Programmation « orientée système »
LANGAGE C – POINTEURS (1/5)

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

Objectifs du cours d'aujourd'hui

Accès mémoire : les **pointeurs** :

- Définition et justification (utilité)
- Utilisation
- Passage « par référence »
- Pointeurs / Références

EPFL

Les pointeurs, à quoi ça sert ?

En programmation, les pointeurs servent essentiellement à trois choses :

① à permettre un *partage* d'objet *sans duplication* entre divers bouts de code
☞ **« référence »**

② à pouvoir *choisir des éléments* non connus *a priori* (au moment de la programmation)
☞ **généricité**

③ à pouvoir manipuler des objets dont la *durée de vie* ($\simeq$ portée dynamique) *dépasse* les blocs dans lesquels ils sont déclarés (*portée*, au sens syntaxique)
☞ **allocation dynamique**

Note : les pointeurs n'existent pas dans tous les langages en tant que type explicitement manipulable par le programmeur (p.ex. Java).

EPFL

Les pointeurs, à quoi ça sert ?

Exemple :
Vous souvenez vous de la fin de l'exercice sur les intégrales ?
Comment faire pour ne pas recompiler le programme pour chaque nouvelle fonction ?

Supposons que vous ayez préprogrammé 5 fonctions :

```
double f1(double x);
...
double f5(double x);
```

et vous donnez le choix à l'utilisateur :

```
do {
  printf("De quelle fonction voulez-vous calculer "
         "l'intégrale [1-5] ?\n");
  scanf("%d", &rep);
} while ((rep < 1) || (rep > 5));
```

Comment manipuler de façon générique la réponse de l'utilisateur ?
$\Rightarrow$ avec un **pointeur** sur la fonction correspondante.

EPFL

le programme complet 1/2

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

double f1(double x) { return x*x;            }
double f2(double x) { return exp(x);         }
double f3(double x) { return sin(x);         }
double f4(double x) { return sqrt(exp(x));   }
double f5(double x) { return log(1.0+sin(x)); }

/* Fonction est un nouveau type : pointeur sur des fonctions *
 * prenant un double en argument et retournant un double     */
typedef double (*Fonction)(double);

Fonction demander_fonction(void)
{
    int rep;
    Fonction choisie;
    do {
      printf("De quelle fonction [...] calculer l'intégrale [1-5] ?\n");
      scanf("%d", &rep);
    } while ((rep < 1) || (rep > 5));

    switch (rep) {
      case 1: choisie = f1 ; break ;
      case 2: choisie = f2 ; break ;
```


le programme complet 2/2

```c
      case 3: choisie = f3 ; break ;
      case 4: choisie = f4 ; break ;
      case 5: choisie = f5 ; break ;
    }
    return choisie;
}

double demander_nombre(void) { ... }
double integre(Fonction f, double a, double b) { ... }

int main(void) {
  double a;
  double b;
  Fonction choix;

  a = demander_nombre();
  b = demander_nombre();
  choix = demander_fonction();
  printf("Integrale entre %lf et %lf :\n", a, b);
  printf("%f\n", integre(choix, a, b));
  return 0;
}
```

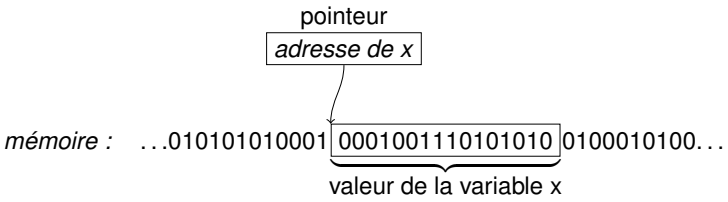
Note : ce programme peut encore être amélioré, notamment en utilisant des tableaux...

Les pointeurs

Une variable est physiquement identifiée de façon unique par son **adresse**, c'est-à-dire l'adresse de l'emplacement mémoire qui contient sa valeur.

Un **pointeur** est une variable qui contient l'**adresse** d'un autre objet informatique.

☞ une « *variable de variable* » en somme



Les pointeurs (2) : une analogie

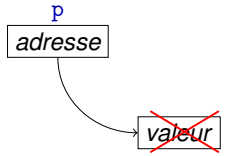
Un pointeur c'est comme la page d'un carnet d'adresse
(sur lesquelles on ne peut écrire qu'une seule adresse à la fois) :

déclarer un pointeur	ajouter une page dans le carnet (mais cela ne veut pas dire qu'il y a une adresse écrite dessus !)
allouer un pointeur `p`	aller construire une maison quelque part et noter son adresse sur la page `p` (mais `p` n'est pas la maison, c'est juste la page qui contient l'adresse de cette maison !)

Les pointeurs (2) : une analogie

Un pointeur c'est comme la page d'un carnet d'adresse
(sur lesquelles on ne peut écrire qu'une seule adresse à la fois) :

« libérer un pointeur » `p` (en fait, c'est « libérer la mémoire pointée par le pointeur » `p`)



Aller détruire la maison dont l'adresse est écrite en page `p`.
Cela ne veut pas dire que l'on a effacé l'adresse sur la page `p` ! ! mais juste que cette maison n'existe plus.
Cela ne veut pas non plus dire que toutes les pages qui ont la même adresse que celle inscrite sur la page `p` n'ont plus rien (mais juste que l'adresse qu'elles contiennent n'est plus valide)

Objectifs
Utilité des pointeurs
Pointeurs : définition
Opérateurs
Passage par référence
const pointeurs
Pointeurs et références

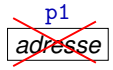
Les pointeurs (2) : une analogie

Un pointeur c'est comme la page d'un carnet d'adresse
(sur lesquelles on ne peut écrire qu'une seule adresse à la fois) :

`p1 = p2`

On recopie à la page `p1` l'adresse écrite sur la page `p2`. Cela ne change rien à la page `p2` et surtout ne touche en rien la maison dont l'adresse se trouvait sur la page `p1` !

`p1 = NULL`



On gomme la page `p1`. Cela ne veut pas dire que cette page n'existe plus (son contenu est juste effacé) ni (erreur encore plus commune) que la maison dont l'adresse se trouvait sur `p1` (c.-à-d. celle que l'on est en train d'effacer) soit modifiée en quoi que ce soit ! !
Cette maison est absolument intacte !

©EPFL 2023
Jean-Cédric Chappelier

Objectifs
Utilité des pointeurs
Pointeurs : définition
Opérateurs
Passage par référence
const pointeurs
Pointeurs et références

Les pointeurs (3) : la pratique

La déclaration d'un pointeur se fait selon la syntaxe suivante :

`type* identificateur;`

Cette instruction déclare une variable de nom *`identificateur`* de type pointeur sur une valeur de type *`type`*.

Exemple :
`int* px;`
déclare une variable `px` qui pointe sur une valeur de type `int`.

L'initialisation d'un pointeur se fait comme pour les autres variables :

`type* identificateur = adresse;`

Exemples :

