הפקולטה למדעי המחשב הטכניון – מכון טכנולוגי לישראל

מבוא לתכנות מערכות

the String class

השלמה עבור התרגול של יניב חמו

Copyright © Yaniv Hamo, all rights reserved.

```
1 : #ifndef STRING1_H__
2 : #define STRING1_H__

3 : #include <iostream.h>

4 : class String {
5 :   int len;
6 :   char *val;

7 :   void copy(const char *);
8 :   void clear();

9 : public:
10:   String();
11:   String(const char *);
12:   String(const String &);
13:   ~String();

14:   String &operator=(const String &);

15:   int length() const { return len; }

16:   char operator[](int) const;
17:   char &operator[](int);

18:   String &operator+=(const String &);
19:   friend String operator+(const String &,
                              const String &);
20:   friend bool operator==(const String&,const String&);
21:   friend bool operator!=(const String&,const String&);
22:   friend bool operator< (const String&,const String&);
23:   friend bool operator> (const String&,const String&);
24:   friend bool operator<=(const String&,const String&);
25:   friend bool operator>=(const String&,const String&);

26:   friend ostream &operator<<(ostream &, const String&);
27: };
28: #endif
```

Copyright © Yaniv Hamo, all rights reserved.

זאת הגירסא הראשונה שנכתוב עבור String – מימוש מחרוזת ב-C++. היא מכילה מספר אי דיוקים וטעויות תיכון, אותם נפתור בגירסא הבאה.
זהו קובץ ה-header של המחלקה String. הוא ימצא בקובץ string.h.
כמו כל קובץ header גם פה יש את שורות 1, 2 ו-28 על מנת למנוע מצב שקובץ זה יכלל פעמיים.

שורה 3: אנו רוצים להשתמש במחלקה ostream על מנת להגדיר את אופרטור ההדפסה, ולכן חייבים לכלול את iostream.h.

שורה 4: מתחילים להגדיר את המחלקה String.

שורות 5-6: השדות הפנימיים הדרושים למימוש המחרוזת; מערך דינמי של תווים ואורך המחרוזת.

שורות 7-8: נגלה כי לרוב נרצה להגדיר מתודות copy ו-clear; copy מעתיקה ערך מסוים לאובייקט מהמחלקה אותה אנו כותבים, ו-()clear מנקה את הזיכרון שהקצה האובייקט. בעזרת מתודות אלו נוכל לממש הרבה מתודות בהמשך.

שורה 9: מציינת מעבר לאיזור הציבורי (המנשק) של המחלקה.

שורה 10: הגדרת constructor (c'tor) חסר ארגומנטים. c'tor כזה הינו הכרחי כדי ששורות הקוד הבאות תפעלנה:

```
String s;
String arr[10];
```

בעת יצירת מערך של אובייקטים, תמיד נקרא ה-c'tor חסר הארגומנטים עבור כל אחד מאלמנטי המערך.
אם c'tor כזה לא קיים (כלומר קיימים רק c'tors עם ארגומנטים) לא יהיה ניתן להגדיר מערך של אובייקטים מהמחלקה (לא יעבור קומפילציה).

שורה 11: כדי שניתן יהיה ליצור אובייקט עם ערך התחלתי שהוא מחרוזת:

```
String s = "abc";
String s("abc");
```

שתי שורות הקוד הללו זהות ומשתמשות ב-c'tor שהוגדר בשורה 11. שימו לב לסימן ה- = בשורת הקוד הראשונה; זהו לא =operator, אלא קריאה ל-c'tor, משום שמגדירים אובייקט חדש.
שימו לב שהגדרנו את הפרמטר בתור * const char על מנת להצהיר שאיננו מתכוונים לשנותו.

שורה 12: כדי שניתן יהיה לכתוב:

```
String s;
String s2 = s;
```

השורה הראשונה קוראת ל-c'tor חסר הארגומנטים, והשורה השנייה ל-c'tor משורה 12.
c'tor זה נקרא גם copy constructor משום שהוא מעתיק אובייקט אחד במחלקה לאובייקט אחר. חייבים לממש אותו במידה והעתקה אינה טריוויאלית (לא מספיק רק להעתיק שדה-שדה) – כלומר אם ישנם מצביעים והקצאת זיכרון דינמית, כמו במקרה שלנו. הוא יקרא גם במקרים הבאים:

```
String func(String s)
{
  return s;  // copy constructor is called
}

String s;
func(s);   // copy constructor is called
```

Copyright © Yaniv Hamo, all rights reserved.

