CS–202 Computer Systems – C Lectures

C02 – POINTERS 1: BASICS

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

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).

Jean-Cédric Chappelier

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.

Jean-Cédric Chappelier

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

```
      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;
}
```

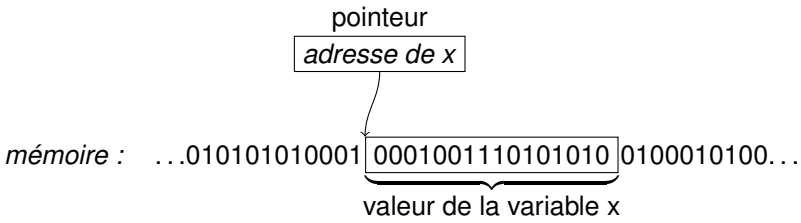
Note : ce programme peut encore être amélioré, notamment en utilisant des tableaux...

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) :

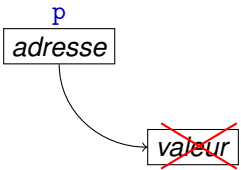
déclarer un pointeur	ajouter une page dans le carnet (mais cela ne veut pas dire qu'il y a une adresse écrite dessus !)
allouer un pointeur `p`	aller construire une maison quelque part et noter son adresse sur la page `p` (mais `p` n'est pas la maison, c'est juste la page qui contient l'adresse de cette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

Jean-Cédric Chappelier

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)

Les pointeurs (2) : une analogie

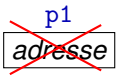
Un pointeur c'est comme la page d'un carnet d'adresse
(sur lesquelles on ne peut écrire qu'une seule adresse à la fois) :

`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 !



Jean-Cédric Chappelier
EPFL

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 */
```


Jean-Cédric Chappelier

Opérateurs sur les pointeurs

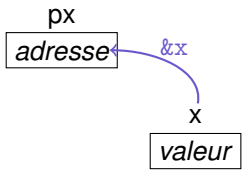
C possède deux *opérateurs* particuliers en relation avec les pointeurs : `&` et `*`.

`&` 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`).

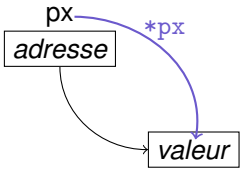
Exemple :

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



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`

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 !

Jean-Cédric Chappelier

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);
```


Jean-Cédric Chappelier

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 ?

Jean-Cédric Chappelier

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);
```


Jean-Cédric Chappelier

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.

Jean-Cédric Chappelier

Pointeurs const et pointeurs sur des const

```c
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).

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.

Jean-Cédric Chappelier

Allocation de mémoire

Il y a **deux façons** d'*allouer de la mémoire* en C.

① *déclarer des variables*
La réservation de mémoire est déterminée à la compilation : allocation statique.

☞ allocation « **sur la pile** »

② **allouer dynamiquement** de la mémoire **pendant l'exécution** d'un programme.

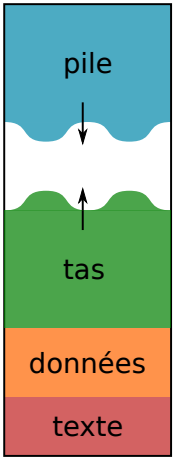
L'allocation dynamique permet également de réserver de la mémoire **indépendamment de toute variable** : on pointe directement sur une zone mémoire plutôt que sur une variable existante.

☞ allocation « **sur le tas** »

Jean-Cédric Chappelier

Pile et tas

Mémoire virtuelle d'un processus :



pile (« *stack* ») : variables locales

Note : taille limite de la pile : `ulimit -s` ; typiquement quelques Mo.

tas (« *heap* ») : allocation dynamique

Note : le tas est en général limité uniquement par l'espace d'adressage (`ulimit -m`)...
...jusqu'à « mettre à genoux » la machine (mémoire virtuelle, swap, ...)

« données » : variables statiques et globales

« texte » : code du programme et constantes

Jean-Cédric Chappelier
EPFL

Allocation dynamique de mémoire

C possède deux fonctions `malloc` et `calloc`, définies dans `stdlib.h`, permettant d'**allouer** dynamiquement de la mémoire.

Note : il existe également `realloc` dont nous parlons plus loin.

```
pointeur = malloc(taille);
```

réserve une zone mémoire de taille *taille* et met l'adresse correspondante dans `pointeur`.

