

OOP-Lecture 1

A series of horizontal lines in white, light gray, and dark gray extending across the bottom of the slide.

תכנות מונחה עצמים

מהווה את ההבדל העיקרי בין שפת C לשפת
C++!

תכנות פרוצדורלי - שפת C

❖ כל מערכת ממוחשבת מכילה עם שני גורמים – מידע ו-אלגוריתם (תכנות דרך פעולה).

❖ שפה פרוצדורלית היא שפה שמדגישה את החלק האלגוריתמי בתכנות.

❖ כל תוכנית היא סט של פרוצדורות (או פונקציות) שעל המחשב לבצע.

❖ בכללי, תוכנה מתוכננת, מתוארת ונבנת תוך שימוש במונחי הפעולות שהיא מבצעת.

❖ מדובר בתכנות Top-down.

תכנות מונחה עצמים - שפת C++

❖ תכנות מונחה עצמים (Object Oriented Programing - OOP) מתמקד בחלק **המידע** במערכת.

❖ הרעיון העיקרי הוא לתכנן מבני / מחלקות מידע (classes) שייצגו את התכונות העיקריות של הבעיה.

❖ **מחלקה (class)** מגדירה בעזרת איזה מידע מייצגים אובייקט בתוכנית ואיזה פעולות ניתן לבצע על אותו מידע.

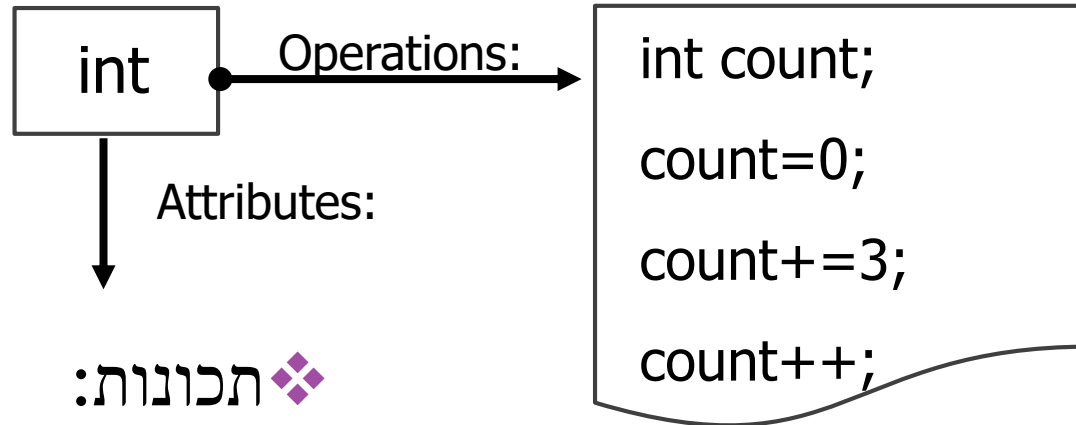
❖ מדובר בתכנות Bottom-up.

תכונות התכנות המונחה עצמים

❖ התכונות החשובות ביותר בתכנות מונחה עצמים הן:

1. (Abstract Data Type) – Abstraction – הפשטה.
2. Encapsulation and Data hiding – כימוס והסתרת מידע.
3. Inheritance – ירושה.
4. Polymorphism – רב צורתיות.
5. Reusability of code – שימושיות מחודשת.

Abstract Data Type



תכונות: ❖

גודל: 4 □

bytes

סוג: מספר □

שלם.

פעולות: ❖

■ אופרטורים אריתמטיים

■ אופרטורים לוגיים

■ bitwise

■ פעולות קלט / פלט

Abstract Data Type (cont...)

❖ כיצד נתאר מכונית?



Abstract Data Type (cont...)

