Chapitre 11

Pointeurs
Et
Fonctions II

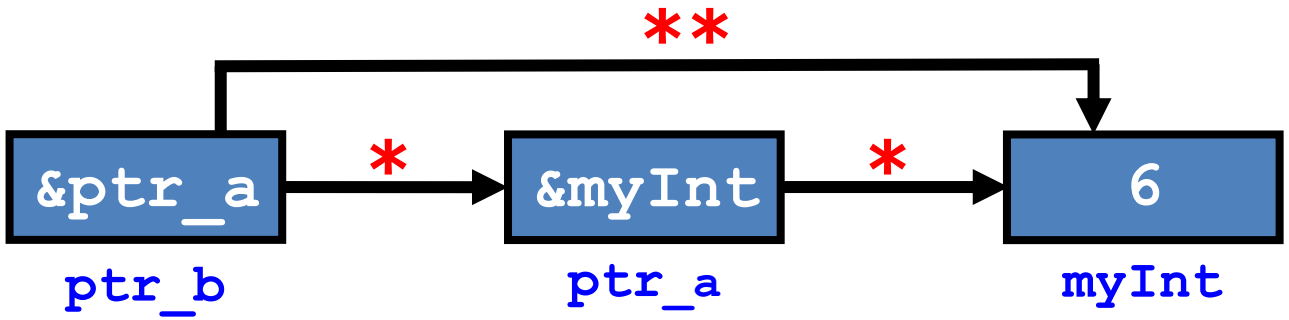
Plan

1. **Pointeurs de pointeurs et arithmétique**
2. Interaction d'un programme avec l'OS
3. Passage de tableaux en paramètre
4. Valeur de retour d'une fonction
5. Pointeur sur des fonctions
6. Callback

Pointeur de pointeur

```
int myInt = 6;
int *ptr_a = &myInt;

int **ptr_b = &ptr_a;
```



La valeur **6** dans la variable `myInt` est **modifiable** par le programme

Le contenu de `ptr_a`, **adresse de la variable** `myInt`, est **modifiable** par le programme

Le contenu de `ptr_b`, **adresse du pointeur** `ptr_a`, est **modifiable** par le programme

Les adresses de `myInt, ptr_a` et de `ptr_b` sont des **constantes** déterminées à la **compilation**

Arithmétique des pointeurs

a. **Affectation** par un pointeur sur le même type

b. **Addition** et **soustraction** via 1 entier

c. **Incrémentation** et **décrémentation** d'un pointeur

d. **Soustraction** de deux pointeurs

a. Affectation

Soient `ptr1` et `ptr2` 2 pointeurs **sur le même type de données**, alors l'instruction

```
ptr1 = ptr2;
```

fait pointer `ptr1` sur le même "objet" que `ptr2`

Casting

```
ptr1 = (double*) ptr2;
```

b. Addition et soustraction via 1 entier

Si `ptr` pointe sur l'élément `a[i]` d'un tableau, alors

`ptr+n` pointe sur `a[i+n]`

`ptr-n` pointe sur `a[i-n]`

b. Addition et soustraction via 1 entier

Exemple : addition/soustraction d'un entier à un pointeur

```
float* ptr = (float*)0x100000;

ptr+1 vaut 0x100004
ptr+2 vaut 0x100008
ptr-2 vaut 0x0FFFF8 //(0x100000-2*4)
```

c. Incrémentation et décrémentation

Si `ptr` pointe sur l'élément `a[i]` d'un tableau alors après l'instruction :

`ptr++;`	`ptr` pointe sur `a[i+1]`
`ptr += n;`	`ptr` pointe sur `a[i+n]`
`ptr--;`	`ptr` pointe sur `a[i-1]`
`ptr -= n;`	`ptr` pointe sur `a[i-n]`

d. Soustraction de 2 pointeurs

Soit `ptr1` et `ptr2` deux pointeurs qui pointent sur le **même tableau**

`ptr2-ptr1`	fournit le nombre **d'éléments** compris entre `ptr1` et `ptr2`

Le résultat de la soustraction `ptr2-ptr1` est :

