נושאים מתקדמים בתכנות מונחה עצמים
הרצאה 3

פרופ' עפר שיר
ofersh@telhai.ac.il

החוג למדעי המחשב
תל-חי
המכללה האקדמית

מבנה ההרצאה

- Smart pointers
- Reference countin

C++03

Smart -> Pointers*

Source: Scott Meyers, More Efficient C++

מהו פוינטר חכם?

פוינטר חכם (smart pointer) הוא אובייקט בתפקיד מצביע המוסיף פונקציונליות ביחס למצביע רגיל (*dumb pointer*).

פוינטרים חכמים מאפשרים ניהול משאבים טוב יותר (מניעת דלף זיכרון) וגם ייעול קוד ע"י מניעת העתקות עמוקות מיותרות (נראה בהמשך ב-Reference Counting).
קלאינט המשתמש ב smart pointers אמור להשתמש בהם כאילו הם היו dumb pointers.
בתקן הישן קיימת מחלקת תבנית סטנדרטית עבור פוינטר חכם: **`auto_ptr`**; בפרק זה נבנה פוינטר חכם כדי להבין את תפקידיו.

פוינטרים חכמים: מוטיבציה

1. פחות תקלות:

א- שחרור אוטומטי של זיכרון

ב- איתחול אוטומטי: בנאי ברירת המחדל מאתחל כתובת ל-NULL

ג- dangling pointers (פוינטרים "רופפים")

2. בטיחות מפני דלף זיכרון בעת זריקת exceptions

3. garbage collection :

בניגוד לשפות אחרות, C++ לא משחררת זיכרון מוקצה דינאמית באופן אוטומטי. שימוש בפוינטרים חכמים מאפשר לשחרר את ההקצאה שעליה מצביע הפוינטר ברגע שחייו של הפוינטר מסתיימים כמשתנה אוטומטי.

פוינטרים חכמים: מוטיבציה (המשך)

4. STL containers :

מבני נתונים ב STL שומרים על הנתונים by value ולא by pointer, לכן הם לא מותאמים להיות פולימורפים. לדוגמה:

```
class Student { /*...*/ };
class Scholar : public Student { /*...*/ };

Student b;
Scholar d;
vector< Student> v;

v.push_back(b); // OK
v.push_back(d); // slicing!!!
```

הבעיה נובעת מכך שהאובייקטים ב v יכולים להיות רק מטיפוס Student.

פוינטרים חכמים: מוטיבציה (המשך)

5. STL containers:

ניתן לפתור את הבעיה ע"י שימוש ב vector של מצביעים:

```
vector<Student*> v;
v.push_back(new Student);    // OK
v.push_back(new Scholar);    // OK too

// cleanup:
for(vector<Student*>::iterator i=v.begin();i!= v.end(); ++i)
    delete *i;
```

אלא שאז צריך להקפיד על שחרור פרטני של כל מצביע.

← ניתן לעשות זאת יותר בפשטות בעזרת פוינטרים חכמים:

```
vector<SmartPtr<Student> > v;
v.push_back(new Student);    // OK
v.push_back(new Scholar);    // OK too
```

בנאים ומפרקים

ניתן לקבוע מה קורה כאשר פוינטר נבנה ומתפרק:

- ניתן להציב NULL לפוינטר שלא מצביע על שום דבר.
- ניתן לפרק את האובייקט עליו מצביע הפוינטר, אם זהו הפוינטר האחרון שמצביע על אותו אובייקט (למניעת דלף זיכרון)

העתקות והשמות

ניתן לקבוע מה קורה כאשר מעתיקים ערך של פוינטר

Dereferencing

מה קורה כאשר פונים לאובייקט עליו מצביע הפוינטר?

template<class T> SmartPtr

פוינטרים חכמים הם templates, כי הטיפוס עליו מצביע הפוינטר חשוב מאוד לתפקודו. קוד התבנית עבור הפוינטר חכם נראה בעיקרון כך:

```
template<class T>
class SmartPtr
{
   public:
      SmartPtr(T* realPtr=0);
      SmartPtr(const SmartPtr<T>& rhs);
      ~SmartPtr();
      SmartPtr<T>& operator=(const SmartPtr<T>& rhs);
      T* operator->() const;
      T& operator*() const;
   private:
      T *pointee;
};
```

template<class T> SmartPtr

הסברים:

1. פוינטר חכם בנוי על בסיס הפוינטר ה"טיפש" **pointee**, שהוא המצביע האמיתי כאן.
2. ההנחה היא ש-**pointee** מצביע על אובייקט שהוקצה דינמית כך שהמפרק של **SmartPtr** יכול לשחרר אותו באמצעות **delete**
3. ייתכן שנרצה לשים את ה copy c'tor ו assignment o'tor ב private כדי למנוע העתקות בין פוינטרים
4. האופרטורים **->** וכן ***** הם **const** כי אובייקט ה **SmartPtr** אינו משתנה על ידם (אם כי האובייקט עליו מצביע **pointee** עשוי להשתנות)

```
template<class T>
class SmartPtr
{
   public:
      SmartPtr(T* realPtr=0) : pointee(realPtr){}
      SmartPtr(const SmartPtr<T>& rhs);
      ~SmartPtr() { delete pointee;}
      SmartPtr<T>& operator=(const SmartPtr<T>& rhs);
      T* operator->() const;
      T& operator*() const;
   private:
      T *pointee;
};
```

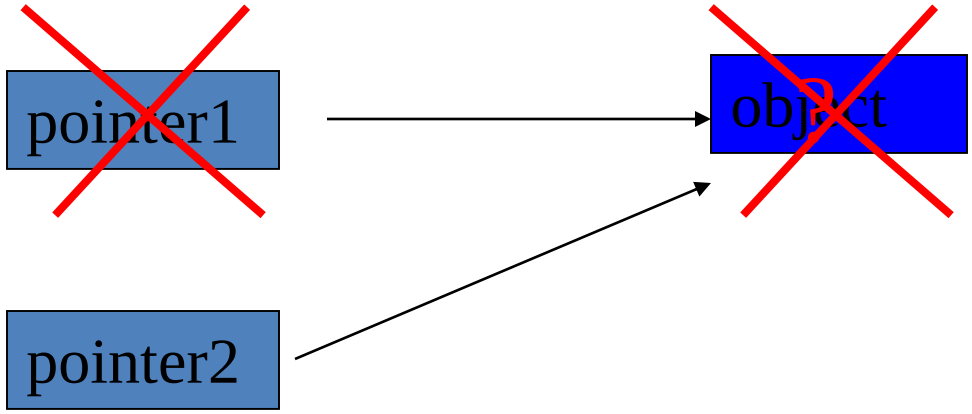
בעיה: dangling pointer

הנחה:

כל האובייקטים עליהם מצביעים פוינטרים חכמים מוקצים דינמית

בעיה:

אם שני פוינטרים מצביעים על אותו אובייקט, פרוק של פוינטר חכם יגרום לאובייקט עליו מצביע הפוינטר השני להיעלם:



בעיית הבעלות (ownership)

תפקיד ה copy c'tor לפתור את בעיית הבעלות ולענות על השאלה "איזה פוינטר הוא הבעלים של האובייקט?"

הדבר חשוב בכדי לקבוע מי יפרק את האובייקט.

פיתרון 1:

העתקה עמוקה בכל פעם שפוינטר מועתק ← כל פוינטר יצביע על האובייקט שלו!

החיסרון:

לא תמיד יודעים על איזה טיפוס של אובייקט מצביע הפוינטר. זה לא חייב להיות T, זה יכול להיות טיפוס שנגזר מ T.

העברת בעלות ב copy c’tor

פיתרון 2:

הבעלות על האובייקט מועברת בזמן ההעתקה:

```
template<class T>
SmartPtr<T>::SmartPtr(SmartPtr<T>& rhs)
{
   pointee = rhs.pointee; // transfer ownership

   rhs.pointee = 0;       // rhs no longer owns
}
```

העברת בעלות באופרטור השמה מעתיק

```
template<class T>
SmartPtr<T>& SmartPtr<T>::operator=( SmartPtr<T>& rhs)
{
    if(this==&rhs)
        return *this;
    delete pointee;       // delete currently owned object
    pointee=rhs.pointee; // transfer ownership
    rhs.pointee=0;        // rhs no longer owns anything
    return *this;
}
```

העברת בעלות: הערות

1. הפרמטר של בנאי ההעתקה (ושל אופרטור ההשמה) אינו **const** כי הוא משתנה בתוך הפונקציה. זה עשוי ליצור בעיה אם עושים שימוש בפונקציה שמניחה שהפרמטר הוא **const**, כמו, למשל, בשימוש ב **SmartPtr** בקוד תבנית המניח זאת.

