

חלוקת הפרויקט לקבצים פקודות קדם מעבד וקומפילציה מותנית

**הדר בינסקי
מבוסס על שקפים של קרן כליף**

תוכן:

- חלוקת הפרויקט לקבצי header וקבצי source
- תהליך הקומפילציה
- קדם המעבד
- בעיות ב- include'ים כפולים
- קומפילציה מותנית
- מאקרו
- עבודה עם ספריות מוכנות

חלוקת הפרויקט לקבצי h וקבצי c

- עד כה השתמשנו בפונ' שקיבלנו משפת C ואשר נכתבו בספריה שאליה ביצענו `include`. (לדוגמא, `#include <string.h>`)
- כדי להתכונן לקראת יצירת ספריה משלנו נחלק את התכניות שלנו ל-2 קבצים (ולקובץ נוסף בו תכתב פונ' `main()`, הסבר בהמשך):
 - `<file_name>.h` קובץ זה יכלול את ה:
 - `prototype`'ים של הפונקציות
 - `struct`'ים
 - `include`'ים.
 - `typedef`'ים.
 - `define`'ים.
 - לקובץ זה נעשה `include` מהתכנית שתרכזה להשתמש בפונקציות המוגדרות בו.
 - `<file_name>.c` – יכיל `include` לקובץ ה-`h` התואם ויממש את הפונקציות שבו
- `include` לספריה שכתבנו יהיו בתוך "" (גרשיים) ולא בתוך `<>`

דוגמא – ספריה לניהול תווים (1)

// prototypes

```
int isSmallLetter(char);  
int isCapitalLetter(char);  
int isAlpha(char);  
int isDigit(char);
```

הקובץ character.h

דוגמא – ספריה לניהול תווים (2)

#include "character.h"

הקובץ character.c

```
int isSmallLetter(char ch)
{
    return (ch >= 'a' && ch <= 'z');
}
int isCapitalLetter(char ch)
{
    return (ch >= 'A' && ch <= 'Z');
}
int isAlpha(char ch)
{
    return (isSmallLetter(ch) || isCapitalLetter(ch));
}
int isDigit(char ch)
{
    return (ch >= '0' && ch <= '9');
}
```

דוגמא – ספריה לניהול תווים (3)

```
#include <stdio.h>
```

```
#include "character.h"
```

הקובץ main.c

```
void main()
```

```
{
```

```
    char ch=0;
```

```
    printf("Please enter a charachter: ");
```

```
    ch = getchar();
```

```
    printf("Is '%c' a digit? %d\n", ch, isDigit(ch));
```

```
    printf("Is '%c' a small letter? %d\n", ch, isSmallLetter(ch));
```

```
    printf("Is '%c' a capital letter? %d\n", ch, isCapitalLetter(ch));
```

```
    printf("Is '%c' a letter? %d\n", ch, isAlpha(ch));
```

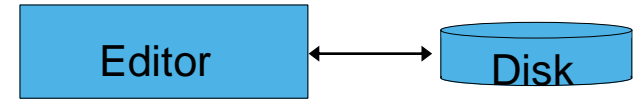
```
}
```

ספריות שלנו - סיכום

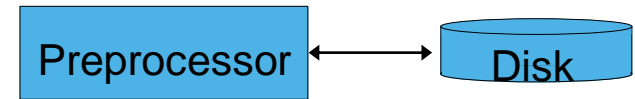
- ניתן לכתוב את כל הקוד בקובץ אחד, אבל אם אנחנו כותבים קוד כללי, שיתכן ויהיה שימושי גם במקומות אחרים, נעדיף לחלק את הקוד לקובץ ספריה נפרד
- מי שירצה להשתמש בספריה שלנו, יתעניין במה יש לה להציע, ולא באיך היא מבצעת את הפעולות, וכך הוא יוכל להסתכל בקובץ ה-header בלבד בו יש ריכוז של כל הפונקציות
- בהמשך נלמד כיצד ליצור ספריה משלנו בה כמובן לא יהיה קובץ main מאחר והיא מיועדת לשימושם של אחרים...

תהליך הקומפילציה

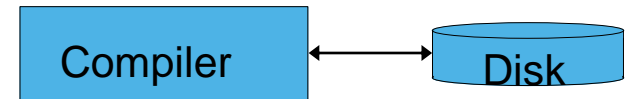
התוכנית נכתבת בעורך טקסטואלי ונכתבת לדיסק



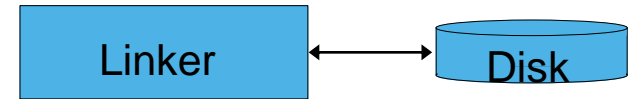
קדם המעבד עובר על הקוד ומבצע החלפות טקסטואליות נחוצות (כל הפקודות המתחילות ב- #, כמו define ו- include). התוצר הוא קובץ טקסטואלי



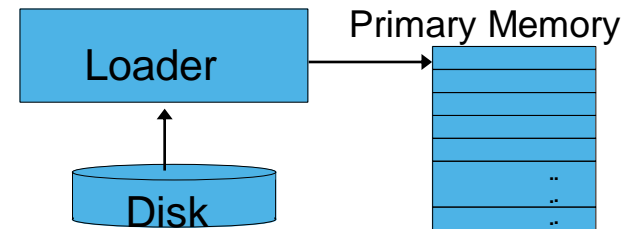
עבור כל קובץ c, הקומפילר יוצר קובץ obj בשפת מכונה ושומר אותו על הדיסק. בשלב זה רק נבדקת תקינות הסינטקס בפונקציות.



