TP POO C++ n°1

Sommaire

Les objectifs de ce tp sont de découvrir la programmation orientée objet en C++.

Rappels

La programmation orientée objet consiste à **définir des objets logiciels et à les faire interagir entre eux**.

Concrètement, un **objet** est une **structure de données** (**ses attributs** c.-à-d. des variables) qui définit son **état** et une **interface** (**ses méthodes** c.-à-d. des fonctions) qui définit son **comportement**.

Une **classe déclare des propriétés communes** à un ensemble d'objets.

Un **objet** est créé à partir d'un **modèle** appelé **classe**. Chaque objet créé à partir de cette classe est une **instance** de la classe en question.

Travail demandé

Nous allons successivement découvrir les notions suivantes :
– l'utilisation de paramètres par défaut et de liste d'initialisation
– l'écriture d'un destructeur
– l'implémentation d'un constructeur de copie et de l'opérateur d'affectation
– la surcharge d'opérateurs

On vous fournit un programme `testPileChar.cpp` où vous devez décommenter progressivement les parties de code source correspondant aux questions posées.

Notion de pile

Une **pile (« stack » en anglais)** est une **structure de données basée sur le principe « Dernier arrivé, premier sorti », ou LIFO (*Last In, First Out*)**, ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

Une **pile** est utilisée en général pour **gérer un historique de données** (pages webs visitées, ...) ou **d'actions** (les fonctions « Annuler » de beaucoup de logiciels par exemple). La pile est utilisée aussi pour tous les paramètres d'appels et les variables locales des fonctions dans les langages compilés.

Voici quelques fonctions communément utilisées pour manipuler des **piles** :
– « Empiler » : ajoute ou dépose un élément sur la pile
– « Dépiler » : enlève un élément de la pile et le renvoie
– « La pile est-elle vide ? » : renvoie « vrai » si la pile est vide, « faux » sinon
– « La pile est-elle pleine ? » : renvoie « vrai » si la pile est pleine, « faux » sinon
– « Nombre d'éléments dans la pile » : renvoie le nombre d'éléments présents dans la pile
– « Taille de la pile » : renvoie le nombre maximum d'éléments pouvant être déposés dans la pile
– « Quel est l'élément de tête ? » : renvoie l'élément de tête (le sommet) sans le dépiler

La classe PileChar

Le but est de réaliser une **structure de pile pouvant traiter des caractères** (type `char`).

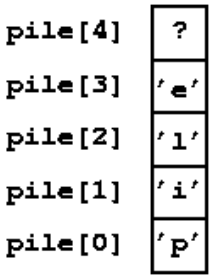
La classe `PileChar` contient **trois données membres privées** :
– deux entiers strictement positifs, nommés `max` et `sommet`, et
– un pointeur sur un caractère, nommé `pile`.

La donnée membre `max` contient la taille de la pile créée pour cette instance de la classe, autrement dit le nombre maximum de caractères qu'il sera possible d'y mettre.

La donnée membre `sommet` indique le numéro de la case dans laquelle on pourra empiler le prochain caractère. Ce n'est donc pas exactement le sommet de la pile, mais un cran au dessus.

Le pointeur sur un caractère `pile` désigne le tableau de caractères, alloué dynamiquement (avec `new`) pour mémoriser le contenu de la pile.

Dans cet exemple, on a successivement empilé les quatre lettres du mot "pile". Ici, `max` vaut 5 (le nombre maximal de lettre empilable). et `sommet` vaut 4 puisque le dernier élément empilé est le 'e' du mot "pile", et que le prochain empilage se fera en `pile[4]`.



Paramètre par défaut

Il y a évidement un **constructeur** qui prend en paramètre un entier strictement positif pour préciser la taille désirée pour la pile, future valeur pour `max`. On précisera pour ce paramètre, une valeur par défaut de `50` (c'est une **constante** !).

Le langage C++ offre la possibilité d'avoir des **valeurs par défaut pour les paramètres d'une fonction (ou d'une méthode)**, qui peuvent alors être sous-entendus au moment de l'appel.

Cette possibilité, permet d'écrire qu'un seul constructeur profitant du mécanisme de valeur par défaut :

```
#define TAILLE_PAR_DEFAUT 50

// Active les affichages de debuggage pour les constructeurs et destructeur
#define DEBUG

class PileChar {
   private:
      unsigned int max;
      unsigned int sommet;
      char *pile;
   public:
      PileChar(int taille=TAILLE_PAR_DEFAUT); // je suis le constructeur (par défaut) de la
          classe PileChar
};
```

PileChar.h

Liste d'initialisation

Un meilleur moyen d'affecter des valeurs aux données membres de la classe lors de la construction est la **liste d'initialisation**. On va utiliser cette technique pour définir le constructeur de la classe `PileChar` :

```
PileChar::PileChar(int taille/*=TAILLE_PAR_DEFAUT*/) : max(taille), sommet(0) // c'est la
    liste d'initialisation
{
   pile = new char[max]; // allocation dynamique du tableau de caractère
   #ifdef DEBUG
   cout << "PileChar(" << taille << ") : " << this << "\n";
   #endif
}
```

PileChar.cpp

La liste d'initialisation permet d'utiliser le constructeur de chaque donnée membre, et ainsi d'éviter une affectation après coup. La liste d'initialisation doit être utilisée pour certains objets qui ne peuvent pas être contruits par défaut : c'est le cas des références et des objets constants.

