

# תכנות מונחה עצמים בשפת C++

## מחלקות

אוהד ברזילי  
אוניברסיטת תל אביב

# מחלקות (classes)

- תזכורת: מחלקה היא טיפוס נתונים מופשט עם מימוש (אולי חלקי)
- זהו תאור סטטי, גלופה (cookie cutter)
- עצם (אובייקט) הוא מופע של מחלקה, עצם הוא יישות של זמן ריצה
- מחלקה היא גם טיפוס וגם מודול



# המחלקה Point

■ נקודה במישור הדו-מימדי

■ שאילתות

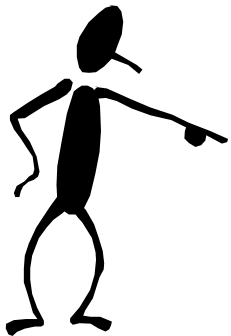
○ שיעורים קרטזיים  $(x, y)$

○ שיעורים קוטביים  $(rho, theta)$

○ מרחק (distance)

■ פקודות

○  $translate$  ,  $rotate$  ,  $scale$  (הזזה)



# המחלקה Point

## ■ חלק מהתאור המופשט:

- $x: \text{POINT} \rightarrow \text{REAL}$
- $\text{translate}: \text{POINT} \times \text{REAL} \times \text{REAL} \rightarrow \text{POINT}$

## ■ אקסיומה לדוגמא:

- $x(\text{translate}(p1,a,b)) = x(p1) + a$

# תכונות ומאפיינים

הפונקציות של ה ADT הופכות למאפיינים (features) של המחלקה (ב C++ נקראים members, איברים)

3 סוגים של פונקציות: בנאים, פקודות, שאילתות

2 אלטרנטיבות מימוש: זמן או מקום

בדוגמא של Point אם נבחר במאפיינים הקרטזיים של הנקודה (x,y) בתור נתונים (שדות של המחלקה) נוכל להציג את rho ו theta בתור חישובים



# תכונות ומאפיינים



- איברי המחלקה המיוצגים על ידי מקום (נתונים)  
נקראים **תכונות המחלקה (attributes)**
  - בשפת C++ מקובלים השמות **השדות של המחלקה**  
ולפעמים גם **data members**

- איברי המחלקה המיוצגים על ידי זמן (חישוב)  
נקראים **שגרות המחלקה (routines)**
  - בשפת C++ מקובלים השמות **המתודות של המחלקה**  
**(methods)**. (המינוח העברי - שיטות - אינו מקובל) וכן  
**.member functions**

# סיווג מאפייני המחלקה כפי שרואה זאת הלקוח

- אין ערך מוחזר – פרוצדורה (command)
- יש ערך מוחזר
  - עם ארגומנטים – פונקציה
  - בלי ארגומנטים
- ממומש ע"י זכרון – תכונה (שדה, attribute)
- ממומש ע"י חישוב – פונקציה (חסרת ארגומנטים)

# מאפייני המחלקה

## סיווג פנימי

- זיכרון: שדה
- חישוב: רוטינות
  - בלי ערך מוחזר – שגרות (פרוצדורות)
  - עם ערך מוחזר - פונקציות
- בחלק משפות התכנות (לדוגמא Eiffel אבל לא C++) אין הבדל תחבירי בין גישה לשדה ובין פונקציה חסרת פרמטרים
- ב-C# תכונות של המחלקה שיש להם גישה תחבירית אחודה נקראות `properties`
- בשפות אלו הסתרת המימוש מהלקוח טובה יותר





# POINT עם C++ - ל C מ

בשפת C

```
typedef struct POINT
{
    float x,y;
} POINT;
```

```
float rho(POINT *this)
{
    return sqrt((this->x * this->x) +
                (this->y * this->y));
}
```

```
main()
{
    float r;
    POINT p = {3,4};
    r = rho(&p);
}
```

# מ C ל- C++ עם POINT

בשפת C

```
typedef struct POINT
{
    float x,y;
} POINT;
```

בשפת C++

```
class POINT
{
    float x,y;
};
```

הפך להיות  
typedef struct  
class

# POINT מֵ C++ לִ C

בשפת C

```
float rho (POINT *this)
{
    return sqrt((this->x * this->x) +
                (this->y * this->y));
}
```