❖ בדרך ה"ישנה":

```
sCarName;      char*
iNumOfDoors;    int
bAutomatic;     bool
                //...
```

❖ שאלות:

- ומה אם אנחנו צריכים 100 מכוניות?
- כיצד נתאר את הפעולות: "פתיחת דלת במכונית",
"התנעת מכונית" וכו'

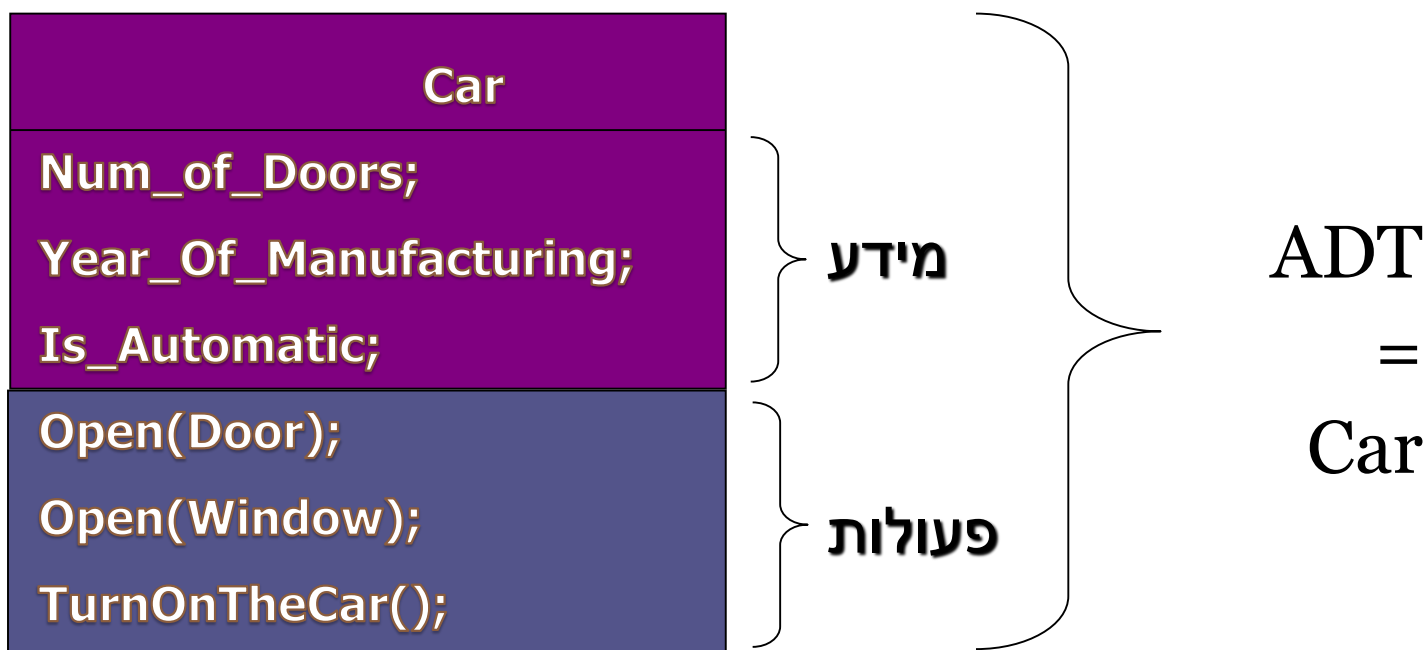
Abstract Data Type - הגדרה

❖ ADT הוא מפרט של סט מידע וסט הפעולות שניתן לבצע על אותו מידע (לדוגמא, מספרים ממשיים, מחסנית, מכונית...).

❖ *ADT מהווה ומכיל:*

- 1. טיפוס נתונים חדש.
- 2. סט של פעולות. סט הפעולות הזה מהווה ממשק (*interface*) לטיפוס החדש.
- 3. פעולות הממשק הם המנגנון האחד היחיד המאפשר גישה למבנה טיפוס הנתונים.

מכונת כ- ADT



❖ היכולת לאגד ביחד מידע עם פעולות – מאפשרת ליצור טיפוס חדש של נתונים.

□ וזאת בדיוק ההגדרה של כימוס.

❖ דוגמאות נפוצות ל-ADT:

□ טיפוסים פרימיטיביים (בנויים בשפה): **int, bool, float**

□ טיפוסים מוגדרים על ידי המשתמש: **Stack, Queue, Tree**

Car, Student, Text-editor.