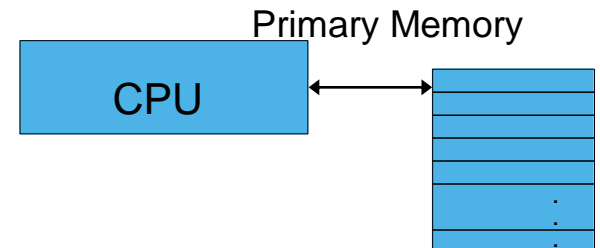
ה- linker קושר את קובץ ה- obj עם הספריות, ויוצר קובץ exe המוכן להרצה ונשמר בדיסק. כלומר, מקשר בין קריאה לפונקציה, למימוש שלה.



בעת ההרצה, טוענים את התוכנית מהדיסק לזיכרון הראשי



עם הרצת התוכנית, ה- cpu עובר על כל פקודה ומריץ אותה



פקודות של קדם-מעבד

- פקודות של קדם-מעבד הן פקודות המתחילות עם הסימן #
- פעולת ה- include היא פקודת קדם-מעבד (preprocessor) אשר מחליפה בקובץ הקוד במקום פקודת ה-include את תוכן הקובץ

a.h

```
// prototypes  
void aFoo1();  
int  aFoo2();
```

main.c

```
#include <stdio.h>  
  
#include "a.h"  
  
void main()  
{  
    aFoo1();  
    aFoo2();  
}
```



main'.c

שאותו ציינו בפקודה

```
contents of <stdio.h>  
int printf(...)  
...  
  
// prototypes  
void aFoo1();  
int  aFoo2();  
  
void main()  
{  
    aFoo1();  
    aFoo2();  
}
```

בעיתיות אפשרית (1) בפקודת include

a.h

```
// prototypes  
void aFoo1();  
int aFoo2();
```

- יתכן ונעשה include לקובץ מסוים יותר מפעם אחת:

b.h

```
#include "a.h"  
  
// prototypes  
void bGoo1();  
int bGoo2();
```

main.c

```
#include <stdio.h>  
  
#include "a.h"  
#include "b.h"  
  
void main()  
{  
    aFoo1();  
    bGoo1();  
}
```



```
#include <stdio.h>
```

```
// prototypes  
void aFoo1();  
int aFoo2();
```

```
#include "a.h"
```

```
// prototypes  
void bGoo1();  
int bGoo2();
```

```
void main()  
{  
    aFoo1();  
    bGoo1();  
}
```



```
#include <stdio.h>
```

```
// prototypes  
void aFoo1();  
int aFoo2();
```

```
// prototypes  
void aFoo1();  
int aFoo2();
```

```
// prototypes  
void bGoo1();  
int bGoo2();
```

```
void main()  
{  
    aFoo1();  
    bGoo1();  
}
```

בעיתיות אפשרית (1) בפקודת include

- יתכן ונעשה include לקובץ מסוים יותר מפעם אחת:

a.h
// prototypes
void aFoo1();
int aFoo2();

b.h
#include "a.h"

// prototypes
void bGoo1();
int bGoo2();

main.c
#include <stdio.h>

#include "a.h"
#include "b.h"

void main()
{
 aFoo1();
 bGoo1();
}

#include <stdio.h>

// prototypes
void aFoo1();
int aFoo2();

#include "a.h"

// prototypes
void bGoo1();
int bGoo2();

