

Vincent Castille  
Nicolas Lichnowski  
Mathieu Lapeyre

Groupe 3  
1<sup>ère</sup> année  
2004 / 2005

# **Devoir de C++ n° 2** **Structures de données**

Devoir tuteuré sous la direction de Didier Mathieu

# Sommaire

## I) Mapping

- Commentaires.....	3
- CString.h.....	8
- CString.cpp.....	11
- Exo_01.cpp.....	17
- Makefile_01.....	20

## II) Buddy System

- Commentaries.....	4
- CString.h.....	22
- CString.cpp.....	25
- TestString.cpp.....	35
- Makefile.....	37

## III) Listes chaînés

- Commentaires.....	5
- CString.h.....	39
- CString.cpp.....	42
- CTrou.h.....	48
- CTrou.cpp.....	51
- CExceptions.h.....	55
- CExceptions.cpp.....	57
- Exo_01.cpp.....	59
- Makefile_01.....	61

## I) Mapping

On a développé une classe `CString` dans laquelle la taille des granules est paramétrable. Nous avons déclaré des données membres `static Sz_Mapping` et `Sz_NTCTS` qui représentent respectivement la taille du vecteur de booléen (`VMapping` qui est lui aussi `static`). Nous utilisons un pointeur de `char` `static` vers un tableau de `NTCTS` pour ranger les chaînes. Ces 4 variables sont définies `static` car il ne faut pas qu'elle change à chaque création d'une chaîne (ex : On ne va pas créer un deuxième vecteur de booléen et un deuxième pointeur de `char`). Nous avons aussi deux autres données membres qui ne sont pas `static` `m_Taille` qui représente la taille d'une chaîne et `m_Ptr` qui représente la position du premier élément de la chaîne. Pour la gestion mémoire nous utilisons trois fonctions essentielles :

- `FindPos` (`static` toujours pour les mêmes raisons) qui permet de rechercher le nombre de granules nécessaires pour l'insertion de la nouvelle chaîne. Cette fonction nous renvoie la position.

`FindPos` prend comme paramètre le nombre de granules. On parcourt le vecteur `VMapping` à la recherche de granules libres. A chaque fois qu'un granule est trouvé on incrémente un compteur. Si ce granule est le premier trouvé alors on sauvegarde sa position et on met le booléen `Premier` à `false` (le booléen `Premier` représente le premier granule). Sinon si le granule n'est pas libre, on réinitialise le compteur `Pos` (variable de sauvegarde de la position du premier granule) à 0 ainsi que `Premier` à `true`. Si à la fin du parcours on a toujours pas trouvé suffisamment de granules libres on double la taille du tableau. Après avoir longuement réfléchi sur une méthode pour réallouer juste ce qu'il fallait, on s'est dit qu'il était préférable de doubler la taille. Cela évite de réallouer trop souvent. Dans tous les cas on retourne la position `Pos`.

- `Realloc` (`static` toujours pour les mêmes raisons) qui permet de réallouer de l'espace mémoire si nécessaire.

`Realloc` prend comme paramètre la taille à rajouter. On rajoute à `Sz_NTCTS` la valeur `Increment * Granule`, pour `Sz_Mapping` on rajoute simplement `Increment`. On réalloue la taille du tableau `NTCTS` puis on redimensionne `VMapping`.

- `NbGranule` qui renvoie le nombre de granules nécessaires à rechercher pour une taille donnée. Par une fonction mathématique toute simple ; on renvoie la partie entière de  $\frac{1}{4} * Taille + 1$ .

On a implémenté plusieurs constructeurs : un par défaut, un par copie d'une `CString`, un par copie d'un `NTCTS` et un par copie d'une `string` puis un destructeur. Les algorithmes de ces constructeurs sont identiques, seule chose supplémentaire pour la copie d'un `NTCTS` c'est qu'il faut calculer la taille.

On a implémenté deux fonctions membres `ToUpper` et `ToLower`. Le nom de ces fonctions est suffisamment explicite pour expliquer leur fonctionnement. On a également implémenté `AfficheVect` et `AfficheTab` qui permet d'afficher le vecteur de mapping et le tableau de `CString`.

**On a implémenté plusieurs opérateurs :**

- `operator <<` **qui affiche caractère par caractère dans le flux.**

- `operator []` **qui renvoie l'élément qui lui est passé. Si l'élément demandé est supérieur à la taille on renvoie le dernier élément.**

- `operator =` **qui va libérer de l'espace mémoire dans le vecteur `VMapping` pour réserver la taille de la chaîne du deuxième opérande. Et enfin recopie caractère par caractère de la nouvelle chaîne.**

**P.S : On a essayé d'implémenter un `operator +` sans y parvenir (malheureusement).**

## **II) Buddy System**

**On a développé une classe `CString` dans laquelle nous avons déclaré en `static` la taille des granules `m_GranuleSize`, la taille du tableau de mapping `m_MapSize`, la taille du tableau de caractère `m_BaseSize`, le tableau de mapping `m_Base` et le tableau de mapping `m_Map` car il ne faut pas créer ces variables à chaque fois que l'on crée une nouvelle chaîne de caractères. Les autres données membres (`m_Indice` et `m_Taille`) ne sont pas en `static` car elles sont propres à chaque chaîne.**

**Dans les fonctions `private`, on a écrit `Assign` qui renvoie l'indice de la plage mémoire à allouer. Dans cette méthode, on calcule la taille de mapping nécessaire puis on fait appel à la fonction `NbBits` qui retourne le nombre de bits à allouer et alloue si nécessaire dans le tableau de mapping.**

**La fonction `Free` prend pour paramètres l'indice dans le tableau de caractères et la taille de la chaîne et met les bits correspondants à 0.**

**Nous avons défini plusieurs constructeurs: un par défaut, un autre par recopie d'une `NTCTS`, un par recopie d'une `CString` et un constructeur qui initialise la nouvelle chaîne avec un caractère (espace par défaut).**

**Cette classe permet 4 opérateurs:**

- `operator =` **qui effectue la copie d'une `CString`. Il libère tout d'abord l'espace nécessaire pour la copie, change l'indice du premier opérande, modifie la taille du premier opérande puis effectue la copie dans une boucle `for`.**

- `operator =` **qui effectue cette fois-ci l'affectation d'une `NTCTS`. Il calcule la taille de la `NTCTS` puis libère l'espace nécessaire, modifie l'indice et la taille avant d'effectuer la copie dans une boucle `for`.**

- `operator <<` **qui affiche caractère par caractère dans le flux la chaîne qui lui est passée en paramètre.**

- `operator >>` **qui crée une nouvelle chaîne (`NTCTS`) dans laquelle on saisie puis on fait appel à l'opérateur `=`. Avant la sortie il supprime la chaîne créée.**

- `operator []` **qui renvoie le caractère de la chaîne demandé. Renvoie le dernier élément si l'indice passé est supérieur à la taille.**

Deux fonctions membres : `ToUpper` et `ToLower` qui se passent de commentaires.

Deux fonctions d'affichage ont été écrites pour permettre la vérification et aussi pour déboguer le programme.

Dans le fichier `CString.h` on a implémenté une classe d'exception. Elle permet de capter les exceptions dues à l'allocation.

### III) Listes chaînées

Nous avons décidé d'implémenter trois classes `CString`, `CTrou`, et `CException`.

La classe `CString` ne gère pas la mémoire mais permet d'implémenter les fonctions utiles pour l'utilisateur. Dans cette classe on va définir la classe `CTrou` en friend. On a défini deux données membres `m_Taille` et `m_Indice` pour respectivement la taille de la chaîne et la position dans le tableau. On a défini `m_Trou` en static un pointeur vers un `CTrou` et un pointeur `m_Base` qui pointe vers le premier élément du tableau de NTCTS. Nous avons une donnée membre `protected` et static `m_BaseSize` qui représente la taille du tableau alloué en mémoire.

Nous utilisons plusieurs constructeurs : un par défaut, un par recopie de `CString` et un autre par recopie de `char *` puis un qui prend en paramètre la taille ainsi qu'un caractère pour le recopier sur toute la longueur de la chaîne. On a aussi défini un destructeur pour libérer de l'espace mémoire.

On a défini plusieurs fonctions membres `ToUpper` et `ToLower` (qui se passent de commentaires !!!!) ainsi que la fonction `Fill` qui remplit la chaîne d'un seul et même caractère.

