Exercices et problèmes corrigés en C++

Exercice 1 : Pointeurs

On considère que l'on dispose d'un tableau de flottants déclaré de la manière suivante :

```
float tab[3];
```

On supposera que ce tableau a également été initialisé.

1) Écrire un programme permettant de calculer la somme des éléments du tableau en utilisant le formalisme tableau.

2) Écrire un programme permettant de calculer la somme des éléments du tableau en utilisant le formalisme pointeur.

3) Reprendre les questions 1) et 2) en considérant un tableau à deux dimensions déclaré comme suit :

```
float tab2D[3][4];
```

et dont on supposera qu'il a été initialisé.

Exercice 2 : Structures et fonctions

On considère le type structure suivant :

```
struct producteur{
  int stock;
  int ventes[12];
}
```

Ecrire une fonction qui initialise les champs `stock` et `ventes` en donnant la valeur 0 à tous les éléments de deux façons :

a) Via une fonction accédant à une variable de type `producteur` en argument;

b) Via une fonction membre définie dans la structure `producteur` sous la forme :

```
void init();
```

Exercice 3 : Chaînes de caractères

Cet exercice reprend et complète un exercice du TP 2.

Écrire une fonction qui prenne en argument une chaîne de caractères et un caractère, et qui renvoie le nombre d'occurrences du caractère dans la chaîne. On considèrera trois en-têtes, correspondant à trois représentations des chaînes de caractères en C++ :

a) Représentation via le type `string` :

```
int nb_occ(string s,char c);
```

b) Représentation via un tableau de caractères :

```
int nb_occ(char s[],char c);
```

c) Représentation via un pointeur vers le premier caractère :

```
int nb_occ(char *s, char c);
```

Problème A : Files d'attente et magasins

Remarque : il est fortement recommandé d'implémenter et de tester les différentes classes de l'exercice sur machine.

Contexte On souhaite modéliser la gestion des files d'attente durant une période de confinement. Pour cela, on considèrera tout d'abord les constituants d'une file (les individus), puis la file elle-même, définie via une allocation dynamique. Enfin, on modélisera des commerces comme héritant de la classe représentant une file.

Partie 1 : Individus

On considère une classe `Personne` représentant des individus en temps de confinement. Cette classe comporte trois membres données :

- Deux membres `nom` et `prenom` de type `string`;
- Un membre `attestation` de type `bool` qui représente le fait que la personne dispose ou non d'une attestation de sortie.

a) Coder deux constructeurs pour la classe `Personne`, respectivement sans argument et avec deux chaînes de caractères en arguments. Dans le premier cas, les membres `nom` et `prenom` seront initialisés avec la chaîne de caractères vide; dans le second, les deux arguments serviront à initialiser ces deux membres données. Enfin, dans les deux cas, la valeur du membre `attestation` sera initialisée à `false`.

b) Créer une fonction permettant de mettre la valeur du booléen `attestation` à `true`.

c) On souhaite créer une classe `Policier` qui dérive de la classe `Personne`. Cette classe devra disposer d'un constructeur basé sur celui de la classe `Personne`, qui initialise la valeur du booléen à `true`; elle devra également comporter une fonction membre `controler` dont l'interface sera la suivante :

```
string controler(Personne);
```

Cette fonction vérifiera la valeur du booléen de l'objet de type `Personne` passé en argument. En fonction de la valeur de ce booléen, elle renverra une chaîne de caractères indiquant le nom complet de la personne, et si celle-ci possède ou non une attestation.

Partie 2 : Files d'attente

On adoptera ici la représentation circulaire d'une file d'attente. On se donnera ainsi un tableau de taille fixée, dans lequel on rangera les individus à la suite dans leur ordre d'arrivée dans la file. Deux indices serviront à repérer respectivement le début et la fin de la file.

1. On considère une classe `File` représentant une file d'attente possédant trois membres données :

   - `gens` de type `Personne *` représentera un tableau d'objets de type `Personne`;
   - `lgfile` de type `int` représentera la taille maximum possible pour la file;
   - `lmax` de type `static const int` représentera la valeur maximale qui puisse être choisie pour `lgfile`;
   - `ideb` et `ifin`, de type `int`, représenteront les indices de début et de fin de la file, respectivement. Ils seront initialisés à la valeur -1.

