

# Algo/C++/Structures de Données - Devoir n° 2 - 2005

© D. Mathieu [mathieu@romarin.univ-aix.fr](mailto:mathieu@romarin.univ-aix.fr)  
I.U.T.d'Aix en Provence - Département Informatique  
Créé le 21/03/2005 - Dernière mise à jour : 23/03/2005

---

Donné le 22 mars 2005.

A rendre au secrétariat le **lundi 9 mai** au matin.

Vous devez effectuer ce devoir par **trinômes** de **trois** étudiants ... que vous composerez à votre gré. Chaque délégué de groupe devra remettre la liste des trinômes au secrétariat **au plus tard** le vendredi 25 mars.

Dans les jours qui viennent, une version papier du sujet vous sera distribuée.

## Sujet

Il vous est proposé de développer une classe `CString` légèrement différente de la classe `string` standard, permettant une meilleure gestion de la mémoire en perdant très peu d'efficacité. La structure interne d'une `string` a été étudiée à plusieurs reprises, en particulier dans l'amphi 14.

Rappelons qu'une `string` standard ne contient qu'un pointeur vers une adresse de mémoire dynamique qui, elle, contient la NTCTS correspondante, **terminée par** `'\0'` (par définition même d'une NTCTS).

Rappelons aussi que l'information "longueur" n'est pas contenue **en dur** dans la chaîne. Chaque appel aux fonctions `length()` et `size()` nécessite donc le comptage des caractères jusqu'à la rencontre de `'\0'`.

Le second inconvénient de cette implémentation, moins visible sur les petits programmes mais qui peut être très pénalisant pour les applications manipulant de gros textes (éditeurs de textes par exemple ...) : du fait de créations/suppressions successives, les NTCTS peuvent être dispersées (fragmentation de la mémoire) sur de nombreuses pages mémoires physiques, et entraîner des **défauts de page** pénalisants.

Enfin, en cas d'urgence, l'utilisateur doit sauvegarder individuellement chaque chaîne, en parcourant tous les objets de type `string`.

La solution que nous proposons présente deux modifications par rapport à une `string` :

- chaque chaîne contient son nombre de caractères stocké directement dans l'objet comme donnée membre,
- toutes les NTCTS sont stockées dans un grand tableau de caractères, lui-même en mémoire dynamique. Cette dernière caractéristique apporte une réponse aux deux derniers inconvénients cités ci-dessus.

La classe `CString` contient donc une donnée-membre **statique** (variable de classe) qui est un **pointeur** vers le tableau de caractères. On peut lui donner une taille initiale par défaut, comme cela a été vu dans d'autres circonstances (par exemple les classes `CStackxx`).

Chaque instance de la classe `CString` (chaque objet) conserve donc deux informations :

- la taille de la NTCTS correspondante (le nombre d'octets de la chaîne),
- un "pointeur" vers le début de la NTCTS dans le tableau. En fait, pour des raisons qui seront évoquées plus loin, il est plus avantageux de conserver **l'indice** de l'élément que son adresse (ils occupent tous deux 4 octets). Cet indice peut être vu comme le **déplacement** de l'octet par rapport au début du tableau (la **base**).

En cas de création d'une nouvelle chaîne, le constructeur de `CString` doit rechercher un nombre d'octets consécutifs suffisant pour y ranger la NTCTS, placer dans la `CString` le rang du début de cette zone et interdire la réaffectation de cette zone à une autre chaîne, tant que le destructeur ne l'aura pas libéré.

Le constructeur (en réalité la classe) doit donc conserver une image précise de tous les octets libres/occupés du tableau des NTCTS. Différentes méthodes de gestion de mémoire sont possibles, parmi lesquelles nous vous proposons de tester les trois suivantes, utilisées habituellement par les gestionnaires mémoire (systèmes d'exploitation ou langages).

## ***Le mapping* mémoire.**

A chaque octet du tableau des NTCTS est associé un bit, à 0 ou à 1 selon que l'octet correspondant est libre ou occupé. On s'aperçoit que cette gestion très fine (chaque octet est comptabilisé individuellement) est coûteuse : il faut 1 bit pour représenter l'état d'un octet, donc de 8 bits. Le taux de perte de mémoire est de 1/8, soit 12,5 %

La solution consiste à ne prendre en compte que des **granules** d'octets (puissances de 2 : 2, 4, 8, etc...), 1 bit représentant l'état d'occupation du granule complet. Des études statistiques de fonctionnement dans un contexte particulier peuvent aider au choix de la meilleure taille de granule. Dans le devoir, vous fixerez une valeur arbitraire, tout en permettant une reconfiguration la plus facile possible en cas de changement.