Destructeur

Le **destructeur** est la **méthode membre appelée automatiquement lorsqu'une instance (objet) de classe cesse d'exister en mémoire** :

- Son rôle est de **libérer toutes les ressources qui ont été acquises lors de la construction** (typiquement libérer la mémoire qui a été allouée dynamiquement par cet objet).
- Un destructeur est une **méthode qui porte toujours le même nom que la classe, précédé de "~"**.
- Il existe quelques contraintes :
  - Il ne possède aucun paramètre.
  - Il n'y en a qu'un et un seul.
  - Il n'a jamais de type de retour.

La forme habituelle d'un destructeur est la suivante :

```
class T {
   public:
      ~T(); // destructeur
};
```

Pour éviter les fuites de mémoire, le destructeur de la classe `PileChar` doit libérer la mémoire allouée au tableau de caractères `pile` :

On le **définit** de la manière suivante :

```
PileChar::~PileChar()
{
   delete [] pile;
   #ifdef DEBUG
   cout << "~PileChar() : " << this << "\n";
   #endif
}
```

PileChar.cpp

Question 1. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter le constructeur et le destructeur de cette classe. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier qu'il y a bien autant d'affichages de destructeurs que de constructeurs.

```
PileChar pile1a; // construit une pile avec une taille par défaut

PileChar pile1b(10); // construit une pile avec une taille de 10 caractères
```

Les services rendus

Différentes **méthodes publiques** doivent être déclarées et définies dans la classe :
- Une méthode `taille()` qui donne comme résultat un entier positif qui est le nombre maximum de caractères qu'il sera possible d'y mettre
- Une méthode `compter()` qui donne comme résultat un entier positif qui est le nombre d'éléments actuellement présents sur la pile
- Une méthode `afficher()` qui affiche entre des '[' et ']' les éléments actuellement présents dans la pile. Sur l'exemple précédant, cette méthode donnerait l'affichage suivant : "[pile]"
- Une méthode `empiler()` qui prend un caractère en paramètre et le place sur le dessus de la pile
- Une méthode `depiler()` qui enlève le caractère du dessus de la pile et le renvoi en valeur de retour

Pour les méthodes `empiler()` et `depiler()`, on prendra garde à vérifier que l'opération demandée est bien possible (attention aux piles pleines et aux piles vides). On affichera un message d'erreur sur `cerr` dans les cas où l'opération ne peut être effectuée.

Question 2. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter ces services. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous.

```
Pile vide !
Pile pleine !
Pile pleine !
Contenu de pile2 : [hello]
Nombre d'éléments présents sur pile2 : 5
Taille de pile2 : 5
Contenu de pile2 : [hell]
Nombre d'éléments présents sur pile2 : 4
```

Constructeur de copie

Le **constructeur de copie** est appelé dans :
- la création d'un objet à partir d'un autre objet pris comme modèle
- le passage en paramètre d'un objet par valeur à une fonction ou une méthode
- le retour d'une fonction ou une méthode renvoyant un objet

Remarque : toute autre duplication (au cours de la vie d'un objet) sera faite par l'opérateur d'affectation (=).

La forme habituelle d'un constructeur de copie est la suivante :

```
class T
{
   public:
      T(const T&);
};
```

Donc pour la classe `PileChar` :

```
PileChar::PileChar(const PileChar &p) : max(p.max), sommet(p.sommet)
{
   pile = new char[max]; // on alloue dynamiquement le tableau de caractère

   unsigned int i;

   // on recopie les éléments de la pile
   for (i = 0; i < sommet ; i++) pile[i] = p.pile[i];
   #ifdef DEBUG
   cout << "PileChar(const PileChar &p) : " << this << "\n";
   #endif
}
```

Deux situations où le constructeur de copie est nécessaire :

```
PileChar pile3a(pile2); // Appel du constructeur de copie pour instancier pile3a

PileChar pile3b = pile1b; // Appel du constructeur de copie pour instancier pile3b
```

Question 3. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter le constructeur de copie. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous et qu'il y a bien autant d'affichages de destructeurs que de constructeurs.

```
Contenu de pile3a : [hell]
Nombre d'éléments présents sur pile3a : 4
Taille de pile3a : 5
Contenu de pile3b : []
Nombre d'éléments présents sur pile3b : 0
Taille de pile3b : 10
```

Opérateur d'affectation

L'opérateur d'affectation (=) est un opérateur de copie d'un objet vers un autre. L'objet affecté est déjà créé sinon c'est le constructeur de copie qui sera appelé.