   הפונקציה **push_back** של **std::vector** עושה שימוש באופרטור **=** עם פרמטר **const**; ניתן לפתור זאת בעזרת Reference Counting .

   פתרון אחר: **const_cast** (בשקף הבא)

2. כאשר smart pointer מועבר כפרמטר לפונקציה, הוא יועבר תמיד by reference. אם הוא מועבר by value, אז בנאי ההעתקה יגרום לארגומנט הפונקציה לאבד בעלות על האובייקט שלו לטובת משתנה פנימי של הפונקציה

פתרון בעיית ה-const

```
template<class T>
SmartPtr<T>::SmartPtr(const SmartPtr<T>& rhs)
{
    pointee = rhs.pointee; // transfer ownership
    SmartPtr<T>& nonConst = const_cast<SmartPtr<T>&>(rhs);
    nonConst.pointee = 0;  // rhs no longer owns
}
```

```
template<class T>
SmartPtr<T>& SmartPtr<T>::operator=(const SmartPtr<T>& rhs)
{
    if(this==&rhs)
        return *this;
    delete pointee;       // delete currently owned object
    pointee=rhs.pointee; // transfer ownership
    SmartPtr<T>& nonConst=const_cast<SmartPtr<T>&>(rhs);
    nonConst.pointee=0;  // rhs no longer owns anything
    return *this;
}
```

האופרטור *

```
template<class T>
T& SmartPtr<T>::operator*() const
{
    // smart pointer processing
    ...
    return *pointee;
}
```

הערות:

1. לפני ה-`return` ייתכן שנרצה לבצע פעולות מסויימות, כגון העתקה עמוקה.
2. הערך מוחזר by reference למקרה שהאובייקט שעליו מצביע ה-smart pointer אינו מטיפוס `T` אלא מטיפוס הנגזר מ `T` (בנוסף, זוהי החזרה יותר יעילה)
3. אפשר להוסיף זריקת exception למקרה שהפוינטר מצביע על כתובת NULL

האופרטור new

מכיוון ש smart pointer הוא אובייקט ולא פוינטר, האופרטורים new ו delete לא פועלים עליו באופן טבעי

בכל זאת, כיוון ש smart pointer נועד לדמות את הפעולה של פוינטר, היינו רוצים להפעיל עליו את האופרטורים האלה.

האופרטור new משתלב עם טיפוס smart pointer בעזרת בנאי ההמרה:

```
SmartPtr(T* realPtr=0);
```

למשל:

```
SmartPtr<Student> pt1 = new Student;
```

כתובת של Student

האופרטור new

או בעזרת אופרטור ההשמה:

```
SmartPtr<T>& operator=(T* ptr);
```

למשל:

```
SmartPtr<Student> pt1;
pt1 = new Student;
```

האופרטור delete

ניתן להפעיל את האופרטור delete אם נגדיר אופרטור המרה ל *T:

```
operator T*(){return pointee;}
```

ואז ניתן לכתוב את הקוד הבא

```
SmartPtr<int> pt1 = new int;
...
delete pt1;
```

המרה ל *T

בעיה:

תהיה תקלה כאשר יופעל המפרק של pt1.

האופרטור `delete`

מסקנה:

אין להפעיל `delete` על smart pointer

או במילים אחרות:

לא כדאי להגדיר אופרטור המרה ל `T*`, כיוון שהגדרה כזאת מאפשרת הפעלת `delete` על smart pointer.

אלא שלהגדרת אופרטור המרה ל `T*` יש יתרונות אחרים.

המרה ל T*

בדיקה האם smart pointer הוא NULL

היינו רוצים לאפשר כתיבה של קוד כזה:

```
SmartPtr<Student> ptr;
if (ptr == 0)
```

או

```
if (ptr)
```

```
if (!ptr)
```

או

ללא אופרטור המרה ל T* תהיה הודעת שגיאה

המרה ל `T*`

בעיות של המרה ל `T*`:

1. ברגע שיש פוינטר חכם ופוינטר טיפש המצביעים על אותו אובייקט קשה לעקוב אחרי הבעלות על האובייקט.
2. כפי שראינו, המרה אוטומטית מ smart pointer ל dumb pointer גורמת לתקלה במפרק

העמסת האופרטור !

```
template<class T>
class SmartPtr
{
public:
    ...
    bool operator!(){return pointee==NULL;}
    ...
};
```

מאפשר לכתוב `if(!ptr)` אך אינו מאפשר `if (ptr == 0)`
או `if (ptr)`

מסקנה:
יש לשקול את היתרונות והחסרונות של כל אחד מהפתרונות

המחלקה auto_ptr

```
template<class T>
class auto_ptr {
   T* ptr;
public:
      typedef T element_type;
      explicit auto_ptr(T *p = 0) throw() : ptr(p) {}
      auto_ptr(const auto_ptr<T>& rhs) throw();
      auto_ptr<T>& operator=(auto_ptr<T>& rhs) throw();
      ~auto_ptr() {delete ptr;}
      T& operator*() const throw() {return *ptr;}
      T *operator->() const throw() {return ptr;}
      T *get() const throw();
      T *release() const throw();
};
```

המחלקה auto_ptr

הערות:

1. לכל אובייקט עליו מצביע auto_ptr<T> יש רק "בעלים" מטיפוס auto_ptr<T> אחד
2. כאשר מבוצעת העתקה – הבעלות על האובייקט עוברת מהאובייקט auto_ptr<T> שבאגף ימין לאובייקט אליו מבוצעת העתקה
3. במחלקה אמור להיות אינדיקטור המראה האם לאובייקט ה- auto_ptr<T> בעלות על האובייקט שעליו הוא מצביע
4. **חשוב:** אובייקט auto_ptr, או כל smart pointer אחר, נועד להצביע רק על אובייקט אחד ולא יכול להוות בסיס לבניית מערך.

C++03

REFERENCE COUNTING

Source: Scott Meyers, More Efficient C++

מוטיבציה

1. לדעת מתי יש למחוק אובייקט שהוקצה ע"י `new`:
אם כמה פוינטרים מצביעים על אותו אובייקט, נרצה למחוק את האובייקט רק כאשר אין יותר פוינטרים שמצביעים עליו

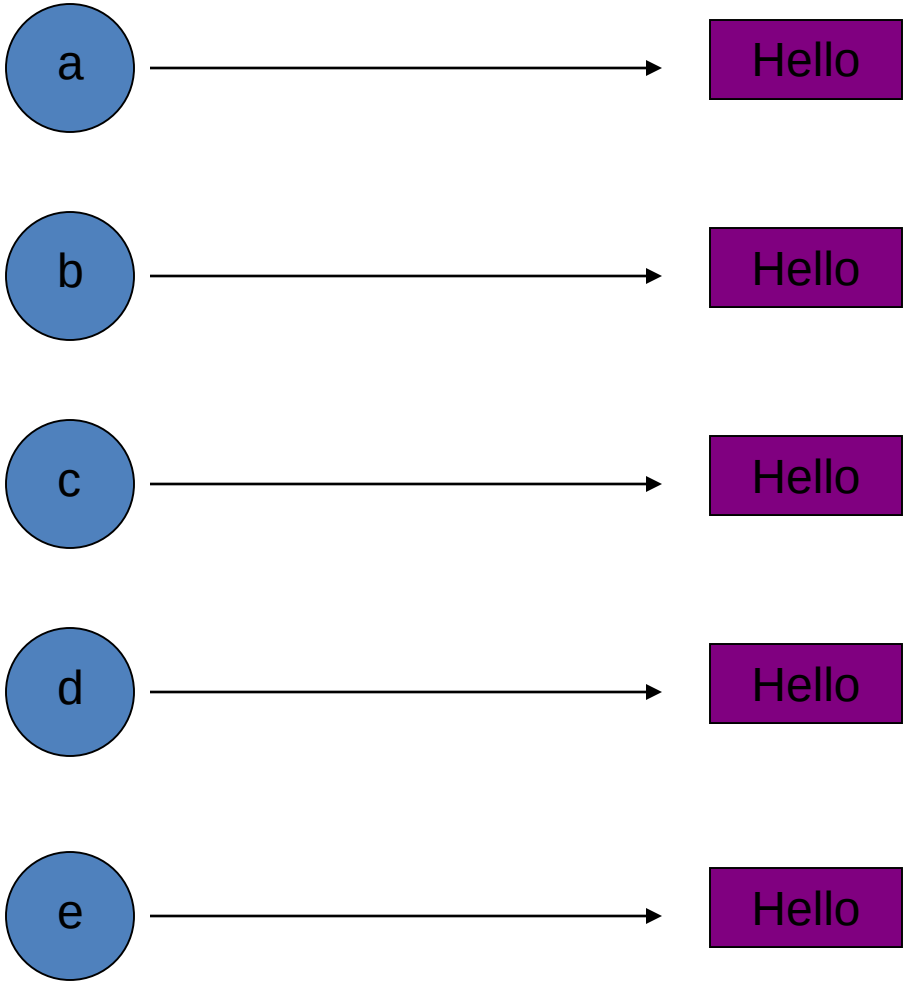
2. חיסכון בזיכרון: אם לכמה אובייקטים יש אותו ערך, אין צורך שיתפסו כמה מקומות בזיכרון.

```
class String {
public:
 String( const char *value="”); // conversion c'tor &
                                 // default c'tor
 String( const String&);       // copy c'tor
 ~String(){ delete []s;}

 String& operator= ( const String&); // assignment o'tor

private:
 int len;
 char *s;
};

String a,b,c,d,e;

a = b = c = d = e = "Hello";
```

