Programmation « orientée système »
LANGAGE C – POINTEURS (2/5)

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

Accès mémoire *et gestion dynamique* :

- Allocation dynamique
- Exemple concret : tableaux dynamiques

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** »

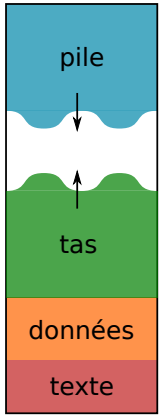
② **allouer dynamiquement** de la mémoire **pendant l'exécution** d'un programme.

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** »

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

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));
```


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]`.

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));
```

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 !!

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;`

«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.

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`, ...)

[hors cours] Monsieur, et en C++... ? [hors cours]

C	C++
`ptr = malloc(sizeof(Type));`	`ptr = new Type;`
`ptr = calloc(n, sizeof(Type));`	`ptr = new Type[n];`
`free(ptr);`	`delete ptr;` ou `delete[] ptr;`
`ptr = NULL;`	`ptr = 0;`

AVERTISSEMENT AUX PROGRAMMEURS JAVA !
N'utilisez `new` **que** pour faire de l'allocation dynamique (cas 3 de l'utilisation des pointeurs) *et pour rien d'autre* !!!
Je n'ai que trop vu de (mauvais) programmeurs C++ (venant de Java) écrire des choses comme :

```
{
    Objet x = new Objet; // x est local donc !
    ...
    delete x; /* la duree de vie de x ne dépasse pas sa portée
               *  ==> l'allocation dynamique est inutile !  */
}
```

A PROSCRIRE !!

(surtout que souvent le `delete` est oublié !!)

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...

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);
```


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 = realloc(ptrold, newsize) ;
```

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*`)`

EPFL

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`

EPFL

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` »

EPFL

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;
}
```


EPFL

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;
}
```


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...)*

Ce que j'ai appris aujourd'hui

Comment utiliser les **pointeurs** pour :

- l'allocation dynamique ;
- en particulier : comment créer des tableaux dynamiques.

Semaine prochaine :

- comment représenter et utiliser des chaînes de caractères ;
- pointeurs sur fonctions ;
- forçage de type (casting).

Puis :

- retour sur le *swap* et le « passage par référence » en Java ;
- lien entre pointeurs et tableaux ;
- arithmétique des pointeurs.