   Écrire les fonctions membres nécessaires pour que la classe `File` soit sous forme canonique.

2. Surcharger l'opérateur `[]` afin que, si `f` est un objet de la classe `File`, `f[i]` (où `i` est un entier entre 0 et `f.lgfile`-1) renvoie la (`i`+1)-ème personne de la file.

3. Surcharger l'opérateur `--` pour que celui-ci représente la sortie de la première personne d'une file. Un appel à cette fonction devra afficher le prenom et le nom de la personne quittant la file, le cas échéant. Pour simplifier, cette fonction n'aura pas de type de retour.[1]

4. Surcharger l'opérateur `+` pour qu'il puisse réaliser la "somme" d'un objet de type `File` et celle d'un objet de type `Personne`, c'est-à-dire ajouter la personne à la file, si tant est qu'il y reste de la place. Un affichage devra permettre de valider si la personne a été ajoutée ou non. Pour simplifier, cette fonction n'aura pas de type de retour.[2]

5. Au vu des contraintes posées par l'implémentation de la classe `File`, comment pourrait-on organiser le code des classes des parties 1 et 2 ?

[1] Si l'on voulait que cet opérateur se comporte comme l'opérateur `--` usuel, il faudrait renvoyer un objet de type `File &`.

[2] Là encore, la logique voudrait que l'on renvoie un objet de type `File`.

Partie 3 : Commerces

Dans cette dernière partie, on utilisera les propriétés de la classe `File` pour définir des structures de données représentant des commerces.

1. Implémenter deux classes `Epicerie` et `Pharmacie` dérivant de la classe `File` et possédant en plus les membres données suivants :

   - Pour la classe `Epicerie`, un entier `stockPaquetsPates`, initialisé à 100;
   - Pour la classe `Pharmacie` : un entier `stockGelHydroAlco`, initialisé à 1;

   Dans les deux cas, l'opérateur `--` devra être redéfini pour diminuer la variable de stock à chaque fois que la première personne de la file sera servie.

2. On souhaite implémenter une classe `Supermarche` héritant à la fois des classes `Epicerie` et `Pharmacie`, mais ne comportant qu'une file. Quelle(s) modification(s) doit-on apporter aux classes précédentes pour cela ?

3. Même question si l'on désire en plus que la classe `Supermarche` comporte un membre donnée de type `Policier` pouvant contrôler l'ensemble des membres de la file d'attente du supermarché.

Correction

Correction de l'exercice 1

1) On peut faire une boucle `for` afin de parcourir tous les éléments du tableau; dans le formalisme tableau, on accède au i-ème élément d'un tableau `tab` en faisant `tab[i]`. Un calcul de la somme des éléments du tableau peut donc être fait comme suit :

```
float s=0;
for(int i=0;i<3;i++){
  s+=tab[i];
}
```

2) Plusieurs solutions sont possibles ici. On commence par la plus semblable à celle de la question précédente :

```
// Solution 1
float s=0;
for(int i=0;i<3;i++){
  s+=*(tab+i);//NB: ne modifie pas le pointeur tab !
}
```

On utilise ici le fait que `tab+i` pointe vers le (i+1)-ème élément du tableau.

Une autre possibilité consiste à utiliser un pointeur auxiliaire, et à ne se servir de l'entier `i` que pour compter le nombre d'itérations de la boucle :

```
// Solution 2
float s=0;
for(float *p=tab, int i=0;i<3;i++,p++){
  s+=*p;
}
```

Le pointeur auxiliaire `p` est ainsi incrémenté à chaque itération.

