```cpp
/**
 * @file cstring.cpp
 * @brief implementation of String methods
 * @date 07.03.2019.
 * @author Oussama Lagha
 * @author Max Caduff
 */

#include "cstring.h"
#include <cstring>


char *String::allocateMemory(const size_t nb) {
    size = nb;
    char *tmp = new char[nb + 1];
    tmp[nb] = '\0';
    return tmp;
}

String::String() {
    chain = allocateMemory(0);
}

String::String(const String &orig) : String(orig.chain) {}

String::String(const char *c) {
    chain = allocateMemory(strlen(c));
    strcpy(chain, c);
}

String::String(char c) {
    chain = allocateMemory(1);
    chain[0] = c;
}

String::String(bool b) {
    if (b) {
        chain = allocateMemory(4);
        sprintf(chain, "%s", "true");
    } else {
        chain = allocateMemory(5);
        sprintf(chain, "%s", "false");
    }
}

String::String(int integer) {

    size_t size = snprintf(chain, 0, "%d", integer);
    chain = allocateMemory(size);
    snprintf(chain, size + 1, "%d", integer);

}

String::String(double dbl) {

    size_t size = snprintf(chain, 0, "%f", dbl);
    chain = allocateMemory(size);
    snprintf(chain, size + 1, "%f", dbl);
}

String::~String() {
    delete[] chain;
```

```
}

size_t String::length() const {
    return size;
}

const char *String::toChar() const {
    return chain;
}

char &String::charAt(const size_t pos) const {
    return (*this)[pos];
}

bool String::equals(const char *c) const {
    return *this == c;
}

bool String::equals(const String &str) const {
    return *this == str;
}

String &String::assign(const char *c) {
    return *this = c;
}

String &String::assign(const String &str) {
    return *this = str;
}


String &String::append(const char *c) {
    return *this += c;
}

String &String::append(const String &str) {
    return *this += str;
}


String String::concat(const char *c) const {
    return *this + c;
}

String String::concat(const String &str) const {
    return *this + str;
}

String String::substring(int begin, size_t end) const {
    // we could also send an empty str begin out of bounds.
    if (begin > (int) size || begin < int(0 - size))
        throw std::out_of_range("begin out of bounds");

    if (0 == end || end > size)
        end = size;

    if (begin >= (int) end)
        throw std::invalid_argument("begin >= end");

    if (begin < 0)
        begin = (int) size + begin;
```

```
    size_t newLen = end - begin;
    char subString[newLen];
    for (int k = 0; k < newLen; k++)
        subString[k] = chain[k + begin];
    subString[newLen] = '\0';
    return String(subString);
}

char &String::operator[](const size_t pos) const {
    if (pos >= size)
        throw std::out_of_range("out of bounds");
    return *(chain + pos);
}


bool String::operator==(const char *c) const {
    return !strcmp(chain, c);
}

bool String::operator==(const String &str) const {
    return str == chain;
}

bool String::operator!=(const char *c) const {
    return !(*this == c);
}

bool String::operator!=(const String &str) const {
    return !(*this == str);
}

String &String::operator=(const char *c) {
    delete[] chain;
    chain = allocateMemory(strlen(c));
    chain = strcpy(chain, c);
    return *this;
}

String &String::operator=(const String &str) {
    return *this = str.chain;
}

String &String::operator+=(const char *c) {
    size_t oldSize = size;
    char *newChain = allocateMemory(oldSize + strlen(c));
    strcpy(newChain, chain);
    strcpy(newChain + oldSize, c);
    delete[] chain;
    chain = newChain;
    return *this;
}

String &String::operator+=(const String &str) {
    return *this += str.chain;
}

String String::operator+(const String &str) const {
    return *this + str.chain;
}

String String::operator+(const char *str) const {
    char tmpChar[size + strlen(str)];
```

```
    strcpy(tmpChar, chain);
    return String(strcat(tmpChar, str));

}

String operator+(const char *lhs, const String &rhs) {
    char tmpChar[rhs.size + strlen(lhs)];
    strcpy(tmpChar, lhs);
    return String(strcat(tmpChar, rhs.chain));
}

String operator+(const char c, const String &rhs) {
    char tmpChar[rhs.size + 1];
    tmpChar[0] = c;
    strcpy(tmpChar + 1, rhs.chain);
    return String(tmpChar);
}


std::ostream &operator<<(std::ostream &os, const String &str) {

//   we don't do: os << str.chain; in case of \0 in the chain
    for (size_t i = 0; i < str.size; i++)
        os << str[i];
    return os;
}

std::istream &operator>>(std::istream &is, String &str) {

    char tmp[1000];
    std::cin.getline(tmp, 1000);
    delete[] str.chain;
    str.chain = str.allocateMemory(std::min((int) strlen(tmp), 1000));
    strncpy(str.chain, tmp, 1000);

    return is;

}
```

```
/**
 * @file cstring.h
 * @brief header of String methods
 * @date 07.03.2019.
