TP POO C++ : La classe PileChar

TP POO C++ : La classe PileChar

Les objectifs de ce TP sont de 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

Travail demandé

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`).

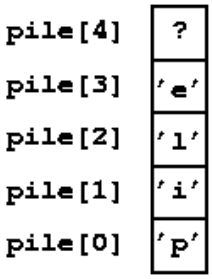
La classe `PileChar` contient **trois données membres privées** :
- deux entiers strictement positifs, nommés `max` et `sommet`, et
- un pointeur sur un caractère, nommé `pile`.

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 taille sera fixée à l'initialisation donc `max` sera déclarée comme donnée membre **constante**.

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]`.



Lorsqu'on instancie un objet `PileChar`, on doit pouvoir choisir sa `taille`. Cela nous donne la première déclaration de la classe `PileChar` :

```
#define DEBUG

class PileChar
{
```

```
   private:
      const unsigned int max;
      unsigned int sommet;
      char *pile;

   public:
      PileChar(unsigned int taille);
};
```

PileChar.h

Liste d'initialisation

La liste d'initialisation permet d'utiliser le constructeur de chaque donnée membre, et ainsi d'éviter une affectation après coup. C'est donc le meilleur moyen pour initialiser les membres. D'autre part, la liste d'initialisation doit être obligatoirement utilisée pour certains objets qui ne peuvent pas être contruits par défaut : c'est le cas des références et des membres données constants (ici `max`).

Ici, le membre `max` est une constante au niveau de l'**objet** : chaque instance pourrait avoir une taille de pile différente.

On va utiliser la liste d'initialisation pour définir le constructeur de la classe `PileChar` qui prend en paramètre un entier strictement positif pour préciser la taille désirée pour la pile, future valeur pour `max` :

```
PileChar::PileChar(unsigned int taille) : max(taille), sommet(0), pile(0) // c'est la liste
    d'initialisation
{
   // TODO : allocation dynamique du tableau de caractère

   #ifdef DEBUG
   cout << "PileChar(" << taille << ") : " << this << "\n";
   #endif
}
```

PileChar.cpp

Question 1. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter le constructeur et réaliser l'allocation dynamique pour la pile. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester.

```
/* Question 1 */
cout << "Question 1 :\n";
PileChar pile1(10);
```

Ce qui doit donner :

```
Question 1 :
PileChar(10) : 0x7fff88a415f0
```

Paramètre par défaut

On souhaiterait pouvoir instancier des objets `PileChar` avec une taille par défaut de `50`. C'est donc une **constante** mais au niveau de la **classe**. Pour cela, il faut lui ajouter le mot clé `static` et, exceptionnellement, l'initialiser dans la déclaration de la classe.

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 :

```
class PileChar
{
   private:
      static const int tailleDefaut = 50; // constante de classe
      // ou anciennement :
      //enum { tailleDefaut = 50 };
      const unsigned int max; // constante d'objet
      unsigned int sommet;
      char *pile;

   public:
      PileChar(int taille=tailleDefaut); // paramètre par défaut
};
```

PileChar.h

Question 2. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter le constructeur par défaut. Décommenter les parties de code de cette question dans le fichier `testPileChar.cpp` et tester.

```
/* Question 2 */
cout << "Question 2 :\n";
PileChar pile2;
```

Ce qui doit donner :

```
Question 2 :
PileChar(50) : 0x7ffff8d14880
```

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 pour le tableau de caractères `pile`.

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

```
PileChar::~PileChar()
{
   // TODO : libération de la mémoire allouée pour la pile

   #ifdef DEBUG
   cout << "~PileChar() : " << this << "\n";
   #endif
}
```

PileChar.cpp

Question 3. Compléter les fichiers `PileChar.cpp` et `PileChar.h` fournis afin d'implémenter 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.

```
/* Question 3 */
cout << "Question 3 :\n";
PileChar pile3a;
PileChar pile3b(20);
PileChar* pile3c = new PileChar(30);
delete pile3c;
```

Ce qui doit donner :

```
Question 1 :
PileChar(10) : 0x7fffc95ed4b0

Question 2 :
PileChar(50) : 0x7fffc95ed4c0

Question 3 :
PileChar(50) : 0x7fffc95ed4d0
PileChar(20) : 0x7fffc95ed4e0
PileChar(30) : 0x23b00d0
~PileChar() : 0x23b00d0

~PileChar() : 0x7fffc95ed4e0
~PileChar() : 0x7fffc95ed4d0
~PileChar() : 0x7fffc95ed4c0
~PileChar() : 0x7fffc95ed4b0
```

