



# תכנות מונחה עצמים בשפת C++ ירושה

---

אוהד ברזילי  
אוניברסיטת תל אביב



# ירושה

---

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

כל הזכויות שמורות למחברים



# ירושה

---

- מספקת שימוש חוזר ויכולת הרחבה
- יחסים בין מחלקות: מחלקה יכולה להיות הרחבה, צמצום או הרכבה של מחלקה או מחלקות אחרות
- דוגמא: מצולעים ומלבנים

# class POLYGON

```
/**
 * @inv: same_count_as_implementation:
 *           count == vertices.size()
 * @inv: at_least_three: count >= 3
 */
class POLYGON
{
public:
    /** constructor */
    POLYGON(int n) : count(n) { /* ... */ }

    /** Number of vertices */
    const int count;

    /** Length of perimeter */
    float perimeter() const { /* ... */ }
```



# class POLYGON

---

```
/** Display polygon on screen */  
void display() const { /* ... */ }
```

```
/** Rotate by angle around center */  
void rotate(POINT center, float angle)  
{ /* ... */ }
```

```
/** Move by a horizontally, b vertically */  
void translate(float a, float b) { /* ... */ }
```

```
private:
```

```
/** Successive points making up polygon */  
list<POINT> vertices;  
};
```

# דוגמא למימוש המתודות

---

```
void POLYGON::rotate(POINT center, float angle)
{
    for(list<POINT>::iterator iter = vertices.begin();
        iter != vertices.end(); iter++)
    {
        POINT& p = *iter;
        p.rotate(center, angle);
    }
}
```

# דוגמא למימוש המתודות

```
/** Length of the perimeter */
float POLYGON::perimeter() const
{
    POINT curr, prev;
    float rslt = 0.0;

    list<POINT>::const_iterator iter = vertices.begin();
    prev = curr = *iter;
    iter++;
    while(iter != vertices.end())
    {
        curr = *iter;
        rslt += curr.distance(prev);
        prev = curr;
        iter++;
    }
    rslt += vertices.back().distance(vertices.front());
    return rslt;
}
```

# POLYGON מ יורש RECTANGLE

```
/** class RECTANGLE
 * @inv: four_count: count == 4
 * @inv: first_side:
 *     vertices.begin()->distance(*(vertices.begin()++)) == side1
 * 3 more like this for the other sides...
 */
class RECTANGLE : public POLYGON
{
public:
    /** set up rectangle centered at center,
     * with side lengths s1 and s2 and orientation angle
     */
    RECTANGLE(POINT center, float s1, float s2, float angle)
    { /* ... */ }

    /** Length of the perimeter. Redefinition
     */
    float perimeter() const
    {
        return 2 * (side1 + side2);
    }
}
```



# POLYGON יורש מ RECTANGLE

...

**private:**

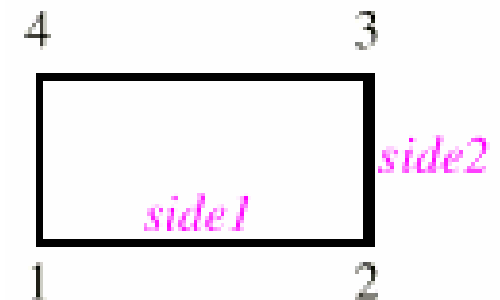
```
/** The two side lengths */
```

```
float side1, side2;
```

```
/** Length of diagonal */
```

```
float diagonal;
```

```
};
```





# מלבן ומצולע

---

○ כל תכונות המצולע (המתודות) עדיין תקפות לגבי מלבן:

● היקף (שהוגדרה מחדש)

● קודקודים

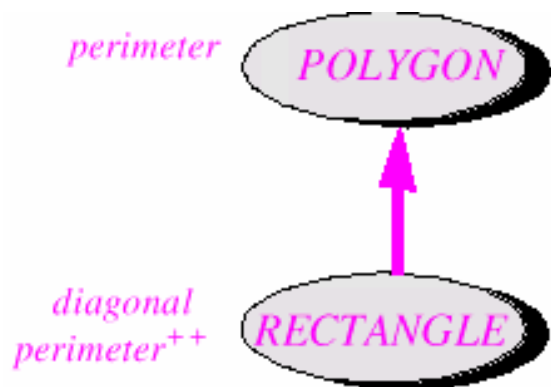
● סיבוב

● הזזה

● ...