void main()
{
 aFoo1();
 bGoo1();
}

#include <stdio.h>

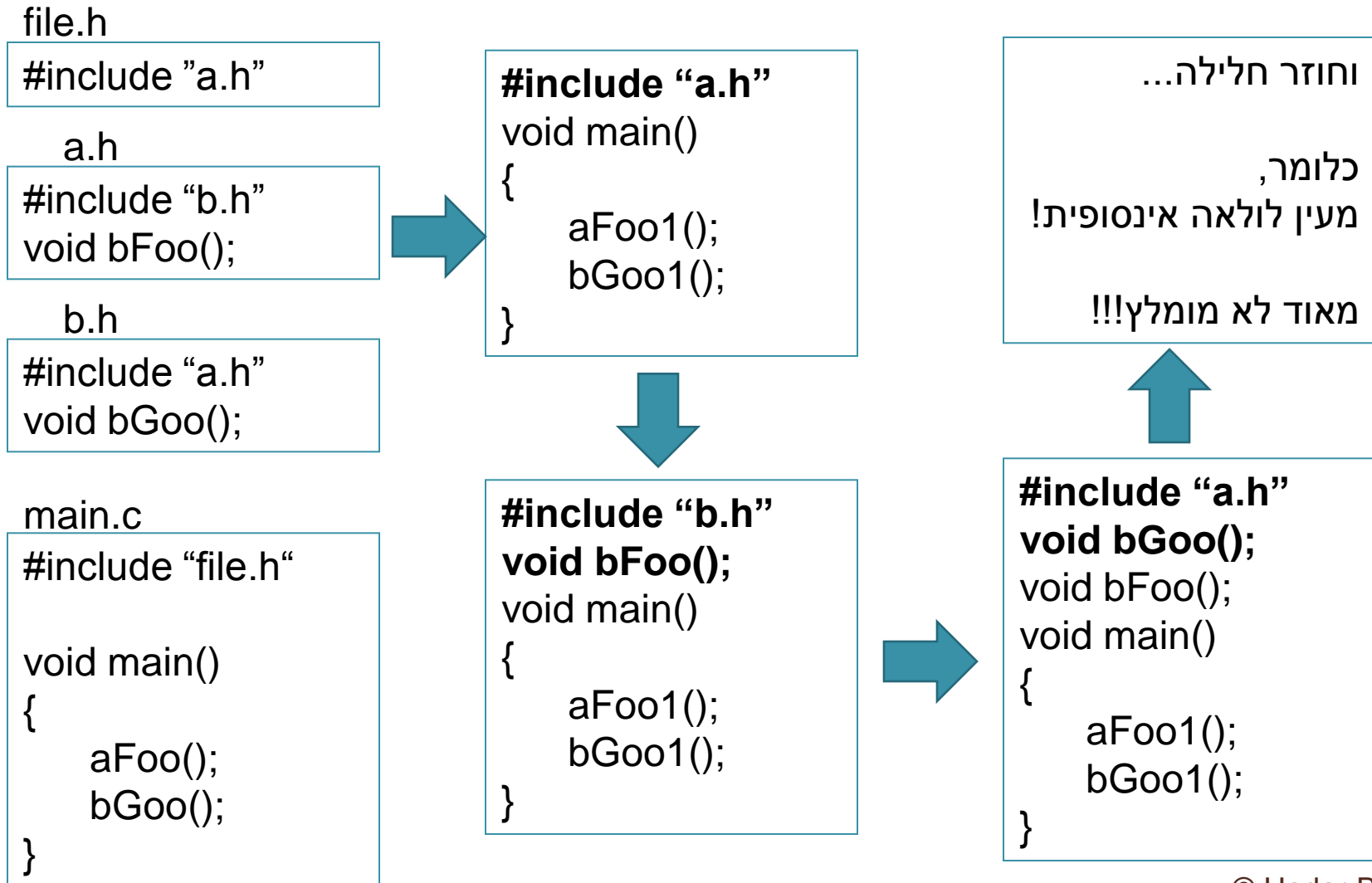
//
vo
in
//
vo
in
//
vo
in

void main()
{
 aFoo1();
 bGoo1();
}

מיותר מאחר
והקומפיילר רואה את
ההצהרה על
הפונקציות שמוגדרות
ב- a.h יותר מפעם
אחת

בעיתיות אפשרית (2) בפקודת include

- יתכן ונעשה include לקובץ מסוים מספר רב של פעמים:



הפתרון: קומפילציה מותנית

- ראינו בעבר את הפקודה `#define` לצורך הגדרת קבוע מסוים
- פקודה זו מוסיפה את הקבוע (`text1`) שהוגדר לטבלת סימולים של התוכנית עם הערך (`text2`) במידה וטרם הוגדר. במידה וכבר הוגדר דורסת את ערכו.

```
#define text1 text2
```
- ניתן גם לכתוב פקודת `#define` ללא ערך, רק כדי להכניס קבוע מסוים לטבלת הסימולים
- ניתן למחוק קבוע מטבלת הסימולים ע"י פקודת `#undef`
- ניתן לבדוק האם קבוע מסוים הוגדר בטבלת הסימולים בעזרת הפקודה `#ifdef` או אם לא הוגדר בעזרת הפקודה `#ifndef`
- במידה והתנאי מתקיים, הקומפיילר יהדר את קטע הקוד הבא עד אשר יתקל ב- `#endif` או ב- `#else` או ב- `#elseif`

הפתרון עם קומפילציה מותנית

a.h

```
#ifndef __A_H
#define __A_H
// prototypes
void aFoo1();
int aFoo2();
#endif // __A_H
```

b.h

```
#ifndef __B_H
#define __B_H

#include "a.h"

// prototypes
void bGoo1();
int bGoo2();
#endif // __B_H
```

main.c

```
#include <stdio.h>

#include "a.h"
#include "b.h"

void main()
{
    aFoo1();
    bGoo1();
}
```



טבלת הסימולים:
__A_H
__B_H

main'.c לאחר
preprocessor

```
#include <stdio.h>

// prototypes
void aFoo1();
int aFoo2();
// prototypes
void bGoo1();
int bGoo2();

void main()
{
    aFoo1();
    bGoo1();
}
```

כעת יש לנו ב-main
פעם אחת בלבד את
ההגדרות מכל קובץ

```
C:\WINDOWS\system32\cmd.exe
max is 13
max is 13
Press any key to continue .
```

מאקרו

- ניתן להשתמש בפקודת `define` כדי לבצע פעולה מסוימת, ולא רק לצורך הגדרת קבועים
- דוגמא: הגדרת מאקרו המוצא מקסימום בין 2 מספרים

```
#include <stdio.h>
```

```
#define max(a, b)  a > b ? a : b
```

```
void main()
{
    printf("max is %d\n", max(13, 5));
    printf("max is %d\n", max(5, 13));
}
```

מה שהקומפיילר רואה
לאחר `precompile`

```
#include <stdio.h>
```

```
void main()
{
```

```
    printf("max is %d\n", 13 > 5 ? 13 : 5);
    printf("max is %d\n", 5 > 13 ? 5 : 13);
}
```

מאקרו - דגשים

- בין שם המאקרו לסוגריים של הפרמטרים אסור שיהיה רווח, אחרת מתקבלת שגיאת קומפילציה שלא ממש עוזרת להבנת הבעיה

`#define max(a, b) a > b ? a : b`

- בכתיבת המאקרו מאוד מומלץ לעטוף כל פרמטר ב-`()`:

`#define mult1(x, y) (x) * (y)`

`#define mult2(x, y) x * y`

`void main()`

`{`

`printf("Result is %d\n", mult1(2+4, 6/2));`

`printf("Result is %d\n", mult2(2+4, 6/2));`

`}`

התוצר של קדם-המעבד
הוא:

`void main()`

`{`

`printf("Result is %d\n", (2+4) * (6/2));`

`printf("Result is %d\n", 2+4 * 6 / 2);`

`}`

סיכום - ההבדל בין מאקרו לפונקציה

- ראינו כי ניתן לממש פונקציות בעזרת מאקרו

- הבדלים בין פונקציה למאקרו :

