Gestion dynamique de mémoire [mm]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 20 mai 2018

Table des matières

C - Gestion dynamique de mémoire

Mots-Clés Gestion dynamique de mémoire ■
Requis Structuration de l'information, Classes ■
Difficulté ●●○ (1 h) ■

Introduction
Ce module explique la gestion dynamique de la mémoire, aborde le clonage d'objets (recopie et affectation) et donne la forme canonique de Coplien.

1 Gestion dynamique de mémoire

1.1 Gestion dynamique de mémoire

Le langage dispose de primitives adaptés à la **gestion dynamique** de la mémoire.

L'allocation dynamique de mémoire
La primitive d'allocation réclame des blocs de mémoire dans la zone de la mémoire appelée le **tas** (*heap*, en anglais) (dit aussi *mémoire dynamique* ou *RAM vive*) et la met à la disposition du programme. Cette opération est dite **allocation dynamique** (en opposition à l'allocation statique) parce que l'espace mémoire est réservé lors de l'exécution du programme (*run time*) et non lors de la compilation. Par analogie, les entités créées de cette façon sont dites **dynamiques**. La durée de vie d'une entité créée dynamiquement n'est pas limitée à la portée dans laquelle elle a été créée : elle existe depuis son point de création jusqu'à sa destruction ou jusqu'à la fin du programme.

La libération dynamique
Lorsque celle-ci est devenue inutile, la primitive de libération permet de détruire l'entité dynamique : la mémoire allouée est libérée et redonnée au système.

C : Pour les utiliser

```c
#include <stdlib.h>
```

Volume du tas
Il dépend de l'implémentation. Certaines implémentations permettent au programmeur de contrôler la taille du tas. Ces programmes sont capables de vider le tas, s'ils réalisent de grosses demandes de blocs mémoire, et qu'ils ne restituent pas au système lorsqu'ils ne sont pas utilisés. Un programme devrait être conçu pour faire face à la situation où le tas est épuisé puisque tous les langages ne fournissent pas par défaut un gestionnaire du tas.

1.2 Allocation dynamique de mémoire

En algorithmique, on suppose que le système possède une place mémoire sans limite et que l'instruction d'allocation se termine correctement. Dans les langages de programmation, des mécanismes existent pour savoir si le système d'exploitation a échoué dans son essai de réservation d'espace mémoire pour la variable pointée.

C : Allocation d'un élément

```
T *p;
p = (T*)malloc(sizeof(T));
```

Explication
Réserve une zone mémoire du nombre d'octets spécifié en paramètre et renvoie un pointeur sur la zone mémoire allouée. La fonction renvoie `NULL` en cas d'erreur.

C : Allocation d'un tableau

```
T *p;
p = (T*)calloc(nelems,sizeof(T));
```

Explication
Alloue une zone mémoire du nombre d'éléments multiplié par la taille d'un élément. La zone allouée est initialisée avec tous les bits à zéro et la fonction renvoie une adresse de type générique `void*` (`NULL` en cas d'erreur).

C : Réallocation d'une zone

```
realloc(ptr,nouvtaille)
```

Explication
Modifie la taille de la zone mémoire pointée par `ptr`. Le pointeur doit soit être `NULL`, soit contenir l'adresse d'une zone mémoire préalablement allouée. Le deuxième paramètre donne la nouvelle taille de la zone mémoire. Si elle est plus petite, les informations de la partie supprimée sont perdues. Si elle est plus grande, les informations de la partie ajoutée n'est pas initialisée (contenu indéterminé).

1.3 Libération dynamique de mémoire

Libération de mémoire
Elle ne doit être appliquée qu'à des pointeurs pointant sur une zone allouée dynamiquement.

C : Libération d'un élément

```
free(p)
```

Explication
Libère et restitue au système d'exploitation la zone mémoire pointée par le pointeur `p` et rend indéterminée la valeur de `p`.

C : Libération d'un tableau

```
free(p)
```

Explication
Libère la zone mémoire allouée au pointeur `p`.

Libération n'implique pas initialisation à nil
Lors de la libération mémoire, la variable pointeur **n'est pas** initialisée à la valeur `Nil`. Elle contient toujours la même adresse mémoire mais celle-ci ne correspond plus à celle d'une variable pointée puisque la variable pointée a été rendue au système d'exploitation. L'utilisation de cette variable pointée provoque une erreur d'exécution avec arrêt immédiat de l'algorithme.

1.4 Exemples

Attention
La libération d'une zone pointée peut avoir un effet de bord lorsque cette zone est pointée par d'autres pointeurs, ou correspond à une variable.

1.5 Création et destruction de tableaux

Création de tableaux

L'utilisation de tableaux **nécessite** un constructeur par défaut, c.-à-d. un constructeur pouvant ne prendre aucun paramètre lors de son invocation. Un tel constructeur peut :

- Ne prendre effectivement aucun paramètre.
- Avoir des paramètres qui acceptent tous une valeur par défaut.

