תכנות מונחה עצמים בשפת C++

אוהד ברזילי

אוניברסיטת תל אביב

המצגת מכילה קטעים מתוך מצגת של פרופ' עמירם יהודאי ע"פ הספר:
Object-Oriented Software Construction, 2nd edition,
by Bertrand Meyer (Prentice Hall) .

וכן מתוך הספר:

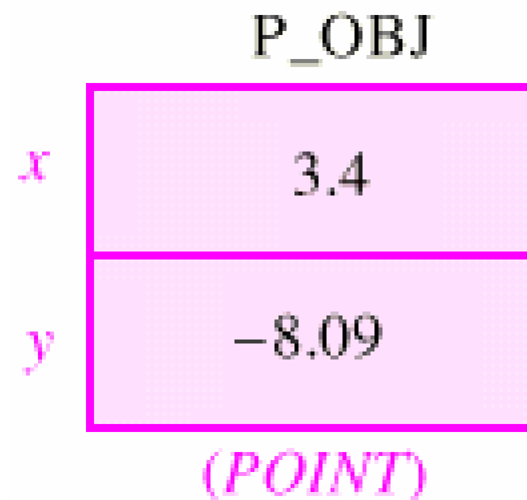
The C++ Programming Language 3rd Edition
by Bjarne Stroustrup

כל הזכויות שמורות למחברים

- עצם הוא מופע זמן-ריצה (instance) של מחלקה
- עצם הוא מופע ישיר של בדיוק מחלקה אחת, המחלקה היוצרת, ואולי מופע של מחלקות נוספות כתוצאה מירושה
- מבנה הנתונים המייצג עצם מכיל שדות נתונים בלבד ללא פונקציות (מתודות)

POINT Class

```
class POINT
{
   float x, y;
   // Routines...
};
```



Simple Book

```
class BOOK1
{
  string title;
  int date, page_count;
};
```

title	"The Red and the Black"
date	1830
page_count	341

(BOOK1)

Writer Class

```
class WRITER
{
  string name, real_name;
  int birth_ year, death_ year;
};
```

name	"Stendhal"
real_name	"Henri Beyle"
birth_year	1783
death_year	1842

(*WRITER*)

עצמים

- נסווג את השדות לסוגים לפי טיפוסיהם:
  - טיפוסים יסודיים (פרימיטיבים): חלק משפת התכנות. כגון: int, char, float, bool (אבל לא string)
  - טיפוסים מורכבים (user defined types)
- שדה מורכב: עצם מוכל או עצם מוצבע?

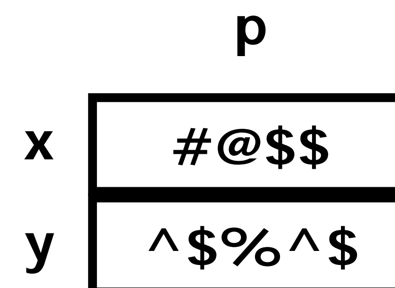
תת עצמים

	(BOOK2)
title	"The Red and the Black"
date	1830
page count	341
	name: "Stendhal" real_name: "Henri Beyle" birth_year: 1783 death_year: 1842 (WRITER)

	(BOOK2)
title	"Life of Rossini"
date	1823
page count	307
	name: "Stendhal" real_name: "Henri Beyle" birth_year: 1783 death_year: 1842 (WRITER)

עצמים, מצביעים והתייחסויות

```
Point p;
Point *pp1, *pp2;
pp1 = &p;
pp2 = pp1;
Point &rp1 = p;
Point &rp2;
```



עצמים, מצביעים והתייחסויות

```
Point p;
Point *pp1, *pp2;
pp1 = &p;
pp2 = pp1;
Point &rp1 = p;
Point &rp2;
```

pp2

*&%$

pp1

!@#$@

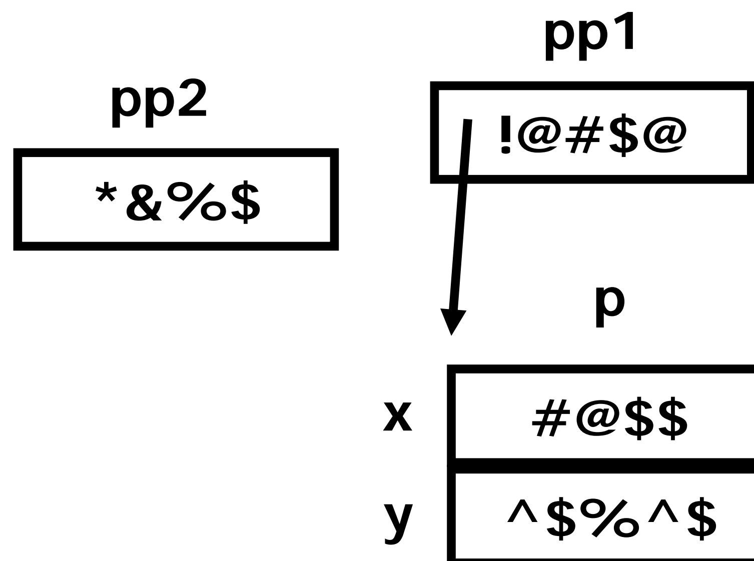
p

x #@$$

y ^$%^$

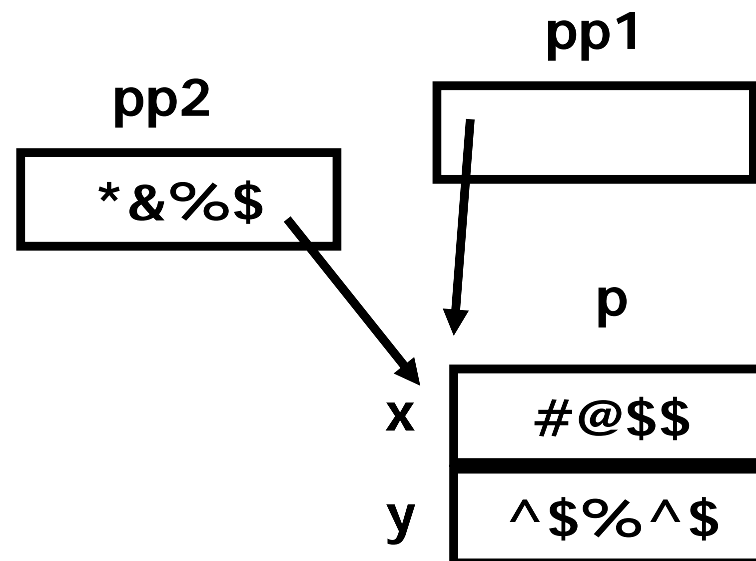
עצמים, מצביעים והתייחסויות

```
Point p;
Point *pp1, *pp2;
pp1 = &p;
pp2 = pp1;
Point &rp1 = p;
Point &rp2;
```