# מונחי ירושה

- אם B יורש מ-A אזי A הוא ההורה של B ו-B נקרא צאצא (ישיר) של A
- בטרמינולוגיה נוספת A נקרא מחלקת הבסיס (base) ו-B נקרא מחלקה נגזרת (derived)
- בטרמינולוגיה שלישית A נקרא מחלקת-על (super) ו-B תת מחלקה (subclass)
- ב JAVA מקובל לאמר כי B מרחיבה את A (extends)



## הגדרות נוספות

---

- כל תכונה של מחלקה (שדה \ מתודה) הוגדרה במחלקה או נורשה ממחלקת בסיס כלשהי
- השמורה (class invariant) של המחלקה היא תוצאת ה AND הלוגי של שמורת המחלקה עצמה עם שמורת מחלקת (או מחלקות) הבסיס אם קיימת (קיימות)
- בנאים אינם עוברים בירושה



# רב צורתיות - polymorphism

---

- היכולת של מצביע להתייחס בזמן ריצה לעצמים ממחלקות שונות
- תכונה זו מושגת בעזרת ירושה
- העצם המוצבע אינו משנה את טיפוסו. מצביע לטיפוס מסוים עשוי להצביע בפועל לטיפוס ממחלקה נגזרת
- כאשר עצם מטיפוס מסוים מקבל השמה של עצם מטיפוס נגזר, אין כאן polymorphism מלא, אלא פריסה (slicing)

# דוגמא

```
void f(POLYGON p, RECTANGLE r, TRIANGLE t)
{
    POLYGON *pp;
    RECTANGLE *pr;
    TRIANGLE *pt;

    pp = &p;
    pp = &r;
    pp = &t;

    pr = &p;    // ERROR
    pr = &r;
    pr = &t;    // ERROR

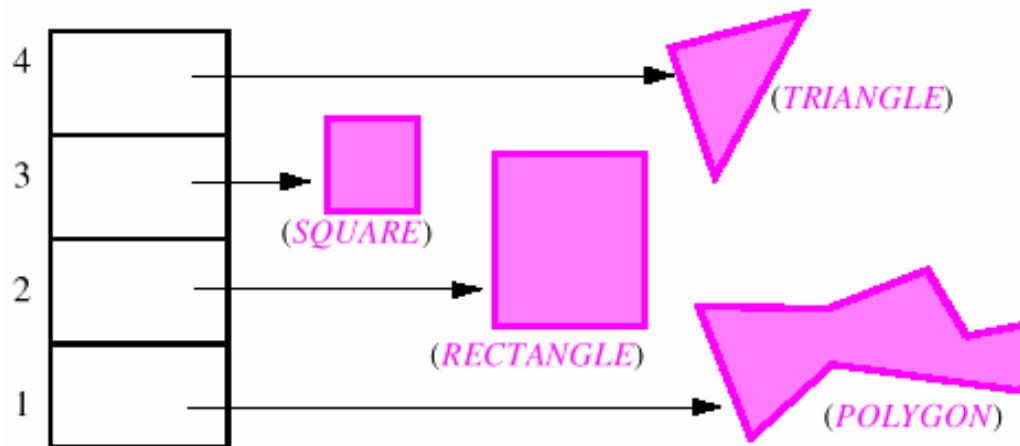
    pt = &p;    // ERROR
    pt = &r;    // ERROR
    pt = &t;

    p = r;     // SLICING
    p = t;     // SLICING
}
```

תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

# מבנה נתונים פולימורפי

- מבנה נתונים המכיל אברים מטיפוסים שונים, כולם צאצאים של אב משותף
- לדוגמא: מערך של מצולעים המכיל בפועל סוגים של מצולעים



# מבנה נתונים פולימורפי

---

```
int j;
float a,b;
...

void g(POLYGON *p, RECTANGLE *r, TRIANGLE *t, SQUARE *s)
{
    vector<POLYGON *> poly_arr;

    poly_arr[0] = p;
    poly_arr[1] = r;
    poly_arr[2] = s;
    poly_arr[3] = t;
    poly_arr.at(j)->rotate(a,b);
}
```





# העברת ארגומנטים למתודה

---

- השמה של מצביעים מתרחשת בצורה מרומזת בכל פעם שאנו מעבירים מצביע כארגומנט לפונקציה
- זוהי השמה של הטיפוס האקטואלי לטיפוס הפרמטר הפורמלי
- גם השמה זו צריכה לציית לכללי הפולימורפיזם

# העברת ארגומנטים למתודה

```
void expecting_polygon(POLYGON *p);
void expecting_rectangle(RECTANGLE *r);

void f()
{
    POLYGON p;
    RECTANGLE r;
    TRIANGLE t;

    expecting_polygon(&p); // OK
    expecting_polygon(&r); // Also good

    expecting_rectangle(&r); // OK
    expecting_rectangle(&p); // Error
    expecting_rectangle(&t); // Error
}
```

# זימון מתודה

○ בקריאה  $x.f$ , כאשר  $x$  מבוסס על טיפוס  $C$ , התכונה  $f$  (מתודה או שדה) חייבת להיות מוגדרת ב- $C$  או באחד מאבותיו הקדמונים עם הרשאות גישה מתאימות

○ טיפוס  $T$  מבוסס על טיפוס  $C$  אם:

- אם  $C$  אינו תבנית אזי  $T$  זהה ל  $C$
- אם  $C$  הינו תבנית אזי  $T$  הוא כל גזירה תבניתית של  $C$ . לדוגמא:  
הטיפוס `vector<POLYGON>` מבוסס על הטיפוס `vector`

○ דבר זה נבדק בזמן קומפילציה

## זימון מתודה - דוגמא

---

```
void h(POLYGON *p, RECTANGLE *r)
{
    p->perimeter();           // legal
    r->diagonal;              // illegal
    r->perimeter();           // legal
    p->side1;                 // illegal
}
```

# ציות לכלל הטיפוס

---

- בהשמה  $x=y$  הטיפוס של  $y$  חייב להתאים (לציית - conform) לטיפוס של  $x$
- $x$  ו- $y$  עשויים להיות פרמטר פורמלי ואקטואלי של פונקציה
- טיפוס  $U$  מתאים לטיפוס  $T$  רק אם מחלקת הבסיס של  $U$  היא צאצא של  $T$ .
- כל הארגומנטים האקטואלים המועברים לפונקציה חייבים להיות צאצאים של הפרמטרים הפורמלים שלהם

## מופע ישיר

---

- מופע ישיר של מחלקה C הוא עצם שנוצר ע"י בנאי של C
- מופע של מחלקה C הוא מופע ישיר של אחד מהצאצאים של C
- מצביע למחלקה מטיפוס T עשוי בזמן ריצה להצביע למופע של T
- ניתן לתאר זאת ע"י הפרדה בין טיפוס סטטי וטיפוס דינאמי



## טיפוס סטטי ודינמי

---

- טיפוס סטטי של עצם הוא הטיפוס שהוגדר (בהכרזה על הטיפוס) עבורו
- טיפוס דינאמי של עצם הוא הטיפוס שלפיו נוצר העצם (טיפוס הבנאי). טיפוס דינאמי של עצם הוא קבוע ואינו יכול להשתנות לאורך חיי העצם
- הטיפוס הדינאמי של מצביע או הפנייה הוא הטיפוס הדינאמי של העצם המוצבע
- הטיפוס הדינאמי של מצביע או הפנייה חייב להיות נגזרת של הטיפוס הסטטי שלו