Exemple

Le constructeur `Point::Point(int x=0,int y=0)` est un constructeur par défaut.

Le pourquoi

Le besoin d'un constructeur par défaut est lié à la syntaxe de construction des tableaux. Par exemple, pour un tableau à une dimension :

```
TElement idObjet[tailleTableau];
```

Comme vous pouvez le constater, il n'y a pas de place pour des paramètres de construction. La construction doit donc se faire sans paramètre, d'où la nécessité d'un constructeur par défaut.

Comment faire sans constructeur par défaut

Que faire si l'on doit créer un tableau d'objets sans constructeur par défaut ? La solution est un peu alambiquée :

- On créé un tableau de pointeurs.
- On appelle le constructeur souhaité sur chacun des pointeurs.

Résumé : Les différents types de tableaux

A partir d'une même classe `T`, on peut avoir :

Type de tableau	Construction des objets	Destruction	Remarques
Tableau statique d'instances statiques	T tableau[TAILLE];	Automatique	Constructeur par défaut obligatoire. Tout est détruit automatiquement

Tableau statique d'instances dynamiques	T *tableau[TAILLE];
	for (int i=0; i<TAILLE;i++)
	tableau[i]=new T(params);
		for (int i=0;i<TAILLE;i++)
		delete tableau[i];
			Le constructeur étant appelé par l'utilisateur, n'importe lequel fait l'affaire.
			Attention, chaque objet doit être détruit individuellement

Tableau dynamique d'instances statiques	T *tableau; tableau = new T[TAILLE]	delete [] tableau;	Constructeur par défaut obligatoire Seul le tableau doit être détruit explicitement
Tableau dynamique d'instances dynamiques	typedef T* PT; PT *tableau[TAILLE]; tableau = new PT[TAILLE]; for (int i=0;i<TAILLE;i++) tableau[i]=new T(params);	for (int i=0;i<TAILLE;i++) delete *tableau[i]; delete [] tableau;	Les éléments doivent être détruits individuellement, avant de détruire le tableau

1.6 La désallocation n'est pas simple

La libération mémoire est nécessaire pour ne pas épuiser le TAS et elle est à réaliser dès qu'une zone n'a plus d'utilité. Mais la gérer n'est pas une opération simple. Certains langages de programmation (JAVA, PYTHON, LISP...) la gèrent automatiquement.

Ramasse-miettes

La gestion automatique de récupération de mémoire est nommée **ramasse-miettes** (*garbage collector* en anglais).

Une stratégie prudente

Il est très important de réfléchir sur les variables et sur la libération de la mémoire dynamique. Les erreurs liées à ces mécanismes sont souvent très difficiles à cerner. Une stratégie prudente est de toujours se poser les questions suivantes :

- Est-ce que je libère tout ce que j'ai alloué ?
- Tous les pointeurs sont-ils initialisés ?
- Est-ce que je libère plusieurs fois certaines données ?
- La mémoire dynamique est-elle initialisée correctement ?

La modularité et les types abstraits, et surtout les constructeurs et destructeurs, facilitent ces manipulations et permettent de cacher les détails délicats. Tout ce qu'un constructeur alloue doit être libéré par le destructeur. Le couplage des constructeurs et destructeurs avec les allocations et libérations est l'une des techniques qui permettent de limiter les risques d'erreurs durant l'exécution d'un programme.

1.7 Les schémas mémoire

Il est primordial de connaître l'état des variables en cours d'exécution d'un algorithme grâce à un schéma mémoire.

Schéma mémoire d'un algorithme
Représente l'ensemble des variables et de leurs valeurs à une étape précise de son déroulement.

Utilité d'un schéma mémoire
Un schéma mémoire délimite trois parties distinctes :

- Le nom de l'algorithme.
- La partie des variables où toutes les variables définies seront représentées par :
  - Une valeur (ou par `?` si la variable n'a pas encore de valeur) pour les variables de type primitif.
  - Une flèche (pointant sur une case dessinée dans la partie droite u schéma) pour les variables de type Pointeur.
- La partie des instances où chaque case aura été créée par l'allocation : il y a autant de cases à représenter qu'il y a d'allocations dans l'algorithme.

Erreurs courantes
Les erreurs à éviter :

- Oublier de représenter une variable.
- Donner une mauvaise valeur à une variable.
- Représenter les instances sans une case associée.
- Oublier de préciser l'étape (au cours du déroulement de l'algorithme) représentée par le schéma mémoire.

2 Clonage d'objets : recopie et affectation

Recopier un objet dans un autre est opération assez fréquente. Deux fonctionnalités y sont dédiées en C++ : le constructeur par recopie et l'opérateur d'affectation.