 * @author Oussama Lagha
 * @author Max Caduff
 */

#ifndef CSTRING_H
#define CSTRING_H

#include <cstdlib>
#include <cstdio>
#include <iostream>

class String {

public:
    /**
     * Constructeur sans arguments, créé juste un \0.
     */
    String();

    /**
     * Constructeur de String à partir d'une autre String. (copie)
     * @param orig la String à partir de laquelle construire cette String
     */
    String(const String &orig);

    /**
     * Constructeur de String à partir d'un tableau de char. (finissant par \0)
     * @param c le tableau de caractère à partir duquel construire cette String.
     */
    explicit String(const char *c);

    /**
     * Constructeur de String à partir d'un caractère.
     * @param c le caractère à partir duquel construire cette String.
     */
    explicit String(char c);

    /**
     * Constructeur de String à partir d'un entier.
     * Calcule la longueur de la chaîne de caractères pouvant contenir le
     * nombre i à partir de la valeur de retour de la fonction snprintf.
     * @param i le nombre entier à partir duquel construire cette String.
     */
    explicit String(int integer);

    /**
     * Constructeur de String à partir d'un double.
     * Calcule la longueur de la chaîne de caractères pouvant contenir le
     * nombre d à partir de la valeur de retour de la fonction snprintf.
     * @param d le double à partir duquel contstruire cette String.
     */
    explicit String(double d);

    /**
     * Constructeur de String à partir d'un booléen.
     * Construit une String contenant "true" si b est vrai, "false" sinon.
     * @param b le booléen à partir duquel construire cette String.
     */
```

```
explicit String(bool b);

/**
 * Destructeur
 */
~String();

/**
 * Renvoie la longueur du tableau de caractères String, sans le \0 final.
 * @return la longueur de cette String.
 */
size_t length() const;

/**
 * Renvoie une référence immuable vers la chaîne de caractères
 * contenue par cette String.
 * @return un pointeur sur la chaîne de caractères associée à cette String.
 */
const char *toChar() const;

/**
 * paraphrase de l'opérateur [].
 * @throw out_of_range si l'index n'est pas valide.
 * @param pos l'index du caractère.
 * @return une référence sur le caractère à l'index pos de cette String.
 */
char &charAt(const size_t pos) const;

/**
 * paraphrase de l'opérateur ==.
 * @param c la chaîne de caractères avec laquelle comparer cette String.
 * @return vrai si cette String est identique à s, faux sinon.
 */
bool equals(const char *c) const;

/**
 * paraphrase de l'opérateur ==.
 * @param str la String avec laquelle comparer cette String.
 * @return vrai si les deux String sont identiques, faux sinon.
 */
bool equals(const String &str) const;

/**
 * paraphrase de l'opérateur =
 * @param c le tableau de char à copier dans cette String, doit finir par \0
 * @return une référence sur cette String modifiée.