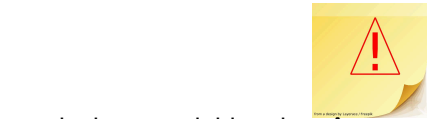
Pour aider à la spécification de la taille, on utilise souvent l'opérateur `sizeof` qui retourne la taille mémoire d'un type (donné explicitement ou sous forme d'une expression).
(Le type de retour de `sizeof` est `size_t`. `printf`/`scanf` : `%zu`)

Par exemple pour allouer de la place pour un `double` :

```
pointeur = malloc(sizeof(double));
```


EPFL

Allocation dynamique : `calloc`



Si l'on souhaite allouer de la mémoire consécutive pour plusieurs variables de même type (typiquement un tableau, dynamique), on **préfèrera** `calloc` à `malloc` :

```
void* calloc(size_t nb_elements, size_t taille_element);
```

Par exemple pour allouer de la place pour 3 `double` consécutifs :

```
pointeur = calloc(3, sizeof(double));
```

> *The use of `calloc()` is strongly encouraged when allocating multiple sized objects in order to avoid possible integer overflows.*
>
> [`malloc` man-page in OpenBSD]

Le problème ?

☞ `p = malloc(n * sizeof(Type))` peut engendrer un overflow sur la multiplication et allouer en fait bien moins que `n` cases, ce qui peut ensuite conduire à un débordement mémoire sur `p[i]`.

EPFL

Et ça peut vraiment arriver ?

Voici un exemple de ce bug dans le code du server OpenSSH 3.1 :

```
unsigned int nresp;
char** reponse;
...
nresp = packet_get_int();
if (nresp > 0) {
    response = malloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; ++i)
        response[i] = packet_get_string(NULL);
}
...
```

(tiré de la fonction `input_userauth_info_response()` dans `auth2-chall.c`)

Où est le bug ?

Différences entre `malloc` et `calloc`

`calloc` est protégé contre l'erreur de débordement de la multiplication,

mais en plus `calloc` initialise à 0 (le contenu de) la zone allouée.

Avec `malloc` la mémoire n'est pas initialisée.

Pour initialiser de la mémoire : `memset` (défini dans `string.h`) :

`memset(`*`pointeur, valeur, taille`*`);`

Exemple : `memset(ptr, 255, sizeof(*ptr));`

Conseil : initialisez **toujours toute** la mémoire utilisée.

Par exemple :

```
struct Machin bidule;
memset(&bidule, 0, sizeof(bidule));
```


Jean-Cédric Chappelier

Test d'allocation correcte

Les fonctions `malloc` et `calloc` retournent `NULL` si l'allocation n'a pas pu avoir lieu.

Pour cela, on écrit souvent l'allocation mémoire comme suit

```
pointeur = calloc(nombre, sizeof(type));
if (pointeur == NULL) {
  /* ... gestion de l'erreur ... */
  /* ... et sortie (return code d'erreur) */
}
/* suite normale */
```

Libération mémoire allouée

`free` permet de **libérer** de la mémoire allouée par `calloc` ou `malloc`.

`free(`*`pointeur`*`)`

libère la zone mémoire allouée au pointeur `pointeur`.

C'est-à-dire que cette zone mémoire peut maintenant être utilisée pour autre chose.
Il **ne** faut **plus y accéder !**...

Je vous conseille donc par mesure de prudence de faire suivre tous vos `free` par une commande du genre :

`pointeur` `= NULL;`

Règle absolue : *Toute zone mémoire allouée par un* `[cm]alloc` *doit impérativement être libérée par un* `free` *correspondant !*
(☞ « garbage collecting »)

Veillez à bien vous assurer que c'est le cas dans vos programmes
(attention aux structures de contrôle !)

Allocation mémoire : exemple

```
int* create_long_live_int()
{
  int* px = NULL;
  // ...
  px = malloc(sizeof(int));
  if (px != NULL) {
    *px = 20; /* met la valeur 20 dans la zone  *
               * mémoire pointée par px.        */
    // ...
  }
  return px;
}

{ // ... ailleurs
  int* q = create_long_live_int();
  printf("%d\n", *q);
  // ...
  return q;
}

{ // ... encore ailleurs, plus loin
  int* r = ...;
  // ...
  free(r); // quand on n'en as plus besoin
  r = NULL;
}
```

Note : sauf exception (*très* grosse taille), on ne fait pas `malloc` et `free` dans la même portée !!

L'allocation dynamique n'est que le « cas d'utilisation **numéro 3** ». Elle n'est pas utile pour les cas 1 et 2 !!