כלומר בכל פעם שיש צורך בהעתקת ערך. שימו לב שבהעברת reference או החזרת reference הוא לא נקרא, משום שלא מועתק דבר.
שוב, אנו מעתיקים מהאובייקט הנתון כפרמטר, לא משנים אותו, ולכן ה-const.

שורה 13: הגדרת ה-destructor (d'tor) – הכרחית במקרה שלנו מכיוון שיש שימוש בהקצאת זיכרון דינמית, שאותו נצטרך לשחרר עם הריסת האובייקט.

שורה 14: על מנת שהשורה הבאה תפעל בצורה נכונה:

```
String s1, s2("abc")
s1 = s2
```

שימו לב להבדל בין =operator לבין copy c'tor; =operator נקרא על אובייקט קיים, ועליו לשחרר את הזיכרון המוקצה כבר, ולהקצות חדש עם ערך זהה לאובייקט המועתק. copy c'tor נקרא רק כשיוצרים אובייקט חדש, ולכן הוא לא צריך לדאוג לשחרור זיכרון כלשהו.
=operator מחזיר reference לאובייקט עליו הוא פועל, על מנת לאפשר:

```
(a = b) = c
```

שהוא ביטוי חוקי ומשמעותו הצבת b לתוך a, ולאחר מכן הצבת c לתוך a. אם לא היינו מחזירים reference, היה חוזר העתק (שהיה נוצר ע"י ה-copy c'tor) זמני שאליו לא ניתן להציב דבר.

שורה 15: getter method – מתודה אשר מחזירה את ערכו של שדה פנימי, במקרה זה של אורך המחרוזת. מכיוון שמתודות כאלה לרוב מסתכמות במשפט return יחיד, נהוג להגדירן כ-inline (כלומר המימוש כבר בגוף המחלקה).
שימו לב ל-const לאחר הסוגריים בחתימת הפונקציה. const זה מצהיר כי מתודה זו לא משנה את האובייקט עליו היא פועלת. מאוד חשוב להקפיד ולהצהיר בצורה זו על כל מתודה שאינה משנה את ערך האובייקט, על מנת לתת פונקציונליות גדולה ככל האפשר לאובייקטים קבועים של מחלקה זו. לדוגמא:

```
void fund(const String &s)
{
  int len = s.length();
}
```

מכיוון ש-s הוא const, לא היינו יכולים להפעיל עליו את המתודה ()length (לא היה עובר קומפילציה), אלמלא היינו מבטיחים לקומפיילר שהיא לא תשנה את s.

שורות 16-17: אנו רוצים לאפשר גישה נוחה לתווים בתוך המחרוזת, לקריאה ולכתיבה. שורה 17 מאפשרת לכתוב:

```
String s = "abcde";
cout << s[2];
s[3] = 'x';
```

כלומר קריאה של התו באינדקס 2, וכתיבה לתו באינדקס 3.
ה-int שמתקבל כפרמטר הינו מה שבא בין הסוגריים המרובעות. היינו יכולים להגדיר כל טיפוס כבא בין הסוגריים המרובעות, לדוגמא:

```
char &operator[](double k)
```

ומתודה זו הייתה נקראת אם היו כותבים:

```
String s;
s[2.4] = 'd'; // meaningless example
```

Copyright © Yaniv Hamo, all rights reserved.

אנו גם רוצים לאפשר גישת קריאה בלבד עבור אובייקטים קבועים:

```
void func(const String &s)
{
  cout << s[2];
}
```

אך לא רוצים לאפשר שינוי של תווים באובייקטים כאלו, ולכן operator[] const לא מחזירה reference.
יותר מדויק לרשום ש-const [] operator מחזירה const char &, ואת זה נתקן בגירסא 2.
הקומפיילר יקרא ל-const []operator על אובייקטים קבועים, ול-[]operator עבור שאר האובייקטים.

שורה 18: חיבור מחרוזת לנוכחית. =+operator מחזיר reference לאובייקט עליו הוא פועל, על מנת לתמוך ב-

```
(a += b) += c
```

שהוא ביטוי חוקי ומשמעותו היא להוסיף את b ל-a ולאחר מכן להוסיף את c ל-a. שוב, אם לא היינו מחזירים reference, פעולת ה- =+ השנייה הייתה מתבצעת על אובייקט זמני.