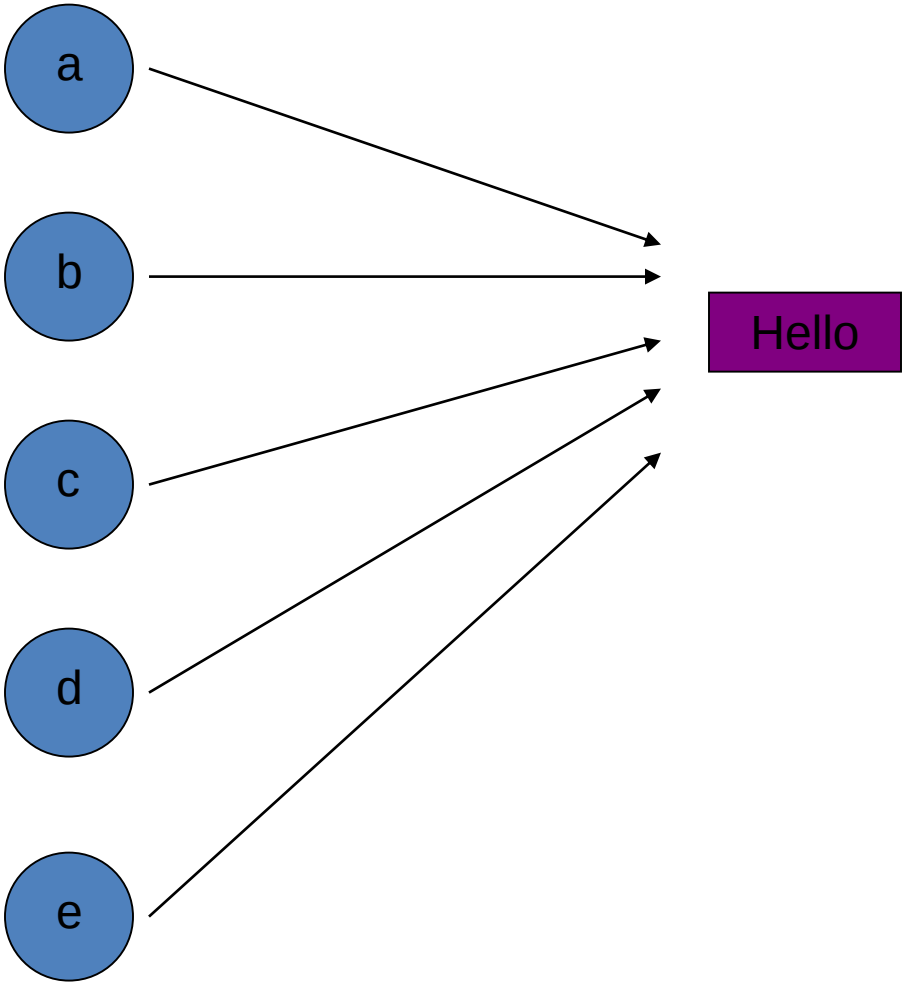
operator=

```
String& String::operator=(const String& st) {
   if ( this == &st )
      return *this;
   delete []s;
   len = st.len;
   s = new char[len+1];
   strcpy(s, st.s);
   return *this;
}
```

התוצאה: העתקה עמוקה



מצב רצוי: העתקה רדודה (!)



בעיות

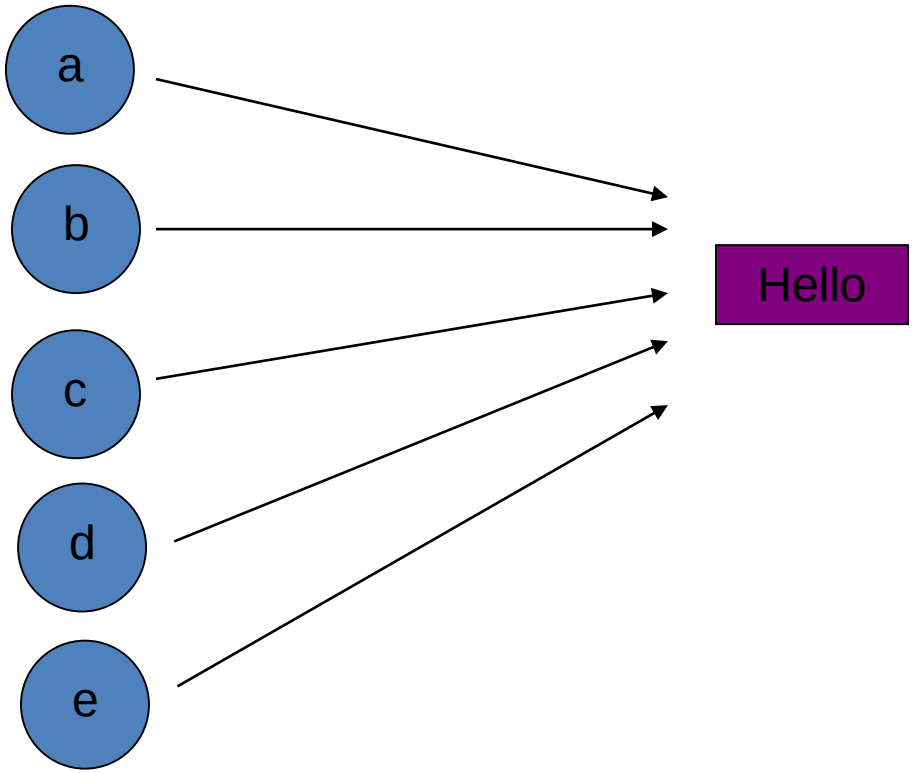
1. שינוי באובייקט אחד יביא לשינוי באובייקטים האחרים.
2. מחיקת אובייקט אחד יגרום למחיקת האובייקטים האחרים.

פתרונות

1. Copy on write – ברגע שאובייקט משתנה הוא יוצר לעצמו עותק משלו
2. Reference counting – ספירת האובייקטים המצביעים על אותו מקום בזיכרון. אובייקט ימחק את המקום עליו הוא מצביע **רק אם הוא היחיד** שמצביע עליו.

כדי לממש את 1 יש צורך ב reference counting, לכן נתחיל עם פתרון 2.