- פענוח המאקרו קורה בזמן קומפילציה (בשלב ה- preprocessor)

בעוד שקריאה לפונקציה מתרחשת בזמן ריצה, ויש לקפוץ למיקום הפונקציה בזיכרון

- שימוש במאקרו מנפח את ה- EXE מאחר והוא נפרש ומשוכפל בכל

קריאה, בניגוד לפונקציה

- לשימוש במאקרו יש חיסרון שלא ניתן לדבג אותו בעזרת F10, וכן

הוא יותר קשה להבנה, כאשר הוא כולל הרבה פעולות

דוגמא לשימוש בקומפילציה מותנית ובמאקרו

- נרצה לכתוב תוכנית המכילה הדפסות לצרכי debug
- נרצה לאפשר כמה סוגים של הדפסות. למשל:
 - הדפסת ההודעה בציון שם הקובץ והשורה מהם נובעת ההודעה
 - הדפסת ההודעה ללא נתונים נוספים
 - לא להדפיס הודעות בכלל
- לא נרצה לעשות if'ים בקוד שלנו, ולכן נשתמש בקומפילציה מותנית

דוגמא לשימוש

בקומפילציה

מותנית (1)

```
#define DEBUG_WITH_DETAILS
```

```
#ifdef DEBUG_SIMPLE
```

```
    #define PRINT(str) printf("%s\n", str);
```

```
#else
```

```
    #ifdef DEBUG_WITH_DETAILS
```

```
        #define PRINT(str) printf("%s (%d): %s\n", __FILE__, __LINE__, str);
```

```
    #else
```

```
        #define PRINT(str) ;
```

```
    #endif
```

```
#endif
```

```
void foo()
```

```
{
```

```
    PRINT("--> foo");
```

```
    PRINT("<-- foo");
```

```
}
```

```
void main()
```

```
{
```

```
    PRINT("--> Main");
```

```
    foo();
```

```
    PRINT("<-- Main");
```

```
}
```

__FILE__ הוא מאקרו קיים לשם הקובץ הנוכחי
__LINE__ הוא מאקרו קיים לשורה הנוכחית בקובץ

C:\WINDOWS\system32\cmd.exe

```
e:\_general tests\c tests\argument_list.c <24>: --> Main
e:\_general tests\c tests\argument_list.c <18>: --> foo
e:\_general tests\c tests\argument_list.c <19>: <-- foo
e:\_general tests\c tests\argument_list.c <26>: <-- Main
Press any key to continue . . . _
```

דוגמא לשימוש בקומפילציה מותנית (2)

```
#define DEBUG_SIMPLE
```

```
#ifdef DEBUG_SIMPLE
```

```
    #define PRINT(str) printf("%s\n", str);
```

```
#else
```

```
    #ifdef DEBUG_WITH_DETAILS
```

```
        #define PRINT(str) printf("%s (%d): %s\n", __FILE__, __LINE__, str);
```

```
    #else
```

```
        #define PRINT(str) ;
```

```
    #endif
```

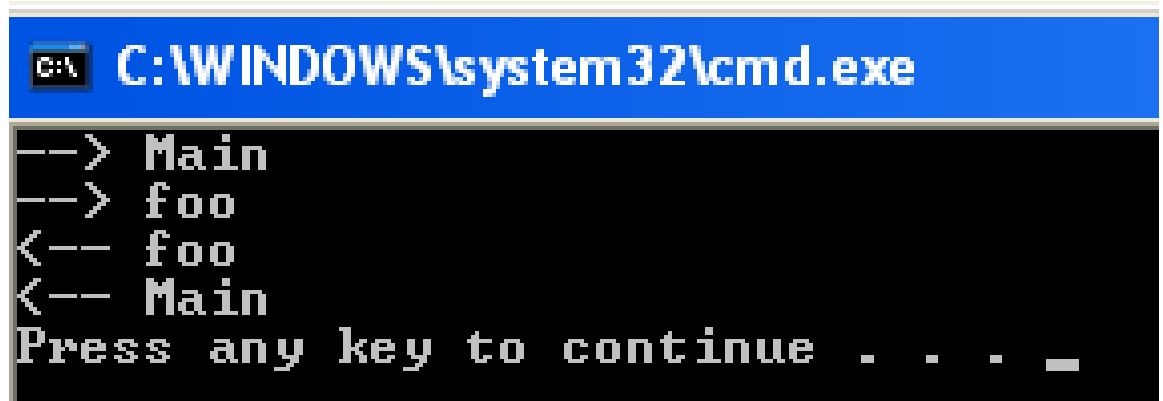
```
#endif
```

```
void foo()
```

```
{  
    PRINT("--> foo");  
    PRINT("<-- foo");  
}
```

```
void main()
```

```
{  
    PRINT("--> Main");  
    foo();  
    PRINT("<-- Main");  
}
```



```
C:\WINDOWS\system32\cmd.exe  
--> Main  
--> foo  
<-- foo  
<-- Main  
Press any key to continue . . . _
```