 */
String &assign(const char *c);

/**
 * paraphrase de l'opérateur =
 * @param str la String à copier dans cette String.
 * @return une référence sur cette String modifiée.
 */
String &assign(const String &str);


/**
 * paraphrase de l'opérateur +=
 * @param c le tableau de char à concaténer à cette String.
 * @return une référence sur cette String modifiée.
 */
```

```
String &append(const char *c);

/**
 * paraphrase de l'opérateur +=
 * @param str la String à concaténer à cette String.
 * @return une référence sur cette String modifiée.
 */
String &append(const String &str);

/**
 * paraphrase de l'opérateur +
 * @param str la String à ajouter.
 * @return la concaténation des deux éléments dans une nouvelle String.
 */
String concat(const String &str) const;

/**
 * paraphrase de l'opérateur +
 * @param str le tableau de char à ajouter. (doit finir par \0)
 * @return la concaténation des deux éléments dans une nouvelle String.
 */
String concat(const char *c) const;


/**
 * Renvoie une sous-chaîne de cette String comprise entre les indices begin
 * et end. begin peut être négatif, on part alors de la fin de la String. Si
 * end > size ou end == 0, alors end = size.
 * @throw out_of_range si begin est en dehors des bornes de la String:
 * (-size <= begin < size)
 * @throw invalid_argument si begin >= end
 * @param begin le début de la sous-chaîne de caractères. (begin inclus)
 * @param end la fin de la sous-chaîne de caractères (end exclu).
 * @return une nouvelle String composée de la sous-chaîne de caractères
 * comprise entre les indices begin et end de cette String.
 */
String substring(int begin, size_t end = 0) const;

/**
 * Renvoie une référence au n-ième caractère de cette String. [0; size-1]
 * @throw out_of_range si l'index n'est pas valide.
 * @param pos l'index du caractère.
 * @return une référence sur le caractère à l'index pos de cette String.
 */
char &operator[](const size_t pos) const;

/**
 * Compare cette String avec une chaîne de caractères.
 * @param c la chaîne de caractères avec laquelle comparer cette String.
 * @return vrai si cette String est identique à c, faux sinon.
 */
bool operator==(const char *c) const;

/**
 * Compare cette String avec une autre
 * @param str la String avec laquelle comparer cette String.
 * @return vrai si les deux String sont identiques, faux sinon.
 */
bool operator==(const String &str) const;

/**
 * Compare cette String avec une chaîne de caractères.
```

```
 * @param c la chaîne de caractères avec laquelle comparer cette String.
 * @return faux si cette String est identique à c, vrai sinon.
 */
bool operator!=(const char *c) const;

/**
 * Compare cette String avec une autre
 * @param str la String avec laquelle comparer cette String.
 * @return faux si les deux String sont identiques, vrai sinon.
 */
bool operator!=(const String &str) const;

/**
 * Surcharge de l'opérateur = pour permettre l'affectation à la String
 * @param c le tableau de char à copier dans cette String, doit finir par \0
 * @return une référence sur cette String.
 */
String &operator=(const char *c);

/**
 * Surcharge de l'opérateur = pour permettre l'affectation à la String
 * @param str la String à copier dans cette String.
 * @return une référence sur cette String modifiée.
 */
String &operator=(const String &str);


/**
 * Surcharge de l'opérateur += pour permettre la concaténation de cette
 * String avec un tableau de char. (qui doit finir par \0)
 * @param c le tableau de char à concaténer à cette String.
 * @return une référence sur cette String modifiée.
 */
String &operator+=(const char *c);

/**
 * Surcharge de l'opérateur += pour permettre la concaténation de cette
 * String avec une autre String.
 * @param str l'autre String à concaténer à cette String.
 * @return une référence sur cette String modifiée.
 */
String &operator+=(const String &str);

/**
 * Surcharge de l'opérateur + pour permettre la concaténation de cette
 * String avec une autre dans une nouvelle String.