שורה 19: חיבור מחרוזות. הסיבה שאופרטור זה מוגדר כפונקציה ולא מתודה (לא חלק מהמחלקה), היא שאנו רוצים לתמוך בשתי השורות הבאות:

```
s = s1 + "abc";
s = "abc" + s1;
```

אם היינו מגדירים אופרטור זה בתור מתודה, הוא היה נקרא רק במקרה הראשון, כי רק אז האובייקט מצד שמאל, עליו מתבצעת הקריאה, הוא מסוג String. כדי שהוא יקרא גם במקרה השני, בשביל סימטריות, אנו חייבים להגדיר אותו כפונקציה מחוץ למחלקה, ופונקציה כזו כבר מקבלת שני פרמטרים; אובייקט שמאלי ואובייקט ימני, ונוכל לטפל בשני המקרים.
שימו לב שלמרות שהגדרנו אופרטור חיבור המקבל שני אובייקטים מסוג String, הוא יפעל בכל המקרים הבאים:

```
String s1, s2;
s1 + "abc";
"abc" + s1;
s1 + s2;
```

בשני המקרים הראשונים, בהם הפרמטר הוא char *, הקומפיילר יצור אובייקט זמני מסוג String, מאותו char *, ע"י קריאה ל-c'tor משורה 11, ואז יפעיל את אופרטור החיבור על שתי ה-Strings.
הגדרנו את האופרטור כ-friend משום ואנו חושבים שהוא יזדקק לגישה לשדות הפנימיים לצורך מימושו. מאוחר יותר נגלה שטעינו, ונתקן זאת בגירסא 2.

שורות 20-25: אופרטורי השוואה. מוגדרים כפונקציות מטעמי סימטריות כמו operator+. הן פעולות בוליאניות שהתשובה עליהן היא כן או לא, ולכן מוגדרות כמחזירות bool. על מנת להשוות נצטרך לגשת לשדות הפנימיים, ולכן הן מוגדרות friend. בגירסא הבא נראה כיצד לממש אותן ללא שימוש ב-friend.

Copyright © Yaniv Hamo, all rights reserved.

שורה 26: אופרטור הדפסה. נועד לתמוך במקרה הבא:

```
String s;
cout << s;
```

אופרטור זה אינו יכול להיות ממומש כמתודה, מכיוון שמתודות מופעלות על האובייקט משמאל, ובמקרה זה זוהי מחלקה שאיננו יכולים לשנות. לכן חייבים להגדיר אותו כפונקציה. הוא מקבל מצד שמאל אובייקט ממחלקת ostream, ומצד ימין אובייקט שלנו, String. הוא מחזיר &ostream, שזהו אותו זרם הפלט שקיבל כפרמטר, על מנת לתמוך בשרשור, לדוגמא:

```
String s;
int x=5;
cout << s << x;
```

שני הערכים הולכים לזרם cout.

שאלות ותשובות:

- *למה אין ערך חזרה ב-c'tors שורות 12-10 ?*

משום ש-c'tors אינם מחזירים אף ערך. אין משמעות לערך חזרה מביטוי כמו:

```
String s;
```

- *אבל c'tors מקצים זיכרון, וזה עלול להיכשל, אז איך נדווח על שגיאה?*

דיווחי שגיאות ב-C++ לא מבצעות ע"י ערכי חזרה, אלא ע"י מנגנון שנקרא exceptions, אותו לא נלמד בתרגול. למתעניינים, דוגמא לתפיסת שגיאה בהקצאת זיכרון:

```
#include <stdexcept>
#include <iostream>

using std::bad_alloc;
using std::cerr;

int main()
{
  int *a;
  try {
    a = new int[1000000];
  }
  catch(bad_alloc &exp) {
    cerr << "Not enough memory";
  }
  return 0;
}
```

- *למה את operator+= הגדרת כמתודה ולא כפונקציה?*

משום שבכל שימוש רצוי בו, יהיה אובייקט של String מצד שמאל. לא נרצה שמישהו יכתוב

```
"abc" += "def";
```

- *למה operator+ מחזיר String ולא &String ?*

מכיוון ואיננו רוצים לתמוך בשורה חסרת משמעות כזו:

```
(a+b) = c;
```

יותר נכון להגדירו כמחזיר const String.