❖ ADT מאפשר להגדיר מאפיינים חיוניים מבלי להיכנס יותר מידי לעומק המימוש הפרטני.

❖ הממשק ב-ADT מספק רשימת פעולות אפשריות ("מה אפשר לעשות?") ולא הגדרות מימוש ("איך אפשר לעשות זאת?")

Abstract Data Type - הגדרה

❖ ADT הוא מפרט של סט מידע וסט הפעולות שניתן לבצע על אותו מידע (לדוגמא, מספרים ממשיים, מחסנית, מכונית...).

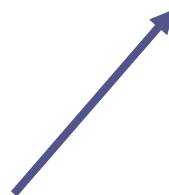
❖ *ADT מהווה ומכיל:*

❖ 1. טיפוס נתונים חדש.

❖ 2. סט של פעולות. סט הפעולות הזה מהווה ממשק (**interface**) לטיפוס החדש.

❖ 3. פעולות הממשק הם המנגנון האחד היחיד המאפשר גישה למבנה טיפוס הנתונים.

(Encapsulation)
כימוס



מחלקות - Classes

- ❖ בשפת C++ (וגם Python ו-Java ועוד..) אנו משתמשים במחלקות (classes) על מנת להגדיר ADTs.
- ❖ מחלקה בשפת C++ היא הייצוג של ישות מידע (טיפוס מוגדר על ידי המשתנה – user defined type).
- ❖ Class היא שילוב של ייצוג מידע ומתודות (methods) שהן פונקציות פנימיות של המחלקה המשתמשות לתפעול המידע.
- ❖ אובייקטים (Objects) הם מופעים של מחלקות. כלומר, הקשר בין אובייקטים למחלקות זהה לקשר בין משתנים לטיפוסים.
- ❖ הגדרת מחלקה לא מקצה זיכרון או אחסון לאף אובייקט!

מחלקה - Class

- The syntax:
class ClassName
{
 //attributes and operations
};

Classes & Objects

הקדמה

- ❖ כל מחלקה מכילה:
 - **Data Members** – המידע המאופסן.
 - **Methods** (= member functions) – הפעולות שניתן לבצע על אותו מידע מאופסן.
- ❖ התוכנית תגדיר / תייצר מגוון **אובייקטים** מה**מחלקות** הקיימות. ואז היא תוכל להשים ערכים ל- data member של האובייקטים ולהפעיל את המתודות שלהם.

Classes & Objects

❖ מחלקות הם טיפוסים מידע חדשים! וכך יש להתייחס אליהן.

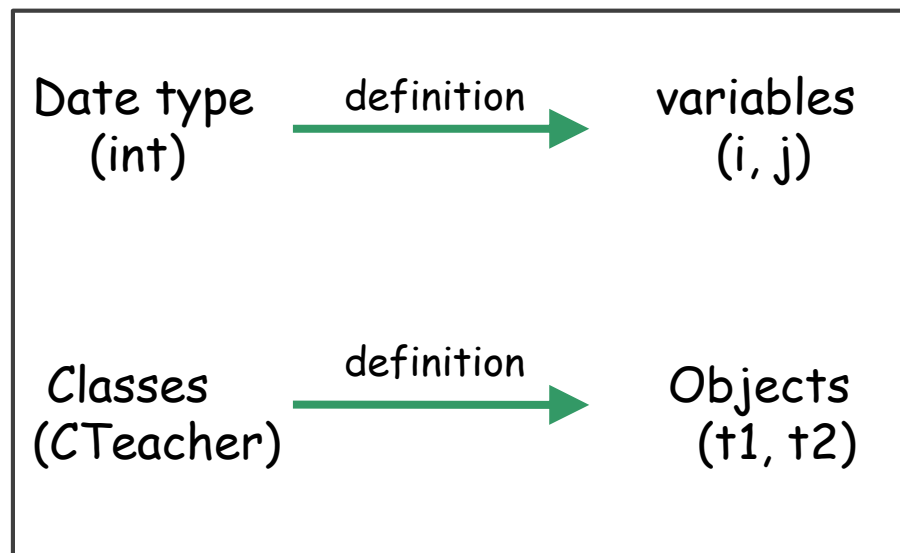
❖ מחלקה מסייעת למתכנת לייצר טיפוס חדש ויש להתייחס לטיפוס זה בדיוק כפי שמתייחסים לטיפוסים הפרימיטיביים הקיימים בשפה: `int, float, char, etc.`

מה ניתן לבצע עם הטיפוסים הפרימיטיביים?