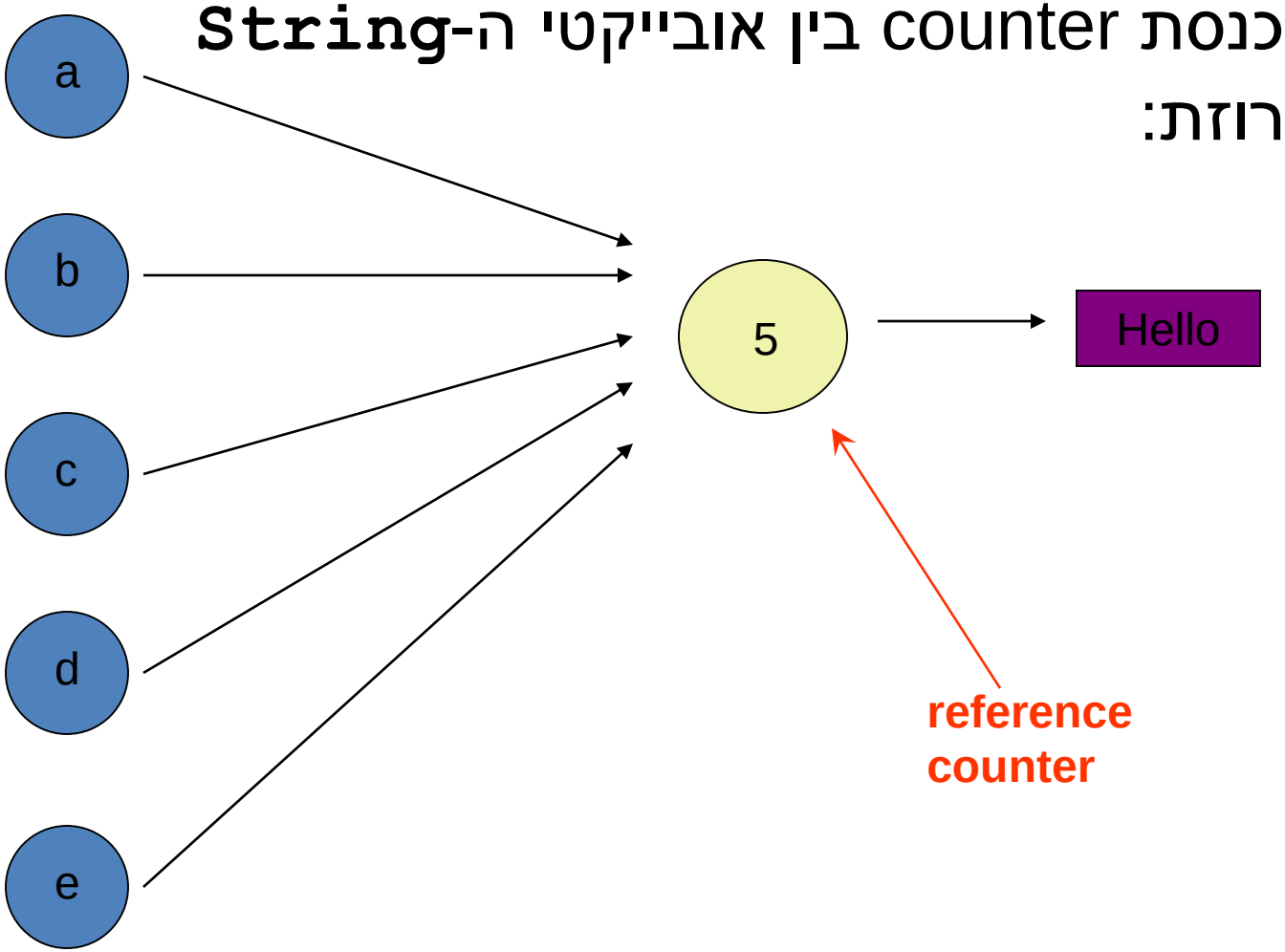
מימוש reference counting



הבעיה: לא ניתן לספור כמה אובייקטים מצביעים על "Hello" כי מדובר בטיפוס פרימיטיבי **char***

מימוש reference counting

פיתרון: הכנסת counter בין אובייקטי ה-**`String`**
לבין המחרוזת:



מימוש

מגדירים ב private של המחלקה **String** רשומה (struct) או מחלקה בשם **StringValue**, המקשרת בין **String** ל **char***

מדוע struct?

כדי לאפשר לפונקציות המחלקה של **String** לגשת לכל ה members של **StringValue**

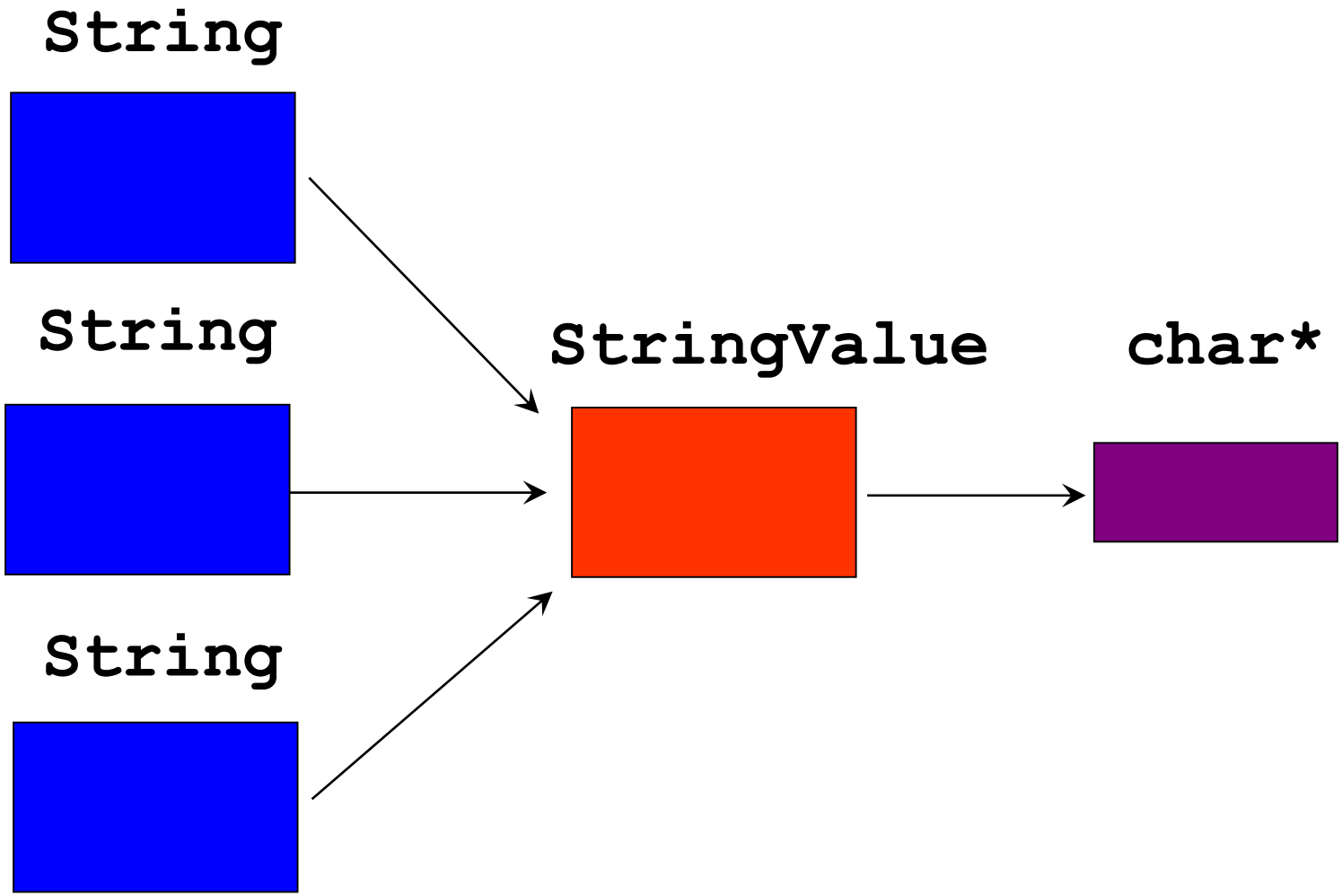
למה הרשאת גישה private?

כדי למנוע גישה מחוץ למחלקה **String**

Reference counting - 1 גירסה

```
class String {
public:
    String(const char *val=""); // conversion c'tor
    String(const String&);       // copy c'tor
    ~String();
    String& operator= (const String&);
    // ...
private:
    struct StringValue {
        int refCount;
        char* data;
        StringValue(const char *initValue);
        ~StringValue();
    };
    StringValue *value;
};
```

Reference counting - גירסה 1



בנאי ומפרק של StringValue

```
String::StringValue::StringValue(const char *initValue)
:refCount(1)
{
   data = new char[strlen(initValue)+1];
   strcpy(data,initValue);
}

String::StringValue::~StringValue()
{
   delete[]data;
}
```

בנאים במחלקה String

```
String::String(const char *initValue)
:value(new StringValue(initValue))
{}

String::String(const String& rhs)
: value(rhs.value)
{
   ++value->refCount;
}
```

העתקה רדודה

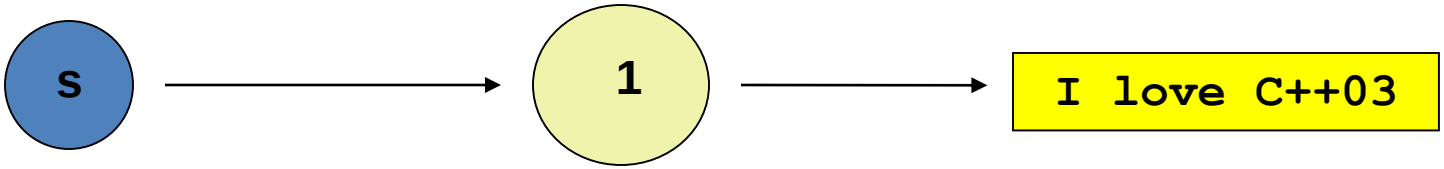
הערה: העתקת אובייקט ה StringValue היא העתקה רדודה של הכתובת. מספר אובייקטים String יכולים להצביע על אותו מופע value