Il est également possible de n'utiliser que des pointeurs, et pas de variable entière :

```
// Solution 3
float s=0;
for(float *p=tab;p<tab+3;p++){
  s+=*p;
}
```

3) Dans le cas d'un tableau à deux dimensions, on peut effectuer une double boucle en utilisant le formalisme tableau :

```
int s=0;
for(int i=0;i<3;i++){
  for(int j=0;j<4;j++){
    s+=tab2D[i][j];
  }
}
```

La difficulté de la manipulation de ce tableau en termes de pointeur réside dans le fait que `tab2D` n'est pas réellement un pointeur vers un pointeur de flottants, mais un pointeur vers un tableau de flottants de taille 4 (donc `tab2D` est de type `*float[4]`).

```
// Formalisme pointeurs pour tableau 2D
float s=0;
float *p;
for(int i=0;i<3;i++){
  p=tab2D[i];//tab2D[i] est bien une adresse
  for(int j=0;j<4;j++){
    s+=*(p+j);
  }
}
```

En affectant `tab2D[i]` au pointeur `p`, on peut ainsi accéder aux éléments du tableau correspondant à la i-ème ligne (càd la première dimension fixée à la valeur i).

Correction de l'exercice 2

a) L'implémentation doit modifier la valeur de la variable de type `producteur` passée en paramètre, il est donc nécessaire de faire un passage par pointeur ou par référence. En utilisant ce dernier formalisme, on peut alors implémenter la fonction comme suit :

```
void init(producteur &p){
  p.stock=0;
  for(int i=0;i<12;i++){
    p.ventes[i]=0;
  }
}
```

Un appel à cette fonction pourra être effectué dans la fonction `main` de la façon suivante :[3]

```
producteur p;
init(p);
```

b) On introduit une fonction membre dans la structure `producteur`, qui devient alors :

```
struct producteur{
  int stock;
  int ventes[12];
  void init();
}
```

Tout appel à cette fonction se fera à partir d'une variable déclarée :

```
producteur p;
p.init();
```

[3] Il faudra a minima déclarer la fonction dans le programme principal avant de l'appeler dans le main !

La fonction `init` aura alors accès directement aux champs de la structure `producteur`.

Le code de la fonction `init` s'écrit en ajoutant le préfixe propre à la structure :

```
void producteur::init(){
  stock=0;
  for(int i=0;i<12;i++){
    ventes[i]=0;
  }
}
```

Correction de l'exercice 3

1. En utilisant le type `string` :

```
int nb_occ(string s, char c){
  int n=0;
  for(char cs:s){
    if  (cs==c){
      n++;
    }
  }
  return n;
}
```

   Rappel : Une telle boucle `for` est valide depuis C++11.

2. En utilisant le type `char*` :

```
int nb_occ(char *s,char c){
  int n=0;
  while(*s!='\0'){
    if (*s==c){
      n++;
    }
    s++;
  }
  return n;
}
```

   Remarque : Une telle fonction ne modifie pas la valeur originelle du pointeur (le passage du pointeur en argument se fait par valeur).

3. En utilisant un tableau de `char` :

```
int nb_occ(char s[],char c){
  int n=0,i=0;
  while(s[i]!='\0'){
    if (s[i]==c){
      n++;
```

```
        }
        i++;
    }
    return n;
}
```

On notera qu'il n'est pas nécessaire de connaître la taille du tableau, puisqu'on sait qu'une chaîne de caractères se termine nécessairement par `'\0'`. Si la taille du tableau était connue, elle pourrait être passée en paramètre.

Correction du problème A

Remarque préliminaire : la correction ci-dessous fournit essentiellement des extraits de code permettant d'implémenter les classes demandées. Elle discute également des difficultés potentielles liées à cette implémentation.
Il y a évidemment plusieurs implémentations possibles d'une même fonction : sur papier, une implémentation sera acceptée dès lors qu'elle respecte les contraintes demandées et ne déclencherait pas d'erreur majeure de compilation.