- **positif**, si `ptr1` précède `ptr2`
- **zéro**, si `ptr1 == ptr2`
- **négatif**, si `ptr2` précède `ptr1`
- **indéfini**, si `ptr1` et `ptr2` ne pointent pas dans le même tableau

d. Soustraction de 2 pointeurs

Exemple

```
float *ptr1=(float*)0x100000;
float *ptr2=(float*)0x100004;
```

`ptr2-ptr1` vaut **1**
`ptr1-ptr2` vaut **-1**
`ptr2-ptr2` vaut **0**

Plan

1. Pointeurs de pointeurs et arithmétique
2. **Interaction d'un programme avec l'OS**
3. Passage de tableaux en paramètre
4. Valeur de retour d'une fonction
5. Pointeur sur des fonctions
6. Callback

Interaction d'un programme avec l'OS

La fonction `main()`

Valeur **retournée** à l'OS avec `return`

Valeurs **reçues** de l'OS avec

une **liste d'arguments**

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

argument count *argument vector*

des **variables d'environnement**

Interaction d'un programme avec l'OS

Valeur de `return` récupérée par le système d'exploitation

Programme C : prog.c

```
int main(void)
{
   …
   return -1;
}
```

Fichier batch : test.bat

```
..
prog.exe
if errorlevel -1 goto err
goto ok
err: echo.....
```

Interaction d'un programme avec l'OS

Valeurs transmises au lancement d'un programme

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

Exemple

```
C:\> myProgFind Salut poem.txt
```

Arguments	[0]	[1]	[2]

`argc = 3`

`argv[0] = "myProgFind"`	nom du programme
`argv[1] = "Salut"`	premier argument
`argv[2] = "poem.txt"`	deuxième argument
`argv[3] = NULL`	chaine vide

Interaction d'un programme avec l'OS

Variables d'environnement

La fonction **`main()`** possède un **3ème** argument **`char *env[]`**, qui est également un tableau de pointeurs sur des chaînes de caractères.

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

Chaque élément de **`env`** contient un **`char*`** de la forme :

`envvar = valeur`

Par exemple **`PATH = C:\Windows\System32`**

Plan

1. Pointeurs de pointeurs et arithmétique
2. Interaction d'un programme avec l'OS
3. **Passage de tableaux en paramètre**
4. Valeur de retour d'une fonction
5. Pointeur sur des fonctions
6. Callback

Tableau 1D comme argument

Quand on passe un tableau en argument à une fonction, on passe une copie de son adresse

Exemple
```
float tab[4]={1,2,3,4};
unefonction(tab);
```

L'argument de `unefonction()` est **un pointeur sur un `float`**

```
void unefonction(float* t) ou void unefonction(float t[])
```

On ne peut pas récupérer la taille d'un tableau dans une fonction

→ on passe donc cette information en argument

```
void unefonction(float *t, short size)
```

Tableau 2D comme argument de fonction

```
void print_vla_2d(int nb_rows, int nb_cols, int array[nb_rows][nb_cols])
{
    for (int i = 0; i < nb_rows; ++i)
    {
        for (int j = 0; j < nb_cols; ++j)
        {
            printf("%d\t", array[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

Tableau 2D comme argument de fonction

```
…

int main(void)
{
  const int nb_rows = 2;
  const int nb_cols = 3;
  int arr[nb_rows][nb_cols];
  … // Init values in arr
  print_vla_2d(nb_rows, nb_cols, arr);

  return 0;
}
```

Passage de chaîne de caractères en paramètre

Appel similaire à celui pour les tableaux.

Exemple :

```
char str[10]="Bonjour";
unefonction(str);
```

L'argument de `unefonction()` est un **pointeur sur une variable de type char**.

`void unefonction(char* str)` **ou** `void unefonction(char str[])`

Comme la longueur de la chaîne est connue par le caractère `'\0'`, il n'est pas nécessaire de passer cette information en argument.

Arguments en nombre variable

Appel de fonction avec un nombre variable d'arguments.

Appel de fonction :