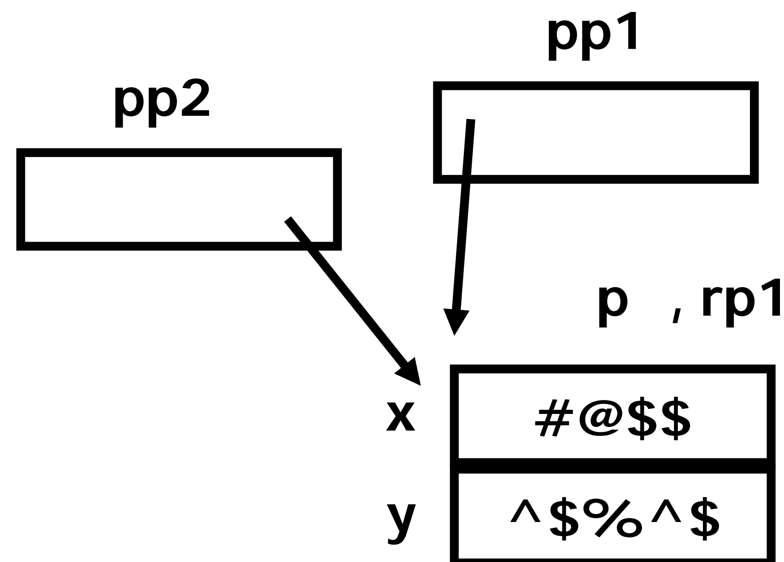
עצמים, מצביעים והתייחסויות

```
Point p;
Point *pp1, *pp2;
pp1 = &p;
pp2 = pp1;
Point &rp1 = p;
Point &rp2;
```



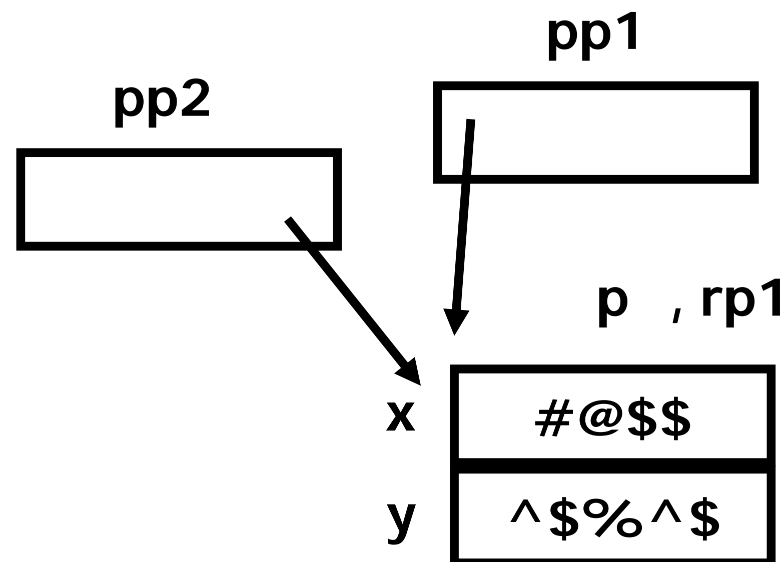
עצמים, מצביעים והתייחסויות

```
Point p;
Point *pp1, *pp2;
pp1 = &p;
pp2 = pp1;
Point &rp1 = p;
Point &rp2;
```



עצמים, מצביעים והתייחסויות

```
Point p;
Point *pp1, *pp2;
pp1 = &p;
pp2 = pp1;
Point &rp1 = p;
Point &rp2;   // ERROR. Unattached reference
```



עצמים, מצביעים והתייחסויות

- נשים לב להבדלים בין מצביע (pointer) להפנייה (reference) בשפת C++:
  - הפנייה הינה שם (לפעמים נוסף) לאובייקט קיים. לא ניתן לשנות את הקישור בין הפנייה ובין העצם שעליו היא מצביעה
  - מצביע הוא ישות עצמאית, עצם בפני עצמו, שעשויה להכיל כתובת של אובייקט קיים. ניתן לעדכן את ערכו של מצביע ובכך להחליף את העצם המוצבע.

עצמים, מצביעים והתייחסויות

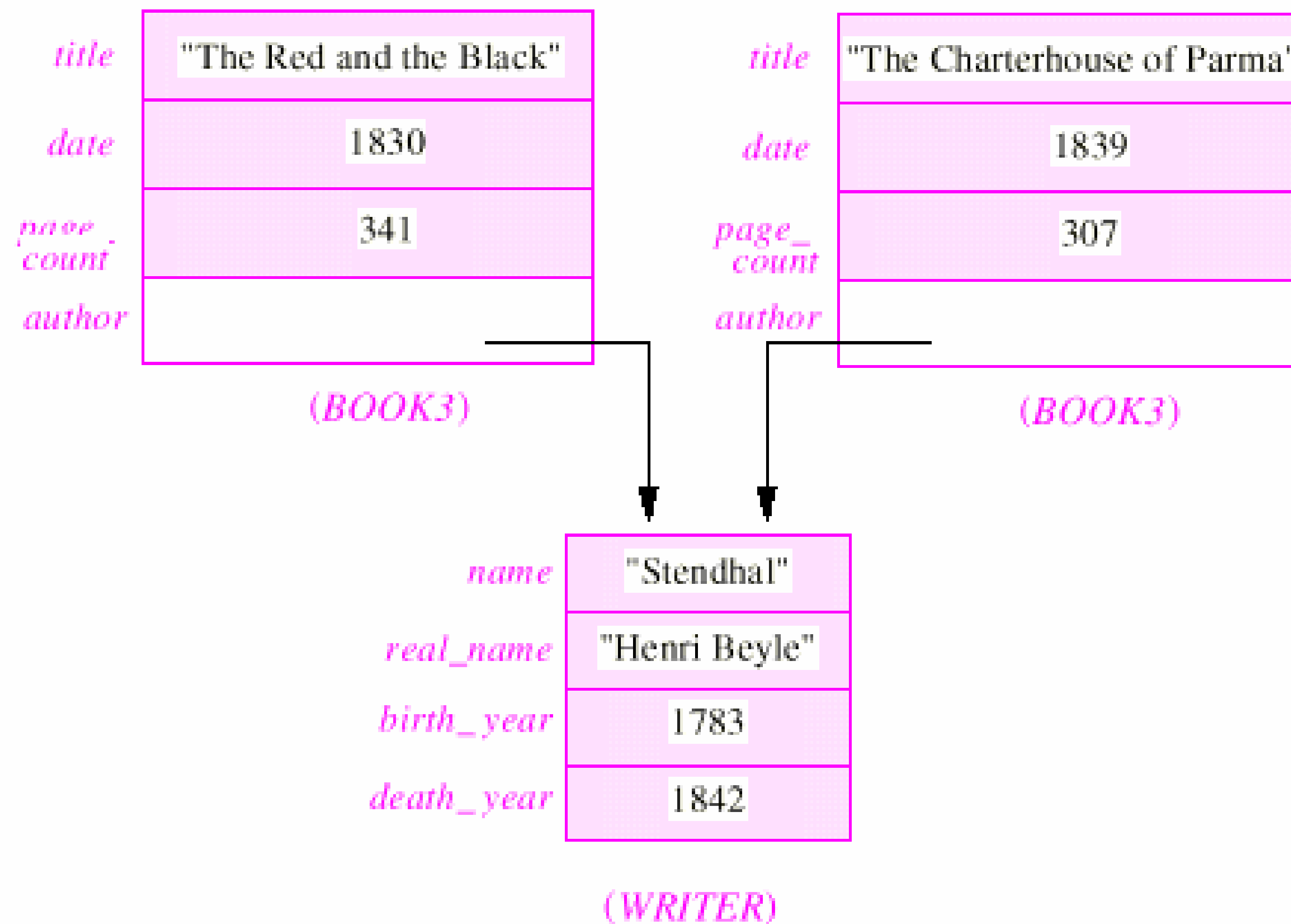
- בעת יצירת עצם מקומי (על המחסנית) אנו בעצם יוצרים גם את העצם וגם את ההפניה עבורו, השם שדרכו ניתן יהיה לגשת אליו
- בעת יצירת הפנייה בלבד אנו נותנים שם נוסף לעצם קיים
- לא נתבלבל עם העברת נתונים לפונקציה by reference. העתקה או אי העתקת העצם לא תלויה בעצם עצמו אלא בהגדרת שיטת ההעברה בפונקציה