On a défini quelques opérateurs :

- `operator +` qui crée une nouvelle `CString` qui a pour taille la taille des deux chaînes qu'on lui a passé et va recopier successivement la première au début de la nouvelle et la deuxième à la fin de la nouvelle.

- `operator []` qui fait la même chose que précédemment.

- `operator <<` qui écrit caractère par caractère dans le flux.

- `operator >>` qui va allouer une nouvelle chaîne `Chaine` dans laquelle on lira au clavier. Grâce à l'opérateur `=` on la recopiera dans la chaîne passée en paramètre de l'opérateur `>>`. Sans oublier de supprimer `Chaine`.

- Deux `operator =`, un qui prend pour paramètre un `char *` (il nous est utile pour l'opérateur précédent) et l'autre un `CString`. Ce dernier crée une nouvelle

`CString` dans laquelle va être recopié la première chaîne passée en paramètre puis la deuxième. La `CString` a pour taille la taille des deux chaînes passées en paramètre.

**Il nous reste à commenter deux fonctions `private` nommées `Assign` et `Free`.**

`Assign` capte les exceptions générées par `CTrou`. Si on a besoin de plus d'espace on augmente la taille du tableau de `NTCTS` et on réalloue (reste à faire un traitement si le tableau `NULL`). Si on est sur le dernier trou on supprime le suivant et on ne réalloue rien.

**La fonction `Free` permet de libérer la `Taille` en octets à la position `Indice`.**

Les données membres de la classe `CTrou` sont `m_Taille` (taille du trou), `m_Indice` (position du trou), `m_Suivant` (pointeur vers le trou suivant). Nous avons une fonction `private` `Allocate` qui prend la taille de la zone mémoire à allouer et retourne l'indice de cette zone.

Nous avons deux constructeurs : un par copie et l'autre qui prend un indice `Taille` et un pointeur vers le suivant (valeur par défaut : `NULL`) ainsi qu'un destructeur.

Dans les fonctions membres, nous avons une fonction `CheckNext` qui vérifie l'intégrité de la liste de trous. Si le trou suivant suit directement le trou courant alors on réalise la fusion de ces deux là.

La fonction `BeginAt` en `inline` (une seule ligne) retourne `true` si le trou est à l'indice passé en paramètre.

La fonction `AllocateFirst` alloue la taille demandée dans le premier trou qui convient. Cette fonction vérifie si la taille du trou sur lequel on est est supérieure à la taille demandée au quel cas on modifie les données membres en fonction sinon si la taille est la même on regarde si le suivant est différent de `NULL`, si oui on prend la place du suivant, sinon on lève une exception (on supprime le suivant sans réallouer). Dans le bloc `try` on fait un appel sur le suivant et dans le bloc `catch` on capte l'exception levée précédemment. Si par hasard le trou courant est le dernier trou de la zone mémoire alors on lève une nouvelle exception, sinon on lève une autre exception pour allouer de l'espace mémoire.

**La fonction `Free` permet de libérer la `Taille` en octets à la position `Indice`.**

Dans la classe `CException`, nous avons implémenté deux classes, la première `CExceptionNeedMoreSpace` va permettre la réallocation, la seconde `CExceptionLastTrou` teste si le dernier trou est la fin de la liste.

# **Listing**

# Mapping CString.h

```

/**
 *
 * @File : CString.h
 *
 * @Authors : V.Castille, N.Lichnowski, M.Lapeyre
 *
 * @Date : 13/02/2005
 *
 **/

#if ! defined __CSTRING_H__
#define __CSTRING_H__

#include <vector>

namespace nsSdD
{
    typedef std::vector <bool> CVBool;

    class CString
    {
    private :
        unsigned m_Taille;
        unsigned m_Ptr;

        static unsigned Sz_Mapping;
        static unsigned Sz_NTCTS;
        static CVBool VMapping;
        static char * Tab_NTCTS;

        //Procédure de calcul du nbre de granule nécessaire
        inline unsigned NbGranule (unsigned & Taille)
        {
            return (unsigned) floor (0.25 * Taille + 1);
        }

        static unsigned FindPos (unsigned Nb);
        static void Realloc (int Increment) throw (std::bad_alloc);

    public :

    ////////////////////////////////////Les Constructeurs////////////////////////////////////

        CString (void); // par défaut
        CString (CString & Cstr); // par recopie
        CString (const char * NTCTS); //par recopie d'une NTCTS
        CString (const std::string & S, unsigned Pos = 0,
            unsigned Nbre = std::string::npos); // string (+ pos +
nbre)

        ~CString (void); // Destructeur

    ////////////////////////////////////Fonctions Membres////////////////////////////////////

        inline unsigned size (void) const {return this->m_Taille;}
        // Accesseur taille

```

```

CString & ToUpper (void);           // Mise en majuscule
CString & ToLower (void);         // Mise en minuscule

    static void AfficheVect (void);
    static void AfficheTab (void);

////////////////////////////////////Les Opérateurs////////////////////////////////////

    char & operator [] (const unsigned);
    CString & operator = (CString &); // Affectation
    CString & operator + (const CString &); // Concanétation

    friend std::ostream & operator << (std::ostream & os, CString & Obj);
        // Injecteur

}; // CString

} // namespace nsSdD

#endif

```

# Mapping CString.cpp

```

/**
 *
 * @File : CString.cpp
 *
 * @Authors : V.Castille, N.Lichnowski, M.Lapeyre
 *
 * @Date : 13/02/2005
 *
 **/

#include <bitset>
#include <vector>
#include <string>
#include <iostream>
#include "CString.h"

#define CSTR nsSdD::CString

using namespace std;

namespace nsSdD
{
    //Possibilité de changer la taille du tableau de NTCTS
    unsigned const Granule = 4;
    unsigned CSTR::Sz_NTCTS = 28;
    unsigned CSTR::Sz_Mapping = Sz_NTCTS / Granule;

    //Initialisation des deux tables: mapping et stockage
    CVBool CSTR::VMapping (Sz_Mapping, false);

    char * CSTR::Tab_NTCTS = (char *) realloc ((void*) Tab_NTCTS,
                                                sizeof (char) * Sz_NTCTS);

    //Procédure de réallocation de tableau dynamique
    void CSTR::Realloc (int Increment) throw (std::bad_alloc)
    {
        Sz_NTCTS += Increment * 4;
        Sz_Mapping += Increment;

        Tab_NTCTS = (char *) realloc ((void*) Tab_NTCTS,
                                      sizeof (char) * Sz_NTCTS);

        //Agrandissement du tableau de Mapping, et remplissage avec des 0
        VMapping.resize (Sz_Mapping, false);
    } // Realloc()

    //Procédure de recherche de Nb granules libre successives
    unsigned CSTR::FindPos (unsigned Nb)
    {
        unsigned Cpt, Pos, j;
        Cpt = j = 0;
        bool Premier = true;

        for ( ; j < Sz_Mapping; ++j)
        {
            //Si on a trouvé le nombre de granule que l'on voulait on sort
            if (Cpt == Nb) break;

            //Si ce n'est pas un granule libre

```

```

    if (VMapping [j])
    {
        Cpt = Pos = 0;
        Premier = true;
    } // if

    //Si c'est un granule libre
    else
    {
        ++Cpt;
        //Si c'est le premier de la série on sauvegarde sa pos.
        if (Premier)
        {
            Pos = j;
            Premier = false;
        } // if
    } // else
} // for ()

//Si aucun granule n'est libre, réallocation du tableau
if (j == Sz_Mapping)
{
    Realloc (Sz_Mapping);
} // if

    return Pos;
} // FindPos

```

////////////////////////////////////Les Constructeurs////////////////////////////////////

```

//Par défaut
CSTR::CString () : m_Taille (0) {}

//Par recopie d'une CString
CSTR::CString (CString & CStr) : m_Taille (CStr.size ())
{
    //Calcul du nombre de granule nécessaire
    unsigned Nb_Granule = NbGranule (m_Taille);

    //Calcul d'une position libre dans le tableau de mapping
    unsigned PosMap = FindPos (Nb_Granule);

    //Correspondance dans le tableau de NTCTS
    m_Ptr = 4 * PosMap;

    //Interdiction de la réaffectation dans cette zone
    for (register unsigned i (0), j (PosMap); i < Nb_Granule; ++i)
        VMapping [j++] = true;

    //Rangement de la string dans le tableau
    for (unsigned i (0); i < m_Taille; ++i)
        *(Tab_NTCTS + m_Ptr + i) = CStr [i];
} // CString (CString)

//Par recopie d'une NTCTS
CSTR::CString (const char * NTCTS)
{
    m_Taille = 0;
    for (register unsigned i = 0; ; ++i)