בנאים במחלקה String

למשל, הפקודה הבאה:

```
String s("I love C++03");
```

תביא למצב המאופיין כך:

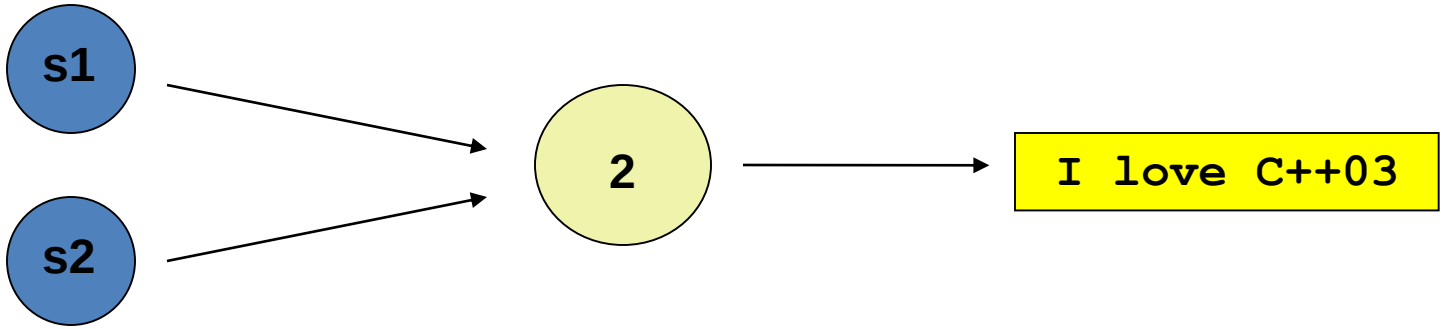


בנאים במחלקה String

והקוד הבא:

```
String s1("I love C++03"), s2 = s1;
```

יביא למצב המאופיין כך:



המפרק של String

```
String::~String()
{
   if(--value->refCount==0)
      delete value;
}
```

פעולות המפרק:

1. הקטנת המונה שבאובייקט ה StringValue
2. אם זהו האובייקט היחיד המצביע על המחרוזת, אז המפרק מוחק את המחרוזת ע"י מחיקת אובייקט ה StringValue

אופרטור השמה מעתיק

```
String& String::operator= ( const String& rhs)
{
   if (value==rhs.value)
      return *this;

   if (--value->refCount == 0)
      delete value;

   value = rhs.value;
   ++value->refCount;

   return *this;
}
```

ביטול ההצבעה הקודמת

העתקה רדודה

Copy-on-Write (COW)

- אובייקטי **String** שונים יכולים להצביע על אותו אובייקט **StringValue** אם המחרוזות שלהם זהות.
- לכן, אם משנים ערך של אובייקט, יש להעתיק אותו תחילה העתקה עמוקה לאובייקט **StringValue** חדש, כדי לא לשנות את האובייקטים שמצביעים על אותו ערך.
- שינוי בתוך מחרוזת ייעשה באמצעות גישה של [ ] ולכן ההעתקה העמוקה תמומש בתוך **operator[]**

```
class String {
public:
    String(const char *val=""); // conversion c'tor
    String(const String&);      // copy c'tor
    ~String();
    String& operator= (const String&);
    const char& operator[](int i) const;
    char& operator[](int i);
    // ...
private:
    struct StringValue  {
        int refCount;
        char* data;
        StringValue(const char *initValue);
        ~StringValue();
    };
    StringValue *value;
};
```

String::operator[]

גרסת ה"קריאה בלבד" - אין שינוי:

```
const char& String::operator[](int i) const
{
    return value->data[i];
}
```

String::operator[]

לעומת זאת באופרטור לכתיבה (שאינו const), ייתכן שהפנייה לאובייקט היא לצורך שינוי המחרוזת:

```
char& String::operator[](int i) {
  if (value->refCount > 1) {
     // the value is shared by other instances
    --value->refCount;
    value = new StringValue(value->data);
  }
  return value->data[i];
}
```

String::operator[]

אין דרך לדעת אם הגישה לאופרטור [] דרך אובייקט שאינו **const** היא לצורך קריאה בלבד או כתיבה, לכן בכל קריאה לאופרטור [] ע"י אובייקט שאינו **const** מתבצעת העתקה לאובייקט **StringValue** חדש.
שיטה זו קרויה:

Copy-on-Write

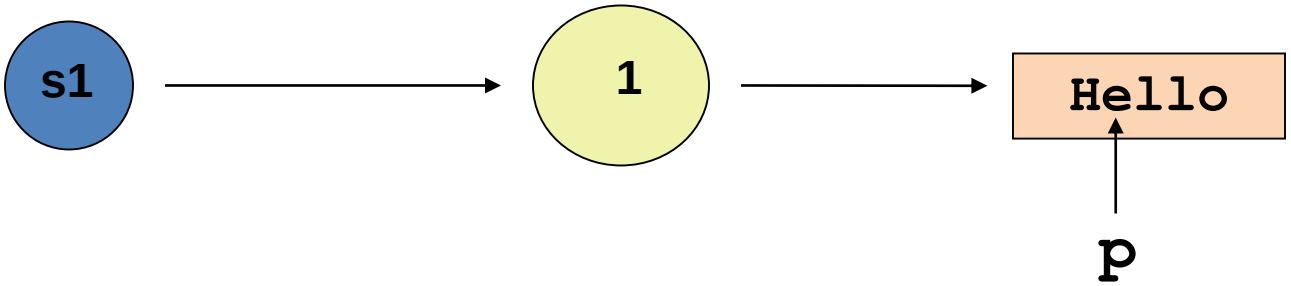
אלא שלא הכל פתור...

Reference counting - 2 גירסה

נתבונן בקוד הבא:

```
String s1 = "Hello";
char *p = &s1[1];
```

וזהו המצב שנוצר:



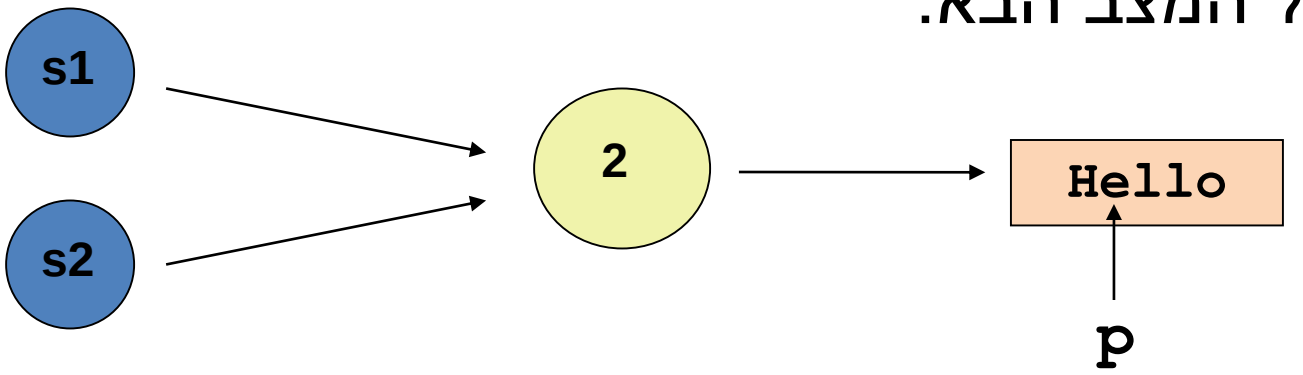
הקריאה ל `operator[]` לא גרמה להעתקה עמוקה כי s1 הוא האובייקט היחיד המצביע על המחרוזת.

Reference counting - 2 גירסה

עכשיו, אם כותבים

```
String s2 = s1;
```

מתקבל המצב הבא:



מה יהיו תוצאות השורה הבאה?

```
*p = 'x';
```

Reference counting - 2 גירסה

הבעיה:

הבנאי של **String** לא יכול לדעת על קיומו של המצביע **p**.

הפיתרון:

להוסיף דגל בוליאני במחלקה **StringValue** שיאמר האם ניתן לשתף את הנתונים עם אובייקט אחר. הדגל יהפוך ל **false** בעת הקריאה הראשונה לאופרטור **[]** .