?

האנלוגיה הבסיסית



Classes & Objects

תזכורת קצרה:

הגדרה ושימוש של מבנים בשפת C:

```
...  
struct CPoint  
{  
    int x;  
    int y;  
};
```

```
int main()  
{  
    struct CPoint P1;  
    P1.x = 5;  
    P1.y = 7;  
    return 0;  
}
```

Classes & Objects

הגדרה:

```
class MyClass {  
    [private:]  
        variables (data members)  
        ...  
        functions (methods)  
        ...  
  
    public:  
        variables (data members)  
        ...  
        functions (methods)  
        ...  
};
```

שימוש:

```
int main() {  
    // define objects of type  
    // class_name  
    MyClass MyObject1;  
    MyClass MyObject2;  
  
    // call a member function  
    MyObject1.func1(...);  
  
    // assign value to data members  
    MyObject1.Index = 12;  
    return 0;  
}
```

Classes & Objects

דוגמא - מחלקת Point

❖ מחלקת Point מייצגת נקודה דו מימדית

```
class Point {  
    public:  
        int m_x, m_y;  
  
        void show() {  
            cout<<"x = "<<m_x<<"", y = "<<m_y<<endl;  
        }  
};
```

```
#include <iostream>  
int main() {  
    Point p1, p2;  
    p1.m_x = 15;  
    p1.m_y = 10;  
    p1.show();  
    cout<<"Please enter x & y values: ";  
    cin>>p2.m_x>>p2.m_y;  
    p2.show();  
  
    return 0;  
}
```

משתני מחלקה - Data member

משתני מחלקה vs משתנים מקומיים

- ❖ מתודות מכירות ו"יודעות" את כל משתני המחלקה של האובייקט שהפעיל אותם.
- ❖ כשמפעילים מתודה על אובייקט מסוים יש למתודה יכולת לגשת ולשנות את כל משתני המחלקה של אותו אובייקט (בלי שהם נשלחים כפרמטר!)
- ❖ בניגוד למשתנים מקומיים (שחיים רק עד סוף הבלוק בו הוצהרו!) משתני מחלקה חיים כל עוד האובייקט שבו הם קיימים חי.

כלל אצבע 1:

אם אתם כל הזמן שולחים את אותו משתנה לכל (או רוב) המתודות של המחלקה, זוהי אינדיקציה טובה שייתכן כי המשתנה צריך להיות משתנה מחלקה!

הסתרת מידע - Information Hiding

- ❖ משתמשי המחלקה יכולים לראות בבירור במה הם יכולים להשתמש וממה להתעלם.
- ❖ נועד להבטיח כי משתמשי המחלקה לא יהיו תלויים במימוש ספציפי של המחלקה העלול בעתיד להשתנות!
- ❖ **public** = כל חברי המחלקה (משתנים ומתודות) שמוגדרים כך נגישים שלכולם! מדובר בממשק (interface) המחלקה.
- ❖ **private** = אף אחד לא יכול לגשת או להפעיל את חברי המחלקה המוגדרים כך מלבד חברי המחלקה האחרים! מהווה מחסום (קיר חוצץ) בין האובייקט והמשתמש (כימס).

משתני מחלקה - Data Members

Private and Public Permissions

- ❖ חברי מחלקה (משתנים ומתודות) יכולים להיות public או private.
- ❖ לחברי מחלקה **public** (פומביים) יש נגישות בכל חלקי הקוד (דרך האובייקט שבו הם מוכלים, ובתנאי שיש גישה אליו!).
- ❖ לחברי מחלקה **private** (פרטיים) ניתן לגשת אך ורק דרך מתודות של אותה מחלקה!
- ❖ public ו- private הן מילים שמורות שברגע שכותבים אחת מהן בהגדרת המחלקה כל חברי המחלקה שמופיעים אחריה מקבלים את הרשאות הגישה שהיא קובעת.
- ❖ ההרשאה הדיפולטית במחלקה (אם לא כתבנו אף מילה) היא private.
- ❖ ניתן לשנות את ההרשאות כמה פעמים שרוצים במחלקה, אך נהוג רק פעם אחת (וכך ננהג).