```

```

    {
        if ((NTCTS) [i] == '\0') break;
        ++m_Taille;
    } //for

    //Calcul du nombre de granule nécessaire
    unsigned Nb_Granule = NbGranule (m_Taille);

    //Calcul d'une position libre dans le tableau de mapping
    unsigned PosMap = CString::FindPos (Nb_Granule);

    //Correspondance dans le tableau de NTCTS
    m_Ptr = 4 * PosMap;

    //Interdiction de la réaffectation dans cette zone
    for (register unsigned i (0), j (PosMap); i < Nb_Granule; ++i)
        VMapping [j++] = true;

    //Rangement de la string dans le tableau
    for (unsigned i (0); i < m_Taille; ++i)
        *(Tab_NTCTS + m_Ptr + i) = (NTCTS) [i];
} // CString (NTCTS)

//Par recopie d'une string
CString::CString (const std::string & S, unsigned Pos /* = 0 */,
                 unsigned Nbre /* = std::string::npos */)
    : m_Taille (S.size ())
{
    //Calcul du nombre de granule nécessaire
    unsigned Nb_Granule = NbGranule (m_Taille);

    //Calcul d'une position libre dans le tableau de mapping
    unsigned PosMap = CString::FindPos (Nb_Granule);

    //Correspondance dans le tableau de NTCTS
    m_Ptr = 4 * PosMap;

    //Interdiction de la réaffectation dans cette zone
    for (register unsigned i (0), j (PosMap); i < Nb_Granule; ++i)
        VMapping [j++] = true;

    //Rangement de la string dans le tableau
    for (unsigned i (0); i < m_Taille; ++i)
        *(Tab_NTCTS + m_Ptr + i) = S [i];
} // CString (string, Pos = 0, Nbre = npos)

//Destructeur
CString::~CString (void)
{
    unsigned Max = NbGranule (m_Taille);
    for (register unsigned i(m_Ptr / 4); i < Max + m_Ptr / 4; i++)
        VMapping [i] = false;
} // ~CString ()

////////////////////////////////////Les Fonctions membres////////////////////////////////////

CString & CString::ToUpper (void)

```

```

{
    for (register unsigned i = 0; i < this->m_Taille; ++i)
        if ((*this)[i] >= 'a' && (*this)[i] <= 'z')
            (*this)[i] = (*this)[i] - 32;
    return *this;
} //ToUpper

CString & CString::ToLower (void)
{
    for (register unsigned i = 0; i < this->m_Taille; ++i)
        if ((*this)[i] >= 'A' && (*this)[i] <= 'Z')
            (*this)[i] = (*this)[i] + 32;
    return *this;
} //ToLower

void CSTR::AfficheVect (void)
{
    for (register unsigned i (0); i < Sz_Mapping; ++i)
    {
        cout << VMapping [i] << "|";
    } // for
}

void CSTR::AfficheTab (void)
{
    for (register unsigned i (0); i < CString::Sz_NTCTS; ++i)
    {
        if ((i % 4) == 0)
            cout << endl;
        cout << Tab_NTCTS [i] << "|";
    } // for
}

////////////////////////////////////Les Opérateurs////////////////////////////////////

ostream & nsSdD::operator << (ostream & os, CString & Obj)
{
    for (register unsigned i (0); i < Obj.m_Taille; i++)
        os.put(*(Obj.Tab_NTCTS + Obj.m_Ptr + i));
    return os;
} // operator <<

char & CSTR::operator [] (const unsigned Nb)
{
    //Nb représente le l'indice de la chaine.
    return (Nb < m_Taille) ? *(Tab_NTCTS + m_Ptr + Nb)
        : *(Tab_NTCTS + m_Ptr +
m_Taille - 1);
} // operator []

CString & CSTR::operator = (CString & Obj)
{
    //Test obligatoire à cause du pointeur
    if (& Obj == this) return *this;

    //Libérer l'emplacement du tableau de NTCTS
    //Il faut mettre à 0 les granules correspondant dans le tableau
    //de mapping
    //Position dans le tableau de mapping

```

```

unsigned Pos = (Obj.m_Ptr / 4);

unsigned Taille = Obj.m_Taille;
for (unsigned i(0); i < NbGranule (Taille); ++i)
    VMapping [Pos++] = false;

Pos = (Obj.m_Ptr / 4);

//Réserve de l'emplacement
for (unsigned i(0); i < NbGranule (m_Taille); ++i)
    VMapping [Pos++] = true;

//Remplissage du tableau de NTCTS
for (unsigned i (0); i < m_Taille; ++i)
    *(Tab_NTCTS + m_Ptr + i) = Obj [i];

return *this;

} // operator =

/*CString & operator + (const CString & CStr)
{
    unsigned int Taille = m_Taille + CStr.m_Taille;
    unsigned NbreGranule = CString::NbGranule (Taille);
    Realloc (NbreGranule);
    for (register char * i = Tab_NTCTS ; i < Tab_NTCTS + m_Taille;
i++) {
        *(i + Tab_NTCTS + m_Ptr) = *(i + m_Ptr);
    }
    for (register char * i = Tab_NTCTS ; i < Taille; i++) {
        *(i + m_Taille + m_Ptr) = *(i + CStr.m_Ptr);
    }
    return *Tab_NTCTS;
} // operator +*/

#undef CSTR
} // namespace

```

# Mapping Exo\_01.cpp

```

/**
 *
 * @File : Exo_01.cpp
 *
 * @Authors : V.Castille, N.Lichnowski, M.Lapeyre
 *
 * @Date : 13/02/2005
 *
 * @Synopsis : Test de la classe CString
 *
 **/

#include <iostream>
#include <cctype>
#include "CString.h"

using namespace std;
using namespace nsSdD;

namespace
{
    void Exo_01 (void)
    {
        cout << "*****Test de la classe CString*****\n\n";

        string ok = "En";
        CString Str2 (ok);

        string toto= "supposant";
        CString Str3 (toto);

        string bjarne = "que";
        CString *Str4 = new CString (bjarne);

        string ik = "la";
        CString Str5 (ik);

        delete Str4;

        string oki = "ERRRR";
        CString Stra(ok);

        CString Stra0 ("bla");

        cout << "Taille de Str2 : " << Str2 << " = " << Str2.size() <<
endl;

        cout << "Vect : ";
        CString::AfficheVect ();
        cout << "\nTab : ";
        CString::AfficheTab ();
        cout << endl;

        cout << Str5 [1] << endl;
        cout << Str3 [2] << endl;

        Str3.ToUpper();
        cout << Str3 << " ";
        Str3.ToLower();
        cout << Str3 << endl;
    }
}

```

```

        //getch();
    } // Exo_01 ()
} // namespace

int main ()
{
    try
    {
        Exo_01 ();
        return 0;

    } // try

    catch (const exception & Exc)
    {
        cout << "\nException standard : " << Exc.what () << endl;

    } // catch ()

    catch (...)
    {
        cout << "\nException inconnue" << endl;

    } // catch ()

} // main ()

```

# Mapping Makefile\_01

```

#/**
#*
#* @File : Makefile_01
#*
#* @Authors : V.Castille N.Lichnowski M.Lapeyre
#*
#* @Date : 27/04/2005
#*
#* @Synopsis : Makefile de la classe CString
#*
#**/
nom = Exo_01
fic = CString
#
# Edition de liens
#
$(nom) : $(nom).o $(fic).o
        g++ -s -o $(nom) $(nom).o $(fic).o -Wall
#
# Compilation de $(fic).cxx en $(fic).o
#
$(fic).o : $(fic).cpp $(fic).h
        g++ -c $(fic).cpp -Wall
#
# Compilation de $(nom).cxx en $(nom).o
#
$(nom).o : $(nom).cpp $(fic).h
        g++ -c $(nom).cpp -Wall
#
#
.PHONY : clean
clean :
        clear;

```

# **Buddy System CString.h**

```

/*
 * @ File: CString.h
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 29/04/2005
 *
 **/

#ifndef __CSTRING_H__
#define __CSTRING_H__

#include <iostream>
using namespace std;

class CString
{
    // L'adresse relative du premier caractère de la chaîne.
    unsigned m_Indice;

    // La taille de la chaîne en octet.
    unsigned m_Taille;

    // La taille d'une granule en octet.
    static unsigned m_GranuleSize;

    // La taille du tableau de mapping en octet.
    static unsigned m_MapSize;

    // La taille du tableau de caractères en octet : dépendant de m_mapSize
    //et m_granuleSize.
    static unsigned m_BaseSize;

    // Le tableau de caractère.
    static char* m_Base;

    // Le tableau de mapping.
    static char* m_Map;

    // Retourne l'indice relatif de la plage mémoire de `Taille` octets
    //disponible et alloue cette plage.
    static unsigned Assign(unsigned Taille);

    // Libère `Taille` octets à l'indice relatif `Indice`.
    static void Free(unsigned Indice, unsigned Taille);

public:

    // Constructeurs

    // Construit une chaîne vide.
    CString();

    // Construit à partir d'une NTCTS classique.
    CString(const char * NTCTS);

    // Alloue et remplit la chaîne du caractère `car`.
    CString(const unsigned Length, char Car = ' ');

    // Constructeur par copie.

