

חלוקה של פרוייקט למספר קבצים

מדוע לחלק תוכנה

תוכנה המבצעת מטלה מסוימת יכולה להיות מורכבת ממספר גדול מאד של שורות קוד. בזמן הפיתוח והתחזוקה של התוכנה, יש צורך בתרגום חוזר של הקוד משפה עילית לשפת המכונה. התרגום (קומפילציה) הזה דורש זמן ולכן יש טעם להפריד את התוכנה הגדולה ליחידות (קבצים) שיתורגמו בנפרד. אילו לא היינו מפרידים, היינו עלולים לתרגם מאות אלפי שורות קוד עבור כל שינוי קטן שנעשה בפרוייקט גדול. סיבה נוספת לחלוקה של תוכנה היא מודולריות. כאשר התוכנה מורכבת מקבצים שונים בעלי משמעות מוגדרת, קל יותר להחליף חלקים או לשנות אותם. הדבר מאפשר גם בניית תוכנית הבנויה מקודים הכתובים בידי אנשים שונים ואפילו בשפות שונות. בדרך כלל משתדלים שבחלוקה לקבצים יהיה הגיון מבני מסויים כך שכל קובץ יכיל אוסף של פונקציות והצהרות הקשורים זה בזה ומבצעים יחד מטלה מסויימת. פרוייקטים גדולים מחולקים למודולים (modules), כל מודול הוא יחידת תוכנה בעלת תפקיד מוגדר שלפעמים אף ניתן להריץ ובדוק באופן עצמאי. לרוב החלוקה לקבצים מהווה העדנה של החלוקה למודולים.

חלוקת תוכנית פשוטה לשני קבצים

כידוע, תרגום של תוכנית משפה עילית לשפת מכונה מתבצעת ע"י תוכנת מחשב בשם compiler. במובן הכללי קומפיילר הוא תוכנית המקבלת קובץ תוכנית בשפה אחת ויוצרת קובץ תוכנית שקולה בשפה אחרת. במקרה הפרטי בו אנו עוסקים מדובר בקומפיילר המתרגם תוכנית ב-C לתוכנית שקולה בשפת המכונה של המעבד עליו אנו עובדים. כאשר מקמפלים פרוייקט הבנוי ממספר קבצים של C, מפעילים את הקומפיילר מחדש עבור כל אחד מהקבצים ובסוף מחברים את תוצאות כל ההפעלות.

ניקח לדוגמה את תוכנית הבאה:

```
#include <stdio.h>

int foo(int a,int b) {
    return a*b;
}

int main() {
    printf("%d\n",foo(4,5));
    return 0;
}
```

נניח שאנחנו רוצים לחלק אותה לשני קבצים, האחד שיכיל את הפונקציה foo והשני את הפונקציה main. נכתוב את שני הקבצים הבאים:

file f1.c:

```
int foo(int a,int b) {
    return a*b;
}
```

```
}
```

file f2.c:

```
#include <stdio.h>
int foo(int a,int b);
int main() {
    printf("%d\n",foo(4,5));
    return 0;
}
```

הקומפיילר שיתרגם את הקוד של f2.c חייב לדעת שמדובר בקוד C תקין למרות שלא מדובר בתוכנית שלמה. ההצהרה של foo המופיעה כאן, נועדה להגיד לקומפיילר כי, foo היא פונקציה הקיימת בפרוייקט אך ממומשת במקום אחר. אילו לא היינו מוסיפים את ההצהרה הזו, הייתה מתעוררת בעיית קומפילציה בזמן שהקומפיילר היה פוגש את הקריאה לפונקציה: foo(4,5).

על מנת לקמפל את התוכנית וליצור קובץ הרצה, עלינו לכתוב:

```
$ gcc -c -Wall f1.c
```

```
$ gcc -c -Wall f2.c
```

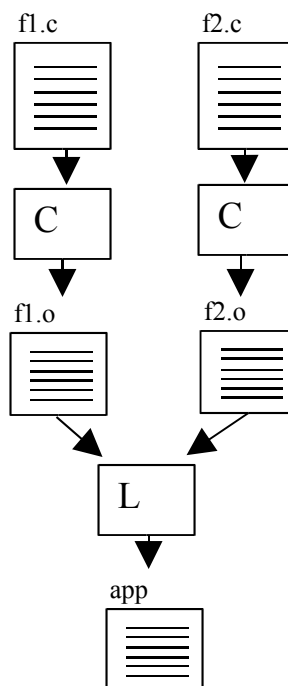
```
$ gcc f1.o f2.o -o app
```

כאשר app הוא שם שרירותי שבחרנו לקובץ ההרצה.