# טיפוס סטטי ודינמי של מצביעים

---

```
POLYGON *p;  
RECTANGLE *r;
```

```
p = new POLYGON();  
r = new RECTANGLE();
```

```
p = r; // p's static type remains POLYGON,  
       // its dynamic type is now RECTANGLE
```



# טיפוס סטטי ודינמי של הפניות\*

---

```
POLYGON& p = *(new POLYGON());  
RECTANGLE& r = *(new RECTANGLE());  
  
p = r; // p's static type remains POLYGON,  
        // its dynamic type is now RECTANGLE
```

---

\* ב C++ כמעט ולא משתמשים בהפניות, מחוץ להקשר של העברה by reference

# קשירה דינאמית (dynamic binding)

---

- בקריאה למתודה של עצם, המתודה שתקרא תתבסס על הטיפוס הדינאמי של העצם. הטיפוס בזמן ריצה
- ברירת המחדל של שפת C++ היא קשירה סטטית. כדי ליצור קשירה דינאמית של פונקציות יש להגדיר אותן כ- virtual. פרטים בתרגול.
- לדוגמא: נרצה לחשב את ההיקף של מצולע כלשהו. בזמן קומפילציה אנו עוד לא יודעים איזה מצולע יוצבע בזמן ריצה

# קשירה דינאמית (dynamic binding)

---

```
POLYGON *p;
```

```
...
```

```
if (chosen_icon == rectangle_icon)
```

```
    p = new RECTANGLE();
```

```
else if (chosen_icon == triangle_icon)
```

```
    p = new TRIANGLE();
```

```
else if ...
```

```
...
```

```
p->perimeter(); // will use feature perimeter
```

```
                // according to the dynamic type.
```



# הגדרה מחדש של תכונות

---

- תכונה שהוגדרה מחדש (override) במחלקה נגזרת מסתירה את התכונה ממחלקת הבסיס
- על הגדרה מחדש לא לשנות את **המשמעות** המקורית של המתודה שאותה היא מגדירה מחדש, אלא רק את **המימוש**
- על החוזה של מחלקת הבסיס להיות מושפע מחוזה מחלקת הבסיס (פרטים בהרצאה על קבלנות משנה)
- קישוריות דינאמית מוסיפה תקורת זמן ריצה קבועה ובלתי תלויה בעומק עץ הירושה (פרטים בתרגול)

# מחלקות מופשטות - מוטיבציה

- ברצוננו לתאר מבחר צורות פתוחות וסגורות
- פעולת ההזזה רלוונטית לגבי כולן, אולם איך נממש אותה?