Copyright © Yaniv Hamo, all rights reserved.

```
1 : #include <cstring.h>
2 : #include <cstdlib.h>
3 : #include "string.h"

4 : void String::copy(const char *s)
5 : {
6 :   if(s) {
7 :     len = strlen(s);
8 :     val = new char [len+1];
9 :     strcpy(val, s);
10:   }
11:   else {
12:     len = 0;
13:     val = NULL;
14:   }
15: }

16: void String::clear()
17: {
18:   delete[] val;
19: }


20: String::String()
21: {
22:   copy(NULL);
23: }


24: String::String(const char *s)
25: {
26:   copy(s);
27: }

28: String::String(const String &str)
29: {
30:   copy(str.val);
31: }


32: String::~String()
33: {
34:   clear();
35: }
```

Copyright © Yaniv Hamo, all rights reserved.

```
36: String &String::operator=(const String &str)
37: {
38:   if(&str == this) return(*this);
39:   clear();
40:   copy(str.val);
41:   return *this;
42: }


43: char String::operator[](int i) const
44: {
45:   if(i<0 || i>=len) exit(1);
46:   return val[i];
47: }


48: char &String::operator[](int i)
49: {
50:   if(i<0 || i>=len) exit(1);
51:   return val[i];
52: }


53: String &String::operator+=(const String &str)
54: {
55:   if(&str == this) return *this;
56:   if(len == 0) copy(str.val);
57:   else {
58:     char *v = new char [len + str.len + 1];
59:     strcpy(v, val);
60:     strcat(v, str.val);
61:     clear();
62:     copy(v);
63:     delete [] v;
64:   }
65:   return *this;
66: }

67: String operator+(const String &str1, const String &str2)
68: {
69:   String s = str1;
70:   s += str2;
71:   return s;
72: }
```

Copyright © Yaniv Hamo, all rights reserved.

```
73:bool operator==(const String &str1,const String &str2)
74:{
75:  return !strcmp(str1.val, str2.val);
76:}


77:bool operator<(const String &str1,const String &str2)
78:{
79:  return strcmp(str1.val, str2.val) < 0);
80:}


81:bool operator!=(const String &str1,const String &str2)
82:{
83:  return !(str1 == str2);
84:}


85:bool operator<=(const String &str1,const String &str2)
86:{
87:  return (str1<str2) || (str1==str2);
88:}


89:bool operator>(const String &str1,const String &str2)
90:{
91:  return !(str1 <= str2);
92:}


93:bool operator>=(const String &str1,const String &str2)
94:{
95:  return !(str1 < str2);
96:}


97 :ostream &operator<<(ostream &out, const String &str)
98 :{
99 :  if(str.val) out << str.val;
100:  return out;
101:}
```

Copyright © Yaniv Hamo, all rights reserved.

שורה 1: ב-C++ כוללים את cstring.h ולא את string.h בשביל הפעולות strcmp, strlen, וכו'

שורה 2: בקובץ זה מוגדר הקבוע NULL (ב-C++ הוא הקבוע 0).

שורה 3: הגדרת המחלקה אותה אנו מממשים.

שורה 4: פונקציית ה-copy שמטרתה העתקת ערך לאובייקט שלנו, בהנחה והוא כרגע לא מאותחל. מאתחלים את שני השדות, len ו-val. בגירסא הבאה נשנה זאת כשנעבור לשימוש ב-init list.

שורה 16: פונקציית ה-clear, שנועדה למחוק שדות של אובייקט קיים.

שורות 20,24,28: מימוש כל ה-c'tors ע"י קריאה פשוטה לפונקצייה copy. שימו לב לשורה 30, בה נשלח השדה val, של אובייקט ה-String הנתון. מכיוון שה-c'tor הינו מתודה, אפשרי מתוכו לגשת לחלק הפרטי של כל אובייקט מהמחלקה.

שורה 32: מימוש ה-d'tor ע"י קריאה פשוטה ל-clear.

שורה 36: אופרטור ההשמה. אופרטור זה משמש לדריסת ערך אובייקט קיים בערך חדש, ולכן מטבעו יצטרך לעשות clear ואז copy. ואת זאת אנו רואים בשורות 39 ו-40.