Private vs. Public

Example - Point2

```
#include <iostream>
int main() {
    Point p1;
p1.m_x = 15; ERROR
    p1.set_x(15);
p1.m_y = 10; ERROR
    p1.set_y(10);
    p1.show();
    p1.set_x(17);
    p1.set_y(5);

    cout<<"x= "<<p1.get_x()<<" y= "<<p1.get_y()<<endl;

    return 0;
}
```

```
class Point {
public:
    void set_x(int x) {m_x = x;}
    void set_y(int y) {m_y = y;}
    int get_x() {return m_x;}
    int get_y() {return m_y;}
    void show() {
        cout<<"x = "<<m_x<<" , y = "<<m_y<<endl;
    }
private:
    int m_x, m_y;
};
```


משתני מחלקה פרטיים

מדוע להשתמש ב-private?

- ❖ מונע גישה ישירה למשתני המחלקה על ידי מתכנתים המשתמשים במחלקה.
- ❖ חשוב לשמירה על כימוס, מודולריות, ובטיחות!
- ❖ חוק חשוב:
- ❖ חייבת להיות סיבה ממש טובה על מנת להגדיר משתנה מחלקה כ-public.
- ❖ בדרך כלל אין כזאת סיבה!!!
- ❖ אז, איך ניגש למשתני מחלקה פרטיים?
- ❖ מתודות set ו-get ...

Methods (Member Functions)

מתודות vs פונקציות (גלובליות)

❖ **מתודות (member functions)** מייצגות את השירותים שהמחלקה מציעה למשתמשים בה. המתודות פועלות על משתני המחלקה של האובייקט ומופעלות אך ורק דרך האובייקט!

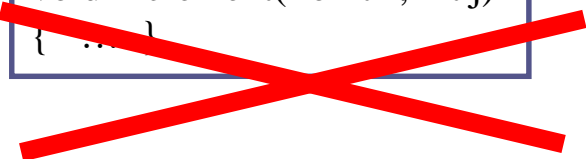
❖ כימוס.

❖ מודלריות.

❖ **פונקציות גלובליות** מיועדות למטרות כלליות ולא נמצאות תחת השירותים של אף מחלקה.

❖ לדוגמא?

```
void Increment(Point P, int j)
{
    ...
}
```



כלל אצבע 2: אם אתם חייבים לשלוח אובייקט לפונקציה גלובלית והיא צריכה גישה כמעט לכל האובייקט, ופועלת על חלקים ממנו, זוהי אינדיקציה טובה שפונקציה זאת בעצם צריכה להיות מתודה של המחלקה!

מימוש חיצוני של מתודות

Example - Point2

```
#include <iostream>
#include "Point.h"
```

```
void Point::set_x(int x) {m_x = x;}
void Point::set_y(int y) {m_y = y;}
```

```
int Point::get_x() {return m_x;}
int Point::get_y() {return m_y;}
```

```
void Point::show() {
    cout<<"x = "<<m_x<<" , y = "<<m_y<<endl;
}
```

```
class Point {
public:
    void set_x(int x);
    void set_y(int y);
    int get_x() ;
    int get_y();
    void show();

private:
    int m_x, m_y;
};
```

Methods (Member Functions)