Jean-Cédric Chappelier

Toujours allouer avant d'utiliser !

Attention ! Si on essaye d'utiliser (pour la lire ou la modifier) la valeur pointée par un pointeur pour lequel aucune mémoire n'a été réservée, une erreur de type `Segmentation fault` ou `Bus Error` se produira à l'exécution.

Exemple :

```
int* px;
*px = 20;    /* ! Erreur : px n'a pas été alloué !! */
```

Compilation : OK

Exécution

➥ Segmentation fault

Conseil : Initialisez **toujours** vos pointeurs. Utilisez `NULL` si vous ne connaissez pas encore la mémoire pointée au moment de l'initialisation :

```
int* px = NULL;
```


Jean-Cédric Chappelier

«Bus Error» ou «Segmentation Fault» ?

C'est en gros la même chose : accès à de la mémoire interdite.

Il y a cependant une subtile différence entre « `segmentation fault` » et « `bus error` ».

« `bus error` » signifie que le noyau n'a pas pu détecter l'erreur d'accès mémoire par lui-même, mais que c'est de la mémoire physique (le matériel) qu'est venu le signal d'erreur.

Jean-Cédric Chappelier

Récapitulons : règles de bon usage

Si vous suivez ces règles, vous vous faciliterez la vie de programmeur avec pointeurs :

- Toute zone mémoire allouée dynamiquement (`malloc`) doit **impérativement** être libérée par un `free` correspondant !

  et c'est celui qui alloue qui doit libérer

  Corollaire : si vous fournissez une fonction qui alloue de la mémoire vous **devez** fournir une fonction « réciproque » qui libère la mémoire, de sorte que celui qui appelle votre première fonction puisse respecter la règle ci-dessus (en appelant la seconde fonction)

- Testez systématiquement vos `malloc`/`calloc` :

```
pointeur = calloc(nombre, sizeof(type));
if (pointeur == NULL) {
  /* ... gestion de l'erreur ... */
  /* ... et sortie (return code d'erreur) */
}
/* suite normale */
```

Récapitulons : règles de bon usage

Si vous suivez ces règles, vous vous faciliterez la vie de programmeur avec pointeurs :

- Pour les allocations multiples, utilisez `calloc` et non pas `malloc`
- Initialisez toujours vos pointeurs.
  Utilisez `NULL` si vous ne connaissez pas encore la mémoire pointée au moment de l'initialisation
- Initialisez toujours **toute** la mémoire utilisée (`memset`).
- ajoutez un *`ptr`* `= NULL;` après chaque `free`
- utilisez toujours `const` dans vos « faux » passages par référence (optimisation)
-  utilisez les outils supplémentaires de votre environnement de développement : options du compilateur, debugger, programmes de surveillance de la mémoire (e.g. `valgrind`, `Address Sanitizer`, ...), programmes de recherche de bugs (`scan-build`, `splint`, `flawfinder`, ...)

Tableaux dynamiques (rappel)

Un **tableau dynamique** est un ensemble d'éléments homogène et *à accès direct* de taille non fixée *a priori*

Interface :

- accès à un élément quelconque (sélecteur)
- modifier un élément quelconque (modificateur)
- insérer/supprimer un élément en fin du tableau (modificateur)
- tester si le tableau est vide (sélecteur)
- parcourir le tableau (itérateur)

Au contraire de Java (`ArrayList`) ou C++ (`vector`), il n'y a pas, en C, de bibliothèque standard fournissant de telles structures de données abstraites. Voyons comment les implémenter...

Jean-Cédric Chappelier

Les tableaux en C (RAPPEL)

En C, on a :

		taille initiale connue *a priori* ? non	taille initiale connue *a priori* ? oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	—(1)	—
	non	(C99) VLA	*type*[N]

(1) N'existe pas en C, mais possible grâce au « flexible array member »

Pour rappel en Java, on utilise :

		taille initiale connue *a priori* ? non	taille initiale connue *a priori* ? oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	ArrayList	
	non	*type*[]	

Tableaux dynamiques : exemple d'utilisation

```
vector v;

// initialisation
if (vector_construct(&v) == NULL) {
  ... // erreur
}

// utilisation
if (vector_push(&v, 2) == 0) {
  ... // erreur
}

...

// libération mémoire
vector_delete(&v);
```


Jean-Cédric Chappelier

Tableaux dynamiques : exemple d'implémentation

```
typedef int type_el; // pour définir le type d'un élément

typedef struct {
   size_t size;      // nombre d'éléments utilisés dans le tableau
   size_t allocated; // nb élements déjà alloués
   type_el* content; // tableau de contenu (alloc. dyn.)
} vector;
```

d'où : **réallocation dynamique** quand on dépasse la taille allouée

☞ allocation dynamique par blocs de taille fixe (`allocated` est un multiple de la taille des blocs)

Comment ?