Une taille de granule de 4 octets conduit à une perte de 1/32 de mémoire, soit environ 3%, ce qui est beaucoup plus raisonnable.

Lorsque le tableau initial des NTCTS est réservé, sa taille permet de déterminer la taille de la table de *mapping* :

$$\text{Nb\_octets\_table\_mapping} = \text{Nb\_octets\_table\_NTCTS} / (8 * \text{Taille\_Granule})$$

La recherche d'une place libre pour ranger une nouvelle NTCTS de  $N$  caractères consiste à rechercher  $N$  bits consécutifs à 0 (mais qui peuvent être à cheval sur plusieurs octets !).

La libération de la mémoire consiste à remettre à 0 tous les bits correspondant à la NTCTS détruite.

**Remarque** : la bibliothèque standard du C++ offre une classe, appelée **bit\_vector**, qui pourrait vous rendre quelques services !. Elle est décrite dans le document **SGI** du C++ mis à votre disposition sur mon site.

## La gestion mémoire arborescente (*buddy system*)

La recherche des séquences de bits libres est relativement coûteuse. Au risque de perdre encore un peu de place mémoire, on peut allouer les espaces mémoires par multiples de puissances de 2 : 2, 4, 8, .... granules.

La recherche est beaucoup plus efficace : les séquences de bits  $< 8$  ne sont **jamais** à cheval sur 2 octets. Les séquences de bits  $\geq 8$  bits sont des séquences d'**octets** !!!

## Les listes chaînées

Une autre solution consiste à mémoriser les différents espaces non affectés de la table des NTCTS dans une liste chaînée de "trous", chacun conservant la taille du trou et la position du trou suivant.

Très facile à mettre en oeuvre après les TPs de C++, elle est très efficace pour rechercher un espace mémoire de taille  $\geq$  à la taille requise. Lorsque l'espace à allouer est inférieur à la taille d'un "trou" disponible, la partie restante constitue à son tour un espace mémoire disponible.

Néanmoins, lorsqu'un espace est libéré, il faut chaque fois vérifier s'il suit ou s'il précède **immédiatement** un autre trou, auquel cas ils doivent être fusionnés en un trou plus volumineux.

La technique peut être raffinée en gérant des listes de "trous" de tailles différentes, et en cherchant en premier lieu un espace dans la liste de taille la plus adaptée.

## La classe `CString`

Outre ce qui a été signalé plus haut, il est au minimum nécessaire de doter la classe `CString` de quelques "fonctions de base" indispensables :

- constructeur par recopie et affectation
- quelques constructeurs qui s'inspirent de ceux de `string`,

- quelques opérateurs/fonctions permettant d'accéder, de modifier, allonger, concatener, etc.

Vous devez (essayer d')implémenter ces trois méthodes de gestion de la mémoire, ainsi que quelques jeux d'essais pour prouver leur (bon) fonctionnement.

En cas de manque de place de mémoire, vous pouvez envisager de réallouer un tableau de taille supérieure, selon une des techniques envisagées en TP. Notez que, malgré le mouvement **global** du tableau, les chaînes restent à leur place **relative**

## Exemple

Après la séquence suivante, en supposant que la taille des granules est de 4 octets, et en utilisant la première méthode (*mapping* de la mémoire) :

```
CString Ch1 ("En");
CString Ch2 ("supposant");
CString Ch3 ("que");
CString Ch4 ("la");
```

le contenu de la classe `CString` est le suivant :

Les espaces peuvent en fait être n'importe quel caractère. La table de *mapping* ne comporte qu'un seul 0 de rang 6 signalant que seul le granule commençant au déplacement 24 (= 6 x 4) est libre .

Après l'instruction suivante :

```
Ch2 = Ch1;
```

le contenu de la classe `CString` devient :

L'espace occupé par `Ch2` a **virtuellement** été libéré (les bits 1, 2 et 3 ont été remis à zéro, mais le contenu du vecteur n'a pas été modifié). Puis le granule 1 a été de nouveau affecté pour la chaîne `ch2` et le contenu de `Ch1` y a été recopié.

Outre celles qui ont été précédemment signalées (concaténations, `assign()`, etc.), il serait intéressant que de nombreuses opérations sur les `strings` restent possibles sur les `CString`, comme par exemple :

```
const CString B ("Bonjour");
cout << B << '\n';
cout << CString ("Hello");
Ch3 [2] = 'x';
cout << Ch4.at (0);
```