 * @param str la String à ajouter.
 * @return la concaténation des deux éléments dans une nouvelle String.
 */
String operator+(const String &str) const;

/**
 * Surcharge de l'opérateur + pour permettre la concaténation de cette
 * String avec un tableau de char dans une nouvelle String.
 * @param str le tableau de char à ajouter. (doit finir par \0)
 * @return la concaténation des deux éléments dans une nouvelle String.
 */
String operator+(const char *str) const;


/**
 * Surcharge de l'opérateur + de char* pour permettre la concaténation d'un
```

```
     * char* avec une string.
     * @param lhs le char* de gauche.
     * @param rhs la string à ajouter.
     * @return la concaténation des deux éléments dans une nouvelle String.
     */
    friend String operator+(const char *lhs, const String &rhs);

    /**
     * Surcharge de l'opérateur + de char pour permettre la concaténation d'un
     * char avec une string.
     * @param lhs le char de gauche.
     * @param rhs la string à ajouter.
     * @return la concaténation des deux éléments dans une nouvelle String.
     */
    friend String operator+(const char c, const String &rhs);

    /**
     * Surcharge de l'opérateur << pour envoyer une string sur un flux.
     * @param os le stream sur lequel envoyer la string.
     * @param rhs la string à afficher.
     * @return une référence sur le flux pour permettre le chainage.
     */
    friend std::ostream &operator<<(std::ostream &os, const String &str);

    /**
     * Surcharge de l'opérateur >> pour lire une string depuis un flux.
     * @param is le stream depuis lequel lire la string.
     * @param rhs la string à populer avec la lecture.
     * @return une référence sur le flux pour permettre le chainage.
     */
    friend std::istream &operator>>(std::istream &is, String &str);

private:

    /**
     * alloue la mémoire pour les nouvelles strings, et stocke la taille.
     * @param nb le nombre de cases mémoire à réserver, '\0' non inclus.
     * @return un pointeur sur la zone réservée.
     */
    char *allocateMemory(const size_t nb);

    // représentation du texte sour forme de char*
    char *chain;
    // size keeps the length of the string, \0 not included.
    size_t size;

};

#endif /* CSTRING_H */
```

```
/**
 * @file main.cpp
 * @brief program entry point
 * @date 07.03.2019.
 * @author Oussama Lagha
 * @author Max Caduff
 */

#include "cstring.h"

using namespace std;

/*
 * programme de test
 */
int main(int argc, char **argv) {


    /* test 1 : constructeur par défaut (chaîne de caractères vide)*/
    String s1;
    cout << "test 1 : s1 : " << s1 << endl;

    /* test 2 : constructeur à partir d'une chaîne de caractères*/
    String s2("Strings !");
    cout << "test 2 : s2 : " << s2 << endl;

    /* test 3 : constructeur à partir d'une instance de la classe String*/
    String s3(s2);
    cout << "test 3 : s3 : " << s3 << endl;

    /* test 4 : constructeur à partir d'un caractère*/
    String s4('p');
    cout << "test 4 : s4 : " << s4 << endl;

    /* test 5 : constructeur à partir d'un entier*/
    String s5(98);
    cout << "test 5 : s5 : " << s5 << endl;

    /* test 6 : constructeur à partir d'un réél*/
    String s6(765.456);
    cout << "test 6 : s6 : " << s6 << endl;

    /* test 7 : constructeur à partir d'un booléen*/
    String s7(true);
    cout << "test 7 : s7 : " << s7 << endl;

    /* autres méthodes tests */

    /* test 8 : length test*/
    cout << "test 8 : s3 length = " << s3.length() << endl;

    /* test 9 : toChar test */
    printf("test 9 : s2: %s\n", s2.toChar());