Reference counting - 2 גירסה

```
class String
{
public:
// ...
private:
    struct StringValue
    {
        int refCount;
        bool shareable;
        char* data;
        StringValue(const char *initValue);
        ~StringValue();
    };
    StringValue *value;
};
```

Reference counting - 2 גירסה

```
String::StringValue::StringValue( const char
*initValue): refCount(1),shareable( true ) {
   data = new char[strlen(initValue)+1];
   strcpy(data,initValue);
}
```

Reference counting - 2 גירסה

```
String::String(const String& rhs)
{
  if(rhs.value->shareable)
  {
    value = rhs.value;
    ++value->refCount;
  }
  else
    value = new StringValue(rhs.value->data);
}
```

Reference counting - 2 גירסה

```
String& String::operator= (const String& rhs)
{
  if (value==rhs.value)
    return *this;
  if(--value->refCount == 0)
    delete value;
  if(rhs.value->shareable) {
    value = rhs.value;
    ++value->refCount;
  }
  else
    value = new StringValue(rhs.value->data);
  return *this;
}
```

Reference counting - 2 גירסה

```
char& String::operator[](int i)
{
  if (value->refCount > 1)
  {
    --value->refCount;
    value = new StringValue(value->data);
  }

  value->shareable = false;

  return value->data[i];
}
```

Reference counting - 2 גירסה

הערה:

מנגנון ה Copy-on-Write מבוסס על ההנחה לפיה שינוי במחרוזת עליה מצביעה המחלקה **String** ייעשה באמצעות האופרטור **[]**. שינויים עשויים להיעשות גם בדרכים אחרות, כמו, למשל, האופרטור ***** (*dereferencing operator*). במקרה זה יש לשקול האם להכליל את מנגנון ה COW גם לאופרטורים הנוספים או לא להעמיס את האופרטורים האלה.

מחלקת בסיס ל Reference counting

Reference counting (RC) עשוי להיות שימושי במחלקות רבות. כפי שראינו, הוספת תכונה זו למחלקה דורשת עבודה רבה. היינו רוצים "לבודד" את הקוד הקשור ב RC כך שניתן יהיה להוסיף אותו לכל מחלקה באופן מודולרי, תוך הכנסת מינימום שינויים באותה מחלקה.

ניתן להגדיר מחלקה מיוחדת שתכיל (כמעט) את כל הקוד הדרוש לביצוע RC ולרשת ממחלקה זו כדי להוסיף RC לכל מחלקה שנרצה.

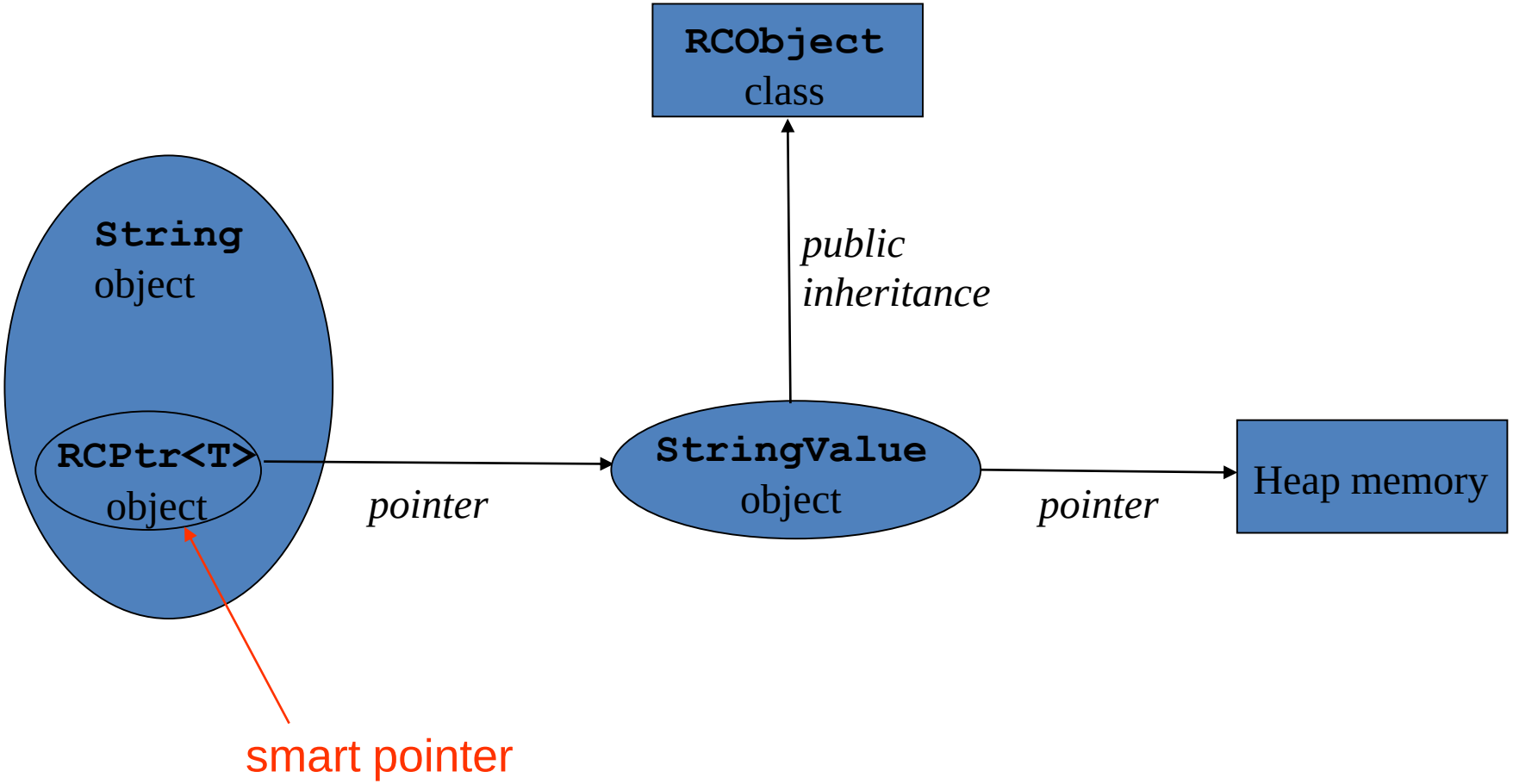
מחלקת בסיס ל Reference counting

שלב 1: יצירת מחלקת בסיס **`RCObject`** ובה הפונקציות הקשורות ב RC. כל מחלקה שתרצה להשתמש ב RC תירש מ **`RCObject`**.

שלב 2: יצירת מחלקה template המהווה מצביע חכם, אשר באמצעותה תנהל המחלקה **`String`** (או כל מחלקה אחרת) את כל הפעולות הקשורות ב RC.

התוצאה: במחלקה **`String`** וב **`StringValue`** אין קוד הקשור ב RC.

דיזיין הפתרון



המחלקה RCObject

```
class RCObject {
protected:
  RCObject(): refCount(0), shareable(true){}
  RCObject(const RCObject&):refCount(0),shareable(true){}
  RCObject& operator=(const RCObject&) {return *this;}
  virtual ~RCObject()=0 {}
public:
  void addReference(){++refCount;}
  void removeReference(){if(--refCount==0) delete this;}
  //object should be on the heap!!!
  void markUnshareable(){shareable=false;}
  bool isShareable() const {return shareable;}
  bool isShared() const {return refCount>1;}
private:
  int refCount;
  bool shareable;
};
```

המחלקה RCObject

הסברים:

1. הבנאים של **RCObject** מאתחלים את **refCount** ל 0, דבר שעשוי להיות מנוגד לאינטואיציה. הסיבה היא שהאובייקט שיוצר את אובייקט ה **RCObject** כבר ידאג לאתחל את **refCount** ל 1.
2. אופרטור ההשמה אינו עושה דבר! הסבר:

i. סביר להניח שלא ייעשה שימוש באופרטור זה: אין השמות בין אובייקטי **RCObject**, אלא בין האובייקטים המצביעים עליהם.

ii. אנו לא מצפים שתהיה השמה בין אובייקטי **StringValue**, אלא בין אובייקטי ה **String** המצביעים עליהם. במקרה זה אובייקטי ה **String** ידאגו לשנות את הערך של **refCount**

iii. הערך של **refCount** נקבע לפי מספר אובייקטי ה **String** המצביעים על אובייקט ה **StringValue**. אובייקט ה **StringValue** לא יכול לדעת מי מצביע עליו ולכן אופרטור ההשמה שלו לא משנה את **refCount**.