2.1 Constructeur de recopie

Motivation

Le **constructeur par recopie** est très important car il permet d'initialiser un objet par clonage d'un autre. Attention, j'ai bien dit initialiser, ce qui signifie que l'objet est en cours de construction. En particulier, le constructeur par recopie est invoqué dès lors que l'on passe un objet par valeur à une fonction ou une méthode.

Syntaxe

```
K::K(const K& o);
```

Explication

Syntaxe du constructeur par recopie d'une classe `K`. Attention ! Il est extrêmement important de passer l'objet recopié par référence sous peine d'entrainer un appel récursif infini !

En l'absence de constructeur de recopie

Si vous ne fournissez pas explicitement de constructeur par recopie, le compilateur en génère automatiquement un pour vous. Celui-ci effectue une recopie binaire optimisée de votre objet... ce qui est parfait si celui-ci ne contient que des éléments simples.

En revanche, si votre objet contient des pointeurs, ce sont les valeurs des pointeurs qui vont être copiées et non pas les variables pointées, ce qui dans de nombreux cas, conduira directement à la catastrophe.

Quand doit on fournir un constructeur de recopie ?

Vous devez fournir un constructeur de recopie dès lors que le clonage d'un objet par recopie binaire brute peut entraîner un dysfonctionnement de votre classe, c'est à dire, en particulier :

- Utilisation de mémoire dynamique.
- Utilisation de ressources systèmes (fichiers, sockets, etc.).

2.2 Opérateur d'affectation

Mise en place

L'opérateur d'affectation et le constructeur de recopie sont très proches dans le sens où ils sont requis dans les mêmes circonstances et qu'ils effectuent la même opération : cloner un objet dans un autre. Il y a tout de même une différence fondamentale : « l'opérateur

d'affectation écrase le contenu d'un objet déjà existant et donc totalement construit ». Ce qui signifie que dans la majorité des cas, il faudra commencer par « nettoyer » l'objet à la manière d'un constructeur avant d'effectuer l'opération de clonage dessus.

Syntaxe

```
K& K::operator=(const K& o);
```

Explication

Syntaxe de l'opérateur d'affectation d'une classe `K`.

2.3 Initialisation ou Affectation ?

Règle : Initialisation ou Affectation

Il est parfois délicat de savoir si l'on a affaire à une affectation ou une initialisation car la syntaxe du signe « = » peut être trompeuse. Il existe pourtant une règle simple : Toute opération d'initialisation ou d'affectation dans une déclaration est l'affaire d'un constructeur.

Résumé

Le tableau suivant résume quelques cas qui doivent être lus séquentiellement et où `T` et `U` sont des classes quelconques :

Instruction	Description	Méthode mise en jeu
T t1;	Initialisation par le constructeur par défaut	T::T(void);
T t2(params);	Initialisation par un constructeur quelconque	T::T(liste params);
T t3(t1);	Initialisation par le constructeur de recopie	T::T(const T&);
T t4();	Piège : c'est le prototype de la fonction t4 qui ne prend pas de paramètre mais renvoie un objet de type T.	
T t5=t1	Initialisation par le constructeur de recopie Cette ligne est à remplacer par T t5(t1); qui fait exactement la même chose mais est moins ambigue du point de vue de la syntaxe.	T::T(const T&);
t5=t2	Affectation à l'aide de l'opérateur d'affectation	T & T::operator=(const T&);

2.4 Constructions et Destructeur (Juillet 2017)

Constructeur sans paramètre

Si l'on ne définit pas de constructeur dans une classe, le constructeur par défaut se contente de réserver l'espace mémoire pour l'objet et ses données membres mais elles ne sont pas initialisées. Pour des pointeurs, il convient donc d'initialiser les variables.

Constructeurs avec paramètres

Pour des tableaux dynamiques, on peut donc définir un constructeur avec paramètres qui s'occupe de la création et de l'initialisation.

Constructeurs et initialiseurs (1)

Dans le cas de constructeurs multiples de même nombre de paramètres, c'est le type des paramètres qui détermine le constructeur appelé.

Constructeurs et initialiseurs (2)

Dans le cas de constantes ou de références, l'initialiseur **doit** les initialiser. Exemple :

```
#include <iostream>
using namespace std;
class A
{
private:
   const int ci1 = 7; // initialisation a la declaration
   const int ci2; // ou utilisation d'un initialiseur
   double& cf; // initialiseur obligatoire
public:
   A(double& f): ci2(5), cf(f) {}
   A(int i, double& f) : ci2(i), cf(f){}
   void afficher(){ cout<<ci1<<" "<<ci2<<" "<<cf<<endl; }
};
int main()
{
   double f1 = 5.5;
   A a1(f1);
   f1 += 100; // modification du double d'origine
   a1.afficher(); // 7, 5, 105.50

   double f2 = 22.25;
   A a2(100, f2);
   f2 *= 100; // modification du double d'origine
   a2.afficher(); // 7, 100, 2225.0
}
```

Constructeur et copie d'objet

Le constructeur de recopie effectue, par défaut, une copie membre à membre. Paf conséquent, dans le cas de tableaux dynamiques (par exemple), il faut donc s'il s'occupe de l'allocation mémoire, sous peine de pointer vers les mêmes éléments.