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 4. 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.

```
/* Question 4 */
cout << "Question 4 :\n";
PileChar pile4(5);

pile4.depiler(); // pile vide !

pile4.empiler('h');
pile4.empiler('e');
pile4.empiler('l');
pile4.empiler('l');
pile4.empiler('o');
pile4.empiler(' '); // pile pleine !
pile4.empiler('!'); // pile pleine !

cout << "Contenu de pile4 : "; pile4.afficher();
cout << "Nombre d'éléments présents sur pile4 : " << pile4.compter() << endl;
cout << "Taille de pile4 : " << pile4.taille() << endl;

char c = pile4.depiler(); // on récupère 'o'
cout << "Caractère dépilé : " << c << endl;

cout << "Contenu de pile4 : "; pile4.afficher();
cout << "Nombre d'éléments présents sur pile4 : " << pile4.compter() << endl;
```

Ce qui doit donner :

```
Question 4 :
PileChar(5) : 0x7fff530119d0
Pile vide !
Pile pleine !
Pile pleine !
Contenu de pile4 : [hello]
Nombre d'éléments présents sur pile4 : 5
Taille de pile4 : 5
Caractère dépilé : o
Contenu de pile4 : [hell]
Nombre d'éléments présents sur pile4 : 4
```

Constructeur de copie

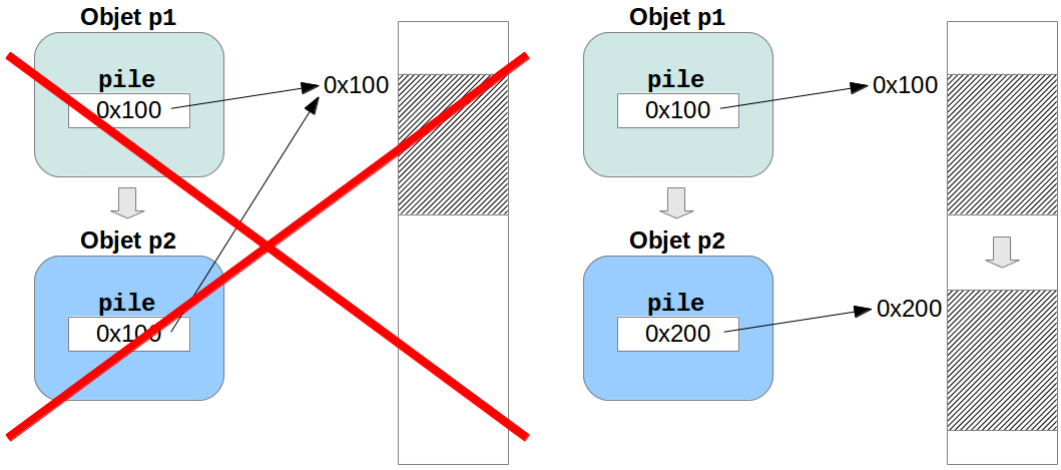
Le **constructeur de copie** est appelé dans :
- la création d'un objet à partir d'un autre objet pris comme modèle
- le passage en paramètre d'un objet par valeur à une fonction ou une méthode
- le retour d'une fonction ou une méthode renvoyant un objet

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&);
};
```

Il faut faire très attention avec les classes qui manipulent de la mémoire dynamique et des pointeurs :



Donc pour la classe `PileChar` :

```
PileChar::PileChar(const PileChar &p) : max(p.max), sommet(p.sommet), pile(0)
{
   // TODO :
   // 1. on alloue dynamiquement le tableau de caractère


   // 2. on recopie les éléments de la pile


   #ifdef DEBUG
   cout << "PileChar(const PileChar &p) : " << this << "\n";
   #endif
}
```

Question 5. 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.

```
cout << "Question 5 :\n";
// Deux situations où le constructeur de copie est appelé :
PileChar pile5a(pile4); // Appel du constructeur de copie pour instancier pile5a
PileChar pile5b = pile1; // Appel du constructeur de copie pour instancier pile5b

cout << "Contenu de pile5a : "; pile5a.afficher();
cout << "Nombre d'éléments présents sur pile5a : " << pile5a.compter() << endl;
cout << "Taille de pile5a : " << pile5a.taille() << endl;

cout << "Contenu de pile5b : "; pile5b.afficher();
cout << "Nombre d'éléments présents sur pile5b : " << pile5b.compter() << endl;
cout << "Taille de pile5b : " << pile5b.taille() << endl;
```

Ce qui doit donner :