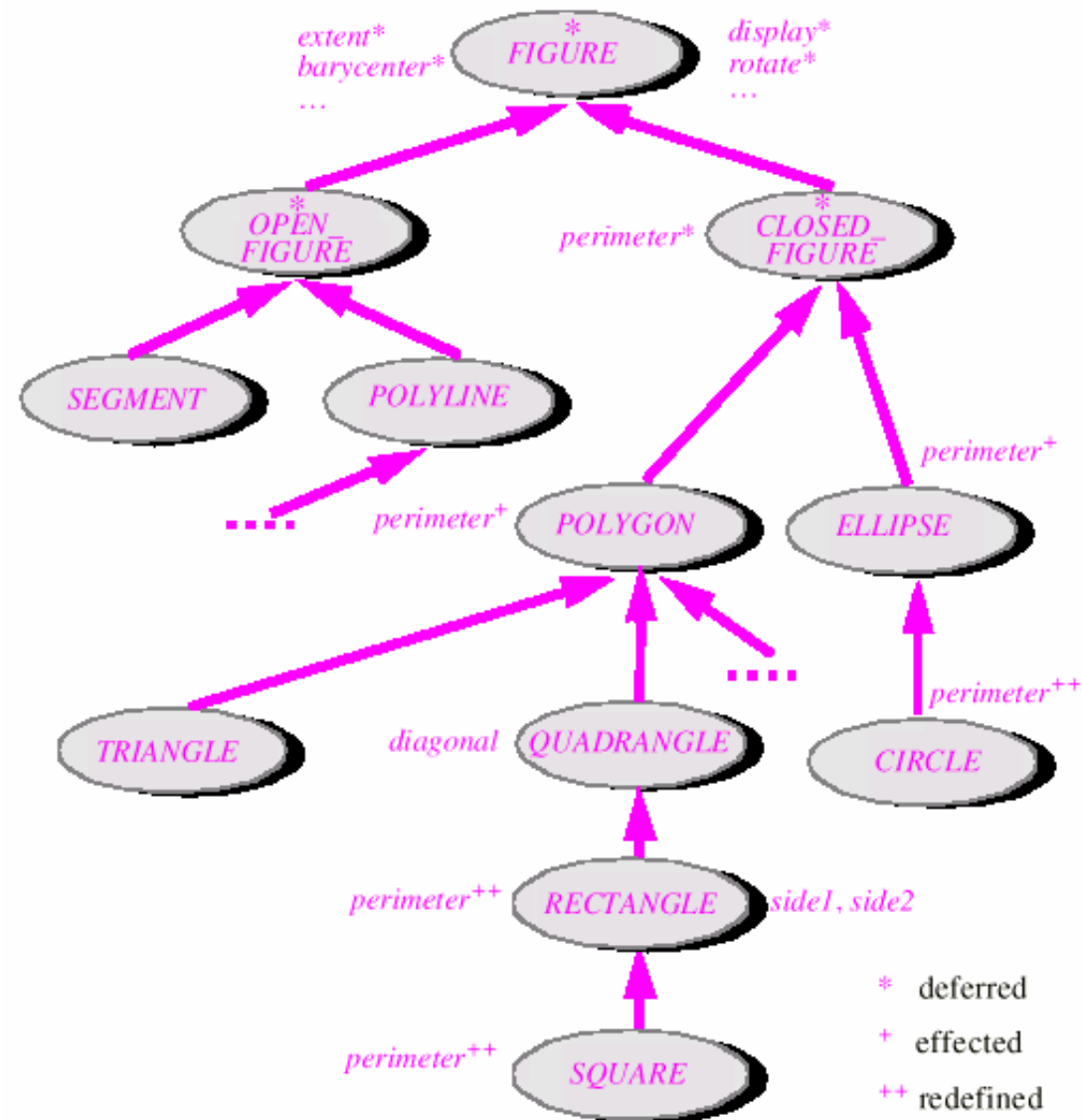
```
class FIGURE {
public:
    /** Move by a horizontally, b vertically*/
    virtual void translate(float, float)
    {
        error("FIGURE::translate"); // inelegant
    }
    // ...
};
```

- ברור שלמחלקות מסוימות שירשות מ FIGURE יהיה מימוש, אבל ל translate אן מימוש כללי.

- נרצה למנוע קוד כזה:

```
FIGURE f; // silly: "shapeless FIGURE"
```

# היררכיית הטיפוס FIGURE



# מחלקות מופשטות (דחיות)

- מחלקה אשר לא ניתן לייצר ממנה מופע נקראת מחלקה מופשטת (abstract class)
- בשפות מונחות עצמים אחרות היא מכונה:
  - JAVA
  - ללא מימוש כלל: ממשק (interface)
  - מימוש חלקי: מחלקה מופשטת (abstract)
  - Eiffel: מחלקה דחוייה (deferred)
- תחבירית, ב C++ מחלקה מופשטת היא מחלקה עם **מתודה וירטואלית טהורה** אחת לפחות שהוגדרה בה או שנורשה ממחלקת בסיס ולא ניתן לה מימוש