- בשפת Java הפנייה היא יישות נפרדת מהעצם המוצבע (ממש כמו מצביעים ב ++C)
- בשפת Java לא ניתן ליצור עצמים מקומיים כלל. עצמים נוצרים רק ע"י הקצאת זיכרון מפורשת בפקודת new
- ניתן לראות זאת כאילו כל המשתנים הלא פרימיטיביים ב java הם מצביעים עם גישה לשדותיהם ע"י נקודה (.) במקום ->

עצם מוכל או מוצבע?

- תת עצמים תופסים יותר זכרון
- עצמים מוצבעים קל יותר לשתף בין מספר ישויות
- ב ++C לא כל המצביעים מאותחלים אוטומטית (ל- null) כך שקיימות 3 אפשרויות:
  - המצביע מתייחס לעצם אחד מסוים
  - מצביע לכתובת 'זבל'
  - המצביע הוא null כלומר לא משויך לעצם מסוים

מוצבע



לא משויך לאף עצם

title	"Candide, or Optimism"
date	1759
page_count	120
author	

(BOOK3)

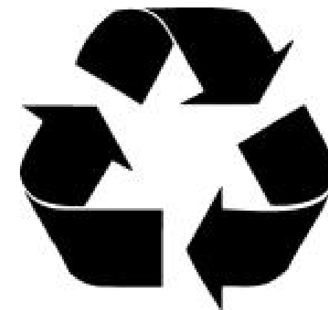
זהות של עצמים

- שני עצמים נפרדים עשויים להיות זהים
- שוויון שדות אינו תנאי מספיק לזהותם של שני עצמים

```
class WRITER {
   string name;
   int birth_year, death_year;
};

class BOOK {
   string title;
   int date;
   WRITER *author;
};
```

מעגלים ביחס הלקוחות

- מחלקה עשויה להיות לקוחה של עצמה
- נבדיל בין מעגל של עצמים ומעגל של מחלקות



```
class PERSON1
{
   string name;
   PERSON1 *loved_one, *landlord;
};
```

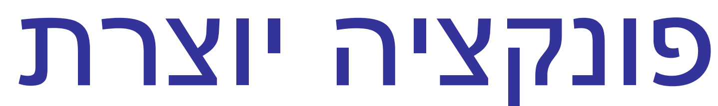
מצביע לעצמו

name "Almaviva"
landlord
loved_one
(PERSON1)

name "Figaro"
landlord
loved_one
(PERSON1)

"Susanna" name
landlord
loved_one
(PERSON1)

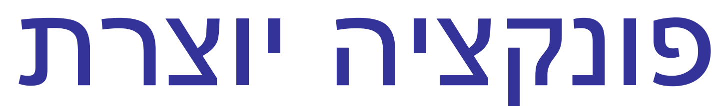
פונקציה יוצרת



- יצירת ואתחול עצמים מפורשת
- כאשר בפונקציה כלשהי מגיעים להצהרה `T obj` קורים הדברים הבאים:

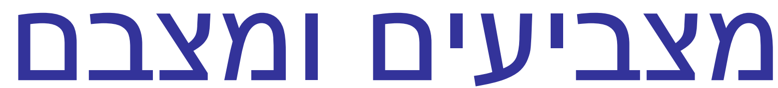
1. מוקצה מקום על המחסנית עבור עצם מטיפוס T (עבור כל אחד משדותיו)
2. שדותיו מאותחלים לפי כללי אתחול ברירת המחדל (בהמשך)
3. השם obj משויך לעצם שנוצר

פונקציה יוצרת



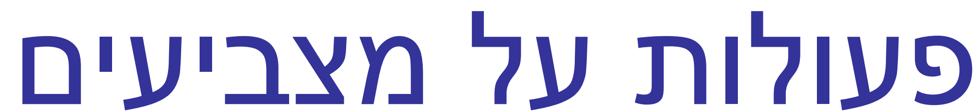
- יצירת ואתחול עצמים מפורשת
- כאשר בפונקציה כלשהי מגיעים לשורה `T *x = new T` קורים הדברים הבאים:

1. מוקצה מקום עבור עצם מטיפוס T (עבור כל אחד משדותיו)
2. x מקבל את כתובתו של העצם שהוקצה

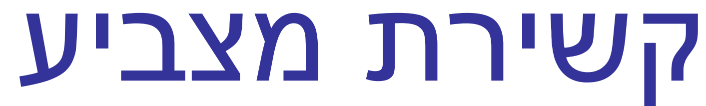


מצביעים ומצבם

- מצביע יכול להיות קשור או לא קשור לעצם
- מצביע p יקשר לעצם ע"י:
  - p=new SomeClass(...)
  - או ע"י p=p1 , כאשר p1 הוא מצביע קשור
- p יהפך ללא קשור ע"י:
  - p=null
  - או ע"י p=p1 כאשר p1 הוא null
- הפעלת מתודה דרך מצביע שאינו קשור תגרום לטעות זמן ריצה



- השמה של מצביעים אינה מבצעת העתקה של האובייקט המוצבע והיא מותרת אם המצביע אינו מוגדר כ const (ראה שיעור 3)
- ההשוואה בין מצביעים p1==p2 תחזיר true רק אם הם מצביעים על אותו עצם או ששניהם null
- ההשוואה בין עצמים obj1==obj2 תעבוד רק אם הוגדר עבור המחלקה T של obj1 אופרטור השוואה
- ההשמה p=null שימושית כדי לבדוק בהמשך האם p!=null (שכן למצביע עלול להיות גם ערך 'זיבלי')




```
class PERSON2
{
   string name;
   PERSON2 *loved_one, *landlord;

   /** Attach the loved_one field of current
       object to l. */
   void set_loved (PERSON2 *l)

   {
       loved_one = l;
   }
}
```