כאשר תרגמנו את f1.c השתמשנו באופציית הקומפילציה '-c'. אופציה זו, מורה לקומפיילר לתרגם את הקוד שבקובץ אך לא ליצור ממנו קובץ הרצה אלא רק קובץ תרגום שיוכל להתחבר מאוחר יותר לקבצים מתורגמים אחרים. קובץ המכיל תרגום של קטע קוד אך איננו מהווה קובץ ריצה, ניקרא **object file** והוא מאופיין בדרך כלל ע"י הסימט 'o.' או 'obj.'. ב - gcc כאשר אין מציינים במפורש את שם ה - object file ניתן לו השם המקורי עם הסימט 'o.'. ראוי להעיר כי אין קשר בין השם object file לפרדיגמת התכנות object oriented.

לאחר שתרגמנו את f1.c ו f2.c לשני קבצי object בשם f1.o ו- f2.o, חיברנו אותם ביחד לקובץ הרצה ע"י הפקודה gcc f1.o f2.o -o app. כעת הפעלה של app מפעילה את התוכנית כולה. לפעולה של חיבור קבצי object לקובץ ריצה קוראים **linking**. שימו לב שכשדרשנו מ- gcc לתרגם קבצי C ל gcc, object files, תפקדה כ - compiler אך כאשר דרשנו מ gcc לחבר קבצי object וליצור קובץ הרצה, gcc תפקדה כ - linker.

נתאר את התהליך כולו בצורה סכמתית:



השתמשנו כאן ב-gcc פעמיים בתור קומפיילר ופעם בתור Linker.

Header files

נניח שיש לנו קובץ המכיל שתי פונקציות שימושיות:

File utils.c:

```
void f1(double a) {...}
double f2(int c, int b) {...}
```

כל קובץ אחר בפרוייקט שירצה להשתמש בשתי הפונקציות האלה יאלץ להצהיר אליהן ליפני השימוש:

File other.c:

```
void f1(double a);
double f2(int c, int b);
```

```
void foo() {
    f2(7,17);
    ...
    f1(2);
}
```

```
}
```

כתיבת ההצהרות שוב ושוב בכל קובץ המשתמש בפונקציות מהווה שכפול קוד. שכפול קוד מפריע לתחזוקה נוחה של תוכנית מכיוון שכל שינוי שרוצים לעשות יש לעשות במספר מקומות. לעיתים שוכחים לבצע את השינוי בכל המקומות הדרושים ואז נוצרת חוסר עקביות. בכדי להתגבר על הבעיה של שכפול קוד וגם בכדי לחסוך את כתיבת ההצהרות של הפונקציות שוב ושוב, ניתן לכתוב אותן בקובץ ניפרד שישתל בכל קובץ המשתמש בפונקציות. לקובץ הצהרות הנועד להשתלה בקבצים אחרים, קוראים **header file** ומסמנים אותו ע"י הסיומת `h:`.

File utils.h:

```
void f1(double a);  
double f2(int c, int b);
```

File other.c:

```
#include "utils.h"  
void foo() {  
    f2(7,17);  
    ...  
    f1(2);  
}
```

פקודת ה- `include#` גורמת להשתלת הקובץ `utils.h`, כפי שהוא, בתוך הקובץ `other.c`. פקודת הקומפילציה נשארת כמו שהייתה:

```
gcc -c -Wall other.c
```

החלק הראשון בתהליך הקומפילציה נקרא `pre-processing`. זהו עיבוד ראשוני של `text` ללא תרגום ממש. החלק של הקומפיילר האחראי על העיבוד הראשוני ניקרא `pre-processor`. כל הפקודות המתחילות ב- `#` הן בעצם הוראות ל- `pre-processor`. הפקודה `include#` `<<stdio.h` שאנחנו רגילים לכתוב, משמעותה – השתלת הקובץ `stdio.h` בתחילת קובץ הקוד שלנו. כאשר שם הקובץ מופיע בתוך גרשיים, ה- `pre-processor` מחפש אותו בתיקה הנוכחית, אם השם בסוגריים משולשים, הוא מחפש בתיקות מיוחדות המוגדרות במערכת.

