Université de Nice-Sophia Antipolis
Polytech ELEC4

Mardi 23 octobre 2019
Durée: 2h

C++ Allocation
Travaux Dirigés – Séance n. 5

---

1 Objectif

Dans ce TD, nous verrons comment gérer l'allocation dynamique et nous mettrons en évidence le rôle du constructeur de copie et du destructeur.

2 New

Jusqu'à présent les objets que nous avons manipulés étaient désignés par des variables locales à des fonctions, ou globales, c'est-à-dire qu'ils étaient alloués *automatiquement* dans la pile d'évaluation ou dans la zone globale par le support d'exécution lors de l'exécution du programme.

L'an passé, nous avons déjà vu, en C, qu'il était possible de faire de l'allocation dynamique à la demande dans le *tas* (en anglais *heap*), en particulier, grâce à la fonction `malloc`.

En C++, il sera également possible d'allouer dynamiquement des objets dans le tas grâce à l'opérateur **new**. Par exemple, l'opération :

```
new complexe(3.1, 6.56)
```

crée le `complexe` $(3.1, 6.56)$ alloué dans le tas. Le résultat de cette opération est l'adresse de l'objet dans le tas. On pourra alors utiliser un pointeur pour accéder à l'objet.

Après la déclaration suivante, on accédera à l'objet alloué par l'intermédiaire de la variable `pc0`.

```
complexe *pc0 = new complexe(3.1, 6.56);
cout << pc0->getPimg() << endl; // affiche 6.56
```

Pour créer dynamiquement les éléments d'un tableau, on utilise la notation **new** `[]`. Par exemple :

```
int *pi = new int[10];
complexe *pc = new complexe[5];
```

Ci-dessus, `pi` est tableau dont les 10 éléments de type **int** ont été alloués dynamiquement. De façon similaire, le tableau `pc` possède 5 éléments de type `complexe`.

exercice 1) Faites un programme qui reprend les deux déclarations précédentes. Affectez la valeur 10 au deuxième élément de `pi`, puis affichez sa valeur.

exercice 2) Affichez la valeur de `pi[11]`. Que constatez-vous et qu'en déduisez-vous ?

exercice 3) Affichez la valeur du troisième élément de `pc`. Que constatez-vous et qu'en déduisez-vous ?

exercice 4) Est-ce que le constructeur par défaut est obligatoire ?

3 Delete

De façon symétrique, un objet qui a été alloué dynamiquement à l'aide de l'opérateur **new** doit être *détruit* lorsque qu'il devient inutile. Sa destruction se fait à l'aide de l'opérateur **delete** qui exécute le destructeur `~`*nom_classe*`()` qui est implicite à chaque classe, et qui pourra être redéfini, si cela est nécessaire.

exercice 5) Avec l'opérateur **delete**, détruisez le complexe `pc0` $(3.1, 6.56)$ que vous avez créé précédemment.

exercice 6) Est-ce que le destructeur `~complexe` est exécuté ? Mettez-le en évidence.

exercice 7) Essayez maintenant d'afficher la partie imaginaire de `pc0`. Que se passe-t-il ? Expliquez.

La destruction des éléments alloués dynamiquement d'un tableau se fait avec **delete** `[]`.

exercice 8) Détruisez les éléments du tableau `pc`. Est-ce que le destructeur `~complexe` est exécuté pour chacun des éléments du tableau ?

---

exercice 9) À la fin de votre programme (juste avant le **return**), insérez les deux lignes suivantes :

```
complexe c3(1.2,56.9);
cout << "c3=" << c3 << endl;
```

Compilez et exécutez votre programme. Que constatez-vous ? Qu'en déduisez-vous ?

---

exercice 10) Pour ceux qui ont fait du Java l'an passé, quelles sont les différences entre les mécanismes d'allocation dynamique des deux langages ?