```
a = maxListe(3, 5, 10, -654);
c = maxListe(2, 3, 0);
```

Prototype : les paramètres anonymes sont déclarés à la fin de l'entête par la syntaxe : `...`

```
int maxListe(short nb, int x1, ...)
```

Paramètres nommés Paramètres anonymes

Arguments en nombre variable

Définition des macros dans `<stdarg.h>`

`va_list <variable>`

Pointeur permettant l'accès aux arguments

`va_start(<variable>, <dernier_arg>)`

Initialisation de **`<variable>`**. **`<dernier_arg>`** dernier des arguments explicitement nommés dans l'en-tête de la fonction.

`va_arg(<variable>, <type>)`

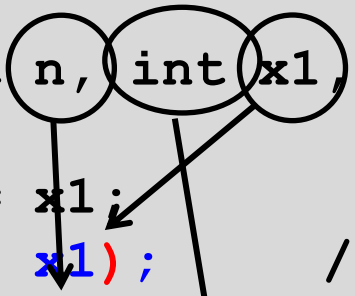
Parcours des arguments anonymes : le premier appel de cette macro donne le premier argument anonyme ; chaque appel suivant donne l'argument suivant. **`<type>`** décrit le type de l'argument.

Arguments en nombre variable - Exemple

Calcul du max d'un ensemble de nombres `m = maxListe(5, -1, 3 , 7, 7, 9);`

```
#include <stdarg.h>

int maxListe(short n, int x1, ...) {
   va_list liste;
   int x, i, max = x1;
   va_start(liste, x1);      /*Initialisation*/
   for (i = 2; i <= n; i++){
      x = va_arg(liste, int); /*Parcours */
      if (x > max)
         max = x;
   }
   return max;
}
```



Plan

1. Pointeurs de pointeurs et arithmétique
2. Interaction d'un programme avec l'OS
3. Passage de tableaux en paramètre
4. **Valeur de retour d'une fonction**
5. Pointeur sur des fonctions
6. Callback

Types de retour

Une fonction peut retourner des pointeurs, par exemple un pointeur sur des entiers

```
int* mafonction(void);
```

Exemple : écrire une fonction **`init()`** qui déclare et initialise une chaîne de caractères à « Bonjour ».

```
char* s = NULL;
s = init();
printf("%s",s);
```

→ Bonjour

Types de retour

⚠
À NE SURTOUT PAS FAIRE !!!

```
char* initFALSE()
{
    char str[] = "Bonjour";
    return str;
}

int main(void)
{
    char *s = NULL;
    s = initFALSE();
    puts(s);
    return 0;
}
```

⚠ Retourne un pointeur sur la variable locale (`str`) à la fonction `init()`, qui va disparaitre à la fin de la fonction

Compilation

```
Warning: function
returns address of a
local variable
```

Types de retour

```
Char *initOK()
{
    char *ptr = malloc((strlen(str)+1)*sizeof(char));
    str[] = "Bonjour"; // 7 chars + '\0'
    strcpy(ptr,str);
    return ptr;
}
int main(void)
{
    char *s = initOK();
    puts(s);
    free(s);
    s = NULL;
    return 0;
}
```

Retourne un pointeur sur le heap: OK

malloc() ⇒ **free()** ⇒ mettre le pointeur à NULL

Plan

1. Pointeurs de pointeurs et arithmétique
2. Interaction d'un programme avec l'OS
3. Passage de tableaux en paramètre
4. Valeur de retour d'une fonction
5. **Pointeur sur des fonctions**
6. Callback

Pointeur sur des fonctions

Adresse d'une fonction

```
00401318 00401322
Process returned 0 (0x0)
execution time : 0.016 s
Press any key to continue.
```

```c
int f1(void){
   return 0;
}

double f2(double x){
   return x;
}

int main(void){
   printf("%p %p\n", f1, f2);
   return 0;
}
```

Pointeur sur des fonctions

Comment mémoriser cette adresse dans une variable (pointeur) ?
Déclarer une **variable** (`ptrFonction`) qui contient l'**adresse d'une fonction**.