```

```

CString(const CString & CStr);

// Destructeur
~CString();

// Opérateurs

// Opérateur d'affectation d'une CString.
CString & operator = (const CString & CStr);

// Opérateur d'affectation d'une NTCTS classique.
CString & operator = (const char * NTCTS);

//Opérateur d'accès à un caractère dans une chaîne
char& operator [] (const unsigned Pos);

// Remplit la chaîne du caractère `car`.
void Fill(char Car);

// I/O
friend ostream & operator << (ostream & Cout, const CString & CStr);
friend void operator >> (istream & Cin, CString & CStr);

////////////////////////////////////fonctions membre////////////////////////////////////

    void ToUpper (void);    //Méthode de mise en majuscule d'une CString
    void ToLower (void);   //Méthode de mise en minuscule d'une CString

// Méthodes de débogage

// Méthode statique affichant le tableau de mapping avec `Mot` octets
//par ligne.
static void AfficherMap(unsigned mot = 4);

// Méthode statique affichant le tableau de caractère.
static void AfficherTab();
}; //CString

////////////////////////////////////

// Exception levée en cas d'erreur d'allocation de la mémoire.
class ExceptionReallocFailure
{
    // Information sur la taille qui n'a pas pu être allouée en octet.
    unsigned m_Size;
public :
    ExceptionReallocFailure(unsigned Size)
    {
        this->m_Size = Size;
    } //ExceptionReallocFailure

    unsigned GetSize()
    {
        return m_Size;
    } //GetSize()
}; //ExceptionReallocFailure

#endif /* __CSTRING_H__ */

```

# **Buddy System**

## **CString.cpp**

```

/*
 * @ File: CString.cpp
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 29/04/2005
 *
**/

#include "CString.h"
#include <stdlib.h>
#include <iomanip>

unsigned CString::m_GranuleSize = 4;
unsigned CString::m_MapSize = 8; // forcément une puissance de 2
unsigned CString::m_BaseSize = m_MapSize * 8 * m_GranuleSize;
char* CString::m_Map = (char*)calloc(sizeof(char), m_MapSize);
char* CString::m_Base = (char*)calloc(sizeof(char), m_BaseSize);

// Constructeurs

// Construit une chaine vide.
CString::CString()
{
    this->m_Taille = 0;
} // CString

// Alloue et remplit la chaine du caractère `car`.
CString::CString(const unsigned Length, char Car)
{
    this->m_Indice = this->Assign(Length);
    this->m_Taille = Length;
    this->Fill(Car);
} // CString

// Constructeur par copie.
CString::CString(const CString & CStr)
{
    this->m_Taille = 0;
    this->operator =(CStr);
} // CString

// Construit à partir d'une NTCTS classique.
CString::CString(const char * NTCTS)
{
    this->m_Taille = 0;
    this->operator =(NTCTS);
} // CString

// Destructeur : libère la place allouée.
CString::~CString()
{
    this->Free(m_Indice, m_Taille);
} // ~CString

// Opérateurs

// Opérateur d'affectation d'une CString.
CString & CString::operator = (const CString & CStr)
{
    this->Free(m_Indice, m_Taille);

```

```

    this->m_Indice = this->Assign(CStr.m_Taille);
    this->m_Taille = CStr.m_Taille;
    for (register char * i = m_Base ; i < m_Base + m_Taille; i++)
        *(i + m_Indice) = *(i + CStr.m_Indice);

    return *this;
} //operator = sur CString

// Opérateur d'affectation d'une NTCTS classique.
CString & CString::operator = (const char * NTCTS)
{
    register unsigned NTCTSSize = strlen(NTCTS);
    this->Free(m_Indice, m_Taille);
    this->m_Indice = this->Assign(NTCTSSize);
    this->m_Taille = NTCTSSize;
    register char* MyBase = (m_Indice + m_Base);
    for (register unsigned i = 0 ; i < NTCTSSize; i++)
        *(MyBase + i) = *(NTCTS + i);

    return *this;
} //operator = sur NTCTS

// Remplit la chaine du caractère `Car`.
void CString::Fill(char Car)
{
    for (register unsigned i = this->m_Indice;
         i < this->m_Indice + this->m_Taille; i++)
        *(m_Base + i) = Car;
} //Fill

//////////////////////////////////// BEGIN //////////////////////////////////////

// Le plus grand unsigned ayant une puissance de 2 immédiatement supérieure
//encodable sur un unsigned.
//volatile const unsigned maxUnsigned = 1 << ((sizeof(unsigned) << 4) - 1);
const unsigned MaxUnsigned = 0x80000000;

// Evite des calculs de puissance plus couteux...
//retourne le nombre de bit à allouer (ou liérer), donc la puissance de
deux
//immédiatement supérieure au nombre de granule.
//Convention d'appel C pour changer un peut...
unsigned NbBits(register unsigned NGranule)
{
    // Si le nombre de granule est nul ou s'il n'a pas de puissance de deux
    //correspondante codable sur un unsigned, on retourne 0.
    if (NGranule > MaxUnsigned)
        return 0;
    register unsigned i;
    for ( i = 1; i < NGranule; i = i << 1);
    return i;
} //NbBits

```

```

/*
 * A SIGNALER : le choix a été fait de représenter le tableau de mapping
 * avec
 * un tableau d'octet pour pouvoir travailler plus bas niveau, notamment en
 * ce
 * qui concerne la gestion des références pendant les parcours, et malgré
 * la
 * nouvelle contrainte de l'initialisation après une réallocation,
 * initialisation automatiquement faite par la classe vector.
 */

// Retourne l'indice de la plage mémoire disponible et alloue cette plage
unsigned CString::Assign(unsigned Taille)
{
    // Premièrement, on calcule le nombre de bits nécessaires pour réserver
    //taille octets en mémoire, donc ce nombre d'octet divisé par la taille
    //en
    //octet d'une granule. On en prend la partie entière supérieure pour ne
    //pas
    //allouer une granule de moins dans certains cas).
    unsigned TailleMap =
        (unsigned)floor((double)Taille/(double)m_GranuleSize);

    TailleMap += (unsigned)(Taille % m_GranuleSize) == 0 ? 0 : 1;

    // Ensuite, on calcule la puissance de deux immédiatement supérieure à
    //ce nombre, c'est le principe du buddy system ou algorithme des frères
    //siamois (on alloue par exemple 4 granules pour 3 granules utilisées.
    unsigned NbBit = NbBits(TailleMap);

    // Cette valeur servira à savoir si on a trouvé ou non un espace de
    // nbBit
    //bits. Elle représente en bits l'indice supérieur au nombre de bits du
    //tableau de mapping. Si cette valeur n'a pas variée à la fin des
    //boucles de
    //recherche, il faudra réallouer de la mémoire...
    unsigned IndiceMap = m_MapSize * 8;

    // L'algorithme prend en compte deux cas sensiblement différents :
    // - le cas où la recherche se fait au niveau du bit,
    // - et le cas où cette recherche se fait au niveau de
    // l'octet.
    // Ce teste sépare ces deux cas :

    if (NbBit < 8)
    {
        // Nous sommes ici dans la recherche d'un octet contenant une
        //séquence
        //de nbBits consécutifs, disposés suivant ce nombre de bits : 1 bit
        //peut se trouver n'importe où dans l'octet, 2 bits à quatre
        //positions
        //différentes et 4 bits à deux positions.

        // Ce masque sert à représenter les bits à rechercher.
        register char Mask = (1 << NbBit) - 1;
    }
}