שורה 38: תהייה בכל אופרטור השמה. שורה זו בודקת שהכתובת של הפרמטר אותו אנו מקבלים אינה במקרה הכתובת שלנו, כלומר שאיזו נשמה טובה לא כתב במקרה:

```
String s;
s = s;
```

זוהי בעיה, מכיוון שכפי שהזכרנו, אופרטור זה טבעו לשחרר את הזיכרון ואז להעתיק ערך חדש. במקרה של הצבה עצמית, שחרור הזיכרון ישחרר גם את הערך החדש, מכיוון ומדובר באותו הזיכרון. לכן חובה לבדוק שאין הצבה עצמית.

שורה 41: אנו רוצים להחזיר reference לאובייקט הנוכחי מסיבות שהזכרנו קודם, ולכן אנו מחזירים את האובייקט הנוכחי, שהוא this* (this הוא רק מצביע, לא האובייקט, ולכן מוסיפים *).

שורות 43, 48: מימוש []operator עבור קבועים ועבור לא-קבועים. שימו לב ל-exit בשורות 45 ו-50 במקרה של חריגה מגבולות המערך. זה כמובן לא נכון לחלוטין, אבל בהיעדר שימוש ב-exceptions, אין לנו משהו טוב יותר לעשות.

שורה 53: =+operator. בדיקת שרשור עצמי, בשורה 55, הינה טעות במקרה זה. אין סיבה שלא לאפשר a+=a (שמבצע a=a+a). נתקן זאת בגירסא 2.

שורה 67: +operator. נשים לב שבמימוש השתמשנו ב-=+operator, שהוא מתודה ציבורית, ולא בשום שדה פרטי במחלקה. לכן היה אפשר לוותר על ה-friend עבור אופרטור זה בקובץ ה-header.

שורות 73,77,81,85,89,93: אופרטורי השוואה. שימו לב שמספיק לממש שניים (== ו- < במקרה זה) על מנת להגדיר את כולם. מכיוון שניגשים לשדה הפנימי val בשורות 75, 79, אופרטורים אלו מוגדרים כ-friend, אבל בגירסא 2 נראה איך לממש אותם ללא friend.

Copyright © Yaniv Hamo, all rights reserved.

שורה 97: אופרטור הדפסה. שימו לב שההדפסה מתבצעת לזרם הנתון ב-out, ולא לזרם ספציפי כגון cout. שורה 100 מחזירה את הזרם הנתון כדי לאפשר שרשור. כיוון שיש גישה לשדות פנימיים, גם פונקצייה זאת היא friend, ושוב בגירסא 2 נפתור זאת כי זה מציק לנו.

Copyright © Yaniv Hamo, all rights reserved.

מעבר לגירסא 2

בגירסא 1 של String היו מספר נקודות הטעונות שיפור:

- לא השתמשנו ברשימת אתחול (init list) ב-c'tors
  תמיד נשאף להשתמש כמה שיותר ברשימת אתחול, ולא בקוד, ב-c'tors שנכתוב. כדי להבין את הסיבה נסתכל בקטע הקוד הבא:

```
class SomeClass1 {
  int x;
public:
  SomeClass1() { x = 5; }
};

class SomeClass2 {
  int x;
public:
  SomeClass2() : x(5) {}
};
```

בעת יצירת אובייקט של SomeClass1, נוצר ה-int, ומאותחל לזבל כלשהו. אז נקרא ה-c'tor, ומציב בו 5. כלומר הייתה הצבה כפולה לתוך x. במקרה של SomeClass2 נעשה שימוש ברשימת אתחול. במקרה כזה ה-int נוצר ומייד מאותחל ב-5; חסכנו פעולה.

- יותר מדי פונקציות חברות.
  מחלקה צריכה להיות כמה שיותר סוציומטית וחסרת חברים, מכיוון שכל פונקציה חברה כזו מאפשרת גישה ישירות לאיברים הפנימיים של המחלקה, תוך עקיפת כל המנשק המסודר, הכולל איזורים פרטיים וציבוריים. ככל שיש יותר פונקציות חברות כך הקוד פחות קריא מכיוון ואנו כבר לא יכולים להבטיח שכל שינוי ערך שדה באובייקט, נמצא תחת בקרה.

- השתמשנו בצורה הישנה `#include <iostream.h>` על מנת לכלול הגדרות ספרייה סטנדרטית. בצורה זו אנו מכניסים את כל מה שמוגדר שם אל מרחב השמות (namespace) שלנו. בגירסא 2 נשתמש בצורה החדשה `#include <iostream>` וידנית נכניס רק את המזהים שנבחר אל תוך מרחב השמות שלנו, ע"י `using`.