```
int* ptr = &i;
int* ptr = NULL; /* ne pointe NULLe part */
```

©EPFL 2023
Jean-Cédric Chappelier

Objectifs
Utilité des pointeurs
Pointeurs : définition
Opérateurs
Passage par référence
const pointeurs
Pointeurs et références

Opérateurs sur les pointeurs

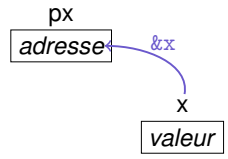
C possède deux *opérateurs* particuliers en relation avec les pointeurs : `&` et `*`.

`&` est l'opérateur qui
retourne l'adresse mémoire de la valeur d'une variable

Si `x` est de type `type`, `&x` est de type `type*` (pointeur sur `type`).

Exemple :

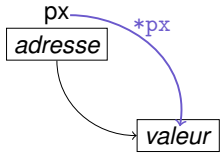
```
int i = 3;
int* ptr = NULL; /* ptr est un pointeur
                  * sur un entier      */
// ...
ptr = &i; // ptr pointe sur la variable i
```



©EPFL 2023
Jean-Cédric Chappelier

Opérateurs sur les pointeurs (2)

`*` est l'opérateur qui **retourne la valeur pointée par une variable pointeur**. Si `px` est de type `type*`, `(*px)` est la valeur de type `type` pointée par `px`.



Exemple :

```
int i = 3;
int* ptr = NULL;

ptr = &tr;
printf("%d\n", *ptr); // affiche la valeur pointee par ptr
```

Notes :

- `*&i` est donc strictement équivalent à `i`
- structures : `p->x` est équivalent à `(*p).x`

EPFL

Houlala !

GARE AUX CONFUSION !

Concernant les pointeurs, C utilise **malheureusement** une *notation identique*, `*`, pour *deux choses différentes* ! (sans parler de la multiplication !)

`type* ptr;` **déclare** une variable `ptr` comme un pointeur sur un type de base `type`	`*ptr` **accède** au contenu de l'endroit pointé par `ptr`

CE N'EST PAS LA MÊME CHOSE !

EPFL

Pointeurs et passage par référence

Comme un pointeur contient l'**adresse mémoire** d'une valeur, si l'on passe un pointeur en argument d'une fonction, *toute modification faite sur cette valeur à l'intérieur de la fonction sera répercutée à l'extérieur*.
$\Longrightarrow$ **effet de bord**

mais très utile en C pour simuler le passage par référence

Utilisez donc **les pointeurs pour faire des passages par référence** !

☞ Rappel : c'est bien pour cela qu'on écrit :

```
scanf("%d", &x);
```


EPFL

Pointeurs et passage par référence (2)

Exemple :