קשירת מצביע

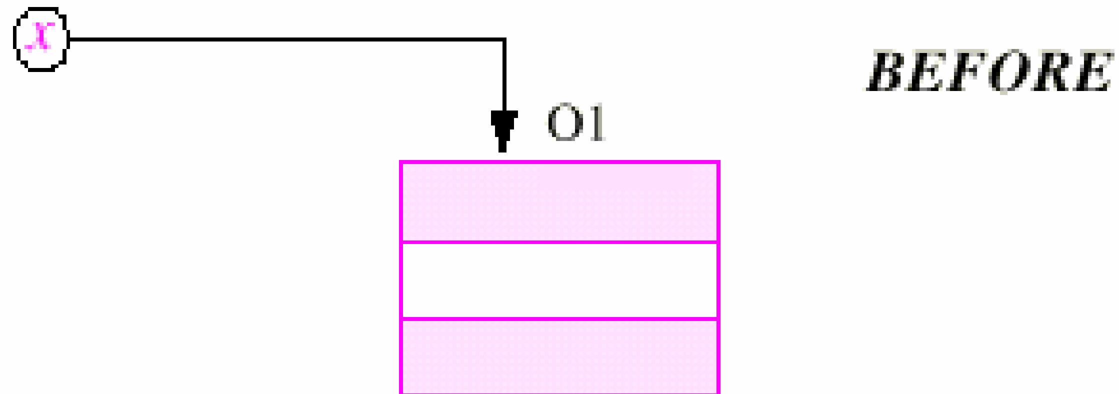
a
O1
name "Almaviva"
landlord
loved_one
(PERSON2)
r
O2
name "Susanna"
landlord
loved_one
(PERSON2)
O3
name "Rosina"
landlord
loved_one
(PERSON2)

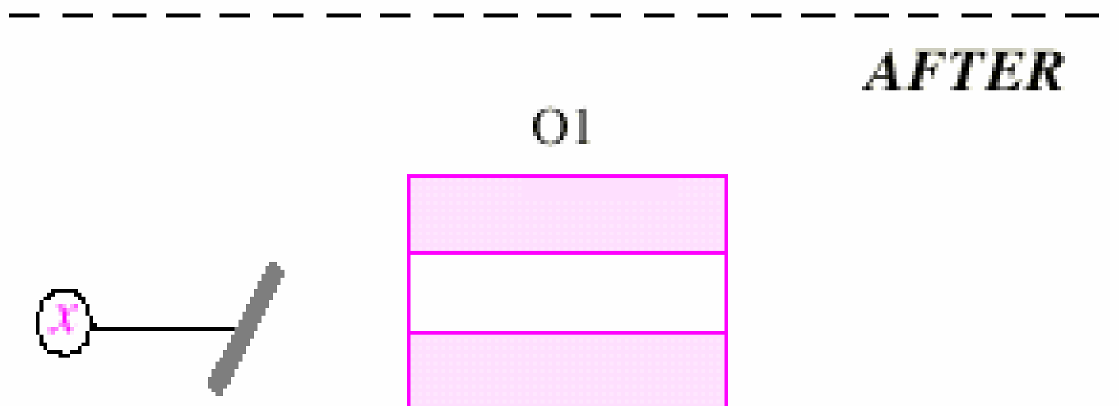
a
O1
name "Almaviva"
landlord
loved_one
(PERSON2)
r
O2
name "Susanna"
landlord
loved_one
(PERSON2)
O3
name "Rosina"
landlord
loved_one
(PERSON2)

Before *a->set_loved (r)* *After*

איפוס מצביע (ללא שחרור!)

$$x = null$$





אתחול, השמה והריסה ב C++

אתחול ברירת מחדל

- אם הוגדר משתנה (עצם) עם ערך אתחול אזי זה יהיה ערכו הראשוני.

  לדוגמא: `int i = 5;`

- עצמים גלובלים, עצמים שהוגדרו בתוך מרחב שמות ועצמים סטטים מקומיים יאותחלו ל-0 אם לא הוגדר אחרת

  לדוגמא:

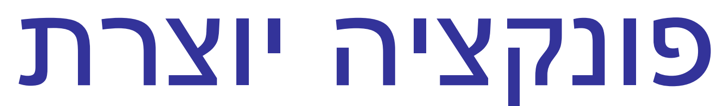
```
int a; // means "int a = 0;"
double d; // means "double d = 0.0;"
```

אתחול ברירת מחדל

- לעצמים מקומיים (משתני מחסנית, משתנים אוטומטיים), ועצמים שהוקצו דינאמית אין אתחול ברירת מחדל

דוגמא:

```
void f()
{
   int x; // x does not have a well-defined value
   // ...
}
```



- במחלקות שנכתוב ניתן (ורצוי!) להחליף את אתחול ברירת המחדל בפונקציה
- פונקציה זו נקראת הבנאי (constructor) של המחלקה
- ניתן להעמיס אותה
- ניתן להעביר לה פרמטרים
- שמה כשם המחלקה שאותה היא מאתחלת ואין לה ערך מוחזר

בנאים ל CPoint

```
class CPoint
{
//...
public:
   CPoint()
   {
        m_x = 0;
        m_y = 0;
   }
   CPoint(int x, int y)
   {
        m_x = x;
        m_y = y;
   }
//...
}
```

בנאי יותר טוב ל CPoint

```
class CPoint
{
//...
public:
  CPoint(int x=0, int y=0)
  {
      m_x = x;
      m_y = y;
  }
//...
}
```

עוד על בנאים

- פונקצית אתחול שהוגדרה כ private לא תאפשר ליצור באמצעותה עצמים
- למחלקה שלא נכתבו עבורה בנאים כלל מספק המהדר (הקומפיילר) בנאי ברירת מחדל (default constructor) זהו בנאי חסר פרמטרים שאינו עושה דבר
- מרגע שנכתב בנאי <u>כלשהו</u> למחלקה לא ניתן עוד להשתמש בבנאי ברירת המחדל
- יש מקרים שבהם נהיה חייבים להשתמש בבנאי חסר פרמטרים ולכן נגדיר כזה לכל מחלקה

- בכל פעם שעצם מסיים את תפקידו, יש לבצע פעולת הריסה מסודרת
- לדוגמא: בסיום חייו של עצם שלאחד משדותיו הוקצה זכרון דינמית יש צורך לשחרר הן את הזיכרון של העצם עצמו והן את הזכרון שהוקצה עבור שדותיו
- לכל עצם נגדיר פונקציית פירוק ששמה כשם המחלקה בצרוף הסימן טילדה (~)
- המפרק הוא יחיד, הוא יקרא אוטומטית לא ניתן לקרוא לו מפורשות ולא ניתן להעביר לו פרמטרים

(destructors) מפרקים

```
class Name {
   const char *s;
// ...
};

class Table {
   Name *p;
   size_t sz;
public :
   Table (size_t s = 15)              // constructor
   { p = new Name[sz = s]; }

   ~Table() { delete[] p ;}           // destructor

   Name *lookup(const char *);
   bool insert (Name *);
};
```

דוגמא

- כל בנאי (גם בנאי ברירת המחדל) קורא לבנאים של כל שדותיו

```
struct Tables {
  int i;
  int vi[10];
  Table t1;
  Table vt [10];
};

Tables tt; // calls t1(15)
```