```

```

// On parcourt tous les octets du tableau...
for (register unsigned int i = 0; i < m_MapSize; i++)
{
    // On récupère ici l'adresse de l'octet courant dans le
    //tableau de mapping.
    char * Octet = &(m_Map[i]);

    // Si cet octet ne comporte que des 1, ce n'est pas la peine de
    // s'y intéresser davantage : on passe à l'octet suivant.
    if ((unsigned char)*Octet == 0xFF)
        continue;

    // Deux octets à accès rapide pour les opération de recherche
    // qui suivent.
    register unsigned char IMask, Indice;

    // Littéralement : pour un masque placé initialement au niveau
    // des poids forts de l'octet (opération de décalage à gauche),
    // on décale le masque de nbBit bits à droite tant que ce
    // masque conserve le même nombre de bits à 1 consécutifs et
    // que ce masque et l'octet courant ont au moins un bit à 1 en
    // commun. L'indice (la position) du masque au niveau du bit
    // est conservée.
    for (Indice = 0,
        IMask = (unsigned char)Mask << (8 - NbBit);
        ((unsigned char)IMask >= (unsigned char)Mask) &&
        (((unsigned char)*Octet & (unsigned char)IMask) != 0);
        Indice++, IMask >>= NbBit);

    // Si le masque qui correspond est toujours supérieur ou égal
    // au masque initial, c'est à dire que l'on n'est pas sorti de
    // la boucle précédente à cause de la condition : "ce masque
    // conserve le même nombre de bits à 1 consécutifs", alors on a
    // trouvé un indice allouable.
    if ((unsigned char)IMask >= (unsigned char)Mask)
    {
        // On mémorise donc l'indice en bit dans le tableau de
        // mapping.

        IndiceMap = 8 * i + Indice * NbBit;

        // Et on alloue cette place.
        m_Map[i] |= IMask;

        // On sort du parcourt du tableau de mapping.
        break;
    } //if
} //for
} //if

```

```

else
{
    // Nous sommes ici dans le cas de la recherche de nbBit/8 octets
    // consécutifs à 0; on mémorise donc ce nombre d'octet (forcément
    // entier ici).
    unsigned NbOctet = NbBit >> 3;

    // i contiendra l'indice en octet dans le tableau.
    register unsigned i;

    // On parcourt le tableau octet par octet...
    for (i = 0; i < m_MapSize; i += NbOctet)
    {
        // Une autre variable pour un autre parcourt...
        register unsigned j;

        // ...depuis i et tant que l'octet correspondant est à 0 et que
        // l'indice représenté par j - i ne dépasse pas le nombre
        // d'octet voulu.
        for (j = i; (j < i + NbOctet) && (m_Map[j] == 0); j++);

        // Si on a trouvé nbOctet octets consécutifs à 0 :
        if (j == i + NbOctet)
        {
            // On alloue cette espace par un nouveau parcourt...
            for (j = 0; j < NbOctet; j++)
                // ... mettant tous les bits des octets à 1.
                m_Map[i + j] = (char)0xFF;

            // On signale que l'indice dans le mapping (en bit) est
            // celui de i (en octet); donc  $i * 8 \iff i \ll 3$ .
            IndiceMap = i << 3;

            // On sort alors du parcourt du tableau de mapping.
            break;
        } //if
    } //for
} //else

// Si indice_map est le même qu'initialement, on doit réallouer de la
// mémoire.
if (IndiceMap >= m_MapSize << 3)
{
    // Dans le cas de cet algorithme, la gestion de la mémoire est
    // faite de manière à toujours être fragmentable en deux parties de
    // tailles égales et étant des puissances de deux. Il faut donc
    // doubler la taille du tableau, quelle que soit la taille de la
    // zone recherchée.
    m_MapSize <<= 1;
    m_Map = (char*)realloc((void*)m_Map, m_MapSize * sizeof(char));
    m_BaseSize <<= 1;
    m_Base = (char*)realloc((void*)m_Base, m_BaseSize * sizeof(char));

    // S'il y a eu une erreur d'allocation...
    if (m_Base == NULL || m_Map == NULL)
    {
        // On lève une exception contenant le nombre d'octets pour
        // lesquels il y a eu un problème.
        throw new ExceptionReallocFailure(m_BaseSize * sizeof(char));
    } //if
}

```

```

// Puisque realloc n'initialise pas la zone et qu'il est important
// de dire que les nouvelles granules sont libres, on les libère "à
// la main" en partant de la moitié du tableau.
for (register unsigned i = m_MapSize >> 1; i < m_MapSize; i++)
    m_Map[i] = 0;

// Enfin, on relance une allocation récursive, même si cela peut
// prendre un temps relativement long pour l'allocation d'un petit
// nombre de granule, cet appel est nécessaire comme le montre
// l'exemple suivant :
// On a initialement un tableau de 16 granules toutes occupées. Si
// on demande 30 nouvelles granules, on a besoin de 32 bits
// consécutifs dans
// le tableau de mapping. Au premier parcourt, on ne trouve pas la
// place et on double la taille du tableau, qui fait alors 32 bits.
// On le parcourt une seconde fois, mais il n'y a que 16 bits
// consécutifs de libre, ce qui n'est pas suffisant. On aura donc
// besoin d'un troisième parcourt après allocation...

return Assign(Taille); // peut-être un peu long...
} //if

// Retourne l'indice correspondant à celui obtenu, donc on multiplie
// l'indice de granule par le nombre d'octet par granule.
return IndiceMap * CString::m_GranuleSize;
} //Assign

// Libère `taille` octets à l'indice `indice`
// (indice étant le déplacement relatif dans le tableau de caractères, donc
// en octet).
void CString::Free(unsigned Indice, unsigned Taille)
{
    // Si on ne libère rien, on n'a évidemment rien à faire...
    if (Taille == 0) return;

    // Premièrement, on calcule le nombre de bits qui ont été nécessaires
    // pour réserver taille octets en mémoire, donc ce nombre d'octet
    // divisé par la taille en octet d'une granule. On prend la partie
    // entière supérieure pour ne pas libérer une granule de moins dans
    // certains cas.
    unsigned TailleMap =
        (unsigned)floor((double)Taille/(double)m_GranuleSize);
    TailleMap += (unsigned)(Taille % m_GranuleSize) == 0 ? 0 : 1;

    // on calcule l'indice dans le tableau de mapping correspondant.
    unsigned IndiceMap = (Indice / m_GranuleSize);

    // Ensuite, on calcule la puissance de deux immédiatement supérieure à
    // ce nombre.
    unsigned NbBit = NbBits(TailleMap);

```

```

// Il y a là encore deux cas distincts :
if (NbBit < 8)
{
    // On calcule l'indice du premier bit dans l'octet ainsi que
    // l'indice de l'octet à modifier.
    IndiceMap = (IndiceMap % 8);

    // On calcule le masque correspondant au nombre de bit à libérer.
    char Mask = (1 << NbBit) - 1;

    // On applique enfin ce masque positionné correctement avec son
    // indice (l'opération (non et) s'applique très bien puisque
    // doivent être remis à 0 les bits qui sont à 1 dans l'octet comme
    // dans le masque.
    m_Map[(int)floor((double)IndiceMap / (double)8)] -=
        Mask << (8 - IndiceMap - NbBit);
} //if
else
{
    // On travaille ici au niveau de l'octet :
    // Il faut calculer l'indice en octet dans le tableau de mapping de
    // l'octet suivant la séquence, donc (indice_map + nbBit) * 8
    register unsigned LastGranule = (IndiceMap >> 3) + (NbBit >> 3);

    // On remet juste la séquence d'octet à 0.
    for (register unsigned i = IndiceMap >> 3; i < LastGranule; i++)
        m_Map[i] = 0;
} //else
} //Free

////////////////////////////////////// END ////////////////////////////////////////

// Opérateurs d'injection
ostream & operator << (ostream & Cout, const CString & CStr)
{
    for (register char * i = CStr.m_Base;
         i < CStr.m_Base + CStr.m_Taille; i++)
        Cout.put(*(i + CStr.m_Indice));
    return Cout;
} //operator <<

void operator >> (istream & Cin, CString & CStr)
{
    //A noter : le new et les malloc, calloc et realloc ne cohabitent pas
    // bien.
    char * Chaine = (char*)malloc(sizeof(char) * 1024);

    // On ne peut pas rentrer plus de 1024 caractères au clavier...
    Cin >> Chaine;
    CStr.operator =(Chaine);
    delete Chaine;
} //operator >>

char & CString::operator [] (const unsigned Pos)
{
    return (Pos < m_Taille) ? *(m_Base + m_Indice + Pos)
        : *(m_Base + m_Indice + m_Taille - 1);
} //operator []