Correction de la partie 1 du problème A

1. Le code des constructeurs sera le suivant :

```
// Constructeur vide avec initialisation vide des champs
// nom et prenom
Personne::Personne() : nom(""),prenom(""),attestation(false){
}

// Constructeur a deux parametres avec initialisation directe
// des champs
Personne::Personne(string sn,string sp) :
nom(sn),prenom(sp),attestation(false){
}
```

2. Soit `ecrire_attestation` la fonction en question, qui n'a pas besoin de retourner d'argument. On propose l'implémentation suivante :

```
void Personne::ecrire_attestation(){
    attestation=true;
    cout<<prenom<<" "<<nom<<" a rempli une attestation."<<endl;
}
```

3. Dans la déclaration de la classe `Policier`, il faudra préciser que celle-ci dérive de la classe `Personne` : il serait donc naturel de déclarer la classe `Personne` avant la classe `Policier`. Cependant, la fonction membre `controler` prend en argument un objet de type `Personne`, et doit accéder aux membres données de cet objet. Comme vu en cours, une solution est la suivante :

```
class Policier;//Declaration de l'existence d'une classe Policier

class Personne{
    // Nom et prenom de l'individu
    string nom,prenom;
    // La personne possede-t-elle une attestation de sortie ?
    bool attestation;

    public:
        // Constructeurs de la classe Personne
```

```
        Personne();
        // Premier argument : nom; deuxieme argument : prenom
        Personne(string,string);
        // Le booleen attestation est necessairement true
        // apres l'execution de cette fonction
        void ecrire_attestation();

        // Declaration de la classe Policier comme amie
        friend class Policier;
};

class Policier : public Personne{
    public:
        Policier(string,string);
        string controle(Personne);
};
```

Le code des fonctions membres de la classe `Policier` pourra alors s'écrire :

```
Policier::Policier(string sn,string sp) : Personne(sn,sp){
    Personne::ecrire_attestation();
}

// Controle d'une personne pour verifier si la
// personne possede une attestation
string Policier::controler(Personne p){
    string s=p.prenom+" ";
    s=s+p.nom;
    if (p.attestation)
        s=s+" a une attestation.";
    else
        s=s+" n'a pas d'attestation.";
    return s;
}
```

Correction de la partie 2 du problème A

1. Selon la définition vue en cours, la forme canonique d'une classe se compose d'un (ou plusieurs) constructeurs, d'un destructeur, d'un constructeur de recopie et d'une surdéfinition de l'opérateur d'affectation =. La définition de l'ensemble de ces fonctions membres est particulièrement recommandée si les membres données de la classe font intervenir de l'allocation dynamique.
   Dans notre cas, il s'agira de définir les quatre fonctions ci-dessous :

```
// Constructeur avec argument de type entier
File::File(int l){
    if(l>lmax){
        cout<<"Taille limite de file atteinte, remplacee par ";
        cout<<lmax<<endl;
    }
    lgfile=l;
    gens = new Personne[l];
    ideb = -1;
    ifin = -1;
};

// Destructeur
File::~File(){
    delete [] gens;
};

// Constructeur de recopie
File::File(File & f){
    lgfile=f.lgfile;
    gens=new Personne[f.lgfile];
    ideb = f.ideb;
    ifin = f.ifin;
    for(int i=ideb;i<=ifin;i++){
        gens[i]=f.gens[i];
    }
};


// Operateur d'affectation
File & File::operator=(const File & f){
    if (this!=&f){
        delete [] gens;
        lgfile=f.lgfile;
        gens=new Personne[f.lgfile];
        ideb = f.ideb;
        ifin = f.ifin;
        for(int i=ideb;i<=ifin;i++){
```

```
            gens[i]=f.gens[i];
        }
    }
    return *this;
};
```

2. La surcharge de l'opérateur `[]` se fait nécessairement via une fonction membre, dont le code peut être le suivant :

```
// Surcharge de []
Personne & File::operator[](int i){
  int j=(ideb+i)%lgfile;
    if ((ideb>-1) && (j<=ifin))
    {
        return gens[j];
    }
}
```