בשפת C++

```
class POINT
{
    float x,y;
    float rho (POINT *this)
    {
        return sqrt((this->x * this->x) +
                    (this->y * this->y));
    }
};
```

הגדרת הפונקציה נכנסה לתוך הגדרת המחלקה (הפונקציה הפכה למתודה)

# POINT עם C++ ל C מ

בשפת C

```
float rho (POINT *this)
{
    return sqrt((this->x * this->x) +
                (this->y * this->y));
}
```

בשפת C++

```
class POINT
{
    float x,y;
    float rho ()
    {
        return sqrt((this->x * this->x) +
                    (this->y * this->y));
    }
};
```

המתודה לא מקבלת את POINT בתור ארגומנט (כי היא מוגדרת בתוכה ולכן ברור שהיא פועלת עליה). עדיין מותר להשתמש ב this אם רוצים

# POINT עם C++ ל C מ

בשפת C

```
float rho (POINT *this)
{
    return sqrt((this->x * this->x) +
                (this->y * this->y));
}
```

בשפת C++

```
class POINT
{
    float x,y;
    float rho ()
    {
        return sqrt(x*x + y*y);
    }
};
```

מומלץ לוותר על this

# POINT עם C++ - ל C מ

בשפת C

```
float rho (POINT *this)
{
    return sqrt((this->x * this->x) +
                (this->y * this->y));
}
```

בשפת C++

```
class POINT
{
    float x,y;
public:
    float rho ()
    {
        return sqrt(x*x + y*y);
    }
};
```

כדי שאפשר יהיה לקרוא למתודה מחוץ למחלקה יש להוסיף את התגית `public:` לפני הגדרת המתודה

# מ C ל- C++ עם POINT

## בשפת C

```
main()
{
    float r;
    POINT p = {3,4};
    r = rho(&p);
}
```

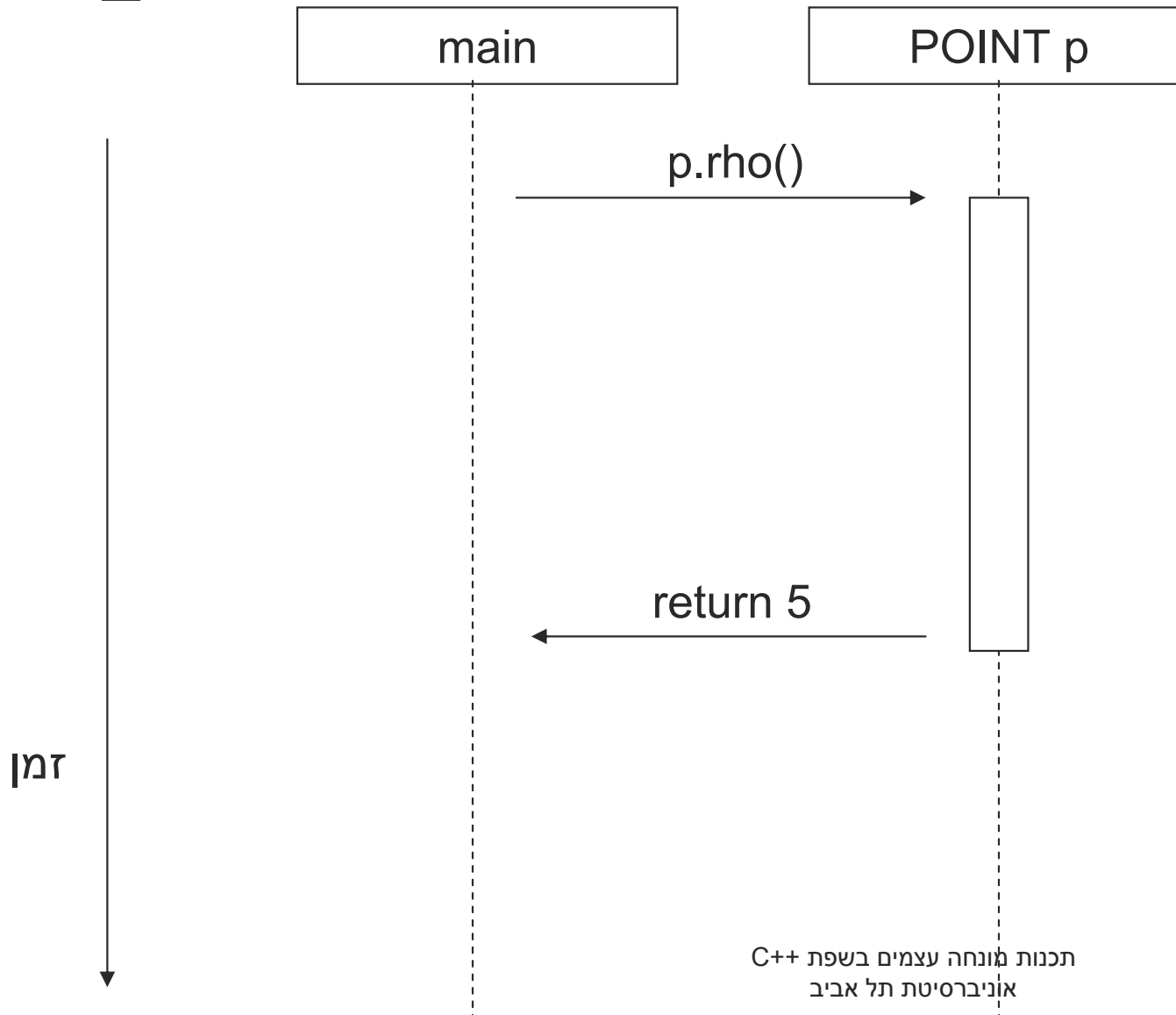
בשפת C אנו מסתכלים על תוכנית כעל אוסף של פונקציות שמקבלות כארגומנטים מבני נתונים