```

```

// Fonction retournant le code binaire de car : pour l'affichage de debug.
char* ToBinary(const register unsigned char Car)
{
    static char Result[9] = "00000000";
    for (register unsigned short i = 0; i < 8; i++)
        Result[i] = ((Car & (0x01 << 7-i)) != 0) ? '1' : '0';
    return Result;
} //ToBinary

void CString::ToUpper (void)
{
    for (register unsigned i = 0; i < this->m_Taille; ++i)
        if ((*this) [i] >= 'a' && (*this) [i] <= 'z')
            (*this) [i] = (*this) [i] - 32;
} //ToUpper

void CString::ToLower (void)
{
    for (register unsigned i = 0; i < this->m_Taille; ++i)
        if ((*this) [i] >= 'A' && (*this) [i] <= 'Z')
            (*this) [i] = (*this) [i] + 32;
} //ToLower

void CString::AfficherMap(unsigned Mot)
{
    cout << right << setfill(' ') << "Contenu de m_map : mots de " <<
    dec << Mot << " Octets" << endl << " Indice | Contenu" << endl;
    for (register unsigned i = 0; i < m_MapSize; i += Mot)
    {
        cout << setw(7) << dec << i*8 << " | ";

        for (register unsigned j = 0; j < Mot; j++)
            cout << ToBinary((unsigned char)m_Map[i + j]) << " ";

        cout << endl;
    } //for
    cout << endl;
} //AfficheMap

void CString::AfficherTab()
{
    cout << setfill(' ') << right << "Contenu de m_base :" << endl <<
    " Granule |" << setw(m_GranuleSize*5 + 2) << " Hexa |" <<
    setw(m_GranuleSize*9 + 2) << " Binaire |" << " Char" << endl;
}

```

```

for (register unsigned i = 0; i < m_BaseSize; i += m_GranuleSize)
{
    cout << setfill(' ') <<
    setw(8) << right << showbase << hex << i/m_GranuleSize << " | ";
    for (register unsigned j = 0; j < m_GranuleSize; j++)
        cout << setw(4) << left << setfill('0') <<
            showbase << hex << (unsigned short)m_Base[i + j] << " ";
    cout << '|';
    for (register unsigned j = 0; j < m_GranuleSize; j++)
        cout << " " << ToBinary((unsigned char)m_Base[i + j]);
    cout << " | ";
    for (register unsigned j = 0; j < m_GranuleSize; j++)
        if (isprint(m_Base[i + j]) || ispunct(m_Base[i + j]))
            cout << (unsigned char)m_Base[i + j]; else cout << '.';
    cout << endl;
} //for

    cout << endl;
} //AfficheTab

```

# **Buddy System**

## **TestString.cpp**

```

/*
 * @ File: TestCString.cpp
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 29/04/2005
 *
**/

#include <iostream>
#include "CString.h"

using namespace std;

int main(int argc, char* argv[])
{
    CString g("gggg");
    CString::AfficherMap(2);
    CString::AfficherTab();
    CString h("\
                1122334455\
                667ghjfdhg\
                uityjjrhjr\
                gfyhsygdfy\
                rtytugjjgf\
                lkjhdgjdkh\
                kjhgjl1223\
                gfhhd34455");

    CString i("gg");
    CString::AfficherMap();
    CString j("gg");
    CString::AfficherMap();
    CString k("gghhhhhhhhhhhhhhhhhhh");
    CString::AfficherMap();
    h = "";
    CString::AfficherMap();
    CString l("gg");
    CString::AfficherMap();
    CString *r = new CString("512");
    cout << i;
    CString::AfficherMap();
    CString::AfficherTab();

    cout << "caractère en position 1 de la chaine: " << g << " : "
         << g[1] << endl;

    g.ToUpper();
    cout << "en majuscule: " << g << endl;

    g.ToLower();
    cout << "en minuscule: " << g << endl;

    return 0;
} //main

```

# **Buddy System Makefile**

```
##**
##*
##* @ File: Makefile
##*
##* @ Authors: V.Castille N.Lichnowski M.Lapeyre
##*
##* @ Date: 29/04/2005
##*
##**/
#
#
TestString: TestString.o CString.o
    g++-3.2 TestString.o CString.o -o TestString
#
TestString.o: TestString.cpp
    g++-3.2 -c TestString.cpp
#
CString.o: CString.cpp CString.h
    g++-3.2 -c CString.cpp
#
all: TestString
```

# Listes chaînée

## CString.h

```

/*
 * @ File: CString.h
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 10/04/2005
 *
**/

#ifndef __CSTRING_H__
#define __CSTRING_H__
#include "CTrou.h"
#include <string>
#include <iostream>

using namespace std;

class CString
{
private:
    // Permet à CTrou d'avoir accès à m_baseSize.
    friend class CTrou;

    unsigned m_Indice;
    unsigned m_Taille;

    static char * m_Base;
    static CTrou * m_Trous;

    /* Retourne l'indice de la plage mémoire disponible et alloue
       cette plage
       (modification des trous);
       première zone qui convient*/
    static unsigned Assign (unsigned Taille);

    // Libère `taille` octets à l'indice `indice`.
    static void Free (unsigned Indice, unsigned Taille);

protected:
    // La taille courante du tableau alloué en mémoire.
    static unsigned m_BaseSize;

public:
    // Constructeurs.
    CString ();
    CString (const char * String);
    CString (const unsigned Length, char Car = ' ');
    CString (const CString & CString);

    // Destructeur : Libère la place.
    ~CString ();

    //fonctions membres
    CString & ToUpper (void);
    CString & ToLower (void);

    // Opérateurs d'affectation.

```

```

CString & operator = (const CString & CString);
CString & operator = (const char * String);

CString & operator + (const CString & CString);

// remplit la chaine du caractère 'car'.
void Fill (char Car);

// si indice > à la taille on renvoie le dernier caractère.
char & operator [] (unsigned Indice);

// Opérateur d'impression.
friend ostream & operator << (ostream & Cout, const CString &
CString);

// Opérateur de saisie.
friend void operator >> (istream & Cin, CString & CString);
}; // classe CString

#endif /* __CSTRING_H__ */

```

# Listes chaînée

## CString.cpp

```

/*
 * @ File: CString.cpp
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 10/04/2005
 *
**/

//#define DEFAULT_LENGTH 100

#include "CString.h"
#include "CExceptions.h"
#include <stdlib.h>

#ifndef DEFAULT_LENGTH
#define DEFAULT_LENGTH 1024
#endif

unsigned CString::m_BaseSize = DEFAULT_LENGTH;
char * CString::m_Base = NULL;
CTrou * CString::m_Trous;

CString::CString()
{
    if (this->m_Base == NULL)
    {
        this->m_Base = (char*) malloc (this->m_BaseSize * sizeof (char));
        // On construit un trou de la Taille du tableau.
        this->m_Trous = new CTrou (0, this->m_BaseSize);
    }//if

    this->m_Taille = 0;
} //CString

CString::CString(const unsigned Length, char Car)
{
    if (this->m_Base == NULL)
    {
        this->m_Base = (char*) malloc (this->m_BaseSize * sizeof (char));
        // On construit un trou de la Taille du tableau.
        this->m_Trous = new CTrou (0, this->m_BaseSize);
    }//if

    this->m_Indice = this->Assign (Length);
    this->m_Taille = Length;
    this->Fill (Car);
} //CString

CString::CString (const CString & CString)
{
    if (this->m_Base == NULL)
    {
        this->m_Base = (char*) malloc (this->m_BaseSize * sizeof(char));
        // On construit un trou de la Taille du tableau.
        this->m_Trous = new CTrou(0, this->m_BaseSize);
    }//if

