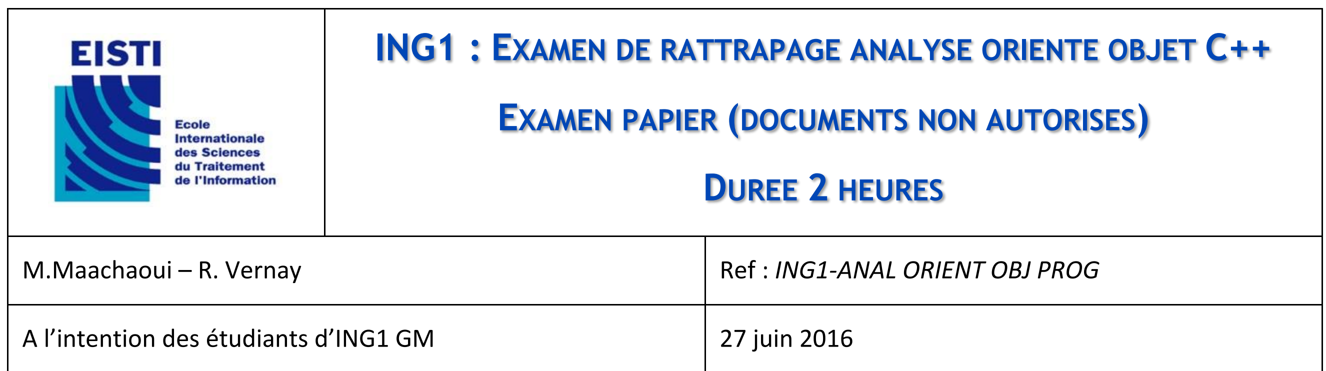

EISTI — Ecole Internationale des Sciences du Traitement de l'Information	ING1 : EXAMEN DE RATTRAPAGE ANALYSE ORIENTE OBJET C++ EXAMEN PAPIER (DOCUMENTS NON AUTORISES) DUREE 2 HEURES
M.Maachaoui – R. Vernay	Ref : *ING1-ANAL ORIENT OBJ PROG*
A l'intention des étudiants d'ING1 GM	27 juin 2016

Modalités

– Durée : 2 heures.

– Vous devez rédiger votre copie à l'aide d'un stylo à encre exclusivement.

– Toutes vos affaires (sacs, vestes, trousse, etc.) doivent être placées à l'avant de la salle.

– Aucun document n'est autorisé.

– Aucune question ne peut être posée aux enseignants, posez des hypothèses en cas de doute.

– Aucune machine électronique ne doit se trouver sur vous ou à proximité, même éteinte.

– Aucune sortie n'est autorisée avant une durée incompressible d'une heure.

– Aucun déplacement n'est autorisé.

– Aucun échange, de quelque nature que ce soit, n'est possible.

Définition du type abstrait pile

L'objectif de ces exercices est de définir le type abstrait pile et d'en proposer deux implantations. La première implantation utilise une représentation de la pile sous forme de tableau (exercice 3). La seconde utilise une structure chaînée (exercice 4). L'exercice 2 est un exemple d'utilisation de la pile.

Exercice 1 : Définition du type abstrait Pile (7 points)

Une pile est une structure de données contenant des données de même type. Le principe d'une pile et que le dernier élément ajouté est le premier à en être retiré. Si on considère une pile d'assiettes, on pose une nouvelle assiette au sommet de la pile. Si on prend une assiette, c'est naturellement celle du sommet de la pile. On dit que la pile est une structure de données de type LIFO (Last In, First Out : dernier entré, premier sorti). On s'intéresse ici à une pile de caractères.

Les opérations sur une pile sont :

- initialiser une pile comme vide. Une pile ne peut être utilisée que si elle a été initialisée,
- empiler un nouvel élément dans une pile,
- dépiler un élément (supprimer le dernier élément ajouté dans la pile),
- accéder à l'élément au sommet de pile (la pile est inchangée),
- connaître le nombre d'éléments contenus dans la pile,
- savoir si une pile est vide ou non.

1.1 Donner la description UML de la classe Pile (de caractères) dont les caractéristiques ont été définies ci-dessus

1.2 Les opérations qui consistent à dépiler et à accéder à l'élément en sommet de pile n'ont un sens que si la pile est non vide. Indiquer deux manières de traiter cet aspect.
Dans la suite, on ne traitera pas cet aspect. On fera l'hypothèse que les opérations sont toujours appelées avec une pile non vide.

1.3 Est-il possible de créer des instances de cette pile ? Pourquoi ? Comment appelle-t-on une telle classe et quel est son intérêt?

1.4 Faut-il définir le destructeur et si oui comment? Justifier la réponse.

1.5 Écrire la classe Pile.