## בשפת C++

```
main()
{
    float r;
    POINT p = {3,4};
    r = p.rho();
}
```

בשפת C++ אנו מסתכלים על תוכנית כעל סדרה של הודעות \ בקשות המועברות בין אובייקטים ובהם הם מבקשים זה מזה שרותים שונים

# מודל העברת ההודעות



בשפות מונחות  
עצמים "נקיות"  
גם `main` עצמו  
היה מתודה של  
עצם כלשהו



# המחלקה POINT

```
/** Two dimensional points */
class POINT
{
    /** Abscissa and ordinate */
    float x, y;

public:
    /** Distance to point (0,0) */
    float rho()
    {
        return sqrt(x*x + y*y);
    }

    /** Angle to horizontal axis */
    float theta(); // will be implemented elsewhere
}
```

# המחלקה POINT

```
/** Distance to other */  
float distance (POINT other)  
{  
    return sqrt((x-other.x)*(x-other.x)+  
                (y-other.y)*(y-other.y));  
}
```

```
/** Move by a horizontal, b vertical */  
void translate(float a, float b)  
{  
    x = x+a;  
    y = y+b;  
}
```

```
/** Scale by a ratio of factor */  
void scale(float factor)  
{  
    x = factor*x;  
    y = factor*y;  
}
```

```
};
```

# המחלקה POINT – מימוש וממשק

- ניתן (ורצוי) לממש את הפונקציות מחוץ לגוף המחלקה.

- הצהרות על המחלקה ותכונותיה יופיעו בקובץ הכותרת (.h) ואילו מימושי הפונקציות יופיעו בקובץ מימוש נפרד (.cpp). תוך שימוש באופרטור השיוך (::)

- ההבדל הוא טכני בלבד – ניתן להתייחס למימושי הפונקציה כאילו נכתבו בתוך ה { } של המחלקה עצמה

```
//point.c
/** Angle to horizontal axis */
float POINT::theta()
{ ... }
```

# המחלקה POINT - הערות

- שימו לב לשימוש בפונקציה `sqrt()`
  - שימוש בפונקציות גלובליות נשאר בעינו
- במתודה `distance` איזה עצם משתנה ואיזה נשאר?
  - כשרשמנו: `x-other.x`
  - כאילו רשמנו: `this->x - other->x`
- לכל מתודה יש ארגומנט מרומז (`implicit`) והוא האובייקט שעליו היא פועלת (`this`)
- בעת ביצוע מתודה המצביע `this` מצביע על העצם הנוכחי