```

```

    this->m_Taille = 0;
    this->operator = (CString);
} //CString

CString::CString (const char * String)
{
    if (this->m_Base == NULL)
    {
        this->m_Base = (char*) malloc (this->m_BaseSize * sizeof (char));
        // On construit un trou de la Taille du tableau.
        this->m_Trous = new CTrou (0, this->m_BaseSize);
    } //if

    this->m_Taille = 0;
    this->operator = (String);
} //CString

CString::~CString()
{
    if (this->m_Taille != 0)
    {
        this->Free (m_Indice, m_Taille);
    } //if
} //~CString

CString & CString::ToUpper (void)
{
    for (register unsigned i = 0; i < this->m_Taille; ++i)
        if ((*this) [i] >= 'a' && (*this) [i] <= 'z')
            (*this) [i] = (*this) [i] - 32;

    return *this;
} //ToUpper

CString & CString::ToLower (void)
{
    for (register unsigned i = 0 ; i < this->m_Taille ; ++i)
        if ((*this) [i] >= 'A' && (*this) [i] <= 'Z')
            (*this) [i] = (*this) [i] + 32;

    return *this;
} //ToLower

CString & CString::operator = (const CString & CString)
{
    if (this->m_Taille < CString.m_Taille)
    {
        this->Free (m_Indice, m_Taille);
        this->m_Indice = this->Assign (CString.m_Taille);
    } //if
    else if (this->m_Taille > CString.m_Taille)
    {
        this->Free (this->m_Indice + CString.m_Taille,
                    this->m_Taille - CString.m_Taille);
    } //else if

    this->m_Taille = CString.m_Taille;
}

```

```

    for (register char * i = m_Base ; i < m_Base + m_Taille ; i++)
    {
        *(i + m_Indice) = *(i + CString.m_Indice);
    }//for

    return *this;
} //operator =

CString & CString::operator + (const CString & CStr)
{
    CString* NewCString = new CString (this->m_Taille + CStr.m_Taille);
    for (register char * i = m_Base ; i < m_Base + this->m_Taille ; i++)
        *(i + NewCString->m_Indice) = *(i + this->m_Indice);

    for (register char * i = m_Base ; i < m_Base + CStr.m_Taille ; i++)
        *(i + this->m_Taille + NewCString->m_Indice) = *(i +
CStr.m_Indice);

    return *NewCString;
} //operator +

CString & CString::operator = (const char * String)
{
    register unsigned LgString = strlen (String);
    if (this->m_Taille < LgString)
    {
        this->Free (m_Indice, m_Taille);
        this->m_Indice = this->Assign(LgString);
    } //if
    else if (this->m_Taille > LgString)
    {
        this->Free (this->m_Indice + LgString, this->m_Taille - LgString);
    } //else if

    this->m_Taille = LgString;
    register char* MyBase = (m_Indice + m_Base);

    for (register unsigned i = 0 ; i < LgString ; i++)
        *(MyBase + i) = *(String + i);

    return *this;
} //operator =

void CString::Fill (char Car)
{
    for (register unsigned i = this->m_Indice ;
        i < this->m_Indice + this->m_Taille ; i++)
        *(m_Base + i) = Car;
} //Fill

////////// private static ////////////////////////////////////////////

unsigned CString::Assign (unsigned Taille)
{
    // Le premier qui convient...

    unsigned Indice;

```

```

try
{
    if (m_Trous != NULL) return m_Trous->AllocateFirst(Taille);
    else Indice = m_BaseSize;
} //try
catch (ExceptionNeedMoreSpace *Exc)
{
    // TODO : Choisir de réallouer automatiquement...
    m_BaseSize = Exc->ReturnValue + Taille;
    m_Base = (char*) realloc ((void*) m_Base,
        m_BaseSize * sizeof (char));
    if (m_Base == NULL)
    {
        // TODO : Erreur de réallocation : nécessite peut-être
        // un tassement du tableau...
        cerr << "Problème de réallocation de la mémoire";
        throw m_Trous;
    } //if

    return Exc->ReturnValue;
} //catch

catch (ExceptionLastTrou *Exc)
{ // On supprime le suivant
  //(dernier trou restant)...
    Indice = m_Trous->GetIndice();
    delete m_Trous;
    m_Trous = NULL;
    if (!Exc->Realloc)
    { // on ne doit pas réallouer de mémoire...
        return Indice;
    } //if
} //catch

m_BaseSize = Indice + Taille;
m_Base = (char*) realloc ((void*) m_Base, m_BaseSize * sizeof (char));
if (m_Base == NULL)
{
    // TODO : Erreur de réallocation : nécessite peut-être
    // un tassement du tableau...
}

return Indice;
}

// Libère `Taille` octets à l'Indice `Indice`.
void CString::Free(unsigned Indice, unsigned Taille)
{
    // Si il y a un premier trou...
    if (m_Trous != NULL && Taille != 0)
    {
        // Si l'Indice est situé avant ce trou
        if (Indice < m_Trous->GetIndice())
        {
            m_Trous = new CTrou (Indice, Taille, m_Trous);
        } //if
        else m_Trous->Free (Indice, Taille);
    } //if
}

```

```

    else if (Taille != 0)
    {
        m_Trous = new CTrou (Indice, Taille);
    } //else if
} //Free

char & CString::operator [] (unsigned Indice)
{
    return (Indice < this->m_Taille) ? *(m_Base + m_Indice + Indice)
                                     : *(m_Base + m_Indice +
m_Taille - 1);
    // si Indice > à la Taille on renvoie le dernier Caractère.
} //operator []

ostream & operator << (ostream & Cout, const CString & CString)
{
    for (register char * i = CString.m_Base ;
         i < CString.m_Base + CString.m_Taille; i++)
        Cout.put (*(i + CString.m_Indice));

    return Cout;
} //operator <<

void operator >> (istream & Cin, CString & CString)
{
    // La Taille d'une chaine entrée au clavier ne doit pas dépasser
    // 1023 Caractères.
    char * Chaine = (char*) malloc (sizeof (char) * 1024);
    Cin >> Chaine;
    CString.operator = (Chaine);
    delete Chaine;
} //operator >>

```

# Listes chaînée

## CTrou.h

```

/*
 *
 * @ File: CTrou.h
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 20/04/2005
 *
 **/

#ifndef __CTROU_H__
#define __CTROU_H__

#define NULL 0

class CTrou
{

    /*! \var unsigned m_Taille
        \brief Contient la Taille du trou.
    */ unsigned m_Taille;

    /*! \var CTrou * m_Suivant
        \brief Contient l'indice (relatif à la base du tableau de
        \caractères) du trou.
    */ unsigned m_Indice;

    /*! \var CTrou * m_Suivant
        \brief Un pointeur sur le trou suivant.
    */ CTrou * m_Suivant;

    /*! \fonction unsigned Allocate (unsigned Taille)
        \param Taille la Taille à allouer
        \return l'indice de la zone mémoire allouée
    */ unsigned Allocate (unsigned Taille);

public:

    /*! \fonction CTrou(unsigned Indice, unsigned Taille,
                        CTrou * Suivant = NULL)
        \brief Construit un nouveau trou.
    */ CTrou (unsigned Indice, unsigned Taille,
             CTrou * Suivant = NULL);

    /*! \fonction CTrou(CTrou & cTrou)
        \brief Constructeur par recopie (nécessaire en cas de
        \passage en
        \paramètre autre que référence ou pointeur).
    */ CTrou (CTrou & cTrou);

    /*! \fonction ~CTrou()
        \brief Détruit le trou.
        \warn (ou warning...) Attention : Le trou suivant peut
        \être perdu
    */ ~CTrou ();