3. La surcharge de l'opérateur `--` peut se faire de deux façons, selon que l'on souhaite une notation préfixée (`i--` ou post-fixée (`--i`). Dans le premier cas, un entier devra être passé en paramètre (fictif) de l'opérateur; dans le second cas, l'opérateur sera déclaré sans paramètre. On donne ci-dessous les deux implémentations :

```
// Surcharge de -- pour faire sortir la premiere personne de la file
// Version pre-fixee (appel : f--, avec f de type File)
void File::operator--(int n){
    if (ideb==-1)
        cout<<"Personne dans la file..."<<endl;
    else
    {
        cout<<gens[ideb].prenom<<" ";
        cout<<gens[ideb].nom<<" est sorti.e de la file"<<endl;
        if (ideb==ifin)
        {
            ideb=-1;
            ifin=-1;
        }
        else
        {
            ideb=(ideb+1)%lgfile;

        }
    }
    cout<<ideb<<ifin<<endl;
}

// Surcharge de -- pour faire sortir la premiere personne de la file
```

```
// Version post-fixee (appel : --f, avec f de type File)
void File::operator--(){
    if (ideb==-1)
        cout<<"Personne dans la file..."<<endl;
    else
    {
        cout<<gens[ideb].prenom<<" ";
        cout<<gens[ideb].nom<<" est sorti.e de la file"<<endl;
        if (ideb==ifin)
        {
            ideb=-1;
            ifin=-1;
        }
        else
        {
            ideb=(ideb+1)%lgfile;

        }
    }
    cout<<ideb<<ifin<<endl;
}
```

4. L'ajout d'un objet de type `Personne` et surtout l'affichage de ses nom et prénom implique que la classe `File` puisse accéder aux membres données d'une variable de classe `Personne`. Pour permettre cela, on peut déclarer la classe `File` comme amie de la classe `Personne`. Le code de l'opérateur surchargé peut alors être :

```
// Surcharge de + pour ajouter une personne dans une file
void File::operator+(Personne p){
    if ((ifin+1)%lgfile==ideb)
    {
        cout<<"La file est pleine, pas d'ajout possible."<<endl;
    }
    else
    {
        if (ideb==-1)
        {
            ideb=0;
            ifin=0;
        }
        else
        {
            ifin=(ifin+1)%lgfile;
        }
        gens[ifin]=p;
        cout<<"Ajout de "<<p.prenom<<" ";
        cout<<gens[ifin].nom<<" dans la file."<<endl;
```

```
    }
    cout<<ideb<<ifin<<endl;
}
```

5. Une première possibilité d'organisation des fichiers consiste à séparer chaque classe et donc à voir des fichiers en-têtes *Personne.h, Policier.h, File.h* et leurs fichiers correspondants *Personne.cpp, Policier.cpp, File.cpp*. Dans ce cas, pour respecter les dépendances inter-classes, il conviendrait d'inclure *Personne.h* dans *Policier.h* et dans *File.h*. Cependant, un script de test important à la fois *Policier.h* et *File.h* définirait alors doublement la classe `Personne`, ce qui conduirait à une erreur. Une meilleure solution serait d'inclure *Personne.h* dans *Policier.h* puis *Policier.h* (et donc *Personne.h*) dans *File.h*. Ainsi, un fichier de test utilisant ces trois classes se contenterait d'inclure *File.h*.
   Une autre solution consiste à créer un unique fichier *.h* (et son équivalent .cpp) regroupant les déclarations de trois classes. Cette approche est plus économe en nombre de fichiers mais aussi moins modulaire, dans la mesure où il sera nécessaire d'importer le code des trois classes même si la classe `Personne` est la seule dont on aura besoin.

Correction de la partie 3 du problème A

1. On ne donne ici que la déclaration et les définitions pour la classe `Epicerie` :

```
// Declaration de la classe Epicerie
class Epicerie : public File{

    // Stock de marchandises;
    int stockPaquetsPates;

    public:
        // Constructeur
        Epicerie(int);

        // Surcharge de l'operateur --
        void operator--();

};

// Definition de ses fonctions membres

Epicerie::Epicerie(int s,int t) : stockPaquetsPates(s), File(t){
    cout<<"Ouverture epicerie"<<endl;
}

void Epicerie::operator--(){
    if (stockPaquetsPates==0)
    {   cout<<"Plus de pates a l'epicerie,";
        cout<<" le client ne sera pas servi.";
    }
    else
    {
        cout<<"Le client a ete servi a l'epicerie.";
        stockPaquetsPates--;
        cout<<"Nb de paquets de pates restant :";
        cout<<stockPaquetsPates<<endl;
    }
    File::operator--();
}
```