Exercice 2 : Utilisation de la pile (3 points)

Une pile peut être utilisée pour afficher un entier en utilisant seulement l'affichage d'un caractère. En effet, le schéma de *Horner* permet de récupérer successivement les chiffres qui constituent l'entier (en faisant des modulos 10 et des divisions entières par 10). Cependant les chiffres sont récupérés dans le sens inverse de l'affichage : on obtient d'abord les unités, puis les dizaines, etc. L'idée est donc de ranger les chiffres dans une pile puis d'afficher le contenu de la pile. Une ébauche de l'algorithme est donnée ci-dessous.

```
1   // Objectif : Afficher un entier chiffre par chiffre en utilisant une pile .
2
3   #include <iostream.h>
4
5   // Afficher l'entier n sur la sortie standard en utilisant la pile p.
6   //
7   // @pre n > 0  est positif
8   // @pre ... // la pile est vide
9   //
10  // Principe : Utiliser seulement l'affichage d'un caractère
11  // et une pile pour conserver les chiffres de n.
12  //
13  void afficher (int n, Pile &p)
14  {
15     // n est positif
16     // la pile est vide
17     // Stocker les chiffres de n dans la pile p
18       do {
19         int unite = n % 10; // unité de n
20         char chiffre = '0' + unite ; // le caractère
21   // correspondant à unite
22         ... // empiler chiffre dans la pile p
23         n = n / 10;
24       } while ( n > 0);
25       // Afficher le contenu de la pile
26       ...
27  }
28
```

2.1 Il ne semble pas logique de voir apparaître la pile en paramètre de la fonction « afficher ». Aurait-on pu la déclarer comme variable locale ?

2.2 Compléter l'algorithme proposé.

Exercice 3 : Réalisation d'une pile avec un tableau (4 points)

On choisit de représenter la pile par un tableau de caractères (les éléments mis dans la pile) et un entier (le nombre d'éléments dans la pile). Le i^e^ élément ajouté dans la pile est à la i^e^ position du tableau.

Le tableau sera déclaré de la manière suivante (CAPACITE étant une constante).

```
char tab [CAPACITE] ;
```

3.1 Est-il nécessaire de définir le constructeur de copie ? Justifier la réponse.
3.2 Écrire une classe *PileTableau* qui respecte cette implantation.
On ne définira ni le constructeur de copie, ni l'opérateur d'affectation, ni le destructeur.
Indication : Quel est le lien entre les classes Pile et *PileTableau* ?
3.3 Quelles seraient les modifications à apporter pour permettre à l'utilisateur de la pile (un autre programmeur) de préciser la capacité de cette pile ?
On décrira les modifications à apporter sans écrire le code correspondant.

Exercice 4 : Réalisation d'une pile avec une structure chaînée (6 points)

Nous décidons de réaliser une deuxième implantation du type abstrait pile mais cette fois sous forme de structure chaînée. Une pile est alors un ensemble de cellules reliées les unes aux autres. Chaque cellule contient un des éléments mis dans la pile et une référence vers une autre cellule (on parle de cellules simplement chaînées). La figure 1 montre l'état de la mémoire lorsque les valeurs de 3, 5 puis 1 ont été successivement insérées dans la pile.

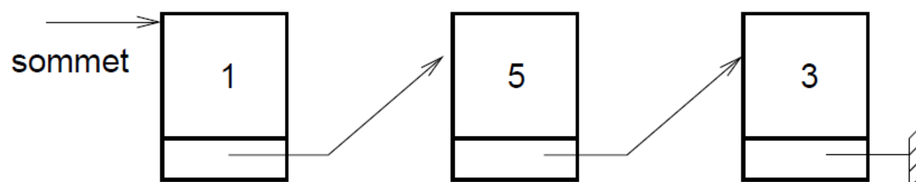

Figure 1. Exemple de représentation en mémoire d'une pile chaînée.

On remarque que pour représenter la pile, il suffit d'avoir accès à la cellule contenant l'information en sommet de pile. Cette cellule possède une référence vers la cellule contenant la valeur précédemment ajoutée.

Ainsi, empiler une nouvelle valeur consiste à créer une nouvelle cellule dont l'information est cette valeur et dont la cellule suivante est l'ancien sommet. Cette nouvelle cellule devient alors le nouveau sommet de la pile. Elle est nécessairement allouée dynamiquement puisqu'elle doit continuer à exister après la fin de l'exécution de la fonction *empiler*. Lorsque l'on veut dépiler, il suffit de supprimer la cellule correspondant au sommet, la cellule suivante devenant le nouveau sommet.

4.1 *Définition d'une cellule*. Une cellule simplement chaînée est caractérisée par l'information qu'elle contient (un caractère dans le cas présent) et une référence vers une autre cellule (qu'on appelle généralement cellule suivante ou cellule précédente).
Voici le texte C++ de la classe Cellule. Indiquer quelles sont les maladresses qui ont été commises dans cette classe.

```
1  class Cellule
2  {
3  public:
4    Cellule ( char info_ , Cellule & suivante_ )
5    // Créer une cellule contenant info_ et liée à suivante_ .
6    {
7        info = info_ ;
8        suivante = suivante_ ;
9    }
10   char info ; // information conservée par la cellule
11   Cellule & suivante ; // la cellule suivante
12 };
```

4.2 *Diagramme UML.* Dessiner le diagramme UML faisant intervenir les classes Pile, Pile-Chaînée et Cellule. On ne fera pas apparaître les opérations (seulement les classes, les relations et les attributs).

4.3 *Classe PileChaînée*. Écrire la classe *PileChaînée*. On ne définira ni le destructeur, ni le constructeur de copie, ni l'opérateur d'affectation.

4.4 *Destructeur*. Pourquoi est-il nécessaire de définir le destructeur de la *PileChaînée* ? Écrire ce destructeur.