- משתני const ומשתני reference יש לאתחל

```
struct X {
  const int a;
  int &r;
};
X x ;     // error: no default
          // constructor for X
```

זמני יצירה והריסה של עצמים

- עצמים מקומיים (אוטומטיים, מחסנית)
  - נוצרים בכל פעם שמגיעים לשורת ההצהרה עליהם
  - נהרסים בכל פעם שיוצאים מהבלוק שבו הוגדרו
- עצמים שהוקצו דינמית
  - נוצרים בפקודת new
  - נהרסים בפקודת delete
- עצמים שהם שדות של עצם מכיל
  - נוצרים כאשר העצם המכיל נוצר
  - נהרסים כאשר העצם המכיל נהרס

זמני יצירה והריסה של עצמים

- אברים במערך
  - נוצרים בכל פעם שהמערך נוצר
  - נהרסים בכל פעם שהמערך נהרס
- משתנה סטטי מקומי (בפונקציה)
  - נוצר כאשר מגיעים לשורת ההצהרה עליו בפעם הראשונה
  - נהרס בסיום התוכנית כולה
- משתנה גלובלי, משתנה מרחב שמות ומשתנה סטטי של מחלקה
  - נוצר ב"תחילת התוכנית"
  - נהרס בסוף התוכנית

זמני יצירה והריסה של עצמים

- עצם זמני
  - נוצר במהלך חישוב הביטוי שבו הוא מופיע
  - נהרס בסיום חישוב הביטוי העוטף

- לא נדון בעצמים שמוקצים ע"י אופרטור הקצאה ובשדות של union (שאולי אין להם כלל בנאי ומפרק)

זמני יצירה והריסה של משתנים מקומיים

```
void f(int i)
{
   Table aa;
   Table bb;
   if (i>0) {
       Table cc;
   // ...
   }

   Table dd;
   // ...
}
```

- aa , bb , ו- dd יווצרו (בסדר זה) בכל קריאה ל f()
- dd , bb ו- aa יהרסו (בסדר זה) בכל יציאה מ f()
- אם i>0 אז cc יווצר לפני dd ויהרס לפני bb

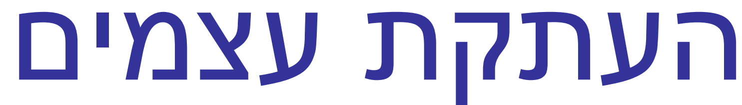
העתקת עצמים

- הבנאי יקרא פעמיים (עבור t1 ו- t3) ואילו המפרק 3 פעמים (עבור כולם)
- t2 ו- t1 חולקים את אותו מערך מכיוון שהאתחול ביצע העתקה נאיבית (שדה-שדה)
- לא ניתן לשחרר את המערך שהוקצה ביצירת t3 מכיוון שהוא נדרס ע"י ההשמה t3=t2
- יש צורך לשלוט הן על תהליך האתחול בעזרת העתקה והן על תהליך ההשמה בעזרת העתקה

```
void h()
{
   Table t1;

   // copy initialization
   // same as Table t2(t1);
   Table t2 = t1;

   Table t3;

   // copy assignment
   t3 = t2;
}
```

התוכנית שגויה !

העתקת עצמים

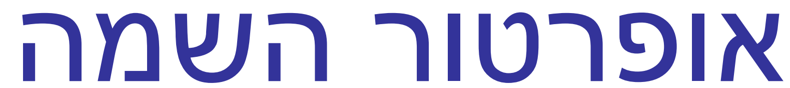


- יצירת עצם תוך כדי אתחולו ע"י סימן "=" היא סוכר תחבירי לבנאי
- בנאי זה מכונה "בנאי העתקה" (copy constructor)
- השמה של עצם לתוך עצם קיים ע"י סימן "=" היא סוכר תחבירי לאופרטור ההשמה
- השמה ואתחול הן שתי פעולות שונות !

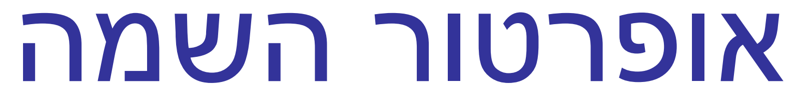
```
class Table {
// ...
   // copy constructor
   Table(const Table &);

   // copy assignment
   Table &operator=(const Table &);
};
```

- **בנאי ההעתקה נקרא בעת יצירת עצם חדש ע"י ארגומנט מאותו טיפוס, העברה של נתון by value והחזרה של נתון by value**

```
// copy constructor
Table::Table(const Table& t) // by reference!
{
  p = new Name[sz=t.sz];
  for(int i=0 ; i<sz ; i++)
     p[i] = t.p[i];
}
```

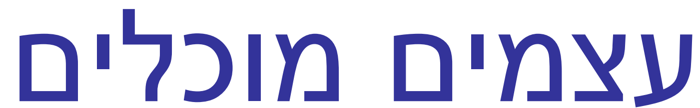


1. הגני מפני השמה עצמית
2. שחררי את האובייקט הישן (על מרכיביו)
3. אתחלי אותו
4. העתיקי שדה-שדה את האובייקט החדש



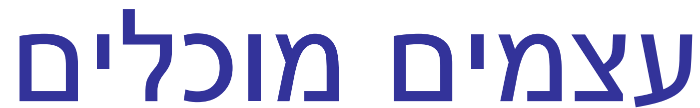

```
// assignment operator
Table &Table::operator=(const Table& t)
{
  if (this!= &t) // beware of t = t
  {
     delete []p;
     p = new Name[sz=t.sz];
     for(int i=0 ; i<sz ; i++)
          p[i] = t.p[i];
  }
  return *this;
}
```

תקציר הפרקים הקודמים...



- בתוך גוף הבנאי (אחרי הסוגריים המסולסלים) העצם כבר בנוי. לאתחל עכשיו את שדותיו יכול להיות מאוחר מדי או לחלופין לא יעיל

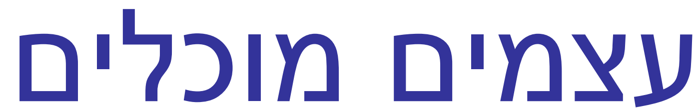
```cpp
class Club {
  string name;
  Table members;
  Table officers;
  Date founded;
  // ...
  Club(const string& n, Date fd);
};
```



עצמים מוכלים

- שורת האתחול מאפשרת לאתחל שדות לפני התחלת ביצוע הבנאי
- השדות יאותחלו לפי סדר הגדרתם במחלקה (ולא לפי סדר הופעתם בשורת האתחול)

```
Club::Club(const string& n , Date fd) : name(n),
   members(), officers(), founded(fd)
{
    // ...
}
```



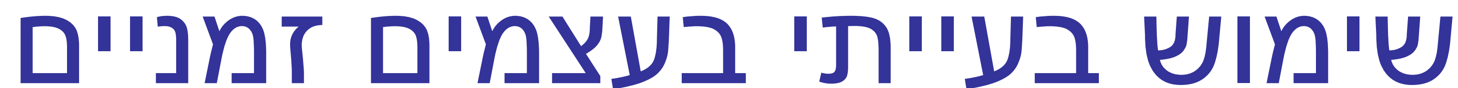
- ניתן להשמיט שדות שהבנאי שלהם לא מקבל ארגומנטים

```
Club::Club(const string& n , Date fd)
: name(n), founded(fd)
{
    // ...
}
```