דוגמא לשימוש

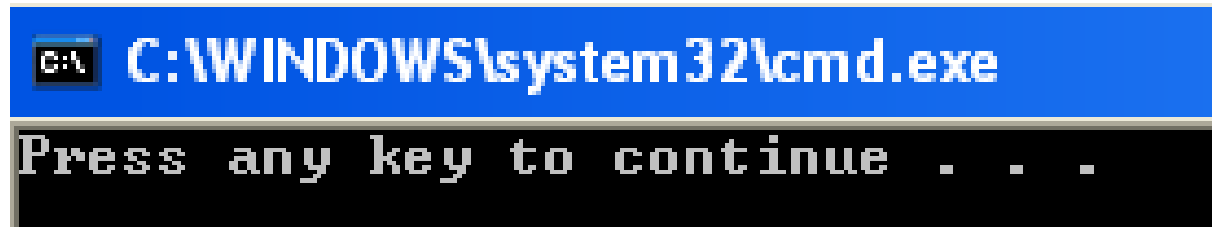
בקומפילציה

מותנית (3)

```
#ifdef DEBUG_SIMPLE
    #define PRINT(str) printf("%s\n", str);
#else
    #ifdef DEBUG_WITH_DETAILS
        #define PRINT(str) printf("%s (%d): %s\n", __FILE__, __LINE__, str);
    #else
        #define PRINT(str) ;
    #endif
#endif
```

```
void foo()
{
    PRINT("--> foo");
    PRINT("<-- foo");
}
```

```
void main()
{
    PRINT("--> Main");
    foo();
    PRINT("<-- Main");
}
```



דוגמא לשגיאת קומפילציה (1)

```
#include <stdio.h>
```

```
void foo(int x)
{
    printf("the number is %d\n");
}
```

```
void main()
{
    for (i=0 ; i < 5 ; i++)
    {
        printf("%d ", i);
        goo(i);
    }
}
```

נקבל את שגיאת הקומפילציה הבאה:
error C2065: 'i' : undeclared identifier

הקומפיילר רק נותן אזהרה שהוא לא מוצא את goo:
warning C4013: 'goo' undefined; assuming
extern returning int

במידה ויש שגיאות קומפילציה, הקומפיילר לא ממשיך לתהליך לינקר

דוגמא לשגיאת קומפילציה (2)

- לעיתים מקבלים שגיאות שקשה לאתר את הבעיה:

```
#include <stdio.h>

#define SIZE=5;

void main()
{
    int arr[SIZE];
}
```

✖	2	error C2008: '=' : unexpected in macro definition
✖	3	error C2143: syntax error : missing ']' before ';'
✖	4	error C2143: syntax error : missing ';' before ']'

- הצגת התוצר של שלב הקדם-מעבד שאותו המחשב מקמפל, היה עוזר לפתור את הבעיה:

```
void main()
{
    int arr[=5;];
}
```

דוגמא לשגיאת לינקר

```
#include <stdio.h>
```

```
void foo(int x)
{
    printf("the number is %d\n");
}
```

```
void main()
{
    int i;

    for (i=0 ; i < 5 ; i++)
    {
        printf("%d ", i);
        goo(i);
    }
}
```

תוכנית זו מתקמפלת (אין שגיאות סינטקטיות בתוך הפונקציות) ולכן הקומפיילר עובר לתהליך הלינקר, ומוציא את השגיאה הבאה:

**error LNK2019: unresolved external symbol
_goo referenced in function _main**

עבודה עם ספריות מוכנות

- בעולם האמיתי נשתמש בספריות מוכנות שהורדנו מהאינטרנט / שקנינו מאישזיהי חברה וכד'
- ספריות כאלו לרוב לא יכילו את הקוד עצמו (כדי לא לחשוף את האלגוריתם שלהם), אלא את הקוד מקודד באופן בינארי המוכן לשימוש, ולא לקריאה
- לצורך ההקבלה: כאשר אנחנו קונים תוכנה מסויימת, אנחנו מקבלים רק את קובץ ההרצה שלה, ולא את הקוד של איך התוכנית כתובה
- בניגוד לקבצי הרצה, לא יהיה בתוך הספרייה פונקציית main
- אם קובץ עם סיומת ס הוא תוצר קומפילציה בינארי של קובץ C אחד, אז ספרייה הינה אוסף של כמה קבצים עם סיומת ס (למעשה סוג של מארז)

יצירת קובץ ספריה

- כדי לייצר ספריה משלנו נפתח פרויקט שאינו מייצר exe, אלא lib. (ראו בשקף הבא).
- בנוסף, נייצר 2 קבצים חדשים בלבד:
 - `<file_name>.h` – קובץ זה יכלול את ה:
 - prototype של הפונקציות
 - struct'ים
 - include'ים.
 - typedef'ים.
 - define'ים.
 - `<file_name>.c` – יכיל `include` לקובץ ה- header התואם ויממש את הפונקציות שבו

קומפילציה ב- windows: יצירת lib

- נפתח פרויקט שאינו מייצר exe, אלא lib. השינוי היחידי בשלב פתיחת הפרוייקט הוא במסך הבא (המסך בו אנחנו תמיד מסמנים