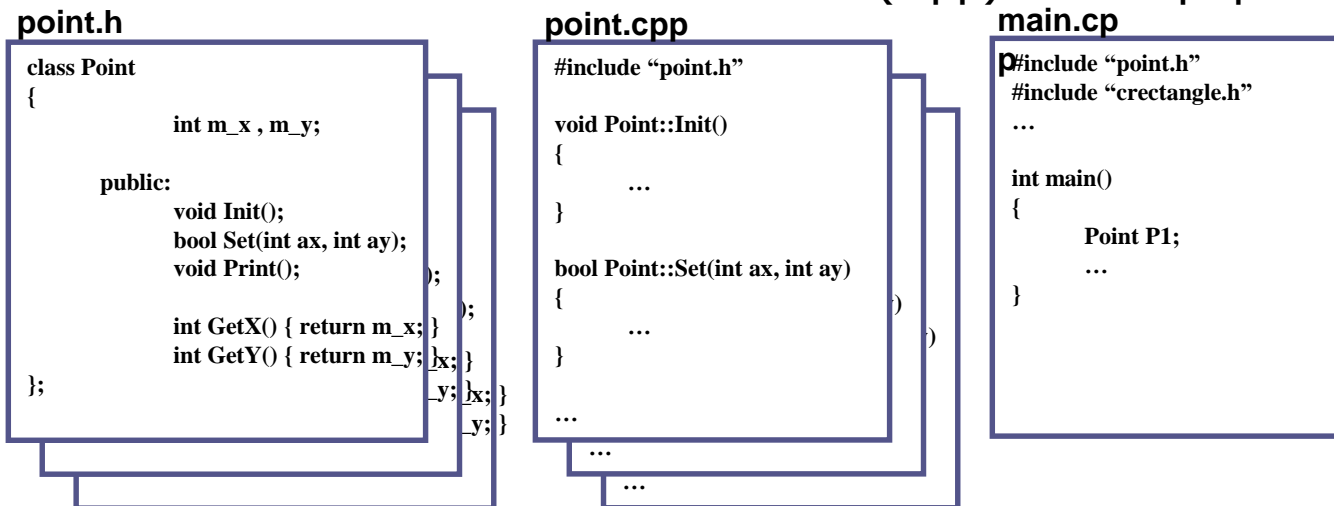
מימוש חיצוני של מתודות

- ❖ מתודות שממומשות בתוך הגדרת המחלקה (קובץ h) מוגדרות אוטומטית כ-**inline functions**.
- ❖ רק פונקציות פשוטות ולא מסובכות.
- ❖ ללא לולאות, switch, קריאות לפונקציות אחרות וכדומה!
- ❖ רוב המתודות ימומשו חיצונית להגדרת המחלקה (קובץ cpp). בהגדרת המחלקה יצוינו רק הצהרות המתודות.
- ❖ מתודה שממומשת חיצונית חייבת להופיע בשמה **המלא**. כלומר – חייב להופיע גם שם המחלקה שהיא ממומשת ואופרטור השייכות "::".

Methods (Member Functions)

עבודה עם מספר קבצים

- ❖ כל מחלקה תמומש בשני קבצים:
 - ❖ קובץ כותרות: (h) header file בו יופיעו הגדרת המחלקה, משתני המחלקה והצהרות המתודות.
 - ❖ קובץ מימוש: (cpp) code file בו ימומשו כל המתודות



inline

מהו inline?

- ❖ קריאה לפונקציה זה דבר יקר (overhead של מחסנית הקריאות).
- ❖ בפונקציות קצרות ה-overhead גדול יותר מעלות הפונקציה עצמה – בזבזני!
- ❖ inline – בקשה מהקומפיילר להעתקת קוד הפונקציה במקום לקרוא לה ישירות.
- ❖ עולה יותר בזמן קומפילציה ובגודל קוד!
- ❖ חוסך הרבה בזמן ריצה!
- ❖ inline בקשה שתיענה רק אם הקומפיילר יחליט שזה סביר!
- ❖ פונקציות פשוטות וקצרות.
- ❖ בלי תנאים, לולאות, מבני בקרה, קוד ארוך וכו'
- ❖ inline שלא נענה – פשוט התעלמות! זה לא שגיאה!

inline יתרונות מול חסרונות

❖ יתרון: יעילות בזמן ריצה!

❖ חסרונות:

❖ עלות בזמן קומפילציה.

❖ מגדיל מאוד את נפח הקוד.

❖ אי וודאות לגבי התוצאה הסופית! (התקבל? לא התקבל?)

❖ מצריך include לספריית מערכת בקובץ ה-H (כפילויות ב-include)

❖ לפונקציה אין כתובת!