חלוקה של פרויקט המכיל structs

כאשר כותבים פרויקט מרובה קבצים, משתדלים שכל קובץ ייתן אוסף שירותים מוגדרים ובכך ליצור תוכנית מודולרית גם ברמת הקבצים. הדוגמה הבאה היא תוכנית המאפשרת שימוש בנקודות ומשולשים על המישור. בחרנו להפריד את הקוד, המשולש והקוד המשתמש בהם. התוכנית הספציפית היא אולי קטנה ומנוונת מדי בשביל הפרדה כזו אבל הדוגמה היא טיפוסית ומעלה בעיות והתלבטויות העולות בתוכניות אמיתיות.

לכל קובץ המספק שירותים נכתוב header file הכולל את ההצהרות שלו ומאפשר לקבצים אחרים להשתמש בו. נתחיל בקובץ המתאר את הנקודה:

file Point.h:

```
struct Point {
    int m_x,m_y;
};

struct Point* createPoint(int x, int y);
void movePoint(struct Point* p, int dx, int dy);
void printPoint(struct Point pnt);
```

כפי שניתן לראות, קובץ ה- header מכיל את ההצהרה על טיפוס המבנה של הנקודה, פונקציה היוצרת נקודה, פונקציה המזיזה נקודה ופונקציה המדפיסה נקודה.

כעת נוסיף את קובץ המימוש של הנקודה:

file Point.c:

```
#include <stdlib.h>
#include <stdio.h>
#include "Point.h"

struct Point* createPoint(int x, int y) {
    struct Point *p = (struct Point*) malloc (sizeof(struct Point));
    p->m_x = x;
    p->m_y = y;
    return p;
}

void movePoint(struct Point* p, int dx, int dy) {
    p->m_x += dx;
    p->m_y += dy;
}

void printPoint(struct Point pnt) {
    printf("(%d,%d)",pnt.m_x,pnt.m_y);
}
```

שימו לב שגם קובץ המימוש צריך לכלול את ה- header המתאים. בדוגמה הקודמת, שקובץ ה- header כלל רק הצהרות על פונקציות זה לא היה הכרחי, אבל כאשר הוא כולל גם הצהרות על טיפוסים, זה הכרחי.

כעת נוסיף גם את קובץ ה- header של המשולש:

file Triangle.h:

```
#include "Point.h"
struct Triangle {
    struct Point m_p1, m_p2, m_p3;
};

struct Triangle* createTriangle(struct Point p1,
                                struct Point p2,
```

```

        struct Point p3);
void printTriangle(struct Triangle trn);

```

מכיוון שמשולש בנוי מנקודות, היינו חייבים לכלול את ה - header של הנקודה כבר בשלב ההצהרות שלו.

נוסיף את קובץ המימוש שלו:

file Triangle.c:

```

#include <stdlib.h>
#include <stdio.h>
#include "Triangle.h"
struct Triangle* createTriangle(struct Point p1,
                               struct Point p2,
                               struct Point p3) {
    struct Triangle *p = (struct Triangle*)malloc(sizeof(struct Triangle));
    p->m_p1 = p1;
    p->m_p2 = p2;
    p->m_p3 = p3;
    return p;
}

```

```

void printTriangle(struct Triangle trn) {
    printf("[");
    printPoint(trn.m_p1);
    printf(",");
    printPoint(trn.m_p2);
    printf(",");
    printPoint(trn.m_p3);
    printf("]");
}