☞ fonction `realloc` :

$$ptrnew = \texttt{realloc}(ptrold, newsize) \ ;$$

Jean-Cédric Chappelier

realloc

`realloc` :

- change la taille de la zone allouée, aussi bien en augmentation qu'en diminution
- déplace le pointeur (« réalloue ») si nécessaire, tout en gardant l'intégrité des données (recopie)
- libère l'ancienne mémoire si nécessaire
- l'ancienne zone mémoire est inchangée si `realloc` échoue (c.-à-d. retourne `NULL`)
- la nouvelle zone mémoire supplémentaire (lorsqu'on augmente) n'est pas initialisée
- si *`ptrold`* est `NULL`, c'est un `malloc(`*`newsize`*`)`
- si *`newsize`* est nulle (et *`ptrold`* n'est pas `NULL`), c'est un `free(`*`ptrold`*`)`

Jean-Cédric Chappelier

Tableaux dynamiques : initialisation

```
vector* vector_construct(vector* v) {
  if (v != NULL) {
    vector result = { 0, 0, NULL };
    result.content = calloc(VECTOR_PADDING, sizeof(type_el));
    if (result.content != NULL) {
      result.allocated = VECTOR_PADDING;
    } else {
      // retourne NULL si on n'a pas pu allouer la mémoire nécessaire
      return NULL;
    }
    // écriture atomique
    *v = result;
  }
  return v;
}
```

VECTOR_PADDING : taille des blocs choisie. Par exemple :

```
#define VECTOR_PADDING 128
```


Jean-Cédric Chappelier

Tableaux dynamiques : libération mémoire

Attention ! Comme on a fourni une fonction faisant l'allocation (`vector_construct`), il faut aussi fournir une fonction pour la libération :

```
void vector_delete(vector* v) {
  if ((v != NULL) && (v->content != NULL)) {
     free(v->content);
     v->content = NULL;
     v->size = 0;
     v->allocated = 0;
  }
}
```

NOTE : « `x->y` » est la même chose que « `(*x).y` »

Jean-Cédric Chappelier

Tableaux dynamiques : ajout d'un élément

Exemple d'ajout d'un élément à la fin du tableau (et retourne la taille (nombre d'éléments) du tableau après ajout, 0 en cas d'échec) :

```
size_t vector_push(vector* v, type_el val) {
  if (v != NULL) {
    while (v->size >= v->allocated) {
      if (vector_enlarge(v) == NULL) {
        return 0;
      }
    }
    v->content[v->size] = val;
    ++(v->size);
    return v->size;
  }
  return 0;
}
```


Jean-Cédric Chappelier

Tableaux dynamiques : augmentation de taille

```
vector* vector_enlarge(vector* v) {
  if (v != NULL) {
    vector result = *v;
    result.allocated += VECTOR_PADDING;
    if ((result.allocated > SIZE_MAX / sizeof(type_el)) ||
        ((result.content = realloc(result.content,
                                   result.allocated * sizeof(type_el)))
         == NULL)) {
      return NULL; /* retourne NULL en cas d'échec ;
                    * v n'a pas été modifié.          */
    }
    // affectation finale, tout d'un coup
    *v = result;
  }
  return v;
}
```


Jean-Cédric Chappelier

SIZE_MAX

SIZE_MAX est défini dans la bibliothèque standard stdint.h depuis C99, sinon :

```
#ifndef SIZE_MAX
#define SIZE_MAX (~(size_t)0)
#endif
```

Les pointeurs

Déclaration : `type* identificateur;`

Adresse d'une variable : `&variable`
Accès au contenu pointé par un pointeur : `*pointeur`

Pointeur sur une constante : `type const* ptr;`
Pointeur constant : `type* const ptr = adresse;`

Allocation mémoire :

```
#include <stdlib.h>

pointeur = malloc(sizeof(type));
pointeur = calloc(nombre, sizeof(type));
pointeur = realloc(pointeur, sizeof(type));
```

Libération de la zone mémoire allouée : `free(pointeur);`

Pointeur sur une fonction : `type_retour (* ptrfct)(arguments...)`

EPFL