המחלקה RCObject

3. שימו לב שאובייקט יכול "להתאבד". בפונקציה **removeReference**, הפקודה **delete this;** מניחה שהאובייקט הוקצה דינאמית משתי סיבות:
   I. רק לאובייקט שהוקצה דינאמית ניתן לעשות **delete**
   II. רק לאובייקט שהוקצה דינאמית אין קריאה אוטומטית למפרק ביציאה מה scope שלו. אם היתה קריאה למפרק עבור אובייקט שכבר נעשה לו **delete** היתה שגיאת ריצה.

   יש צורך להתאים את האובייקט המיועד – ראו להלן.

המחלקה String

```
class String
{
public:
   String(const char *val=""); // conversion c'tor
   const char& operator[](int i) const;
   char& operator[](int i);
    // ...

private:
   RCPtr<StringValue> value;
};
```

smart pointer

שינויים במימושים של String

```
char& String::operator[](int i)
{
  if (value->isShared())
  {
    value = new StringValue(value->data);
  }

  value->markUnshareable();

  return value->data[i];
}
```

StringValue המחלקה

```
class StringValue: public RCObject
{
   void init(const char *initValue);

public:
   char* data;
   StringValue(const char *initValue);
   StringValue(const StringValue& rhs);
   ~StringValue();
};
```

```
void String::StringValue::init(const char *initValue)
{
   data = new char[strlen(initValue)+1];
   strcpy(data,initValue);
}
```

הערות ל **String** ו **StringValue**

1. במחלקה **String** יש עכשיו פוינטר חכם במקום פוינטר "טיפש".
2. ניהול ה Reference counting מתבצע באמצעות הפוינטר החכם **RCPtr**. מלבד זה אין ל **String** נגיעה ל Reference counting
3. הפונקציה **StringValue::init** מרכזת בתוכה קוד המשותף לשני הבנאים של **StringValue** ולכן נכלל בפונקציה מיוחדת ב private של **StringValue**. <u>זוהי גישה נפוצה בהנדסת תוכנה.</u>
4. המחלקה **StringValue** אינה מנהלת מנגנון RC. הכול נעשה במחלקת הבסיס **RCObject**.

מחלקת התבנית RCPtr

```
template<class T>
class RCPtr
{
    T* pointee;
    void init();
public:
    RCPtr(T* realPtr=0): pointee(realPtr) { init();}
    RCPtr(const RCPtr& rhs): pointee(rhs.pointee)
    { init();}
    ~RCPtr(){ if (pointee) pointee->removeReference();}
    RCPtr& operator=(const RCPtr& rhs);
    T* operator->() const{return pointee;}
    T& operator*() const{return *pointee;}
};
```

מימושים של RCPtr

```
template<class T>
void RCPtr<T>::init()
{
  if(pointee==0) return;

  if(pointee->isShareable() == false) {
    pointee= new T(*pointee);
// requires a copy c'tor for T
  }
  pointee->addReference();
}
```

מימושים של RCPtr

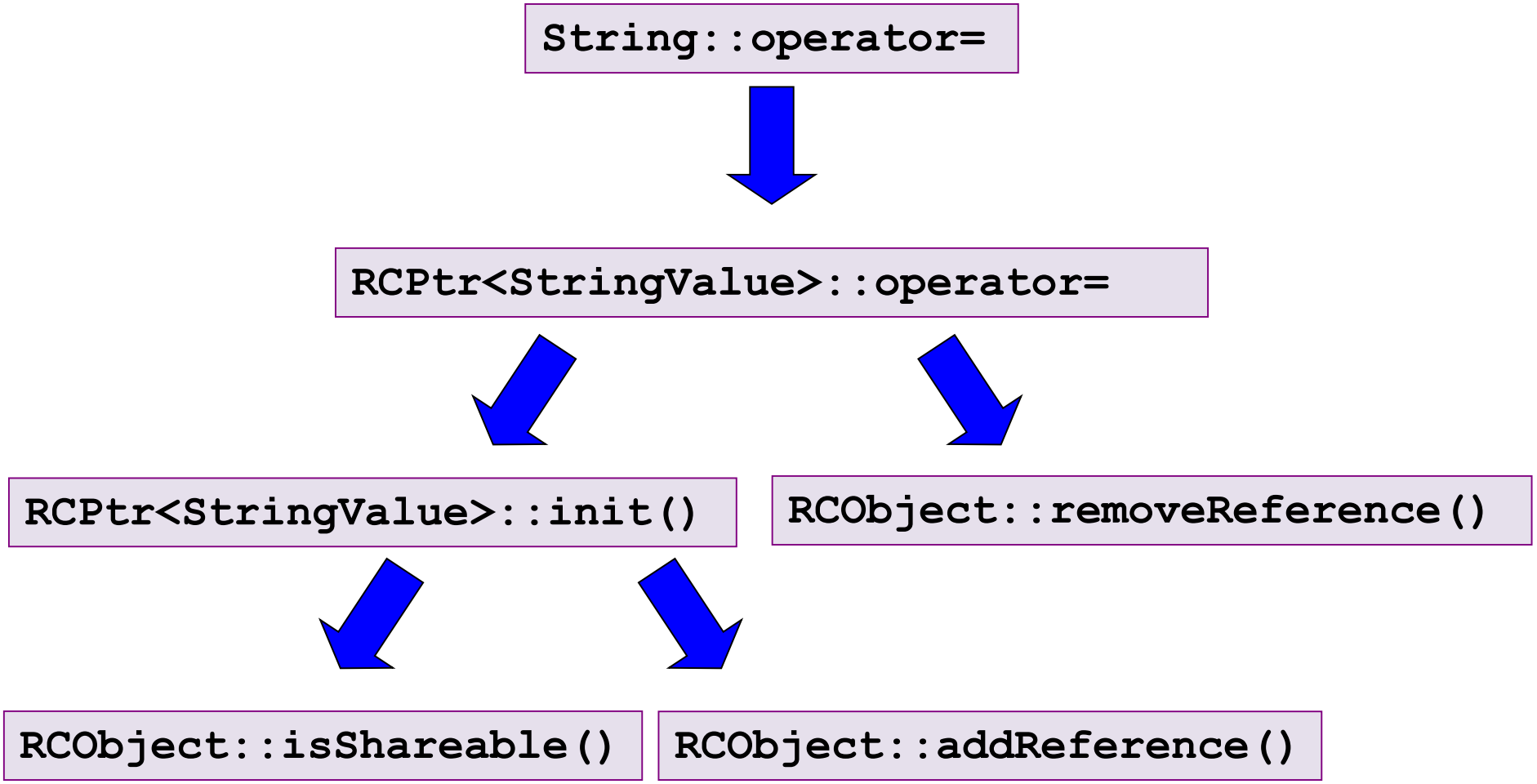
```
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr<T>& rhs)
{
  if (pointee != rhs.pointee)   {
    T* oldPointee = pointee;
    pointee = rhs.pointee;
    init();
    if(oldPointee)
      oldPointee->removeReference();
  }
  return *this;
}
```