La forme habituelle d'opérateur d'affectation est la suivante :

```
class T
{
   public:
      T& operator=(const T&);
};
```


Cet opérateur renvoie une référence sur `T` afin de pouvoir l'utiliser avec d'autres affectations. En effet, l'opérateur d'affectation est associatif à droite a=b=c est évaluée comme a=(b=c). Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable.

La définition de l'opérateur = est la suivante :

```
PileChar& PileChar::operator = (const PileChar &p)
{
   // vérifions si on ne s'auto-copie pas !
   if (this != &p)
   {
      delete [] pile; // on libère l'ancienne pile
      max = p.max;
      sommet = p.sommet;
      pile = new char[max]; // on alloue une nouvelle pile
      unsigned int i;
      for (i = 0; i < sommet ; i++) pile[i] = p.pile[i]; // on recopie les éléments de la
          pile
   }
   #ifdef DEBUG
   cout << "operator= (const PileChar &p) : " << this << "\n";
   #endif
   return *this;
}
```

Ce qui permettra d'écrire :

```
PileChar pile4; // Appel du constructeur par défaut pour créer pile4

pile4 = pile3a; // Appel de l'opérateur d'affectation pour copier pile3a dans pile4
```

Question 4. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter l'opérateur d'affectation =. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous. Expliquer pourquoi le message d'erreur "Pile pleine !" s'affiche ?

```
Taille de pile4 : 50
Pile pleine !
Pile pleine !
Contenu de pile4 : [hello]
Nombre d'éléments présents sur pile4 : 5
Taille de pile4 : 5
```

Surcharge d'opérateurs

La **surcharge d'opérateur** permet aux opérateurs du C++ d'avoir une signification spécifique quand ils sont appliqués à des types spécifiques. Parmi les nombreux exemples que l'on pourrait citer :
– `myString + yourString` pourrait servir à concaténer deux objets `string`
– `maDate++` pourrait servir à incrémenter un objet `Date`
– `a * b` pourrait servir à multiplier deux objets `Nombre`
– `e[i]` pourrait donner accès à un élément contenu dans un objet `Ensemble`

Les opérateurs C++ que l'on surcharge habituellement :
- Affectation, affectation avec opération (`=`, `+=`, `*=`, etc.) : **Méthode**
- Opérateur « fonction » `()` : **Méthode**
- Opérateur « indirection » `*` : **Méthode**
- Opérateur « crochets » `[]` : **Méthode**
- Incrémentation `++`, décrémentation `--` : **Méthode**
- Opérateur « flèche » et « flèche appel » `->` et `->*` : **Méthode**
- Opérateurs de décalage `<<` et `>>` : **Méthode**
- Opérateurs `new` et `delete` : **Méthode**
- Opérateurs de lecture et écriture sur flux `<<` et `>>` : **Fonction**
- Opérateurs dyadiques genre « arithmétique » (`+`,`-`,`/` etc) : **Fonction**

Les autres opérateurs ne peuvent pas soit être surchargés soit il est déconseillé de le faire.

La **première technique** pour surcharger les opérateurs consiste à les considérer comme des **méthodes** normales de la classe sur laquelle ils s'appliquent.

Le principe est le suivant :

```
A Op B se traduit par A.operatorOp(B)

t1 == t2; // équivalent à : t1.operator==(t2)
t1 += t2; // équivalent à : t1.operator+=(t2)
```

On surcharge les opérateurs `==`, `!=` et `+=` pour la classe `PileChar` :

```
class PileChar
{
   private:
      unsigned int max;
      unsigned int sommet;
      char *pile;

   public:
      ...
      bool operator == (const PileChar &p); // teste si deux piles sont identiques
      bool operator != (const PileChar &p); // teste si deux piles sont différentes
      PileChar& operator += (const PileChar &p); // empile une pile sur une autre
};

bool PileChar::operator == (const PileChar &p)
{
   if(max != p.max) return false;
   if(sommet != p.sommet) return false;
   unsigned int i;
   for (i = 0; i < sommet ; i++) if(pile[i] != p.pile[i]) return false;
   return true;
}

bool PileChar::operator != (const PileChar &p)
{
   // TODO
}
```



```
PileChar& PileChar::operator += (const PileChar &p)
{
   unsigned int i, j;

   // Reste-il assez de place pour empiler les caractères ?
   if((sommet + p.sommet) <= max)
   {
      for (i = sommet, j = 0; j < p.sommet ; i++, j++) pile[i] = p.pile[j]; // on recopie
          les éléments de la pile
      sommet += j; // on met à jour le nouveau sommet
   }
   else cerr << "Pile pleine !\n";

   return *this;
}
```