```
Question 5 :
PileChar(const PileChar &p) : 0x7fff6694b250
PileChar(const PileChar &p) : 0x7fff6694b260
Contenu de pile5a : [hell]
Nombre d'éléments présents sur pile5a : 4
Taille de pile5a : 5
Contenu de pile5b : []
Nombre d'éléments présents sur pile5b : 0
Taille de pile5b : 10
```

Opérateur d'affectation

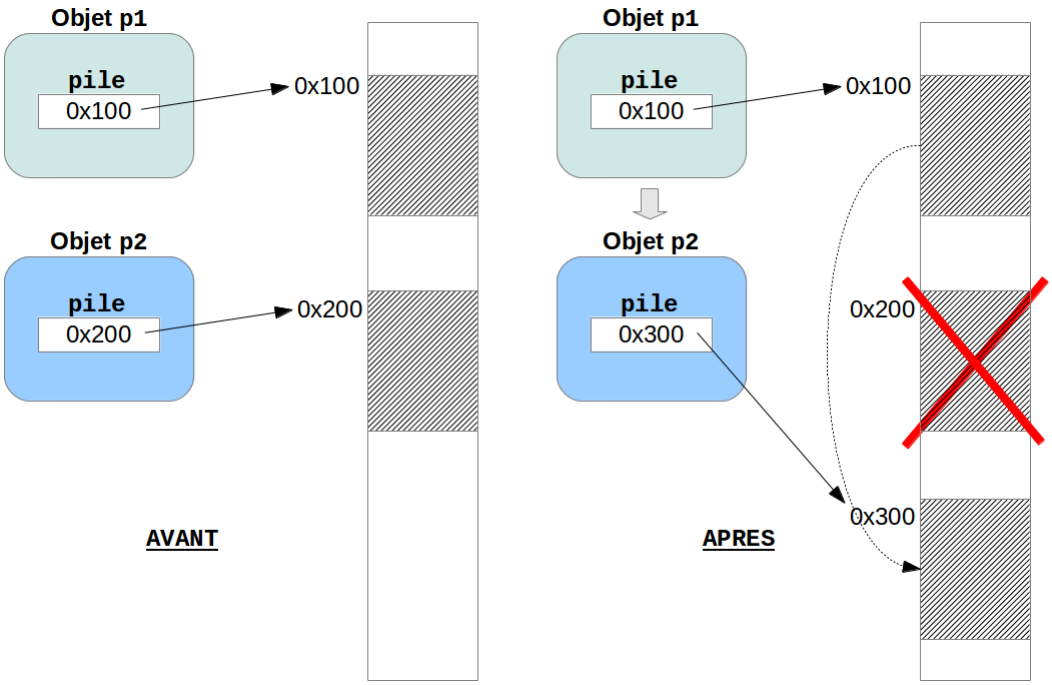
L'opérateur d'affectation (=) est un opérateur de copie d'un objet vers un autre. L'objet affecté est déjà créé sinon c'est le constructeur de copie qui sera appelé.

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.

Il faut faire très attention avec les classes qui manipulent de la mémoire dynamique et des pointeurs :



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)
   {
      // TODO :
      // 1. on libère l'ancienne pile


      // 2. on réinitialise les attributs


      // 3. on alloue une nouvelle pile


      // 4. on recopie les éléments de la pile


   }
   #ifdef DEBUG
   cout << "operator= (const PileChar &p) : " << this << "\n";
   #endif

   return *this;
}
```

Que se passe-t-il si la pile que l'on veut copier a une taille plus grande que l'ancienne pile ? Soit on n'accepte pas la copie soit on a un problème avec l'attribut constant `max` ! On ne peut pas enlever le caractère `const` avec un *cast* du type `const_cast` puisque cela « contredirait » notre choix de départ (même si cela peut se compiler, c'est considéré comme un comportement indéfini et donc non valide). Ici, on va accepter de copier des `PileChar` plus grande à condition d'enlever le qualificatif `const` à l'attribut `max`. Cela permettra aussi de redimensionner nos `PileChar` à l'avenir.