```

```

/*! \fonction void CheckNext()
    \brief Vérifie l'intégrité de la liste de trous : que le trou
    \suivant ne suit pas
    \directement celui-ci. Sinon, réalise la fusion des deux trous
    \juxtaposés...
*/ void CheckNext ();

/*! \fonction inline bool BeginAt(unsigned Indice,
    unsigned & Taille, CTrou * Suivant)
    \brief Retourne true si ce trou commence à l'indice donné en
    \paramètre.
    \paramètre Indice - l'indice à vérifier
    \return true si ce trou commence à l'indice donné en paramètre.
*/ inline bool BeginAt (unsigned Indice)
{
    return this->m_Indice == Indice;
} //BeginAt ()

/*! \fonction unsigned AllocateFirst(unsigned Taille)
    \brief Alloue une zone du tableau en la réservant dans
    \le premier trou
    \rencontré de Taille suffisante.
    \paramètre Taille - la Taille de la zone à allouer.
    \return l'indice de la zone allouée.
*/ unsigned AllocateFirst (unsigned Taille);

/*! \fonction void Free (unsigned Indice, unsigned Taille)
    \brief Modifie la liste de trous de manière à libérer `Taille`
    \octets à l'indice `Indice`.
    \paramètre Indice - l'indice de la zone à libérer.
    \paramètre Taille - la Taille de la zone à libérer.
*/ void Free (unsigned Indice, unsigned Taille);

// Accesseurs.
inline unsigned GetIndice () { return this->m_Indice; }

inline CTrou* GetSuivant () { return this->m_Suivant; }
inline unsigned GetTaille () { return this->m_Taille; }
}; //classe CTrou

#endif /* __CTROU_H__ */

```

# Listes chaînée

## CTrou.cpp

```

/*
 * @ File: CTrou.cpp
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 20/04/2005
 *
**/

#include "CTrou.h"
#include "CString.h"
#include "CExceptions.h"

// Construteurs
CTrou::CTrou (unsigned Indice, unsigned Taille, CTrou * Suivant)
{
    this->m_Taille = Taille;
    this->m_Indice = Indice;
    this->m_Suivant = Suivant;
    this->CheckNext();
} //CTrou

CTrou::CTrou (CTrou & cTrou)
{
    this->m_Taille = cTrou.m_Taille;
    this->m_Indice = cTrou.m_Indice;
    this->m_Suivant = cTrou.m_Suivant;
} //CTrou

// Destructeur
CTrou::~CTrou() {}

unsigned CTrou::Allocate (unsigned Taille)
{
    unsigned ReturnValue = this->m_Indice;
    this->m_Taille -= Taille;
    this->m_Indice += Taille;
    return ReturnValue;
} //Allocate ()

void CTrou::CheckNext ()
{
    if (this->m_Suivant != NULL)
    {
        if (this->m_Suivant->BeginAt (this->m_Indice + this->m_Taille))
        {
            this->m_Taille += this->m_Suivant->GetTaille();
            CTrou* NewSuivant = this->m_Suivant->GetSuivant();
            delete this->m_Suivant;
            this->m_Suivant = NewSuivant;
        } //if
    } //if
} //CheckNext ()

unsigned CTrou::AllocateFirst (unsigned Taille)
{
    unsigned ReturnValue;

```

```

if (this->m_Taille > Taille)
{
    ReturnValue = this->m_Indice;
    this->m_Taille -= Taille;
    this->m_Indice += Taille;
}
else if (this->m_Taille == Taille)
{
    // Si on est juste de la bonne Taille.
    ReturnValue = this->m_Indice;
    if (this->m_Suivant != NULL)
    {
        // on prend la place du suivant...
        this->m_Taille = this->m_Suivant->GetTaille();
        this->m_Indice = this->m_Suivant->GetIndice();
        CTrou* NewSuivant = this->m_Suivant->GetSuivant();
        delete this->m_Suivant;
        this->m_Suivant = NewSuivant;
    }
    else throw new ExceptionLastTrou (false);
    // on dit de me supprimer...
    // sans réallouer
}
else if (this->m_Suivant != NULL)
{
    try
    {
        return this->m_Suivant->AllocateFirst (Taille);
    }
    catch (ExceptionLastTrou *Exc)
    {
        // On supprime le suivant...
        ReturnValue = this->m_Suivant->GetIndice();
        delete this->m_Suivant;
        this->m_Suivant = NULL;
        if (Exc->Realloc)
        {
            // on doit réallouer de la mémoire...
            throw new ExceptionNeedMoreSpace(ReturnValue);
        }
    }
}
else if (this->m_Indice + this->m_Taille == CString::m_BaseSize)
{
    // Si on est le dernier trou et qu'il finit la zone mémoire...
    throw new ExceptionLastTrou (true);
}
else
    throw new ExceptionNeedMoreSpace (CString::m_BaseSize);
return ReturnValue;
} //AllocateFirst ()

void CTrou::Free (unsigned Indice, unsigned Taille)
{
    // Si il y a un trou suivant...
    if (this->m_Suivant != NULL)
    {
        // Si l'Indice est situé avant ce trou

```

```
if (Indice < this->m_Suivant->GetIndice())
{
    this->m_Suivant = new CTrou (Indice, Taille,
                                this->m_Suivant);
    // un CheckNext est effectué au moment de la création
    //(cas de la fusion de 3 trous).
    this->CheckNext();
} //if
else this->m_Suivant->Free (Indice, Taille);
} //if
else
{
    this->m_Suivant = new CTrou (Indice, Taille);
    this->CheckNext();
} //else
} //Free ()
```

# Listes chaînée

## CExceptions.h

```

/*
 * @ File : CExceptions.h
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 20/04/2005
 *
 **/

#ifndef __CEXCEPTIONS_H__
#define __CEXCEPTIONS_H__

class ExceptionNeedMoreSpace
{
    //contient le trou si réallocation plus grande... optimisation
    //CTrou* LastTrou;

    public:
        unsigned ReturnValue;

        ExceptionNeedMoreSpace(/*CTrou* cTrou,*/unsigned ReturnValue);
}; //ExceptionNeedMoreSpace

/*! \exception ExceptionLastTrou
    \brief Exception levée pendant le parcours des trous pour l'allocation
    \d'une nouvelle zone :
    \lit si le dernier trou correspond tout juste à la taille
*/
class ExceptionLastTrou
{
    public:
        bool Realloc;
        ExceptionLastTrou(bool Realloc)
        {
            this->Realloc = Realloc;
        };
};
#endif /* __CEXCEPTIONS_H__ */

```

# Listes chaînée

## CExceptions.cpp

```
/*
 *
 * @ File : CExceptions.cpp
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 20/04/2005
 *
 **/
```

```
#include "CTrou.h"
#include "CExceptions.h"
```

```
ExceptionNeedMoreSpace::ExceptionNeedMoreSpace
    (*CTrou* cTrou,*/unsigned ReturnValue)
{
    //this->LastTrou = cTrou;
    this->ReturnValue = ReturnValue;
}
```

# **Listes chaînée**

## **Exo\_01.cpp**

```

/*
 * @ File: Exo_01.cpp
 *
 * @ Authors: V.Castille N.Lichnowski M.Lapeyre
 *
 * @ Date: 15/04/2005
 *
**/

#include "CString.h"

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    CString g = "gG";
    CString *a = new CString(11,'1');

    CString *b = new CString(3,'2');

    CString *c = new CString(6,'3');

    CString *d = new CString(2,'4');

    CString *e = new CString(2,'5');

    cout << g << *a << *b << *c << *d << *e << endl;

    g.ToUpper ();

    cout << "en majuscules: " << g << endl;

    g.ToLower();

    cout << "en minuscules: " << g << endl;

    CString CStr;

    cin >> CStr;

    cout << CStr << endl;

    return 0;
} //main()

```

# **Listes chaînée**

## **Makefile\_01**

```

#/**
#*
#* @File : Makefile_01
#*
#* @Authors : V.Castille N.Lichnowski M.Lapeyre
#*
#* @Date : 13/02/2005
#*
#* @Synopsis : Makefile de la classe CString
#*
#**/
nom = Exo_01
#
# Edition de liens
#
$(nom) : $(nom).o CString.o CTrou.o CExceptions.o
        g++ -s -o $(nom) $(nom).o CString.o CTrou.o CExceptions.o -Wall
#
# Compilation de CString.cpp en CString.o
#
CString.o : CString.cpp CString.h
        g++ -c CString.cpp -Wall
#
# Compilation de CTrou.cpp en Ctrou.o
#
CTrou.o : CTrou.cpp CTrou.h
        g++ -c CTrou.cpp -Wall
#
# Compilation de CExceptions.cpp en CExceptions.o
#
CExceptions.o : CExceptions.cpp CExceptions.h
        g++ -c CExceptions.cpp -Wall
#
# Compilation de $(nom).cpp en $(nom).o
#
$(nom).o : $(nom).cpp CString.h
        g++ -c $(nom).cpp -Wall
#
#
.PHONY : clean
clean :
        clear;

```