Question 5. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter les opérateurs `==`, `!=` et `+=`. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous.

```
pile3a et pile2 identiques !
pile3a et pile4 différentes !

Taille de pile5 : 6
Contenu de pile5 : [hello]
Nombre d'éléments présents sur pile5 : 5
Taille de pile5 : 6
Pile pleine !
Contenu de pile5 : [hello]
```

On désire aussi surcharger l'opérateur `+=` pour **empiler une simple caractère**. De la même manière, on va surcharger l'opérateur `-=` pour **supprimer `n` caractères depuis le haut de la pile** :

```
PileChar pile6;

pile6 += 'a'; // empile le caractère 'a'
pile6 += 'z'; // empile le caractère 'z'
pile6 += 'e'; // empile le caractère 'e'
...
pile6 -= 2; // supprime 2 caractères du haut de la pile (donc 'e' puis 'z')
```

Question 6. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin de surcharger les opérateurs `+=` et `-=`. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous.

```
Contenu de pile6 : [azerty]
Nombre d'éléments présents sur pile6 : 6
Taille de pile6 : 50
Donner le nombre de caractères à supprimer dans pile6 : 2
Contenu de pile6 : [azer]
Nombre d'éléments présents sur pile6 : 4
```

La **deuxième technique** utilise la surcharge d'opérateurs externes sous forme de **fonctions**. La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres.

Le principe est le suivant :

```
A Op B se traduit par operatorOp(A, B)

t1 + t2; // équivalent à : operator+(t1, t2)
```

Les opérateurs externes doivent être déclarés comme étant des **fonctions amies** (`friend`) de la classe sur laquelle ils travaillent, faute de quoi ils ne pourraient pas manipuler les données membres de leurs opérandes.

Exemple pour une classe `T` :

```
class T
{
   private:
      int x;

   public:
      friend T operator+(const T &a, const T &b);
};

T operator+(const T &a, const T &b)
{
   // solution n° 1 :
   T result = a;
   result.x += b.x;
   return result;

   // solution n° 2 : si l'opérateur += a été surchargé
   T result = a;
   return result += b;
}
```

```
T t1, t2, t3; // Des objets de type T

t3 = t1 + t2; // Appel de l'opérateur + puis de l'opérateur de copie (=)

t3 = t2 + t1; // idem car l'opérateur est symétrique
```

L'avantage de cette syntaxe est que l'opérateur est réellement symétrique, contrairement à ce qui se passe pour les opérateurs définis à l'intérieur de la classe.

Question 7. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin de surcharger l'opérateur `+` sour forme de fonction amie externe. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous. Profiter pour bien repérer les instants du déroulement du programme où ont lieu les appels aux constructeurs, destructeurs et affectations, éventuellement en plaçant des affichages supplémentaires dans votre programme.

```
Contenu de pile7 : [azerhello]
Nombre d'éléments présents sur pile7 : 9
Taille de pile7 : 50
```

Intérêt et utilisation d'une pile

Par exemple, on peut très simplement inverser les éléments contenus dans un tableau ou dans une chaîne de caractères (pour tester un palindrome) en utilisant une **pile**. Il suffit d'empiler les éléments sur une pile puis de reconstituer le tableau (ou la chaîne) inverse en dépilant les éléments.

Question 8. Écrire dans le fichier `testPileChar.cpp` une **fonction** `affiche_inverse()` recevant en paramètre une **pile** et qui dépile les lettres qui y sont contenues en les écrivant à l'écran au fur et à mesure. ATTENTION : Cette fonction ne doit pas modifier le contenu de la pile déclarée dans le `main()` ... Comment s'y prendre ?

```
Contenu de pile8 : [soleil]
Contenu inverse de pile8 :
lielos
Contenu de pile8 : [soleil]
```

Pour finir, on désire pouvoir créer des piles à partir de **chaînes de caractères C** (`const char []`) ou **C++** (`string`). Pour cela, on va déclarer deux nouveaux constructeurs pour la classe `PileChar` :

```
class PileChar
{
   private:
      unsigned int max;
      unsigned int sommet;
      char *pile;

   public:
      ...
      PileChar(const string &init); // pour les chaînes de caractères en C++
      PileChar(const char init[]); // pour les chaînes de caractères en C
      ...
};
```

Question 9. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter ces deux nouveaux constructeurs. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous.

```
Contenu de pile9a : [Bonjour]
Nombre d'éléments présents sur pile9a : 7
Taille de pile9a : 7
Contenu inverse de pile9a :
ruojnoB

Contenu de pile9b : [Hello world]
Nombre d'éléments présents sur pile9b : 11
Taille de pile9b : 11
Contenu inverse de pile9b :
dlrow olleH
```