הורדת המונה של אובייקט
ה StringValue הישן

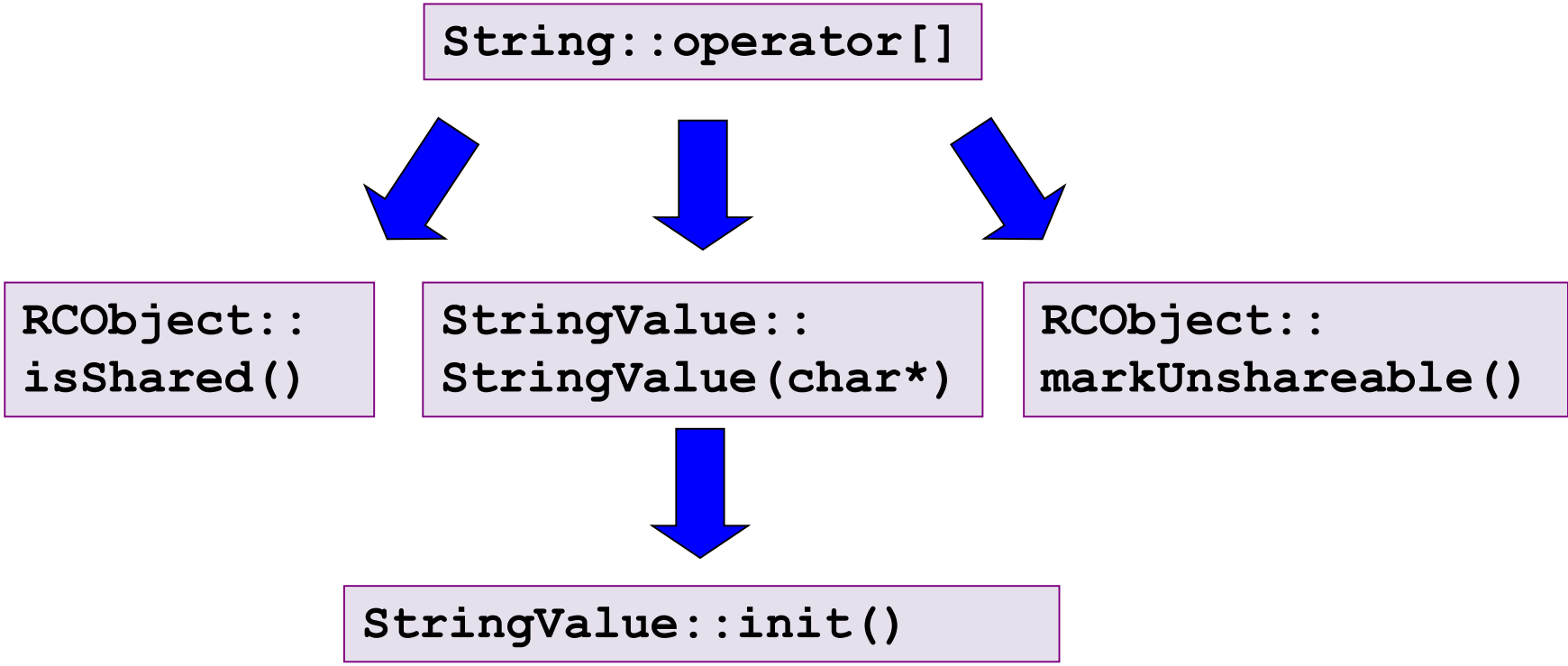
הערות ל **RCPtr**

1. זוהי מחלקה בסיסית של "מצביע חכם"
2. הפונקציה **`init()`** ממלאת תפקיד כמו ב **`StringValue`**
3. המחלקה **`RCPtr`** מנהלת את ה Reference counting באמצעות הפונקציות של מחלקת הבסיס **`RCObject`**
4. רק אובייקט **`RCPtr`** יכול להצביע על אובייקט **`StringValue`**. הדבר מבטיח שאובייקטי ה **`StringValue`** יהיו מוקצים דינאמית ולכן הפעולה **`delete this;`** תהיה חוקית.

ביצוע RC ע"י `operator=` של `String`

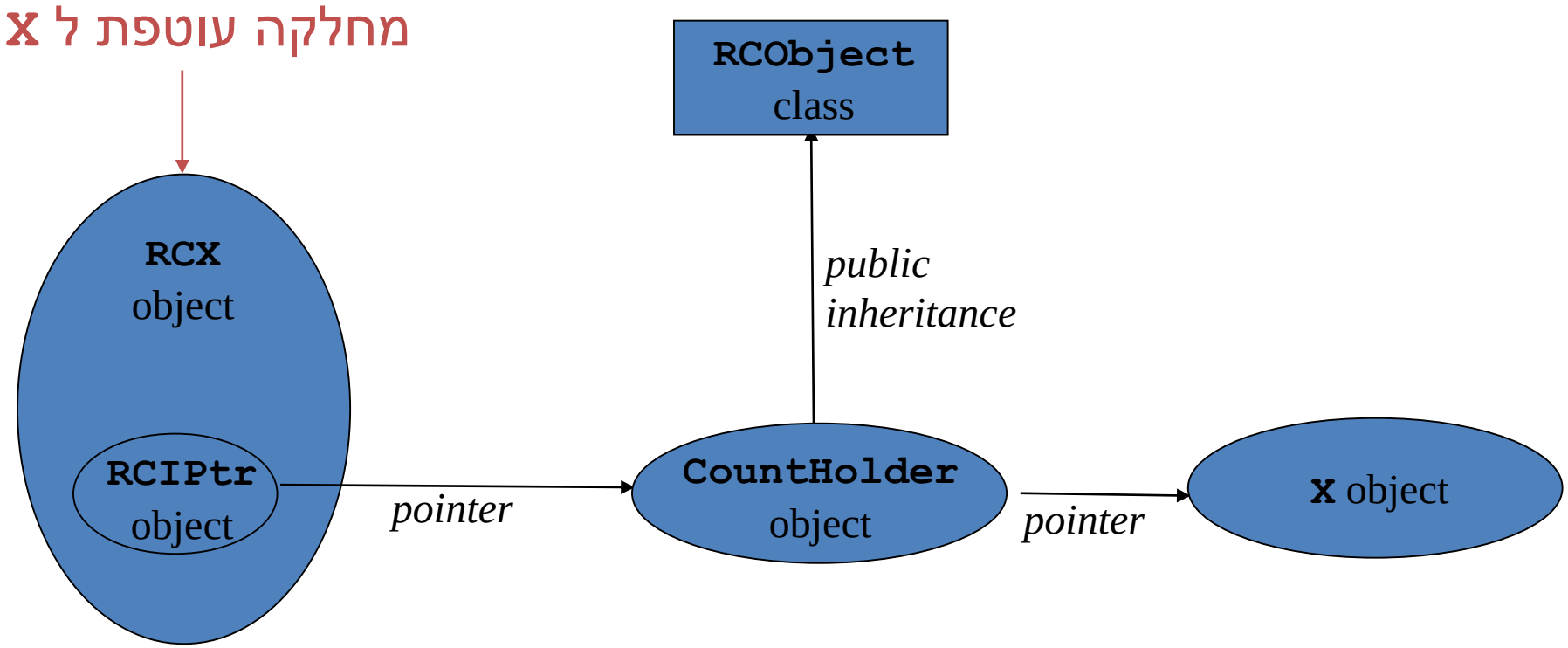


ביצוע Copy-on-Write ע"י `String::operator[]`



ביצוע RC במחלקה שאין אפשרות לשנות את הקוד שלה

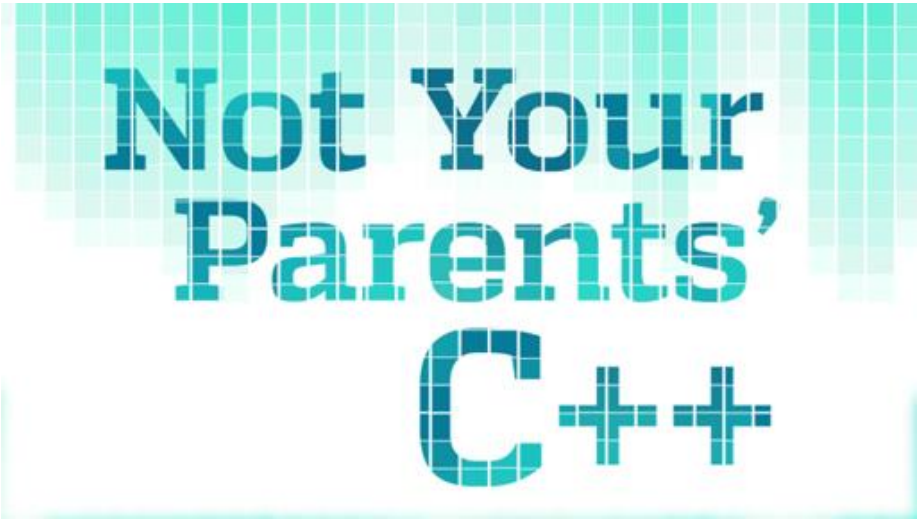
להלן דיזיין למימוש RC במחלקה **x** שאין לנו גישה לקוד המקור שלה:



(indirect לציון I)

סיכום

1. הקוד לביצוע RC הוא יותר מורכב. ביצוע פעולות פשוטות במחלקות שיש בהן RC מערב יותר קריאות לפונקציות ויותר פקודות. לכן, נוסיף RC למחלקה רק כאשר זה משתלם מבחינות אחרות
2. מנגנון ה RC הוא שימושי במחלקה בה מתקיימים התנאים הבאים:
    1. יש שיתוף הצבעות בין אובייקטים. כלומר, הרבה מופעים של המחלקה מצביעים על מספר קטן יחסית של ערכים
    2. הערכים המשותפים הנ"ל הם יחסית יקרים להעתקה עמוקה
    3. מנגנון ה RC נחוץ כדי לפתור את בעיית הבעלות על אובייקטים, כלומר, בכדי שלשאלה "מי רשאי לעשות **`delete`**" לאובייקט יהיה פתרון ברור.



C++11

MODERN REFERENCE COUNTING & SMART POINTERS

C++0x Proudly Presents:

- **`std::shared_ptr`**
- **`std::weak_ptr`**
- **`std::unique_ptr`**

To be continued...