# יחסי ספק - לקוח



■ מחלקה C היא הלקוח (client) של מחלקה S, ומחלקה S היא הספק (supplier) של מחלקה C, אם C כוללת הצהרה מהצורה:  $S \ e$

■  $e$  עשוי להיות שדה, פונקציה, משתנה מקומי (כולל ערך מוחזר) או אפילו ארגומנט של מתודה של C

■  $e$  משויך לעצם ממחלקה S (אח"כ)

■ מחלקה יכולה להיות לקוחה של עצמה:

```
class Person{  
    Person *parent;  
    ...  
};
```

# קריאה למתודה (method call)

- המנגנון היסודי של תכנות מונחה עצמים
- נקרא גם: זימון, שליחת הודעה
- דוגמא: `p1.translate(1.0, -5.4)` פירושו הפעל על העצם `p1` את המתודה `translate` עם הארגומנטים `1.0` ו-`-5.4`
- `p1` נקרא אובייקט המטרה (target)
- `1.0` ו-`-5.4` הם הארגומנטים האקטואלים
- `Object.Procedure(parameters)`
- עקרון המטרה היחידה – כל פעולה בתוכנית מוכוונת עצמים מתבצעת יחסית לעצם מסוים. ה `this` של רגע ההפעלה
- קיומן של פונקציות גלובליות ב `C++` סותר את עקרון המטרה היחידה

# מודול לעומת טיפוס

- השרותים שמספקת המחלקה POINT הופכים אותה למודול
- כל השרותים זמינים עבור כל עצם מהמחלקה POINT, מה שהופך אותה לטיפוס

# קריאה למתודה - המשך

- כל הקוד המתבצע הוא במסגרת של פונקציות (גלובליות או מתודות)

- לכל מתודה יש מטרה (target)

- כאשר המטרה היא האובייקט הנוכחי (this) מקובל להשמיט את שם אובייקט המטרה

- מטרה מפורשת (qualified call)  $x.f()$  ,  $p.y$

- מטרה מרומזת (unqualified call)  $f()$  ,  $y$



# מטרה מרומזת (unqualified call)

- כאשר שורת קוד מופיעה בתוך מתודה של אובייקט `obj`, כל הקריאות למתודות שאינן גלובליות יתבצעו כאשר `obj` הוא אובייקט המטרה שלהן



# מטרה מרומזת (unqualified call)

```
main()
{
    T obj;
    obj.F();
}
```

obj הוא המטרה של הקריאה G()

```
class T
{
    void G();
    void F()
    {
        ...
        G();
    }
};
```

# מטרה מפורשת (qualified call)

- בעת ביצוע גוף של רוטינה  $r$  בהקשר של עצם  $obj$  נתבונן בקריאות מסוג  $x.m$  או  $x.f()$ 
  - אם  $x$  הוא שדה של  $obj$  והוא מטיפוס  $T$  אזי  $T$  הוא המטרה
  - אם  $x$  הוא פונקציה (גלובלית או מתודה) אזי מבצעים את הקריאה והטיפוס של הערך המוחזר הוא המטרה של  $f(x())$
  - אם  $x$  הוא משתנה מקומי של  $r$  והוא מטיפוס  $T$  אזי  $T$  הוא המטרה

# בנץ הוא 'זה'

כאשר העצם הנוכחי (this) הוא obj:

- קריאה לא מפורשת או ביצוע פעולות שאינן קריאה לפונקציה אינן משנות את זהותו של העצם הנוכחי
- קריאות מפורשות גורמות לעצם הנוכחי להפוך למטרה של הקריאה. בסיום ביצוע הקריאה `obj` חוזר להיות העצם הנוכחי
- במהלך ביצוע קריאות לפונקציה גלובלית או סטטית אין עצם נוכחי. בסיום ביצוע הקריאה `obj` חוזר להיות העצם הנוכחי



# העצם הנוכחי

```
main()  
{  
  T obj;  
  obj.F();  
}
```

אין עצם נוכחי

```
class T  
{  
  void G() { ... }  
  void F()  
  {  
    T obj2;  
    obj2.G();  
    G();  
  }  
};
```

# העצם הנוכחי

```
main()
{
    T obj;
    obj.F();
}
```

אין עצם נוכחי  
obj הוא מטרת הקריאה

```
class T
{
    void G() { ... }
    void F()
    {
        T obj2;
        obj2.G();
        G();
    }
};
```