❖ פגיעה בכימוס!!! (המימוש חייב להיות בקובץ H...)

בקשת inline

- ❖ Implicit inline – מממשים את המתודה בתוך הצהרת המחלקה.
- ❖ Explicit inline - המתודה ממומשת מחוץ להצהרת המחלקה, אך:
 - ❖ בקובץ ה-H.
 - ❖ המילה השמורה inline מופיעה לפני מימוש הפונקציה.

דוגמה ל-inline

//File: TwoDigitNumber.h

```
#include <iostream>
```

```
class TwoDigitsNumber {
```

```
    private:
```

```
        char m_firstD, m_secondD; //the digits
```

```
    public:
```

```
        void setFirstDigit(char d1) {m_firstD = d1;}
```

```
        void setSecondDigit(char d2) {m_secondD = d2;}
```

```
        void show();
```

```
};
```

```
inline void TwoDigitsNumber::show() {
```

```
    cout<<"The number is: "<<m_secondD
```

```
    <<m_firstD<<endl;
```

```
}
```

Implicit inline!

Explicit inline!

הכלה (Composition)

Composition

❖ משתני מחלקה לא חייבים להיות משתנים פרימיטיביים. הם יכולים בעצם להיות אובייקטים של מחלקה כלשהי.

```
//File: Line.h
#include "Point.h"
class Line {
    private:
        Point m_p1, m_p2; //Composition!!
    public:
        void setLine(int x1, int y1, int x2, int y2);
        void setLine(const Point& p1, const Point& p2);
        void show();
};
inline void Line::show() {
    cout<<"Line from: ";
    cout<<" To: ";
}
```

יש פה function name overloading. אין צורך ב-
!Point.h Include ל-iostream כי כבר יש אותו ב-Point.h

m_p1.show();
m_p2.show();

```
//File: Line.cpp
#include <iostream>
#include "Point.h"
#include "Line.h"

void Line::setLine(int x1, int y1, int x2, int y2) {
    m_p1.set_x(x1);
    m_p1.set_y(y1);
    m_p2.set_x(x2);
    m_p2.set_y(y2);
}

void Line::setLine(const Point& p1, const Point& p2) {
    m_p1 = p1;
    m_p2 = p2;
}
```

```
//main.cpp file:  
#include "Point.h"  
#include "Line.h"  
  
int main() {  
    Point p1, p2;  
    p1.set_x(15); p1.set_y(10);  
    p2.set_x(0); p2.set_y(0);  
  
    Line line1, line2;  
    line1.setLine(15,10,7,6);  
    line1.show();  
    line2.setLine(p1,p2);  
    line2.show();  
  
    return 0;  
}
```

להראות שבהכלה משתמשים
בפונקציות פומביות (ממשק) של
האובייקטים המוכלים.

Output:

Line from: x=15 y=10

To: x=7 y=6

Line from: x=15 y=10

To: x=0 y=0

משתני מחלקה לא פרימיטיביים

❖ משתני מחלקה יכולים להיות:

❖ אובייקטים (הכלה - composition)

❖ מערך של אובייקטים (מסד נתונים של חברה המכיל את רשימת כל העובדים).

❖ מצביעים לאובייקטים (מחלקת מורה יכול מצביע לקורס שהוא מלמד, ולא את הקורס עצמו. מדוע?) – נקרא גם aggregation.

מצביעים לאובייקטים

❖ בדיוק כמו שיש מצביעים למשתנים רגילים כך אנו יכולים להגדיר גם מצביעים לאובייקטים

❖ int-ל: int

int* ptInt

Point* ptPoint

❖ ל-Point: Point

❖ **חשוב לזכור:** בדיוק כמו במצביעים לפרימיטיביים – גם פה מצביעים לא בהכרח מאותחלים או תקפים (מצביעים לאובייקט קיים). אנו מוכרחים:

❖ להתחיל כל מצביע עם ערך אמיתי ותקף או NULL.

❖ להשים לתוך המצביע כתובת של אובייקט קיים, או:

❖ להקצות אובייקט חדש (על ה-heap) ולהשים את הכתובת לתוך המצביע.