2. La définition de la classe `Supermarche` est un cas d'héritage multiple où les deux classes de bases (`Epicerie` et `Pharmacie`) héritent toutes deux de la même classe, ici la classe `File`. Il est spécifié ici qu'un objet de type `Supermarche` ne doit avoir qu'une seule file de clients, càd qu'il ne doit contenir qu'une seule fois les membres données de la classe `File`. Ceci est possible si l'on fait une déclaration d'héritage virtuel pour les classes `Epicerie` et `Pharmacie` :

```
class Epicerie : public virtual File{ //...
```

```
};
class Pharmacie : public virtual File{ //...
};
```

En plus de ces déclarations, il faut également que la classe `File` dispose d'un constructeur sans argument. Celui-ci peut par exemple être défini via la taille maximale de file autorisée :

```
// Constructeur sans argument pour la classe File
File::File(){
    lgfile=nbmax_pers;
    gens = new Personne[nbmax_pers];
    ideb = -1;
    ifin = -1;
};
```

Le constructeur de la classe `Epicerie` devrait alors être modifié comme suit :

```
Epicerie::Epicerie(int s) : stockPaquetsPates(s){
    cout<<"Ouverture epicerie"<<endl;
}
```

Un appel à ce constructeur provoquerait un appel au constructeur sans argument de `File`. Il est à noter qu'un constructeur de la classe Supermarché pourrait quant à lui spécifier des arguments pour un constructeur de la classe `File`. C'est le cas dans l'exemple ci-dessous (où l'on suppose que la classe `Pharmacie` dispose d'un constructeur à un argument similaire à celui de la classe `Epicerie`) :

```
// Constructeur de la classe Supermarche appelant tous les
// constructeurs des classes dont il herite
Supermarche::Supermarche(int sp,int sg,int tfile) :
Epicerie(sp),Pharmacie(sg),File(tfile){
    cout<<"Ouverture supermarche"<<endl;
}
```

3. En premier lieu, l'ajout d'un membre donnée de type `Policier` impose l'inclusion de la classe `Policier` dans la définition de la classe `Supermarché`. Une re-définition du constructeur `Supermarché` est également souhaitable, dans la mesure où la classe `Policier` ne possède pas de constructeur par défaut ou sans argument. On pourrait ainsi avoir le constructeur suivant :

```
// Constructeur de la classe Supermarche appelant tous les
// constructeurs des classes dont il herite, et incluant
// un objet membre de la classe Policier (dont on suppose
// que le nom est "agent"

Supermarche::Supermarche(Policier p, int sp,int sg,int tfile) :
agent(p), Epicerie(sp),Pharmacie(sg),File(tfile){
    cout<<"Ouverture supermarche"<<endl;
}
```

Pour qu'un contrôle de la file puisse être effectué par le policier, il faut pouvoir accéder aux éléments de la file, de type `Personne`. Afin d'accéder aux membres données de la classe `File` depuis la classe `Supermarche`, il faudra s'assurer que ceux-ci ont le statut `protected` ou `public`[4] et non `private`. Avec cette modification, on pourra alors accéder aux membres données pour écrire une fonction `controler` pour la classe `Supermarche`; le code de cette fonction pourrait alors être :

```
void Supermarche::controler(){
    string saux;
    cout<<"Debut controle de la file"<<endl;
    for(int i=File::ideb;i<=File::ifin;i++){
        cout<<i<<endl;
        saux = officier.controler(File::gens[i]);
        cout<<saux<<endl;
    }
    cout<<"Fin controle de la file"<<endl;
}
```

[4]On rappelle que le statut `protected` rend les membres accessibles aux classes dérivées, mais pas aux utilisateurs de la classe. Le statut `public` rendrait les membres également accessibles à tout utilisateur, ce qui serait contraire au principe d'encapsulation.