- לאחר ביצוע ה destructor עבור עצם מסוים יקראו ה- destructors של כל אחד משדותיו המוכלים.
המפרקים יקראו בסדר הפוך לסדר יצירת העצמים המוכלים

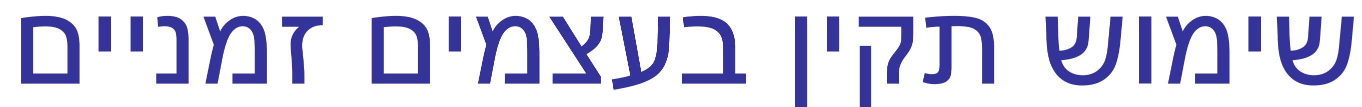
- עצמים ללא בנאי ברירת מחדל (ללא בנאי חסר פרמטרים), הפניות או קבועים חובה לאתחל בשורת האתחול

```
class X {
   const int i;
   Club c;
   Club& pc;
   // ...
   X(int ii, const string& n, Date d , Club& c)
   : i(ii), c(n,d), pc(c) { }
};
```



שימוש בעייתי בעצמים זמניים

```
void f(string& s1 , string& s2 , string& s3)
{
   const char *cs = (s1+s2).c_str();
   cout << cs;

   if(strlen(cs=(s2+s3).c_str())<8
                                     && cs[0]==´a ´)
   {
       // cs used here
   }
}
```




```
void f(string& s1 , string& s2 , string& s3)
{
   cout << s1 + s2;
   string s = s2 + s3;
   if(s.length()<8 && s[0]==´a ´)
   {
      // use s here
   }
}
```

עוד שימוש תקין בעצמים זמניים

```
void g(const string&, const string &);

void h(string& s1 , string& s2)
{
   const string &s = s1 + s2;
   string ss = s1 + s2;
   g(s,ss); // we can use s and ss here
}
```

```
void f(Shape &s , int x , int y)
{
   // construct Point to pass to Shape::move()
   s.move(Point(x ,y));

   // ...
}
```

זה חינם !

- בשפת ++C עבור כל מחלקה מוגדר ע"י המהדר:
  - בנאי ברירת מחדל (בנאי ריק)
    - אלא אם הוגדר בנאי אחר כלשהו
  - בנאי העתקה
  - אופרטור השמה
- שלושתם בעלי הרשאת public
- את שלושתם ניתן לדרוס ע"י כתיבת גרסאות משלנו (בדר"כ רלוונטי עבור מחלקות המכילות מצביעים)

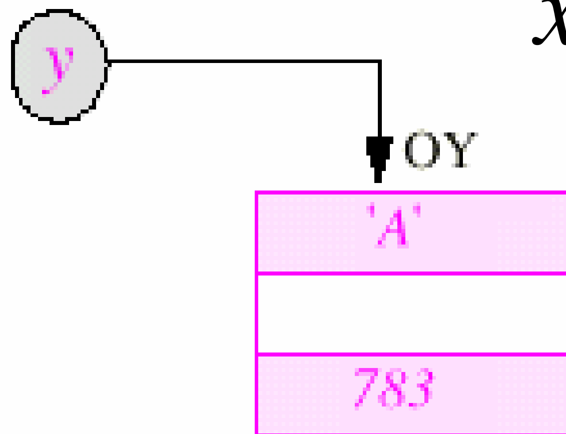
בשפת ++C השפה אינה מספקת
פעולות יסודיות נוספות פרט לאלו
שפורטו בשקף הקודם

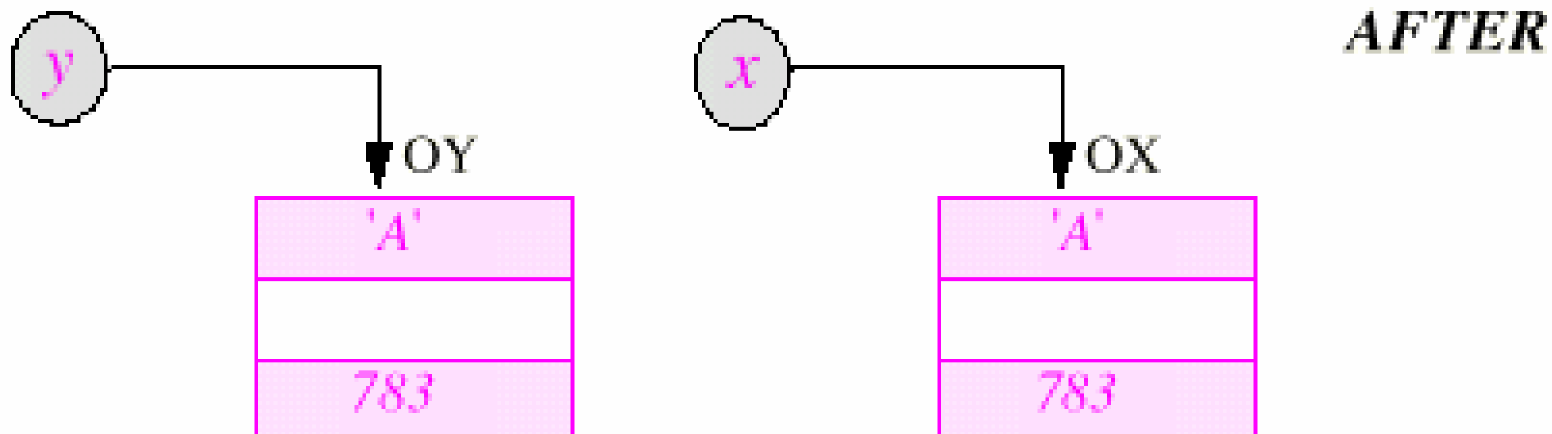
שיבוט העתקה והשוואה

- clone - הינה פעולה אשר יוצרת עותק זהה לזה של העצם המשובט ומחזירה מצביע אליו
- equal - השוואה בין שני עצמים שדה-שדה תמומש ב C++ בדר"כ ע"י האופרטור '=='. פעולה זו על המתכנת לממש בעצמו
- copy – העתקה שדה-שדה. ממומשת ב C++ ע"י אופרטור ההשמה
- ב copy לעומת clone, לא נוצר עצם חדש
- בהקשר הזה ניתן לדבר על deep_copy, deep_equal ו- deep_clone

