possible de créer des tableaux dont la taille n'est connue que lors de l'exécution. Ces tableaux requièrent une gestion de la mémoire précise sous peine de crash. La gestion de ces tableaux dynamiques fait appel aux pointeurs.

```
const unsigned int TailleInitiale = 10;

int *dynArray = (int)malloc(TailleInitiale * sizeof(int));

for(int i=0; i<TailleInitiale; ++i){
    dynArray[i] = i;
    }
…

free(dynArray);
```

Il est primordial de s'assurer que pour chaque appel à `malloc()` il y a un appel à `free()`

Pointer, malloc, free

`malloc()` réserve la mémoire demandée (spécifiée en nombre de **bytes**) et retourne un pointeur de type **`void`** qui peut être *typecasté* sur l'espace mémoire créé ou **`NULL`** en cas d'erreur.

```c
#include <stdlib.h>

const unsigned int N_elem = 10;

int *dynArray = (int*) malloc(N_elem * sizeof(int));

if (dynArray == NULL)  { /* Gestion Error */}

dynArray[0] = 0;         // array like access, preferred!
*(dynArray + 1) = 1;     // ptr like access

free(dynArray);
```

Pointer, malloc, free

`free()` « libère » l'espace mémoire réservé. Il indique au gestionnaire de mémoire que cet espace est à nouveau disponible. Il **n'efface pas** le contenu de la mémoire.

```c
#include <stdlib.h>

const unsigned int N_elem = 10;

int *dynArray = (int*) malloc(N_elem * sizeof(int));

if (dynArray == NULL)  { /* Error */}
dynArray[0] = 0;         // array like access

free(dynArray); // mem is free-ed NOT Erased

free(dynArray); // should trigger an error since already free-ed

free(NULL);    // OK, does nothing
```

calloc

`calloc()` fonctionne comme **`malloc()`**, i.e. réserve la mémoire demandée (spécifiée avec le **nombre** d'éléments et la taille unitaire de l'élément spécifiée en **bytes**) et retourne un pointeur de type **`void`** qui peut être *typecasté* sur l'espace mémoire créé ou **`NULL`** en cas d'erreur. Les valeurs sont initialisées à '0'.

```
#include <stdlib.h>

const unsigned int N_elem = 10;

int *dynArrayC = (int*) calloc(N_elem , sizeof(int));

// dynArray[0] .. dynArray[9] are set to '0'

if (dynArrayC == NULL)  { /* Gestion Error */}

free(dynArrayC);

dynArrayC = NULL; // safety measure, dynArrayC can't be used again
```

Tableau de taille variable – `Pointeur` (C)

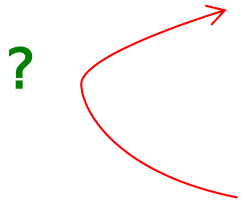
Changement de la taille du tableau (pointeur).

```
const unsigned int TailleInitiale = 10;
const unsigned int NouvelleTaille = 10;
int *newArray = NULL; // pointeur temporaire
int *dynArray = malloc(TailleInitiale * sizeof(int));

newArray = realloc(dynArray , NouvelleTaille * size(int));

if (newArray == NULL) {   // gestion des erreurs
          free(dynArray); // libère la place mémoire
          exit(1);        // quitter de suite avec une erreur!
          }
    else { //Succes!
          dynArray = newArray; // ajuste les pointeurs
          }

for(int i=0; i<NouvelleTaille ; ++i){
          dynArray[i] = i;
          }
free(dynArray); // libère la place mémoire
```

Variable Length Arrays - VLA

En C, le compilateur doit connaître la taille des tableaux statiques au moment de la compilation. Cependant depuis C99 il est possible d'avoir des tableaux statiques dont la taille n'est pas connue à la compilation, mais est définie lors de l'exécution. Cette manière de faire est à proscire le plus possible car la gestion des erreurs pose problème (i.e. crash) s'il n'y a pas assez de place pour le tableau. Un tableau dynamique permet de gérer ces erreurs.

De plus les VLAs sont alloués sur la stack-> attention à la durée de vie du VLA.




```c
int n;

scanf("%d", &n);    // what if input is not a char ?

int myVLA[n];       // what if n = 9999999999999999 ?

myVLA[3]= 22;       // what if 3 > n+1 ?
```

Passage de paramètre à votre exécutable

La fonction main() de votre programme peut recevoir des arguments provenant de la ligne de commande. Cela permet d'ajouter des paramètres dynamiques lors de l'appel de votre programme. Ex:

```
gcc main.c -o output
```

3+1 paramètres: « main.c » « -o » et « ouput » sont passé à `gcc`

Le nom du programme (+ path) « gcc » est toujours fourni à la fonction main().

Il est possible de récupérer ces paramètres dans la fonction main(). Pour cela il faut employer la syntaxe suivante:

```
int main(int argc, const char * argv[])
```

`argc` contient le nombre de paramètre (1..n)

`argv[]` est un tableau de chaine de char contenant les paramètres au format char, il faudra les convertir au format désiré, par exemple avec `atoi()` pour récupérer un entier.

Pour **`gcc main.c –o output`**

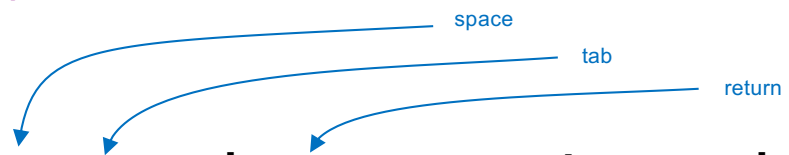
`argc` -> 4

`argv[0]` -> "gcc", `argv[1]` -> "main.c", `argv[2]` -> "-o",
`argv[3]` -> "output", `argv[4]` -> NULL

Attention à tester l'existence du paramètre (via argc) avant d'y accéder (via argv[])

Passage de paramètre à votre exécutable

```
int main(int argc, const char * argv[]){
        printf("argc: %d\n", argc);
        for (int i=0; i < argc; i++)
                printf("argv[%d]: %s\n",i,argv[i]);
        if (argc>=3) {
                int a = atoi(argv[3]);
                printf("argv[3] as int is: %d", a); }
        return 0;
}
```



Avec **./monPrg a string 123 "another String"** **Δ ” ≠ "**

```
argc: 5
argv[0]: ./monPrg
argv[1]: a
argv[2]: string
argv[3]: 123                    <- this is a string
argv[4]: another String                              argv[5]: (null)
argv[3] as int is: 123          <- this is an int
```

Passage de paramètre à votre exécutable

```
int main()
```
ignore les paramètres

```
int main(int argc, const char * argv)
```
paramètres standards

```
int main(int argc, const char * argv[], const char *envp[])
```
paramètres standards + variables d'environnement (non standard)

La fonction `main()` retourne une valeur, en général **0** ce qui indique que tout c'est bien passé.

Une valeur autre que 0 indique potentiellement un problème.

Les constantes EXIT_SUCCESS et EXIT_FAILURE sont définies dans stdlib.h peuvent être utilisée comme valeur de retour.

En fonction de l'OS/compilateur, la valeur retournée peut-être tronquée (8 bits) ou interprétée comme un `unsigned int`.