Centrale Lyon **2024-2025**

ELCa11 Programmation des interfaces graphiques en C++

Examen

Durée 2 heures - Sujet recto-verso -
Aucun document ni ordinateur n'est autorisé.

Exercice 1 (14 points)

On souhaite disposer d'une structure de données de type "Pile" dont le principe est que l'élément dépilé (retiré de la pile) est le dernier qui a été empilé (ajouté à la pile) (principe LIFO : Last In First Out). La classe "PileIntStat" a ainsi été écrite pour gérer des nombres entiers. La déclaration de la classe et la définition (code) des méthodes sont données ci-dessous, ainsi qu'une fonction "main" mettant en œuvre cette classe.

```
class PileIntStat
{
public:
    PileIntStat(int taille=100);
    int get_taille();
    bool est_vide();
    void empile(int v);
    int depile();
private:
    int tab[100];
    int top;
    int taille;
};

int main()
{
    PileIntStat p(10);
    cout << "Taille de la pile : " <<
p.get_taille() << endl;
    p.empile(22);
    p.empile(11);
    p.empile(207);

    int v;
    while (!p.est_vide())
    {
        v = p.depile();
        cout << "Valeur : " << v << endl;
    }
    return 0;
}
```

```
PileIntStat::PileIntStat(int t)
{
    if (t>=100)
        taille = 100;
    else
        taille = t;
    top = -1;
}
int PileIntStat::get_taille()
{
    return taille;
}
bool PileIntStat::est_vide()
{
    return top == -1;
}
void PileIntStat::empile(int v)
{
    top++;
    tab[top] = v;
}
int PileIntStat::depile()
{
    int v = tab[top];
    top--;
    return v;
}
```

Questions :

1. Quel problème le code actuel des méthodes "empile" et "depile" peut-il poser ? Il existe un mécanisme C++ permettant de gérer proprement ces situations. Quel est-il ? Modifier le code de ces méthodes pour le mettre en œuvre et écrire un programme principal (fonction "main") utilisant ce mécanisme. (4 pts)
2. Les données de la pile sont actuellement stockées en mémoire dans un tableau de taille fixe égale à 100 (allocation statique). En reprenant la déclaration de la classe "PileIntStat" et la définition de ses méthodes, écrire une nouvelle classe "PileIntDyn" (déclaration de la classe et définition de ses méthodes) en faisant les modifications et ajouts nécessaires afin

de permettre le stockage des valeurs de la pile dans un tableau dont la taille sera fournie par l'utilisateur au constructeur de la classe (allocation dynamique). **Indication :** allocation / libération de la mémoire par les opérateurs new / delete. (6 pts)

3. En quoi consiste le principe de généricité ? En s'appuyant sur le code précédent, écrire la déclaration d'une classe "Pile" générique (le code (définition) des méthodes n'est pas demandé). Ecrire également une fonction "main" illustrant l'utilisation de cette classe générique. (4 pts)

Exercice 2 (6 points)

On dispose d'une classe "CompteBancaire" dont la déclaration de la classe et la définition des méthodes sont données ci-dessous.

```
class CompteBancaire {
private:
    string numeroCompte;
    double solde;
public:
    CompteBancaire(string nc, double s);
    string getNumeroCompte();
    double getSolde();
    void depot(double montant);
    void retrait(double montant);
};

int main() {
    CompteBancaire cb("2025", 1000.0);
    cout << "Compte numero : " <<
cb.getNumeroCompte() << " solde : " << cb.getSolde()
<< endl;
    cb.depot(500.0);
    cb.retrait(200.0);
    cout << "Compte numero : " <<
cb.getNumeroCompte() << " solde : " << cb.getSolde()
<< endl;

    return 0;
}
```

==> Résultat de l'affichage

```
Compte numero : 2025 solde : 1000
Dépôt effectué avec succès.
Retrait effectué avec succès.
Compte numero : 2025 solde : 1300
```

```
CompteBancaire::CompteBancaire(string nc, double s)
    : numeroCompte(nc), solde(s) {}

string CompteBancaire::getNumeroCompte() {
    return numeroCompte;
}

double CompteBancaire::getSolde() {
    return solde;
}

void CompteBancaire::depot(double montant) {
    if (montant > 0) {
        solde += montant;
        cout << "Dépôt effectué avec succès." <<
endl;
    } else {
        cout << "Montant de dépôt invalide." <<
endl;
    }
}

void CompteBancaire::retrait(double montant) {
    if (montant > 0 && solde >= montant) {
        solde -= montant;
        cout << "Retrait effectué avec succès." <<
endl;
    } else {
        cout << "Retrait impossible." << endl;
    }
}
```

Ecrire le code d'une classe "CompteEpargne" héritant de la classe "CompteBancaire", avec pour attribut privé supplémentaire "tauxInteret" (2 pts) et pour méthodes (outre le constructeur) :

- "getTauxInteret()" pour obtenir la valeur de l'attribut "tauxInteret" (1 pt),
- "appliqueInteret()" pour calculer et ajouter les intérêts au solde actuel du compte (1 pt),
- operateur "<<" pour permettre l'affichage de toutes les informations d'un objet "CompteEpargne" (1 pt).

Ecrire également le code de la fonction « main » mettant en œuvre cette classe (1 pt).