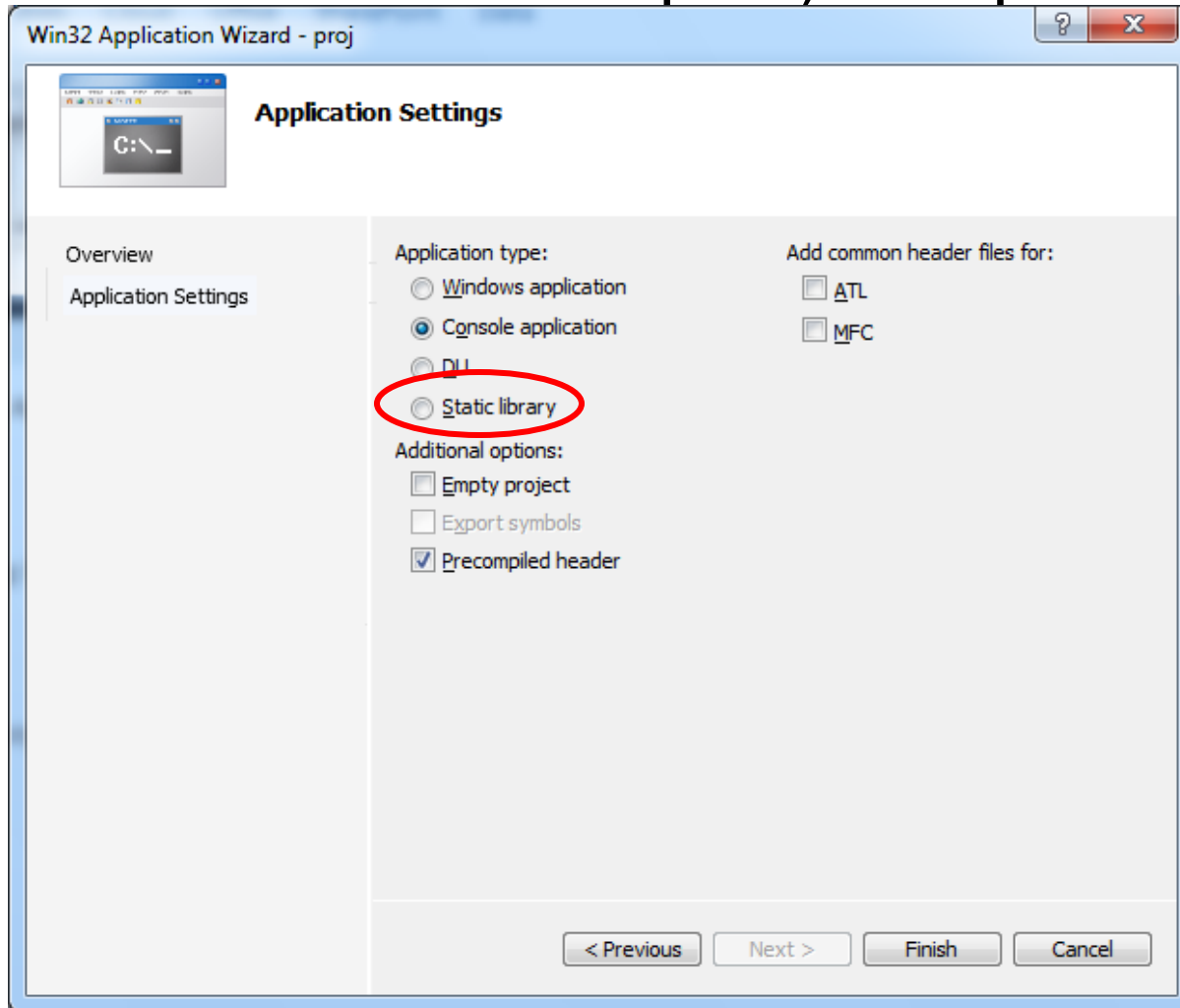
:(Empty project

- נבחר ב:

Static library

וכמובן לא לשכוח:

Empty project

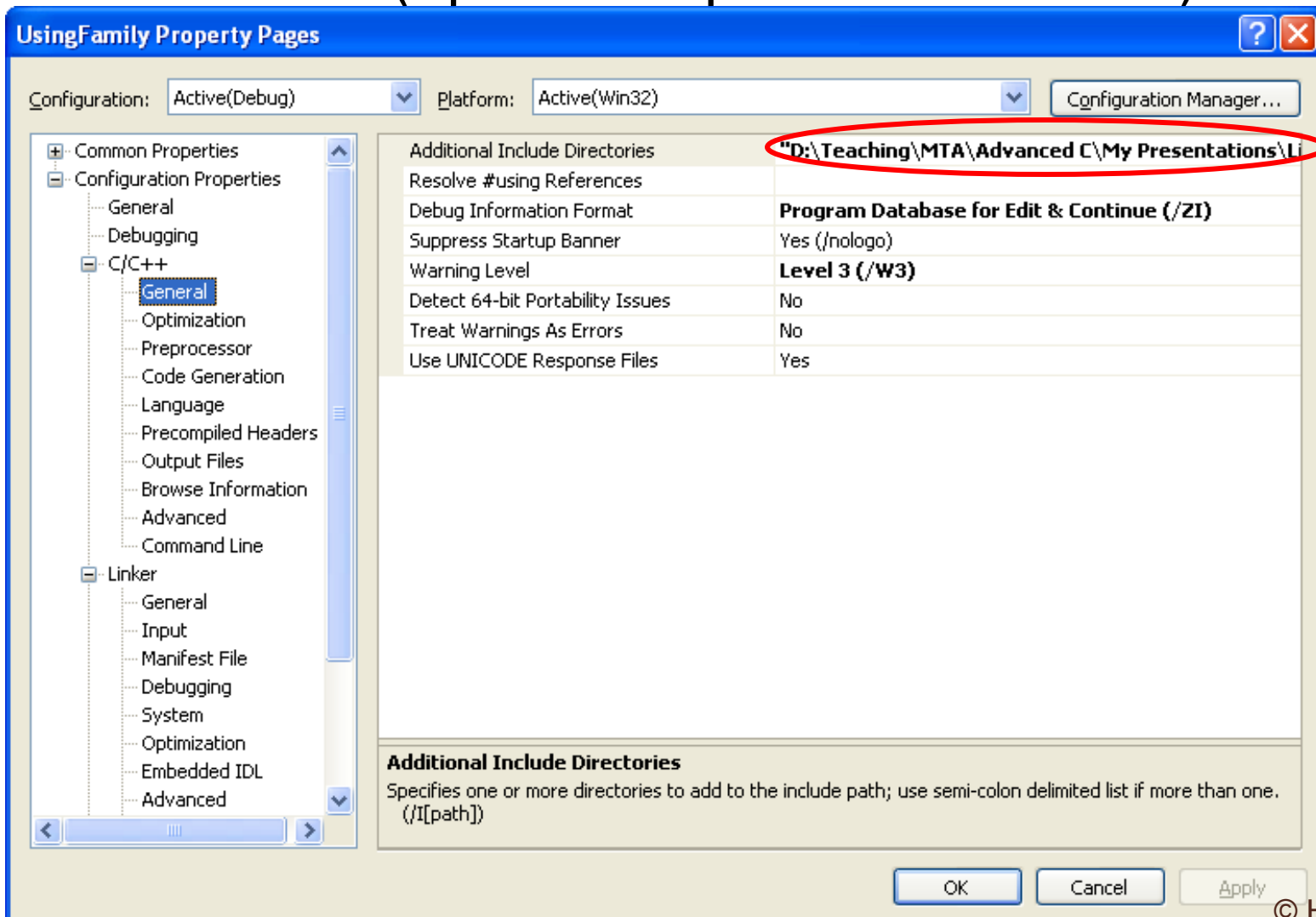


שימוש בקובץ ספריה "חדש"

- כעת נלמד כיצד להשתמש בספריה משלנו, כלומר, ספריה שלא כלולה בהתקנה של סביבת העבודה. זו יכולה להיות ספריה שייצרנו לבד וזו גם יכולה להיות ספריה שהורדנו מהאינטרנט.
- על מנת להשתמש בספריה משלנו עלינו לשנות את מאפייני הפרויקט עליו אנו עובדים כדי שידע היכן למצוא את:
 - קובץ ה-h.
 - קובץ ה-lib.
- בנוסף, עלינו "לומר" בצורה מפורשת ל-Linker כי ברצוננו שישתמש בקובץ ספריה משלנו (זה לא מספיק שאמרנו לפרויקט היכן למצוא את קובץ ה-lib אלא יש גם לומר ל-Linker כי עליו להשתמש בו).

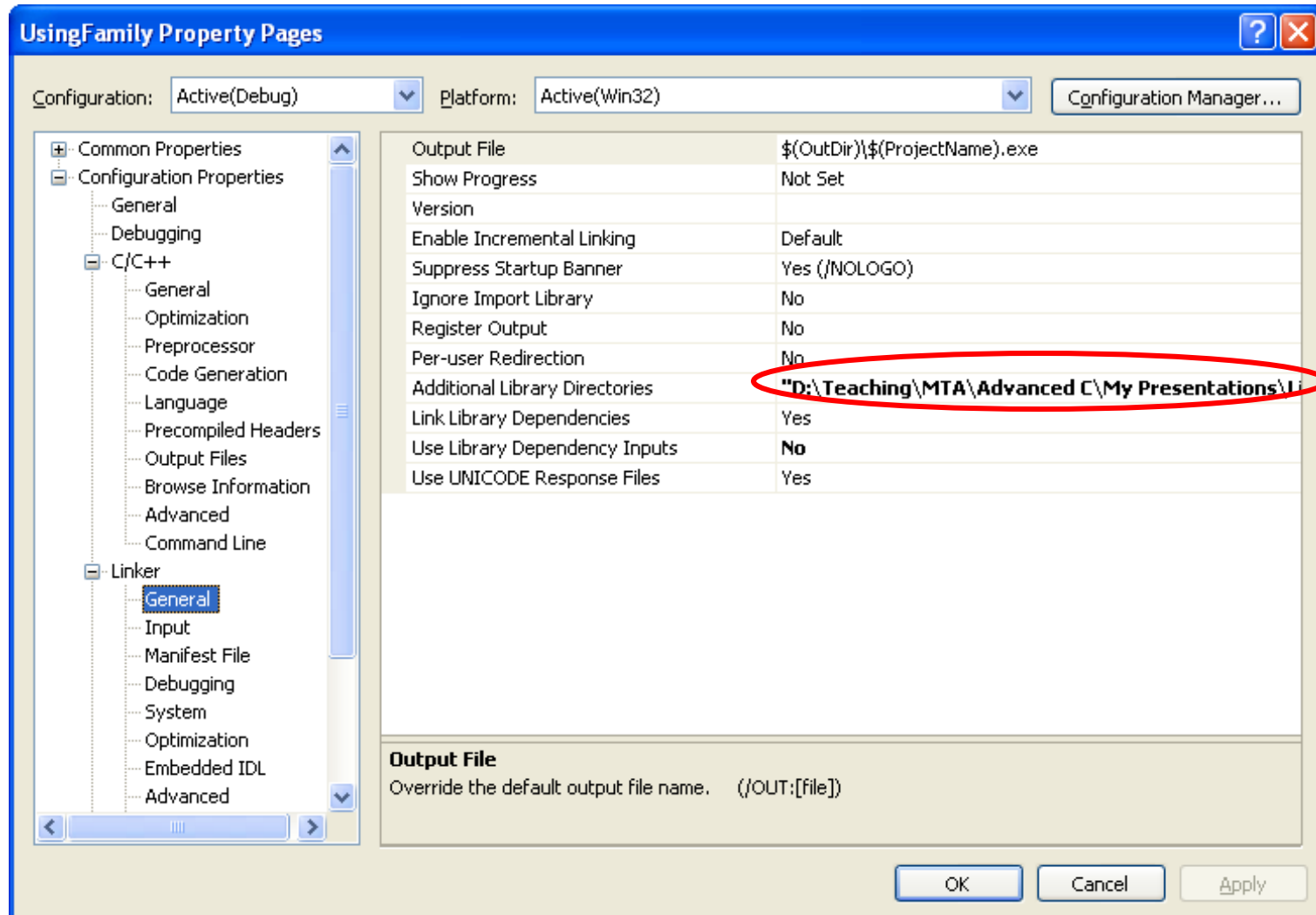
קומפילציה ב- windows: שימוש ב- lib

- נפתח פרוייקט רגיל, ונשנה את הגדרותיו כדי "שידע" היכן למצוא את קובץ ה-H (במידה והוא לא בתיקיית הפרוייקט):