```
void swap(int* a, int* b) {
  int const tmp = *a;
  *a = *b;
  *b = tmp;
}
int main(void) {
  int i = 3, j = 2;
  printf("%d,%d\n", i, j); /* affiche 3,2 */
  swap(&i, &j);
  printf("%d,%d\n", i, j); /* affiche 2,3 */
  return 0;
}
```

Exercice : comment écrire `swap` en Java ?

EPFL

Optimisation

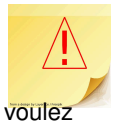
Un moyen d'**éviter la copie locale** du passage par valeur (d'objets complexes) est d'utilise un passage par adresse (pointeur).

Mais comme il s'agit d'une optimisation et non pas d'un vrai passage par référence (c.-à-d. on ne veut pas modifier la valeur passée), on n'autorisera pas la fonction à modifier ses arguments en **protégeant la valeur pointée** par le mot `const`.

Exemple :

```
Matrice addition (Matrice const * m1,
                  Matrice const * m2);
```


Optimisation (2)



Conseil : utilisez toujours `const` dans vos passages par pointeurs sauf si vous voulez **vraiment** modifier la variable pointée.

Note : on utilisera la même optimisation (adresse plutôt qu'objet) pour la valeur de retour lorsqu'il s'agit de structures compliquées :

Par exemple :

```
Matrice* addition (Matrice const * a, Matrice const * b)
{
   Matrice* resultat = malloc(sizeof(Matrice));
   if (resultat != NULL) {
     // algorithme d'addition
     // ...
   }
   return resultat;
}
```

mais attention ! : prévoir le `free` quelque part...

☞ cours de la semaine prochaine

Notez bien le `malloc` et **surtout pas (!!!)** :

```
Matrice resultat; ....; return &resultat;
```


Pointeurs `const` et pointeurs sur des `const`

type `const*` `ptr;` (ou `const` *type*`*` `ptr`)
déclare un **pointeur sur un objet constant** de type *type* : on ne pourra pas modifier la valeur de l'objet au travers de `ptr` (mais on pourra faire pointer `ptr` vers un autre objet).

type`* const ptr = &obj;`
déclare un **pointeur constant sur un objet** `obj` de type *type* : on ne pourra pas faire pointer `ptr` vers autre chose (mais on pourra modifier la valeur de `obj` au travers de `ptr`).

Pour résumer : `const` s'applique toujours au type directement précédent, sauf si il est au début, auquel cas il s'applique au type directement suivant.

Pointeurs `const` et pointeurs sur des `const`

```
int i = 2, j = 3;
int const * p1 = &i;
int* const p2 = &i;

printf("%d,%d,%d\n", i, *p1, *p2); /* affiche 2,2,2 */

*p1 = 5; /* erreur de compilation : on ne peut pas
          * modifier au travers de p1 */
*p2 = 5; /* OK, licite */

printf("%d,%d,%d\n", i, *p1, *p2); /* affiche 5,5,5 */

p1 = &j; /* licite */
p2 = &j; /* erreur de compilation : on ne peut pas
          *  modifier p2 */

printf("%d,%d,%d\n", i, *p1, *p2); /* affiche 5,3,5 */
```

Pointeurs et références

En programmation, il existe la notion de **référence**, proche de la notion de pointeur mais néamoins *subtilement différente*.
(certains langage d'ailleurs, dont C++, offrent les deux : pointeurs *et* références.
Les références **n'existent** par contre **pas en C**.)

Une référence est en fait un identificateur (c.-à-d. *un autre nom*).

À la différence des pointeurs, une référence

- peut avoir une sémantique de = **très** différente des pointeurs
- doit toujours être initialisée
- ne peut jamais être nulle (c.-à-d. ne pas référencer quelque chose)
- ne peut référencer qu'un seul et même objet (tout au long du programme)
- ne peut pas référencer une autre référence
- n'a pas d'adresse en tant que telle (c.-à-d. on ne peut pas avoir de pointeur sur des références. En fait, il est même tout à fait possible qu'une référence n'existe pas en tant que telle dans la mémoire, contrairement à un pointeur, mais ne soit qu'un alias géré par l'éditeur de liens).

EPFL

Pointeurs et références

Pour faire simple : une référence est comme un pointeur qu'on ne peut pas changer (« `* const` », donc) et qui est toujours affecté (à une adresse valide).

L'utilisation des références est donc limitée au cas ① des trois cas d'utilisation des pointeurs : on ne peut pas les utiliser pour la généricité, ni pour l'allocation dynamique.

Les références, par contre, sont beaucoup plus faciles à manipuler que les pointeurs et permettent d'écrire du code plus sûr.

Adage (pour les langages qui ont pointeurs et références, **pas** pour le C donc :-( ) :
utilisez des références quand vous pouvez, utilisez des pointeurs quand vous devez.

EPFL

Ce que j'ai appris aujourd'hui

Comment utiliser les **pointeurs** pour :

- la généricité (p.ex. choix d'éléments non connus *a priori*) ;
- partage de données (sans duplication) ;
- dont en particulier le passage « par référence » (par valeur de pointeur).

Semaine prochaine :

- l'allocation dynamique ;

puis :

- comment représenter et utiliser des chaînes de caractères
- pointeurs sur fonctions
- approfondissements pratiques : retour sur le `swap`, passage d'arguments en Java, copie profonde, ...
- pointeurs et tableaux
- arithmétique des pointeurs

EPFL