```
<type> (* ptrFonction)( <args> );
```

Rappel : pointeur sur un double `double *ptrDouble;`

Exemple d'un pointeur sur une fonction

```
double (*ptrFunction)(int, double);
```

`ptrFunction` est une variable pointeur destinée à contenir l'adresse de fonctions à 2 arguments `(int, double)` retournant un `double`.

Pointeur sur des fonctions

Affectation d'un pointeur de fonction

```
double myFunction(int n, double x){
  return n+x;
}

int main(void) {
  double (*ptrFunction)(int, double);
  ptrFunction = myFunction;

  return 0;
}
```

Pointeur sur des fonctions

Comment appeler une fonction à partir d'un pointeur de fonction ? Par **déréférencement du pointeur de fonction**

```
double myFunction(int n, double x) {
  return (double)n+x;
}

int main(void) {
  double (*ptrFonction)(int, double);
  ptrFonction = myFunction;
  double x =(*ptrFonction)(4, 5.7);
  return 0;
}
```

Opérateur de déréférencement

≡ `x=myFunction(4,5.7)`

Pointeur sur des fonctions

Déclaration d'un **tableau de pointeurs de fonctions**

```
<type> (* tabPtrFonction[taille] )( <args> );
```

Exemple

```
double add(double x, double y){ return x+y;}
double sub(double x, double y){ return x-y;}
...
double (*tabFct[4])(double,double) =
                              {add,sub,mult,div};
tabFct[0]=sub; // => no more 'add' function!
x=(*tabFct[0])(2,3);
```

Résumé

Déclaration d'un pointeur `ptrFct` sur une fonction qui :

reçoit deux `int` et renvoie un `int`

```
int(*ptrFct)(int, int);
```

ne reçoit aucun argument et ne retourne rien

```
void(*ptrFct)(void);
```

reçoit un pointeur sur `int` et renvoie un pointeur sur un `int`

```
int*(*ptrFct)(int*);
```

Déclaration d'un tableau de fonctions qui reçoivent un **pointeur sur** `int` et renvoient un **pointeur sur** `int`.

```
int*(*ptrFct[taille])( int* );
```

Résumé

Initialisation

D'un pointeur sur des fonctions `ptrFct` recevant deux `int` et renvoyant un `int`

```
int(*ptrFct)(int, int) = mafct;
```

D'un tableau de fonctions recevant un pointeur sur `int` et renvoyant un pointeur sur un `int`.

```
int*(*tabPtrFct[taille])( int* ) = {f1, f2, f3,…};
```

Affectation d'un pointeur sur des fonctions `ptrFct` :

```
ptrFct = mafct;
```

```
tabPtrFct[0] = maFct;
```

Plan

1. Pointeurs de pointeurs et arithmétique
2. Interaction d'un programme avec l'OS
3. Passage de tableaux en paramètre
4. Valeur de retour d'une fonction
5. Pointeur sur des fonctions
6. **Callback**

Callback

Callback

*Une **fonction de rappel** "callback" est une fonction qui est passée en argument à une autre fonction. Cette dernière peut alors faire usage de cette fonction de rappel comme de n'importe quelle autre fonction, alors qu'elle ne la connaît pas par avance.* [wikipédia]

Exemples d'application

Programmation événementielle

Associer une action (une fonction) à un événement, par exemple un click souris

Fonction dont le comportement peut être modifié

Par exemple, tri croissant/décroissant ► Passer la fonction de comparaison en argument

Callback

```c
#include <math.h>

double zeroValue(double (*ptrFct)(double))
{
  return (*ptrFct)(0.0);
}
int main(void)
{
   double x,y;
   x = zeroValue(sin);
   y = zeroValue(cos);
   printf("%f,%f",x,y);

   return 0;
}
```

```
0.000000,1.000000
```

Exercices

[Exercices du chapitre 11]