```

לבסוף נוסיף קובץ המכיל את main ומשתמש בנקודות ומשולשים:

file main.c:

```

#include <stdlib.h>
// #include "Point.h"
#include "Triangle.h"
int main() {

    struct Point* p = createPoint(3,2),p1=*p,p2=p1,p3=p1;
    struct Triangle* tr;

    movePoint(&p2,1,1);
    movePoint(&p3,-1,1);

    tr = createTriangle(p1,p2,p3);
    printTriangle(*tr);

    free(p);
    free(tr);
    return 0;
}

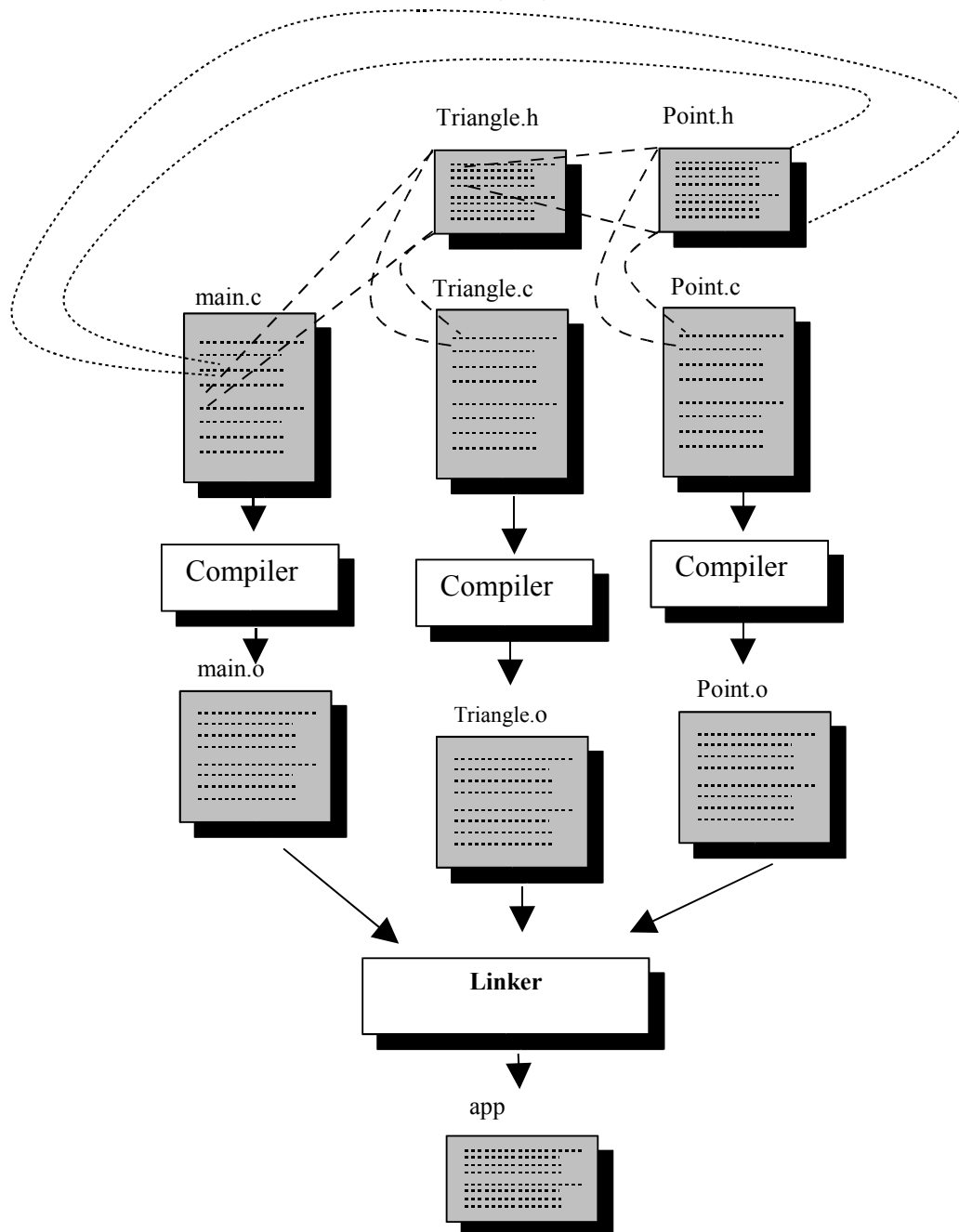
```

בשלב זה ניתן כבר לקמפל את הפרוייקט:

```
$ gcc -c -Wall Point.c
$ gcc -c -Wall main.c
$ gcc -c -Wall Triangle.c
$ gcc main.o Triangle.o Point.o -o app
```

בסופו של התהליך נוצר קובץ הריצה app המורכב מתרגום של כל הקבצים. הסדר של פקודות הקומפילציה (gcc -c) הוא שרירותי. פקודת ה- linking חייבת כמובן להיות אחרונה מכיוון שהיא משתמשת בתוצאות כל פעולות הקומפילציה. סדר קבצי ה- object הניתנים ל- linker גם הוא שרירותי אך אחד ויחיד מבניהם חייב להכיל את main.

