

Année universitaire : 2017 / 2018

LIFAP3 : Algorithmique et programmation avancée

Contrôle mi-parcours  
24 octobre 2017  
Durée : 1h30

Note :

/ 20

coller ici

Nom : .....  
Prénom : .....  
N° d'étudiant : .....  
Signature : .....

coller ici

**Documents et téléphones portables interdits.** Le barème est donné à titre indicatif. Répondez dans les emplacements prévus à cet effet. Travaillez au brouillon d'abord de sorte à rendre une copie propre – nous ne pouvons pas vous garantir une copie supplémentaire. **Il sera tenu compte de la présentation et de la clarté de vos réponses.**

### Exercice 1 : Trace mémoire (5 points)

Supposons le programme principal donné ci-dessous. Dessiner l'état de la mémoire aux trois endroits indiqués en commentaires, et donner la trace écran. Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

```
1  #include <iostream>
2  using namespace std;
3
4  class Point2D {
5  public :
6      float x,y;
7      Point2D () { x = 1.0; y = -1.0; }
8      bool estOrigine() const {
9          /* Dessinez l'état de la mémoire #2 */
10         return x==0.0 && y==0.0;
11     }
12 };
13
14 Point2D * creerTableauPoint2D (unsigned int taille) {
15     Point2D * tab = new Point2D [taille];
16     /* Dessinez l'état de la mémoire #1 */
17     return tab;
18 }
19
20 int main () {
21     Point2D * tabPoints = creerTableauPoint2D(2);
22     tabPoints[1].x = tabPoints[0].x + 1.0;
23     tabPoints[1].y = tabPoints[0].y - 1.0;
24     if (tabPoints[1].estOrigine()) cout << "origine";
25     else cout << "pas origine";
26     delete [] tabPoints;
27     /* Dessinez l'état de la mémoire #3 */
28     return 0;
29 }
```



### État de la mémoire #1

	PILE	
VR main		3 987 546 988
tabPoints		3 987 546 984
	<i>appel à creerTableauPoint2D</i>	
VR creerTab.		3 987 546 980
taille	2	3 987 546 976
tab	1 000	3 987 546 972

	TAS	
	-1.0	1 012
	1.0	1 008
	-1.0	1 004
	1.0	1 000

### État de la mémoire #2

	PILE	
VR main		3 987 546 988
tabPoints	1 000	3 987 546 984
	<i>appel à estOrigine</i>	
VR estO.		3 987 546 983
this	1 008	3 987 546 979

	TAS	
	-2.0	1 012
	2.0	1 008
	-1.0	1 004
	1.0	1 000

### État de la mémoire #3

	PILE	
VR main		3 987 546 988
tabPoints	1 000	3 987 546 984

	TAS

### Trace écran :

pas origine

## Exercice 2 : Structure Etudiant (6 points)

Supposons l'existence de la structure **Etudiant** dont certains champs prennent beaucoup de place en mémoire et dont certains champs sont des pointeurs.

**Question 2.1 :** Donner l'entête, en C++, de la procédure globale **permutationEtudiant** qui échange les champs d'une première variable de type **Etudiant** passée en paramètre avec une deuxième variable passée en paramètre.

```
void permutationEtudiant (Etudiant & etu1, Etudiant & etu2);
```

**Question 2.2 :** Donner l'entête, en C++, de la fonction globale **estInferieurEtudiant** qui retourne vrai si une première variable de type **Etudiant** passée en paramètre est inférieure à une deuxième variable passée en paramètre, et faux sinon.

```
bool estInferieurEtudiant (const Etudiant & etu1, const Etudiant & etu2);
```

**Question 2.3 :** Ecrire, en C++, la procédure globale **trierTableauEtudiants** qui trie par sélection un tableau de variables **Etudiant** passé en paramètre. Vous pourrez utiliser les procédures et fonctions des questions précédentes.

```
void trierTableauEtudiants (Etudiant * tab, unsigned int taille) {
    for (unsigned int i = 0; i < taille-1; i++) {
        unsigned int indmin = i;
        for (unsigned int j = i+1; j < taille; j++){
            if (estInferieurEtudiant(tab[j],tab[indmin])) indmin = j;
        }
        permutationEtudiant(tab[indmin],tab[i]);
    }
}
```

**Question 2.4 :** Donner l'entête, en C++, de la procédure globale **saisirEtudiant** qui saisit au clavier les informations relatives à une variable de type **Etudiant**. Ces informations sont utilisées pour mettre à jour la variable de type **Etudiant** passée en paramètre.

```
void saisirEtudiant (Etudiant & e);
```

**Question 2.5 :** Ecrire, en C++, les instructions permettant de créer sur la pile un tableau de 5 variables **Etudiant**, de saisir les informations des variables, puis de trier le tableau. Vous pourrez utiliser les procédures et fonctions des questions précédentes.

```
Etudiant tabEtudiants [5];
for (unsigned int i = 0; i < 5; i++) saisirEtudiant(tabEtudiants[i]);
trierTableauEtudiants(tabEtudiants,5);
```

### Exercice 3 : Classe, complexité et invariant (9 points)

Dans cet exercice, on s'intéresse à modéliser une calculatrice simple opérant sur des réels. Pour cela, on conçoit une classe **Calculatrice** qui contient :

- Une donnée membre réelle privée **resultat** qui sera affectée avec le résultat des opérations effectuées
- Les procédures membres publiques **addition**, **soustraction**, **multiplication** et **division** qui prennent deux paramètres réels, qui réalisent l'opération entre ces paramètres et mettent le résultat de l'opération dans la donnée membre **resultat**
- Une procédure membre publique **afficher** qui affiche à l'écran le résultat de la dernière opération effectuée. Ce résultat devra valoir zéro à la construction d'une instance de la classe.