Copyright © Yaniv Hamo, all rights reserved.

```
1 : #ifndef STRING2_H__
2 : #define STRING2_H__

3 : #include <iostream>
4 : using std::ostream;

5 : class String {
6 :   int len;
7 :   char *val;
8 :
9 :   char *copy(const char *);
10:   void clear();

11: public:
12:   String();
13:   String(const char *);
14:   String(const String &);
15:   ~String();

16:   String &operator=(const String &);

17:   int length() const { return(len); }

18:   const char &operator[](int) const;
19:   char &operator[](int);

20:   String &operator+=(const String &);
21:   int compare(const String &) const;
22:   void print(ostream &out) const;
23: };

24: const String operator+(const String &,const String &);
25: ostream &operator<<(ostream &, const String&);

26: bool operator==(const String&, const String&);
27: bool operator!=(const String&, const String&);
28: bool operator< (const String&, const String&);
29: bool operator> (const String&, const String&);
30: bool operator<=(const String&, const String&);
31: bool operator>=(const String&, const String&);

32: #endif
```

Copyright © Yaniv Hamo, all rights reserved.

זהו קובץ ה-header. נעבור רק על השינויים מהגירסא הראשונה.
שורות 3-4: ב-C++ נהוג לכלול מחלקות סטנדרטיות בשמן בלבד, ללא h. בסוף. אם כוללים כך ספרייה סטנדרטית, היא תוגדר ב-namespace משלה הנקרא std. על מנת להשתמש במשהו שהוגדר ב-namespace זה, עלינו להביא אותו במפורש ל-namespace הנוכחי. שורה 3 כוללת את הספרייה הסטנדרטית iostream, מתוכה אנו רוצים את המחלקה ostream (כדי שנוכל להגדיר את אופרטור ההדפסה). שורה 4 מביאה את המחלקה ostream אל ה-namespace הנוכחי, ובכך מאפשרת שימוש במחלקה זו.

שורה 9: אנו רוצים להשתמש ב-copy ברשימת אתחול, משהו כמו:
```
String(const char *s) : val(copy(s))
```
ולכן על copy להחזיר char * שהוא העתק של ה-* char אותו היא מקבלת כפרמטר, ומצביע להעתק יכנס ישירות לשדה val ע"י רשימת האתחול.
כיוון שנרצה גם לאתחל את השדה len ברשימת האתחול, copy לא תאתחל את השדה len כפי שעשתה בגירסא הראשונה.

שורה 18: הצורה היותר מדויקת; אם אנו רוצים להבהיר שאין אנו רוצים בהצבה לערך אותו אנו מחזירים, אך עדיין אנו מעוניינים להחזיר אותו כרפרנס כדי לשפר ביצועים, נחזיר אותו כ-const.

שורה 21: מתודה זאת נועדה לאפשר מימוש חיצוני של כל אופרטורי ההשוואה, ללא שנאלץ להגדירם כחברים במחלקה. האופרטורים יוכלו פשוט להפעיל מתודה זאת, ולקבל את יחס ההשוואה בין המחרוזת עליה מופעלת המתודה, לבין המחרוזת הנתונה. הערך המוחזר במקרה של:
```
s1.compare(s2)
```
הוא שלילי אם s1<s2, אפס אם s1==s2, וחיובי אם s1>s2.

שורה 22: מתודה זו נועדה לאפשר מימוש של אופרטור הפלט כלא חבר במחלקה. מקבלת כפרמטר את הזרם אליו יש להדפיס את המחרוזת.

שורות 24-31: הגדרת כל האופרטורים הדורשים סימטריות כפונקציות גלובליות, אך הפעם פונקציות אלו אינן חברות.