בעיית ההכללה הכפולה (או בעיית ההצהרה הכפולה)
נסכם בשרטוט סכמתי את היחסים הדדיים בין הקבצים:



הקווים המקווקוים מסמנים את פעולת השתילה המבוצעת ע"י ה - preprocessor כתוצאה מפקודת #include. שימו לב שהקווים המסמנים את הכלת הקובץ Point.h ב main.c חלשים יותר. למעשה, בקוד שכתבנו לא קיימת הכלה כזאת מכיוון שהפקודה #include"Point.h" נמצאת בהערה בקובץ main.c.

מכיוון שהקובץ main.c משתמש הן במשולשים והן בנקודות, טבעי שיכלול את קבצי ה - header של שניהם. ננסה לעשות זאת ע"י הוצאתה של הפקודה #include "Point.h" מההערה בה היא נמצאת. כעת, כאשר ננסה לקמפל את main.c:

```
$ gcc -c -Wall main.c
```

נקבל שגיאת קומפילציה המתלוננת ש - struct Point הוגדר יותר מפעם אחת.

את הסיבה לשגיאה אפשר לראות בתרשים הסכמתי: הקובץ Point.h הוכלל ב - main.c

פעמיים: פעם ישירות ופעם באופן עקיף דרך Triangle.h.

באופן יותר מפורש, הפקודות:

```
#include "Point.h"
```

```
#include "Triangle.h"
```

גורמות ל - preprocessor לשתול ראשית את הקובץ Point.h:

```
struct Point {
    int m_x,m_y;
};
```

...

```
#include "Triangle.h"
```

ואז לשתול את Triangle.h:

```
struct Point {
    int m_x,m_y;
};
```

...

```
#include "Point.h"
```

```
struct Triangle {
    struct Point m_p1, m_p2, m_p3;
};
```

...

ואז לשתול את שוב את Point.h:

```
struct Point {
    int m_x,m_y;
};
```

...


```

struct Point {
    int m_x,m_y;
};
...
struct Triangle {
    struct Point m_p1, m_p2, m_p3;
};
...

```

כפי שניתן לראות, ב - main הושתלה ההצהרה על struct main פעמיים. לפי חוקי השפה C, אסור ליחידת קומפילציה (קובץ המוכן לתרגום) להכיל הצהרה כפולה על טיפוס. הצהרות כפולות על פונקציות הן מותרות ולכן לא גורמות כאן לשגיאות קומפילציה.

איך ניתן לפתור את בעיית ההכללה הכפולה ?

דרך אחת היא הדרך בה נקטנו בקוד המקורי: מכיוון ש Triangle.h כבר גורם להכללת Point.h לא כללנו אותו. זה איננו פתרון מספק. פתרון זה דורש מהמתכנת המשתמש ב - Triangle לדעת בדיוק איזה קבצי header מוכללים בגללו ואז לא להכליל אותם. בפרוייקט גדול יכול להיות שמדובר בקבצים רבים שנכללים באופן עקיף מאד. לדעת איזה קבצים נכללים באופן עקיף כתוצאה מ - Triangle עלולה להיות עבודה מייגעת. אם בשלב מסוים ירצה המשתמש לשנות את תוכניתו ולא להשתמש ב - Triangle הוא יאלץ לכלול את כל ה - headers שקודם נאסרו עליו. אפשרות אחרת היא לשנות את Triangle באופן הבא:

file Triangle.h:

```

// no include here
struct Triangle {
    struct Point m_p1, m_p2, m_p3;
};

struct Triangle* createTriangle(struct Point p1,...

```

file Triangle.c:

```

#include <stdlib.h>
#include <stdio.h>
#include "Point.h" // <-- moved to here
#include "Triangle.h"
struct Triangle* createTriangle(struct Point p1,...

```

לאחר השינוי נוכל לקמפל את הפרוייקט כך שפקודת ההכללה של Point.h נמצאת מחוץ להערה ב - main.c.