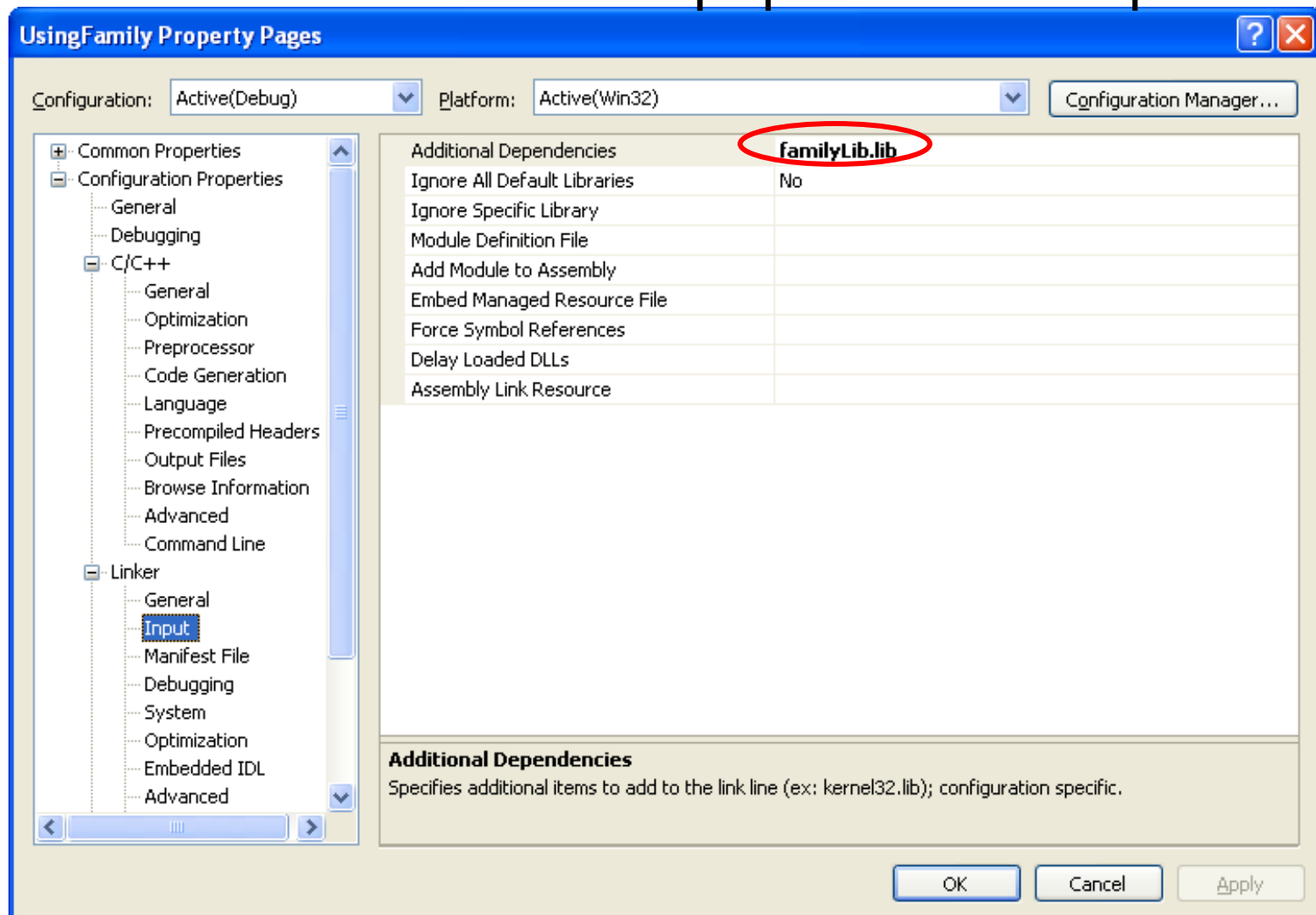
קומפילציה ב- windows: שימוש ב- lib (2)

ואת קובץ ה-LIB:



קומפילציה ב- windows: שימוש ב- lib (3)

ולבסוף שהלינקר גם יכול את קובץ ה-LIB:



קישור דינאמי לספריה לעומת קישור סטטי

- הקישור שעשינו לספריה הינו קישור סטטי
- כלומר, החלפת הספריה בספריה זרה, שרק מימוש הפונקציות שונה, תדרוש קומפילציה מחדש של הקובץ שהשתמש בספריה
- ניתן לבצע קישור דינאמי
- כלומר, כל פעם שתהיה הרצה של התוכנית ויהיה שימוש בפונקציה מהספריה, הקומפילר יחפש את המימוש בזמן ריצה
- לכן ניתן להחליף את הספריה ללא לקמפל מחדש
- שימו לב: החלפת הספריה משמע שהיא תכיל את אותן פונקציות עם אותן חתימות בדיוק, רק המימושים שונים!

מוזמנים ללמוד לבד את הנושא 😊

אז מה למדנו:

- חלוקת הפרויקט לקבצי header וקבצי source
- תהליך הקומפילציה
- קדם המעבד
- בעיות ב- include'ים כפולים
- קומפילציה מותנית
- מאקרו
- עבודה עם ספריות מוכנות