4 Une pile d'entiers

exercice 11) Écrivez la classe `PileChainee` qui définit une pile d'entiers. Les éléments seront alloués dynamiquement et chaînés entre eux (voir le td13 d'elec3). Vous écrirez une classe `noeud` qui contiendra la valeur d'un entier et le lien sur le noeud suivant.

Notez qu'en C++, le pointeur `NULL` est représenté par la constante `nullptr`.

exercice 12) Testez le programme suivant :

```
#include <iostream>
#include <cstdlib>
#include "PileChainee.hpp"

using namespace std;

int main() {
  PileChainee p = PileChainee();
  cout << p.estVide() << endl;
  p.empiler(6);
  p.empiler(15);
  cout << p.sommet() << endl;
  cout << p.estVide() << endl;
  cout << p.sommet() << endl;
  p.depiler();
  cout << p.sommet() << endl;
```

```
  return EXIT_SUCCESS;
}
```

exercice 13) Est-ce que le destructeur `~PileChainee` est exécuté ? Est-ce qu'il fait correctement son travail, c'est-à-dire supprimer tous les éléments de la pile ? Écrivez le destructeur `~PileChainee` qui supprime effectivement tous les éléments de la pile.

exercice 14) Ajoutez à votre classe `PileChainee` la surcharge de l'opérateur `<<` de façon à pouvoir écrire tous les éléments d'une pile sur un `ostream`.

5 Constructeur de copie

exercice 15) Déclarez une variable `p1` de type `PileChainee`.

exercice 16) Empilez la valeur 10 en sommet de votre pile `p1` et affichez la valeur du sommet.

exercice 17) Déclarez une variable `p2` de type `PileChainee` et affectez-lui `p1`. Affichez la valeur du sommet de `p2`.

exercice 18) Appliquez la fonction `depiler` sur `p2` et affichez la valeur du sommet de `p1`, puis celle de `p2`. Que constatez-vous ? Qu'en déduisez-vous ?

Nous avons vu dans le TD 2 que l'affectation de `string` ou de `vector` faisait une copie de la valeur affectée. Cette opération est à définir explicitement à l'aide du `constructeur de copie`.

Le `constructeur de copie` (public) d'une classe `C` possède le prototype suivant :

```
C(const C &o)
```

Un constructeur de copie sert à initialiser l'objet courant **this** à partir de l'objet `o` passé en paramètre.

Ce constructeur est utilisé pour :
— la duplication d'un objet lors de la création d'un nouvel objet ;
— lors d'une transmission d'un paramètre par valeur ;
— au retour d'une fonction renvoyant la valeur d'objet.

Par défaut, chaque classe possède un constructeur de copie implicite, toutefois dans certains cas, et en particulier, lorsque l'objet à copier est formé à partir d'objets dynamiques, il est nécessaire de redéfinir le constructeur de copie par défaut. C'est le cas pour la classe `PileChainee`, comme l'a montré l'exemple précédent.

exercice 19) Écrivez le constructeur de copie de la classe `PileChainee` et vérifiez que la déclaration `PileChainee p2 = p1;` précédente fonctionne maintenant correctement.

Attention : dans le cas d'une affectation, le constructeur de copie n'est pas exécuté automatiquement. Il faut alors surcharger l'opérateur d'affectation. Pour une classe `C`, son prototype est :

```
C &operator=(const C &o);
```

exercice 20) Dans votre classe `PileChainee`, écrivez la surcharge de l'opérateur d'affectation `=`, et remplacez dans votre programme de test, la déclaration : `PileChainee p2=p1;` par une déclaration suivie d'une affectation : `PileChainee p2; p2=p1;`.

---

exercice 21) Dans votre fichier de test, ajoutez la fonction `ecrirePile` qui écrit sur la sortie standard le contenu d'une pile `p` passée en paramètre. Testez avec une transmission de `p` par valeur et par référence. Tracez les appels aux constructeur de copie et au destructeur de la pile. Quelles conclusions pouvez-vous en tirer ?