פתרון זה דורש מהמתכנת המשתמש ב - Triangle להכליל את Point.h לפני הכללת Triangle.h. אם המשתמש לא יעשה זאת ההצהרה של struct Point תחסר להצהרה של Triangle.h. החיוב להכליל את Point.h לפני הכללת Triangle.h יוצר בעיות דומות לחיוב לא

להכליל אותו. שוב, איזה קבצים בדיוק צריך לכלול? מה השינויים שצריך לעשות בקוד אם מחליטים לא להשתמש ב- Triangle?

לבעיית ההכללה הכפולה יש פתרון המנצל פקודות נוספות של ה- preprocessor. ה- preprocessor מאפשר להתעלם מקטעי קוד שלמים אם תג מסוים הוגדר. כמו כן מאפשר ה- preprocessor להגדיר תגים. נשנה את Point.h באופן הבא:

```
#ifndef POINT_H
#define POINT_H
struct Point {
    int m_x,m_y;
};
struct Point* createPoint(int x, int y);
void movePoint(struct Point* p, int dx, int dy);
void printPoint(struct Point pnt);
#endif
```

השינוי הוא תוספת שתי פקודות preprocessor בתחילת הקובץ ואחת בסופה. הפקודה הראשונה משמעותה "if not defined" כלומר אם לא מוגדר התג POINT_H אז יש להתייחס לכל הקוד עד סוף התנאי, עד ה- #endif. הפקודה הבאה מגדירה את התג POINT_H. לאחר השינוי אפשר לקמפל את הפרוייקט המקורי ללא ההערה וכן לכלול את Point.h בכל קובץ ללא חשש שייכלל פעמיים. נתאר איך עובד המנגנון. כאשר מתרגמים את main.c:

```
$ gcc -c -Wall main.c
```

ראשית מופעל ה- preprocessor ואף תג אינו מוגדר. ב- main.c כתובות הפקודות הבאות:

```
#include "Point.h"
#include "Triangle.h"
```

ה- preprocessor ניגש ל- Point.h על מנת להכליל אותו. בתחילת Point.h הוא נשאל אם התג POINT_H מוגדר. מכיוון שהתג לא מוגדר הוא מתייחס לקוד עד ה- #endif כך שכל הקוד של הקובץ מוכלל ב- main.c. הפקודה הראשונה בקוד שאליו הוא מתייחס גורמת להגדרת התג POINT_H. לאחר מכן ניגש ה- preprocessor לכלול את Triangle.h. בתחילת Triangle.h יש דרישה לכלול את Point.h. כאשר ה- preprocessor ניגש ל- Point.h הוא נשאל שוב אם POINT_H מוגדר. מכיוון שהוא אכן מוגדר, ה- preprocessor מתעלם מכל הקוד שב- Point.h ושותל קובץ

ריק ב - Triangle.h . לאחר מכן נשתל הקובץ Triangle.h ב - main.c . בסופו של התהליך, רק עותק אחד של הקוד של Point.h של Point.h נשתל ב - main.c .
ראוי לציין שכל קומפילציה מתבצעת בנפרד ובאופן בלתי תלוי בקומפילציות האחרות. אם לאחר פקודת הקומפילציה הקודמת נבצע את פקודת הקומפילציה הבאה:
\$ gcc -c Wall Triangle.c
יתחיל התהליך מהמצב הראשוני : בהתחלה ה - preprocessor ללא שום תגים מוגדרים ואז התרגום.

המנגנון למניעת הכללה כפולה שתואר כאן הוא דרך תכנות מקובלת ב - C ומיישמים אותה בכל קובץ header גם אם בשלב זה אין סכנה שייכלל פעמיים. את Triangle.h נשכתב באופן הבא:

```
#ifndef TRIANGLE_H
#define TRIANGLE_H
#include "Point.h"
struct Triangle {
    struct Point m_p1, m_p2, m_p3;
};
struct Triangle* createTriangle(struct Point p1,
                               struct Point p2,
                               struct Point p3);
void printTriangle(struct Triangle trn);
#endif
```

יכל להיות שכאשר Triangle יהיה חלק מפרוייקט גדול יותר תהיה סכנה של הכללה כפולה של ה - header שלו. כדאי לתת לתגים שמות הקשורים באופן ברור לשם הקובץ שעליו הם שומרים.