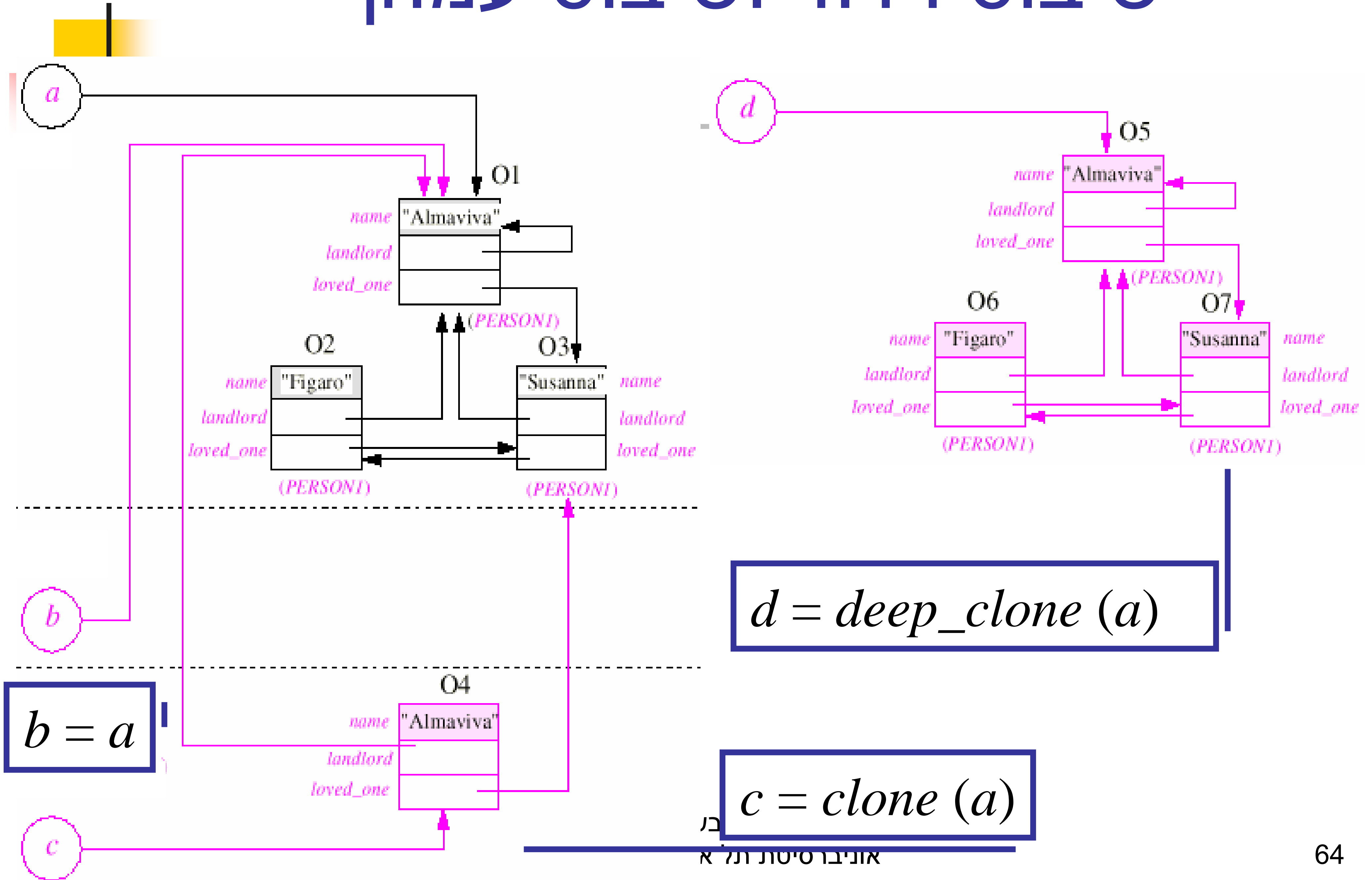
שיבוט עצמים

$$x = clone\ (y)$$



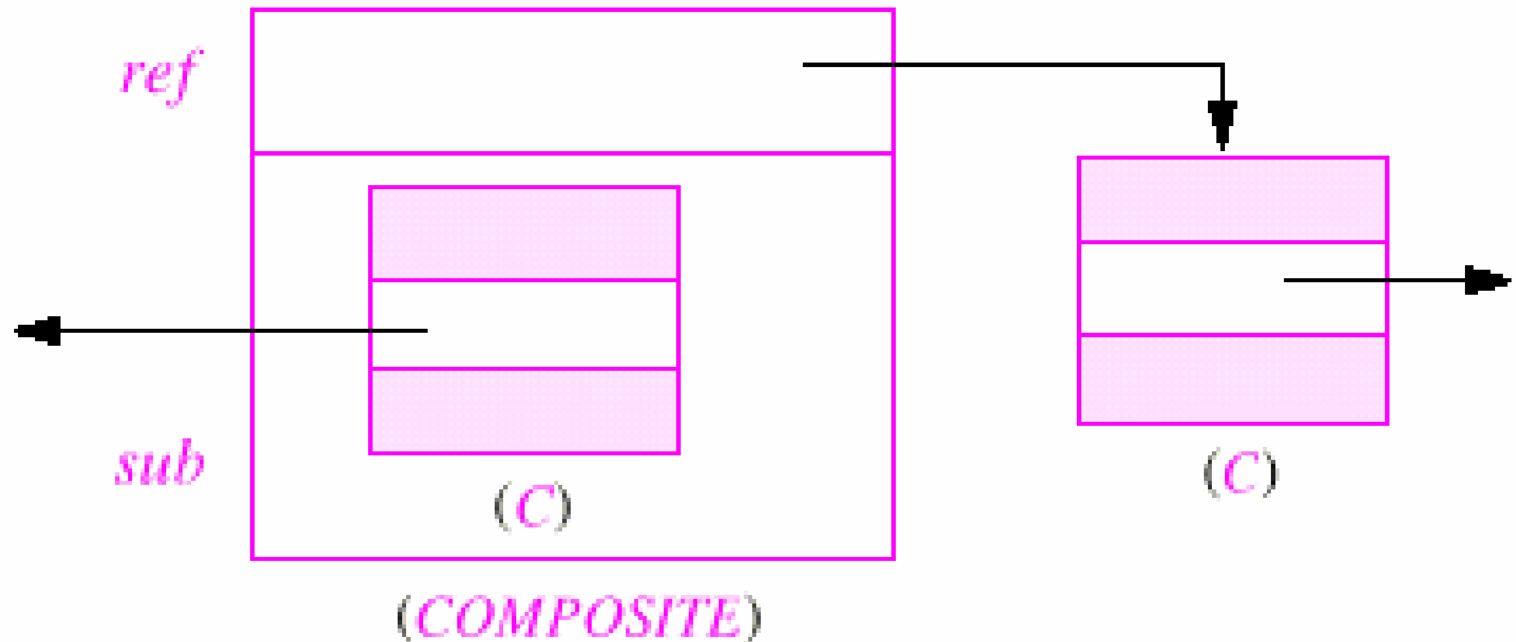


שיבוט רדוד ושיבוט עמוק



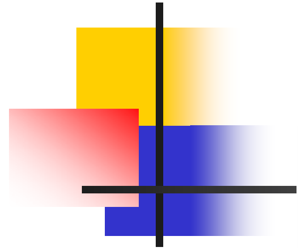
עצם מוכל (expanded type)

```
class COMPOSITE {
  C *ref;
  C sub;
}
```