Question 6. 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 ?

```
/* Question 6 */
cout << "Question 6 :\n";
PileChar pile6; // Appel du constructeur par défaut pour créer pile6

pile6 = pile4; // Appel de l'opérateur d'affectation pour copier pile4 dans pile6

pile6.empiler('o');

cout << "Contenu de pile6 : "; pile6.afficher();
cout << "Nombre d'éléments présents sur pile6 : " << pile6.compter() << endl;
cout << "Taille de pile6 : " << pile6.taille() << endl;
```

Ce qui doit donner :

```
Question 6 :
PileChar(50) : 0x7fff080dd560
operator= (const PileChar &p) : 0x7fff080dd560
Contenu de pile6 : [hello]
Nombre d'éléments présents sur pile6 : 5
Taille de pile6 : 5
```

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 7. 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 ?

```
/* Question 7 */
cout << "Question 7 :\n";
PileChar pile7;

pile7.empiler('s');
pile7.empiler('o');
pile7.empiler('l');
pile7.empiler('e');
pile7.empiler('i');
pile7.empiler('l');

cout << "Contenu de pile7 : "; pile7.afficher();
cout << "Contenu inverse de pile7 : "; affiche_inverse(pile7);
cout << "Contenu de pile7 : "; pile7.afficher();
```

Ce qui doit donner :

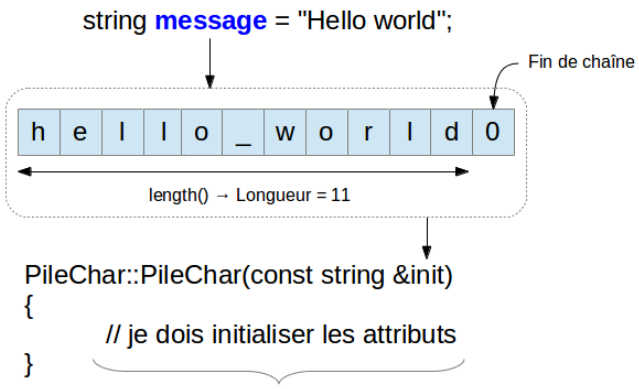
```
Question 7 :
PileChar(50) : 0x7fffc2533570
Contenu de pile7 : [soleil]
Contenu inverse de pile7 :
lielos
Contenu de pile7 : [soleil]
```

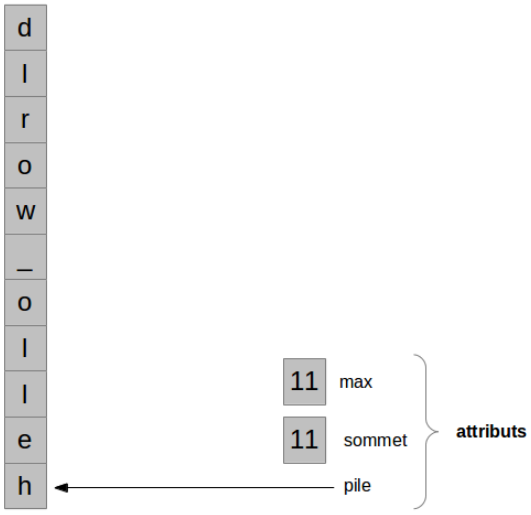
Pour finir, on désire pouvoir créer des piles à partir de **chaînes de caractères C** (types `const char []` ou `const char*`) ou **C++** (type `string`). Pour cela, on va déclarer deux nouveaux constructeurs pour la classe `PileChar` :

```
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
      ...
};
```

Le principe est le suivant (pour le type `string`) :





Question 8. 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.

```
/* Question 8 */
cout << "Question 8 :\n";
PileChar pile8a("Bonjour"); // chaine C

cout << "Contenu de pile8a : "; pile8a.afficher();
cout << "Nombre d'éléments présents sur pile8a : " << pile8a.compter() << endl;
cout << "Taille de pile8a : " << pile8a.taille() << endl;
cout << "Contenu inverse de pile8a : "; affiche_inverse(pile8a);

string message = "Hello world";
PileChar pile8b(message); // chaine C++

cout << "Contenu de pile8b : "; pile8b.afficher();
cout << "Nombre d'éléments présents sur pile8b : " << pile8b.compter() << endl;
cout << "Taille de pile8b : " << pile8b.taille() << endl;
cout << "Contenu inverse de pile8b : "; affiche_inverse(pile8b);
```

Ce qui doit donner :

```
Question 8 :
PileChar("Bonjour") : 0x7fff80a11fb0
Contenu de pile8a : [Bonjour]
Nombre d'éléments présents sur pile8a : 7
Taille de pile8a : 7
Contenu inverse de pile8a :
ruojnoB

PileChar("Hello world") : 0x7fff80a11fc0
Contenu de pile8b : [Hello world]
Nombre d'éléments présents sur pile8b : 11
Taille de pile8b : 11
Contenu inverse de pile8b :
dlrow olleH
```