# פונקציות וירטואליות טהורות

---

```
class Shape { // abstract class
public:
    virtual void rotate(int) = 0 ; // pure virtual function
    virtual void draw() = 0 ; // pure virtual function
    virtual bool is_closed() = 0 ; // pure virtual function
    // ...
};
```



# הגדרה מחדש

- ישנם שני סוגים של הגדרה מחדש:
  - מימוש תכונה קיימת במחלקת הבסיס – המימוש המחודש מסתיר את המימוש המקורי
  - מימוש תכונה מדומה (וירטואלית) – במחלקת הבסיס הוכרזה רק המשמעות של התכונה ללא מימוש
- ב C++ אין הבדל תחבירי בין 2 הסוגים
- תכונות ממומשות במחלקה נקראות התכונות האפקטיביות שלה
- מחלקה נגזרת יכולה להפוך תכונה אפקטיבית ללא זמינה ע"י שינוי הרשאת הגישה לתכונה זו (מ public ל-private)



# סמנטיקה של מחלקות מופשטות

---

- ניתן לבטא משמעות של תכונות ע"י טענות
- ניתן לממש תכונות (אפקטיביות) בעזרת תכונות דחיות (וירטואליות) שעוד לא מומשו
- לדוגמא: מחסנית

# STACK

---

```
/** specification of STACK
 * @inv: count_non_negative: 0 <= count()
 * @inv: empty_if_no_elements: empty() == (count() == 0)
 */
template <class T>
class STACK
{
public:

    /** Number of stack elements */
    virtual int count() = 0;

    /** Top element
     * @pre: !empty()
     */
    virtual T top() = 0;
```



# STACK

---

```
/** Is stack empty? */
bool empty()
{
    return (count()==0);
}

/** Is stack full? */
virtual bool full() = 0;

/** Add x on top
 * @pre: !full()
 * @post: !empty() && top() == x
 * @post: count() == $prev(count())+1
 */
virtual void push(T x) = 0;
```

# STACK

---

```
/** Remove top element
 * @pre: !empty()
 * @post: !full()
 * @post: count() == $prev(count)-1
 */
virtual void pop() = 0;

/** Replace top element by x -- not deferred!!
 * @pre: !empty()
 * @post: !empty()
 * @post: top() == x
 * @post: count() == $prev(count)
 */
void change_top(T x)
{
    pop();
    push(x);
}
};
```



# אפקטיביות של תכונה כשדה

---

○ בשפות שבהן אין הבדל תחבירי בין פונקציות חסרות פרמטרים ובין שדות, ניתן היה לממש תכונה וירטואלית טהורה חסרת פרמטרים במחלקת בסיס גם כשדה במחלקה הנגזרת

○ ב C++ הדבר אינו תקף



# משמעות של ירושה

---

## ○ מודול – הרחבה

- ירושה של תכונות, הוספת תכונות, שינוי מימוש
- העקרון הסגור-פתוח

## ○ טיפוס – ייחוד (specialization)

- יחס is-a. לדוגמא: כלב הוא חיית מחמד
- ארכיטקטורת תוכנה מבוזרת



# שימוש במחלקות מופשטות

---

- כטיפוס נתונים מופשט – ללא מגבלות מימוש
- כמימוש חלקי – ע"י שימוש במחלקות ביניים
- שיתוף ההתנהגות המשותפת. אפשרות לשימוש בהתקשרות חוזרת (callback). לדוגמא: מתודות אפקטיביות אשר משתמשות במתודות דחיות שיוגדרו בשלב מאוחר יותר (אולי ע"י היישום)
- תוכנית עם "חורים"
- מאפשר הבנה וניתוח תוכניות בשפת התוכנית ללא צורך בשפה המתארת מערכות תוכנה (כגון UML)