- **יעילות**
  - גישה לשדות מוכלים שלא דרך dereference של מצביע
- **מודל טוב יותר**
  - מצביע למחלקה S פירושו שהלקוח "יודע על" S
  - עצם כעצם מוכל מעיד על כך שהלקוח מכיל S
  - בפרט, הכלה מרמזת על אי-שיתוף
- **תמיכה אחידה בטיפוסים פרימיטיבים**
  - עצמים מכילים את הטיפוסים היסודיים עצמם ולא מצביע אליהם

אובייקטים מורכבים

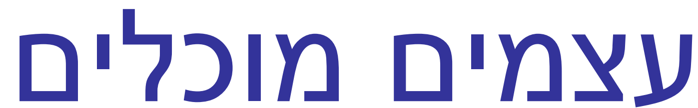


k KEYBOARD1
c CPU1
m MONITOR1
n
(WORKSTATION)

k KEYBOARD1
c CPU2
m MONITOR2
n
(WORKSTATION)

k KEYBOARD1
c CPU3
m MONITOR3
n
(WORKSTATION)

(NETWORK)



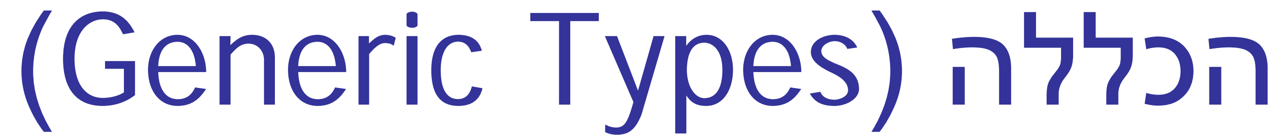
עצמים מוכלים

- לא יתכנו מעגלי ספק-לקוח כאשר הספק הוא עצם מוכל (expanded type)
- יצירה והריסה של האובייקט המוכל תלויה בזמני היצירה וההריסה של העצם המכיל
- בהשמת העצם העוטף לא יופעל בהכרח אופרטור ההשמה של כל אחד משדותיו המוכלים
- כאשר עצם מכיל מצביעים לעצמים אחרים יש לשקול מתי לשחרר את הזכרון של עצמים אלו (למשל אם הם משותפים לכמה עצמים)

#
מתי לשחרר

- לבעיה זו כמה גישות שונות:
- הקצת? תשחרר!
  - כל הקצאה בבנאי תשוחרר במפרק.
  - כל העתקה של עצם אחר (בבנאי העתקה או ע"י אופרטור השמה) תשכפל מידע מוצבע
- ספירת הפניות
  - כל העתקה של עצם אחר המכיל מצביעים תעתיק רק את המצביע ותעדכן מונה.
  - כל שחרור של עצם המכיל מצביעים לא תשחרר את הזכרון המוצבע אלא רק תוריד את המונה ב-1
  - עצם שהוקצה דינמית ושמונה ההצבעות אליו ירד ל-0 ישוחרר

(templates)



הכללה (Generic Types)

- אחרי שתארנו ADT בעזרת טיפוס כללי אך טבעי לבנות מחלקה עם טיפוס כללי
- לכאורה, השימוש בטיפוסים סטטים מיותר – ואולם הוא לצורכי קריאות ופשטות
- הכללה מספקת מנגנון לשימוש חוזר בקוד (ע"י ביטול התלות בטיפוס)
- כתיבה נכונה לא מניחה דבר על הטיפוס הפרמטרי, ואולם הנחות מובלעות המופיעות בקוד, יגרמו לטעות קומפילציה עבור טיפוסים מסוימים

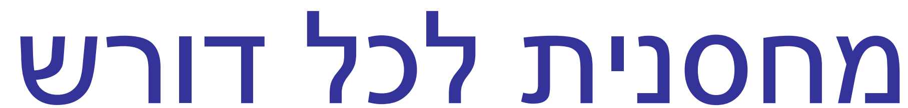
מחסנית של chars

```
class Stack {
   char* v;
   int max_size;
   int top;
public:
   Stack(int s);        // constructor
   ~Stack();            // destructor
   void push(char);
   void pop();
   char top();
};
```

מחסנית כללית

```
template <class T>
class Stack {
   T* v;
   int max_size;
   int top;
public:
   Stack(int s);          // constructor
   ~Stack();              // destructor
   void push(T);
   void pop();
   T top();
};
```

```cpp
template<class T>
void Stack<T>::push(T c)
{
  if(top == max_size)
     // Error Handling Code...
  v[top] = c;
  top = top + 1;
}
```




```
Stack<char> sc; // stack of characters
Stack<complex> scplx; // stack of complex numbers
Stack<list<int>> sli; // stack of list of integers

void f()
{
   sc.push(´c´);

   if(sc.pop() != ´c ´)
        // Logical Error!

   scplx.push(complex(1,2));

   if(scplx.pop() != complex(1,2))
        // Logical Error!
}
```

- עבור כל יצירה של מחלקה עם טיפוס אקטואלי (char, complex, list<int>) יוצר הקומפיילר עותק של התבנית (מחלקה) עבור טיפוס זה
- על המחלקה הנוצרת חלים כל כללי התאמת הטיפוסים הרגילים של ++C:
  - `sc.push(4.5)` טעות קומפילציה

המערך מהדור הישן

```
struct Entry{
   string name;
   int number;
};

Entry phone_book[1000];

void print_entry(int i) // simple use
{
   cout << phone_book[i].name << ´ ´ <<
        phone_book[i].number << ´\ n ´;
}
```

המערך של הדור החדש vector<T>

```
vector<Entry> phone_book(1000);

// simple use, exactly as for array
void print_entry(int i)
{
   cout << phone_book[i].name << ´ ´
              << phone_book [i].number << ´\ n ´;
}

void add_entries(int n) // increase size by n
{
   phone_book.resize(phone_book.size()+n );
}
```

מה עושה הקוד הבא?

```
void f(vector<Entry>& v)
{
  vector<Entry> v2 = phone_book;
  v = v2;
  // ...
}
```

vector

- השימוש ב vector פותר את בעיית ניהול הזכרון, אבל הוא לא פותר את בעיית הגלישה מגבולות המערך
- האופרטור [] לא מבצע בדיקה כי הכניסה המבוקשת קיימת במערך (מדוע?)
- המתודה at(int) מספקת שרות זה
- כדי לבצע בדיקה בכל גישה ניתן להגדיר מחלקה עוטפת (Vec) שבה יוגדר האופרטור [] ע"י קריאה ל – at(int). טכניקה זו נקראת האצלה (delegation)

האצלת סמכויות (delegation)

```
template<class T>
class Vec {
   vector<T> rep;
public:
   Vec() : rep() {}
   Vec(int s) : rep(s) {}
   int size() const {return rep.size();}
   //for all methods of vector<T>...

   T& operator[](int i)
   { return rep.at(i); } // rangechecked

   const T& operator[](int i) const
   { return rep.at(i); } // rangechecked
};
```

עוד על vector

- פעולות על המחלקה vector עם דוגמאות מצויינות באתר MSDN:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcstdlib/html/vclrfvectormembers.asp

- או חפשו ב Google: vector c++ msdn