Affectation et copie d'objet

Elle pose le même cas de problème que le constructeur de recopie en cas d'allocation dynamique de mémoire.

Destructeur

Il sert à libérer la mémoire allouée. Il ne peut n'y en avoir qu'un par classe. Le destructeur est appelé automatiquement à l'issue du bloc dans lequel a été créé l'objet. Et donc, si des allocations dynamiques ont été opérées, il faut écrire son propre destructeur.

3 Forme canonique de Coplien

On dit qu'une classe `T` est sous forme canonique de Coplien si elle fournit les éléments suivants :

Prototype	Fonctionnalité
T::T()	Constructeur par défaut
T::T(const T&)	Constructeur par recopie
T& T::operator=(const T&)	Opérateur d'affectation
T::~T()	Destructeur

Si ces éléments sont codés correctement, alors l'utilisation de cette classe vis à vis de la mémoire est sécurisé. Dès qu'une classe utilise de la mémoire dynamique ou des ressources critiques, il est indispensable de la mettre sous forme canonique de Coplien.

4 Spécificités C/C++

4.1 C : Tableaux dynamiques en paramètres

L'exemple permet de tester les compatibilités et incompatibilités entre trois formes de tableaux assez proches :

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

enum { TY=3, TX=8 }; // nombre de lignes, de colonnes

// matrice dynamique de char ou tableau dynamique de pointeurs de char
void test1(char** tab)
{
   printf("\n");
   int x,y;
   for (y=0 ; y<TY ; ++y){
      for (x=0 ; x<TX ; ++x){
         printf("%c ",tab[y][x]);
      }
      printf("\n");
   }
   printf("\n");
}

// tableau statique de pointeurs de char
void test2(char* tab[])
{
   printf("\n");
   int x,y;
   for (y=0 ; y<TY ; ++y){
      for (x=0 ; x<TX ; ++x){
         printf("%c ",tab[y][x]);
      }
      printf("\n");
   }
   printf("\n");
}

// matrice statique de char
void test3(char tab[][TX])
{
   printf("\n");
   int x,y;
   for (y=0 ; y<TY ; ++y){
      for (x=0 ; x<TX ; ++x){
         printf("%c ",tab[y][x]);
      }
      printf("\n");
   }
   printf("\n");
}

int main()
{
   char mat1[TY][TX];
```

```
   char *mat2[TY];
   char **mat3;
   int y,x;

   // Allocation eventuelle et initialisation avec les memes valeurs
   mat3 = (char**)malloc(sizeof(char*)*TY);
   for (y=0 ; y<TY ; ++y){
      mat2[y] = (char*)malloc(sizeof(char)*TX);
      mat3[y] = (char*)malloc(sizeof(char)*TX);
      for (x=0 ; x<TX ; ++x){
         mat1[y][x] = '0'+x;
         mat2[y][x] = '0'+x;
         mat3[y][x] = '0'+x;
      }
   }

   // Test des appels
   printf( "TEST 1  Passage de: char mat1[TY][TX]\n");
   printf( "appel de test1(char**tab)     : Warming et plantage\n");
   //test1(mat1);
   printf( "appel de test2(char* tab[])   : Warming et plantage\n");
   //test2(mat1);
   printf( "appel de test3(char tab[][TX]): OK\n");
   test3(mat1);
   printf("\n");

   printf( "TEST 2  Passage de: char *mat2[TY]\n");
   printf( "appel de test1(char**tab)     : OK\n");
   test1(mat2);
   printf( "appel de test2(char* tab[])   : OK\n");
   test2(mat2);
   printf( "appel de test3(char tab[][TX]): Warming et resultats incertains ou
    erreurs\n");
   test3(mat2);
   printf("\n");

   printf( "TEST 3  Passage de: char **mat3\n");
   printf( "appel de test1(char**tab)     : OK\n");
   test1(mat3);
   printf( "appel de test2(char* tab[])   : OK\n");
   test2(mat3);
   printf( "appel de test3(char tab[][TX]): Warming et resultats incertains ou
    erreurs\n");
   test3(mat3);
   return 0;
}
```

Les fonctions `test()` affichent chacune un tableau différent en paramètre. Nous obtenons les compatibilités et incompatibilités suivantes :

	`(char ** t)`	`(char * t[])`	`(char t[][TX])`
`char m1[TY][TX]`	NON	NON	OK
`char* m2[TX]`	OK	OK	NON
`char** m3`	OK	OK	NON