גם קבצי ה - header של הספרייה הסטנדרטית מכילים מנגנון המונע הכללה כפולה ולכן אין צורך לחשוש לכלול אותם כאשר רוצים להשתמש בהם.

ניתן ליראות את פלט ה - preprocessor לפני התרגום באופן הבא:

```
$ gcc -E main.c
```

בחינה של פלט זה עשויה להבהיר שאלות לגבי אופן הפעולה של ה - preprocessor.

בעיית המימוש הכפול

בכל ה - object files בפרוייקט הנ"ל הופיעה ההצהרה על struct Point. הדבר היה תקין מכיוון שבכל הקבצים הללו היה צורך לדעת ש struct Point קיימת בקובץ כלשהו. אם לעומת זאת היו הקבצים הללו, מכילים מימוש של פונקציה של Point, הדבר היה גורם להודעת שגיאה בזמן ה - linking. שגיאה כזאת ניתן ליצור לדוגמה ע"י העברת מימוש הפונקציה printPoint (...)-מ-

Point.c ל – Point.h. השינוי הזה היה גורם למימוש הפונקציה להופיע בכל קבצי ה – object file וה- linker לא היה יודע איזה מהמימושים לקשר.
כדאי להבחין בין הבעיה האחרונה של **מימוש כפול** שהיא בעיית **linking** לבין הבעיה הקודמת של **הצהרה הכפולה**, שהיא בעיית **קומפילציה**.

על **headers** הכוללים **headers** ועל **forward declaration**

בקובץ ה – Triangle.h, header כתבנו פקודה ל – preprocessor המכלילה header אחר בתוכו. כדאי להמעיט בהכללות header בתוך header עד כמה שניתן. הכללה כזאת יוצרת תלויות קומפילציה בין קבצים, מאיטה את תהליך הקומפילציה ועלולה לגרום למעגלי הכלה של ה – preprocessor. במקרים מסוימים אפשר להימנע ממנה ע"י הצהרה על קיום טיפוס ללא פרוט. הצהרה על קיום טיפוס מאפשרת יצירת מצביע עליו אך לא מאפשרת כל שימוש הדורש ידיעה של גודלו או אופן סידורו בזיכרון. אי אפשר להעביר טיפוס כזה by value, אי אפשר להתייחס לשדות שלו ואי אפשר להגדיר אותו כשדה של מבנה אחר.

לדוגמה:

file A.h:

```
struct B; // just telling the compiler that a type named B, exist.  
// (called "forward declaration")  
struct A {  
    B* m_p;  
    //B m_b; // error: sizeof B is unkown.  
};  
void foo(B* p);  
//void foo1(B p); // error can't send B object by value
```

file A.c:

```
#include "A.h"  
#include "B.h" // now all the information about B is available.  
void foo(B* p) {  
    B b1; //ok  
    b.m_a++; //ok  
}
```

File B.h:

```
struct B {  
    int m_a, m_b;  
};
```

כעת, כל קובץ שיכלול את A.h לא יכלול את B.h למרות שהקוד של A משתמש ב - B באופן מלא:

file otherFile.c:

```
#include "A.h"  
// .. can use A ..
```

כאן אין תלות קומפילציה בין B.h ל otherFile.c. שינוי בקובץ B.h לא ידרוש קומפילציה מחדשת של otherFile.c. בקובץ Triangle.h לא יכולנו להסתפק בהצהרה על קיום טיפוס בשם Point מכיון שרצינו שהמשולש ממש יכיל נקודות:

file Triangle.h:

```
struct Point;  
struct Triangle {  
    Point m_p1,m_p2,m_p3; // error: sizeof Point is unknown.  
    ...
```

אם היינו בוחרים לממש את המשולש ע"י הצבעות לנקודות, היינו יכולים להימנע מתלות הקומפילציה:

file Triangle.h:

```
struct Point;  
struct Triangle {  
    Point *m_p1,*m_p2,*m_p3; // ok  
    ...
```

הרצון להימנע מתלות קומפילציה לא צריך להכתיב את הארכיטקטורה של התוכנית שלכם. במקרה זה למשל, נראה שהפתרון המקורי הגורם לתלות קומפילציה, טבעי יותר.