**Question 3.1 :** Donner l'implémentation de la classe **Calculatrice** en C++.

```
#include <iostream>
using namespace std;

class Calculatrice {
public :

    Calculatrice () { resultat = 0.0 ; }

    void addition (float valeur1, float valeur2) { resultat = valeur1 + valeur2; }

    void soustraction (float valeur1, float valeur2) { resultat = valeur1 - valeur2; }

    void multiplication (float valeur1, float valeur2) { resultat = valeur1 * valeur2; }

    void division (float valeur1, float valeur2) {
        if (valeur2 != 0.0) resultat = valeur1 / valeur2;
        else {
            cout << "Erreur dans Calculatrice::division : division par zéro" << endl;
            exit(1);
        }
    }

    void afficher () const { cout << resultat << endl; }

private :

    float resultat;
};
```

On ajoute à la classe la procédure membre **puissance** donnée en notation algorithmique ci-dessous.

**Procédure** puissance ( $n$  : réel,  $a$  : entier positif ou nul)

**Précondition** : aucune

**Postcondition** : la donnée membre resultat contient la valeur  $n^a$  ( $n$  puissance  $a$ )

**Paramètres en mode donnée** :  $n$  et  $a$

**Paramètres en mode donnée-résultat** : aucun

**Variables locales** :  $N$ ,  $R$  : réels, et  $A$  : entier positif ou nul

**Début**

```

1  N ← n
2  A ← a
3  R ← 1
4  Tant que A > 0 faire
5      Si A modulo 2 = 0 alors
6          N ← N * N
7          A ← A / 2
8      Sinon
9          R ← R * N
10         A ← A - 1
11     Fin si
12 Fin tant que
13 resultat ← R
Fin puissance
    
```

**Question 3.2** : Compter le nombre de multiplications (lignes 6 et 9) effectuées pour les valeurs de  $a$  du tableau ci-dessous. De quelle complexité semble être la procédure **puissance** ? Justifiez en étudiant brièvement l'algorithme (c.-à-d. sans compter le nombre exact de multiplications pour un  $a$  quelconque).

	$a = 1$	$a = 2$	$a = 4$	$a = 10$	$a = 20$	$a = 100$
nombre de multiplications	1	2	3	5	6	9

La procédure est de complexité  $\log_2 a$ . Si  $a$  est pair, on itère avec un  $a$  divisé par 2. Si  $a$  est impair, on décrémente  $a$ . Mais si on décrémente un nombre impair on obtient un nombre pair. A la fin, on arrive forcément avec  $a$  qui vaut 1 (impair) puis 0 ce qui fait sortir de la boucle. Le fait de diviser  $a$  par 2 quand il est pair et de faire  $n-1/2$  après deux itérations s'il est impair (i.e. au pire on le divise par 2 toutes les deux itérations), nous donne la complexité en  $\log a$ .

Un invariant de boucle possible pour le tant-que de la ligne 4 est :

« A toute itération, on a  $N^A \times R = n^a$  »

**Question 3.3 :** Vérifier les propriétés d'initialisation, de conservation et de terminaison de cet invariant. Et montrer que l'on peut en conclure que la procédure **puissance** affecte bien la valeur  $n^a$  à la donnée membre **resultat**.

Propriété d'initialisation :

Avant de rentrer dans la boucle tant-que, on a  $A = a$ ,  $N = n$  et  $R = 1$  donc on a bien  $N^A \times R = n^a \times 1 = n^a$

Propriété de conservation :

Supposons que la propriété d'invariant est vraie à l'itération précédente  $k-1$ , c'est-à-dire que  $N_{k-1}^{A_{k-1}} \times R_{k-1} = n^a$ .

A l'itération courante  $k$ , on a les deux possibilités :

- Si  $A$  est pair, alors après exécution des instructions des lignes 6 et 7, on a

$$N_k^{A_k} \times R_k = (N_{k-1}^2)^{\frac{A_{k-1}}{2}} \times R_{k-1} = (N_{k-1})^{2 \times \frac{A_{k-1}}{2}} \times R_{k-1} = N_{k-1}^{A_{k-1}} \times R_{k-1} = n^a$$

- Si  $A$  est impair, alors après exécution des instructions des lignes 9 et 10, on a

$$N_k^{A_k} \times R_k = N_{k-1}^{A_{k-1}-1} \times R_{k-1} \times N_{k-1}^{A_{k-1}} = N_{k-1}^{A_{k-1}} \times R_{k-1} = n^a$$

Donc dans les deux cas, on a bien :  $N_k^{A_k} \times R_k = n^a$ , la propriété est donc conservée.

Propriété de terminaison :

A la sortie de la boucle, on a  $A = 0$ . L'invariant de boucle nous dit donc que l'on a

$$N^A \times R = N^0 \times R = 1 \times R = R = n^a$$

Conclusion :

A la ligne 13, on a donc  $R = n^a$ , et on affecte  $R$  à **resultat**, donc la procédure **puissance** affecte bien la valeur  $n^a$  à la donnée membre **resultat**.