

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** »

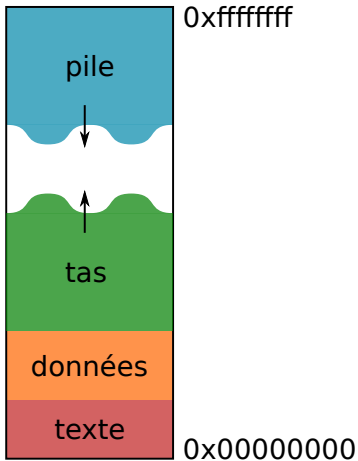
② **allouer dynamiquement** de la mémoire **pendant l'exécution** d'un programme.

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 : variables locales

tas : allocation dynamique

« 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

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* px = NULL;
...
px = malloc(sizeof(int));
if (px != NULL) {
    *px = 20; /* met la valeur "20" dans la zone *
              * mémoire pointée par px.          */
    printf("%d\n", *px);
    ...
    free(px);
    px = NULL;
}
```



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
- ▶ 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, mpatrol, purify, ...), programmes de recherche de bugs (scan-build, splint, lint), flawfinder, ...)



Monsieur, et en C++... ?



C

```
ptr = malloc(sizeof(Type));  
ptr = calloc(n, sizeof(Type));  
free(ptr);  
ptr = NULL;
```

C++

```
ptr = new Type;  
ptr = new Type[n];  
delete ptr; OU delete[] ptr;  
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)

En C, au contraire de Java ou C++, il n'y a pas de bibliothèque standard fournissant de telles structures de données abstraites.

Voyons comment les implémenter...

Tableaux dynamiques (2)

Exemple d'implémentation :

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

typedef struct {
    size_t size;      /* nombre d'éléments dans le tableau */
    size_t allocated; /* taille effectivement allouée */
    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
- ▶ la nouvelle zone mémoire en plus (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)`

Tableaux dynamiques (3)

Création :

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

VECTOR_PADDING : taille des blocs choisie.

Par exemple : `#define VECTOR_PADDING 128`

Tableaux dynamiques (4)



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` »

Tableaux dynamiques (5a)

Augmentation de taille :

```
vector* vector_enlarge(vector* v) {  
    vector* result = v;  
    if (result != NULL) {  
        type_el* const old_content = result->content;  
        result->allocated += VECTOR_PADDING;  
        if ((result->allocated > SIZE_MAX / sizeof(type_el)) ||  
            ((result->content = realloc(result->content,  
                                         result->allocated * sizeof(type_el)))  
             == NULL)) {  
            result->content = old_content;  
            result->allocated -= VECTOR_PADDING;  
            result = NULL;  
        }  
    }  
    return result;  
}
```

`SIZE_MAX` devrait être inclus dans la bibliothèque standard `stdint.h`,
sinon :

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

Tableaux dynamiques (6a)

Exemple d'ajout d'un élément à la fin du tableau (et retourne l'index du dernier élément après ajout, 0 en cas d'échec) :

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


Tableaux dynamiques (6b)

Exemple d'utilisation :

```
vector v;  
if (vector_construct(&v) == NULL) { ... // erreur }  
vector_push(&v, 2);  
vector_delete(&v);
```



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.