Copyright © Yaniv Hamo, all rights reserved.

```
1 : #include <cstring>
2 : #include <cstdlib>
3 : #include "string.h"

4 : char *String::copy(const char *s)
5 : {
6 :   if(s) {
7 :     char *v = new char [strlen(s)+1];
8 :     strcpy(v, s);
9 :     return v;
10:   }
11:   return NULL;
12: }

13: void String::clear()
14: {
15:   delete[] val;
16: }

17: String::String() : val(NULL),len(0)
18: {}

19: String::String(const char *s) :
              val(copy(s)),len(strlen(s))
20: {}

21: String::String(const String &str) :
              val(copy(str.val)), len(str.len)
22: {}

23: String::~String()
24: {
25:   clear();
26: }

27: String &String::operator=(const String &str)
28: {
29:   if(&str == this) return *this;

30:   clear();
31:   val = copy(str.val);
32:   len = str.len;
33:   return *this;
34: }
```

Copyright © Yaniv Hamo, all rights reserved.

```
35: const char &String::operator[](int i) const
36: {
37:   if(i<0 || i>=len) exit(1);
38:   return val[i];
39: }


40: char &String::operator[](int i)
41: {
42:   if(i<0 || i>=len) exit(1);
43:   return val[i];
44: }


45: String &String::operator+=(const String &str)
46: {
47:   // no need to check self assign
48:   if(len == 0) *this = str;
49:   else {
50:     int new_len = len + str.len + 1;
51:     char *v = new char [new_len];

52:     strcpy(v, val);
53:     strcat(v, str.val);
54:     clear();
55:     val = copy(v);
56:     len = new_len;
57:     delete [] v;
58:   }
59:   return *this;
60: }

61: int String::compare(const String &str) const
62: {
63:   return strcmp(val, str.val);
64: }

65: void String::print(ostream &out) const
66: {
67:   if(val) out << val;
68: }
```

Copyright © Yaniv Hamo, all rights reserved.

```
69: const String operator+(const String &str1,
                           const String &str2)
70: {
71:   String s = str1;
72:   s += str2;
73:   return s;
74: }

75: bool operator==(const String &str1,const String &str2)
76: {
77:   return str1.compare(str2) == 0;
78: }

79: bool operator<(const String &str1,const String &str2)
80: {
81:   return str1.compare(str2) < 0;
82: }

83: bool operator!=(const String &str1,const String &str2)
84: {
85:   return !(str1 == str2);
86: }

87: bool operator<=(const String &str1,const String &str2)
88: {
89:   return (str1<str2) || (str1==str2);
90: }

91: bool operator>(const String &str1,const String &str2)
92: {
93:   return !(str1 <= str2);
94: }

95: bool operator>=(const String &str1,const String &str2)
96: {
97:   return !(str1 < str2);
98: }

99 : ostream &operator<<(ostream &out, const String &str)
100: {
101:   str.print(out)
102:   return out;
103: }
```

Copyright © Yaniv Hamo, all rights reserved.

קובץ המימוש.

שורות 1-2: שימו לב שאין h. בסוף שמות הספריות. מכיוון שאילו ספריות משפת C (ניתן לדעת ע"י העובדה ששמן מתחיל ב-c), אין כאן בעיית namespace.

שורה 4: המימוש של copy. רצינו להשתמש ב-copy ברשימת האתחול, ולכן היא כבר לא משנה אף שדה בעצמה, אלא רק מחזירה מצביע להעתק שהיא יוצרת. את ההצבה לשדה val נבצע ברשימת האתחול.

שורה 17, 19, 21: שימו לב לשימוש ברשימת האתחול כדי לאתחל את שני השדות, len המאותחל במספר, ו-val המאותחל בערך החוזר מ-copy. הגופים של כל ה-c'tor ריקים (שורות 18, 20 ו-22) וזהו בדיוק המצב לו אנו שואפים.

שורות 31-32: כיוון ש-copy לא מציבה יותר בשדות, אנו מציבים כאן בעצמנו לשני השדות.

שורה 47: כיוון שאנו שומרים מערך זמני בשורה 51, אין בעיה של הוספה עצמית, a+=a, ולכן אין צורך בבדיקה.

שורה 61: המתודה compare פשוט מחזירה את strcmp על המחרוזת הנוכחית והנתונה.

שורה 65: המתודה print מדפיסה לזרם הנתון, לשימוש באופרטור ההדפסה.

שורה 69: אותו מימוש כמו בגירסא הראשונה, הפעם כבר הבנו שאין צורך להגדיר אותו כ-friend.

שורות 77,81: מימוש של == ו- > ע"י המתודה compare, מה שמאפשר להגדיר את כל אופרטורי ההשוואה כלא חברים.

שורה 101: שימוש במתודה print על מנת להדפיס, במקום גישה ישירה לשדה val, מה שמתיר לנו להגדיר את אופרטור ההשוואה כלא חבר.

Copyright © Yaniv Hamo, all rights reserved.