# העצם הנוכחי

```
main ()
{
    T obj;
    obj.F ();
}
```

obj הוא העצם נוכחי  
obj2 הוא מטרת הקריאה


```
class T
{
    void G () { ... }
    void F ()
    {
        T obj2;
        obj2.G ();
        G ();
    }
};
```

# העצם הנוכחי

```
main()
{
    T obj;
    obj.F();
}
```

obj2 הוא העצם נוכחי

```
class T
{
    void G() { ... }
    void F()
    {
        T obj2;
        obj2.G();
        G();
    }
};
```






# העצם הנוכחי

```
main ()
{
    T obj;
    obj.F ();
}
```

obj הוא העצם נוכחי  
obj הוא מטרת הקריאה

```
class T
{
    void G () { ... }
    void F ()
    {
        T obj2;
        obj2.G ();
        G ();
    }
};
```




# העצם הנוכחי

```
main()
{
    T obj;
    obj.F();
}
```

obj הוא העצם נוכחי

```
class T
{
    void G() { ... }
    void F()
    {
        T obj2;
        obj2.G();
        G();
    }
};
```



# העמסת אופרטורים

- מה עושה  $x.plus(a)$  ?
- ומה עושה  $x+a$  ?
- בשפת Scheme:  $(+ x a)$
- סימן ה'+' הוא אופרטור. זהו שם של פונקציה הנבדלת מפונקציה "רגילה" בתחביר הקריאה לה
- דוגמא: המחלקה `complex`

# אופרטורי complex

```
class complex { // very simplified complex  
  
    double re, im;  
  
    public:  
        complex operator+ (complex) ;  
        complex operator* (complex) ;  
};
```

# אופרטורי complex

```
void f()  
{  
    complex a = {1, 3.1} ;  
    complex b = {1.2, 2} ;  
    complex c = b;  
    a = b+c;  
    b = b+c*a;  
    c = a*b+complex(1,2) ; // יצירת עצם זמני  
}
```

# אופרטורים כמתודות

שם של אופרטור יכול להיות רק אחד מהאופרטורים הסטנדרטים של C++:

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	<i>new</i>	<i>new</i> []	<i>delete</i>	<i>delete</i> []

לא ניתן להעמיס את האופרטורים:

- :: (אופרטור השיוך)
- . (גישה לשדה)
- \* . (גישה לשדה דרך מצביע לפונקציה)

# ייצוא תכונות בררני

- בשפת C++ זמינות של אברי המחלקה נקבעת ביחס למחלקה של הטיפוס הניגש
- שדות של המחלקה (מתודות ונתונים) שיוגדרו כפרטיים (private) ימנעו מאובייקטים של מחלקות אחרות (כאשר העצם הנוכחי הוא מטיפוס של מחלקה אחרת) גישה אליהם
- מחלקה אינה יכולה לאפשר גישות מסוג מסוים בלבד (רק קריאה או רק כתיבה)

# ייצוא תכונות בררני

- מחלקה אינה יכולה למנוע גישה מעצמה
- מחלקה אינה יכולה להפלות בין אובייקטים שונים של אותה מחלקה
- מחלקה יכולה להפלות בין פונקציות ומחלקות שונות ע"י הכרזה עליהן כחברות (friend)
- מחלקה אינה יכולה להגביל תכונות מסוימות בלבד למחלקות מסוימות בלבד



# ייצוא תכונות בררני

```
class A
{
    void f();
    void g();
public:
    void h();
};
```

```
class B
{
    void F()
    {
        A a;
        a.f(); // Error. f() is private
        a.h(); // OK. h() is public
    }
};
```

# ייצוא תכונות בררני

```
class A
{
    void f();
    void g();
public:
    void h();
    friend class B;
};
```

```
class B
{
    void F()
    {
        A a;
        a.f(); // OK. B is a friend
        a.h(); // OK. h() is public
    }
};
```

# ייצוא תכונות בררני

```
class A
{
    void f();
    void g();
public:
    void h();
    friend B::F();
};
```

```
class B
{
    void F()
    {
        A a;
        a.f(); // OK. B::F is a friend
        a.h(); // OK. h() is public
    }
};
```