    /* test 10 : modification du i-ème caractère*/
    s2.charAt(1) = 'p';
    cout << "test 10 : s2 : " << s2 << endl;

    /* test 10_1 : out_of_range test*/
    try {
        cout << "test 10.1 : " << s2.charAt(40) << endl;
    } catch (const std::out_of_range &e) {
```

```
    cout << e.what() << endl;
}

/*test 11 : récupération uniquement du i-ème caractère */
const String s9("Test");
cout << "test 11 : should be 'e' " << s9[1] << endl;

/* test 11_1 :out_of_range test*/
cout << "test 11.1 : ";
try {
    cout << s9[4] << endl;
} catch (const std::out_of_range &e) {
    cout << e.what() << endl;
}
/*true or false test*/
cout << boolalpha;

/* test 12 : egalité test*/
cout << "test 12 : true: " << s9.equals("Test") << endl;

/* test 13 : égalité test*/
String s10("Test");
cout << "test 13 : true: " << s9.equals(s10) << endl;

/* test 14 : assign  d'une chaîne de caractères*/
s1.assign("Strings Lab");
cout << "test 14 : s1: " << s1 << " new length: " << s1.length() << endl;

/* test 15 : assign d'un String*/
s4.assign(s1);
cout << "test 15 : s4: " << s4 << " new length: " << s4.length() << endl;

/* test 17 : append d'une chaîne de caractere*/
s10.append("er main. ");
cout << "test 17 : s10: " << s10 << endl;

/* test 18 :append d'une autre String*/
s10.append(s3);
cout << "test 18 : s10: " << s10 << endl;

/* test 19 : concat une chaine de caractères*/
cout << "test 19 : concat: " << s10.concat("Bonjour..") << endl;

/* test 20 : concat une autre String*/
cout << "test 20 : concat: " << s10.concat(s1) << endl;

/* test 21 : concat returns a new string*/
cout << "test 21 : s10 not modified: " << s10 << endl;

/* test 22 : substring test*/
cout << "test 22 substr. 'Strings !' : " << s10.substring(-9, 0) << endl;

/* test 22.1 : substring invalid_argument test*/
try {
    cout << "test 22.1 : begin > end : "
        << s10.substring(20, 10) << endl;
} catch (const std::invalid_argument &e) {
    cout << e.what() << endl;
}

/* test 22.2 : substring out_of_range test*/
try {
```

```
        cout << "test 22.2 : begin > string length: "
             << s10.substring(25, 0) << endl;
    } catch (const std::out_of_range &e) {
        cout << e.what() << endl;
    }

    /* test 23 : constructing a string from user input*/
    cout << "enter a string (spaces allowed):\n" ;
    cin >> s1;
    cout << "test 23 cin: " << s1 << endl;


    /*   TEST CONST    */

    const String s11 ("constant string");
    const String s12 ("other constant string");

    /* test 24 : length test*/
    cout << "test 24 : length = " << s11.length() << endl;

    /* test 25 : toChar test */
    printf("test 25 : s11: %s\n", s11.toChar());

    /* test 26 : affichage du 3ème caractère (n)*/
    cout << "test 26 : 3rd char (n): " << s11.charAt(2)  << endl;

    /* test 27 : egalité test*/
    cout << "test 27 : true " << s11.equals("constant string") << endl;

    /* test 28 : égalité + substring test*/
    cout << "test 28 : true " << s11.equals(s12.substring(6)) << endl;

    /* test 29 : assign d'un String const*/
    s4.assign(s11);
    cout << "test 29 : s4: " << s4 << " new length: " << s4.length() << endl;

    /* test 30 :append d'une autre String const*/
    s1.append(s11);
    cout << "test 30 : s1 = " << s1 << endl;

    /* test 31 : concat une chaine de caractères*/
    cout << "test 31 : concat: " << s11.concat("Bonjour") << endl;

    /* test 32 : concat une autre String*/
    cout << "test 32 : concat: " << s11.concat(s12) << endl;

    /* test 33 : inegalité test*/
    cout << "test 33 : true : " << (s11 != s12) << endl;



    return EXIT_SUCCESS;
}
```