קובץ **make** רקורסיבי

**מבנה התיקיות:**

****

**כדי לקרוא ל make של כל אחת מהתיקיות השתמשנו בדגל** -C **כדי שהוא יעשה** cd **לתיקייה ויריץ שם** make all

**תמונה שמכילה טקסט, צילום מסך, תוכנה, גופן

התיאור נוצר באופן אוטומטי**

שאלה 1

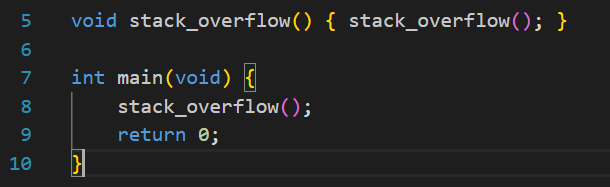
כדי שנוכל לקבל את הcore הרצנו את הפקודה:

ulimit -c unlimited

בעצם נתנו למערכת הפעלה ליצור קבצי core בגודל בלתי מוגבל

תוכנה 1:

גלישה מהמחסנית



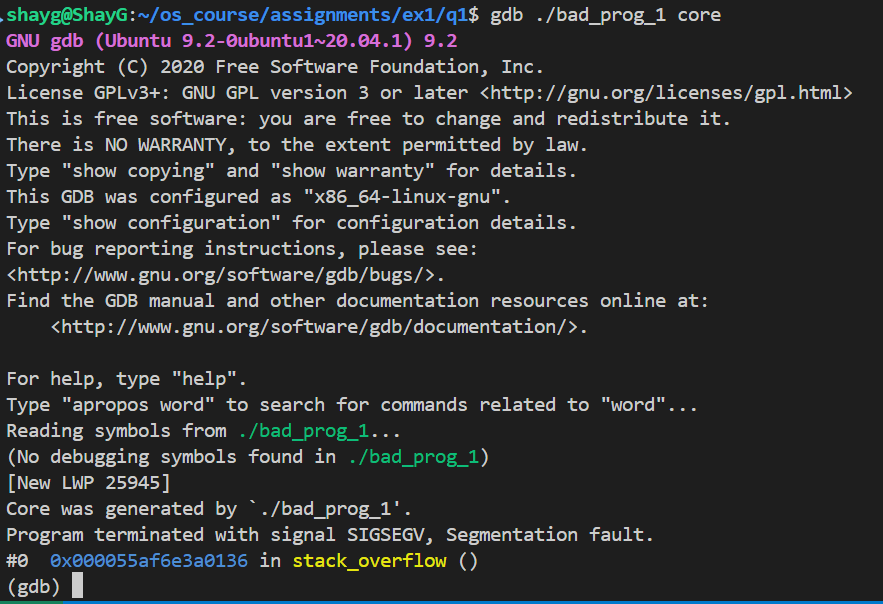
הרצנו את התוכנה הראשונה (שגומרת לstack over flow) כאשר בקימפול **לא** שלחנו את הדגל -g



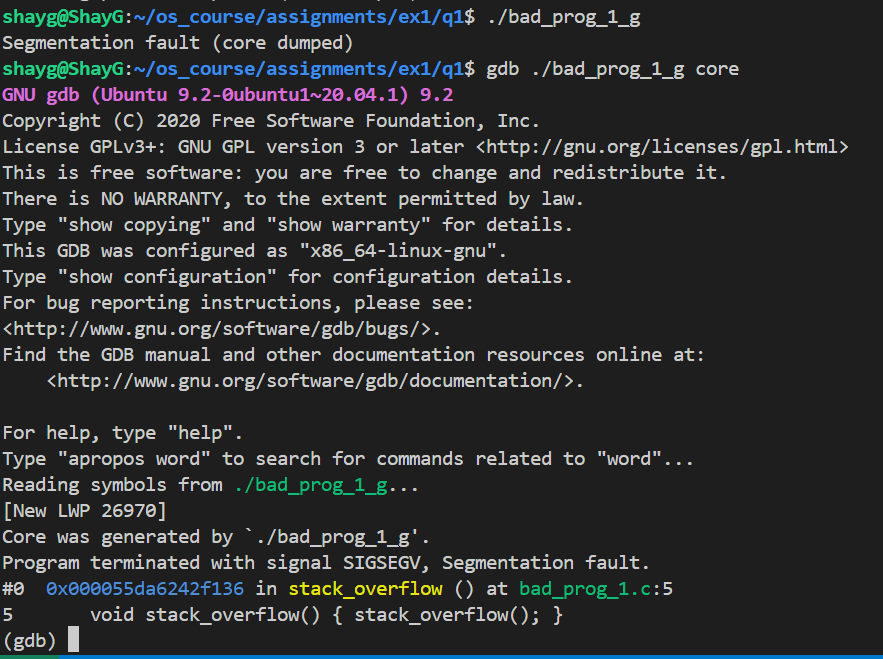
ונוצר לנו הקובץ הבא:



עכשיו נפתח את הcore

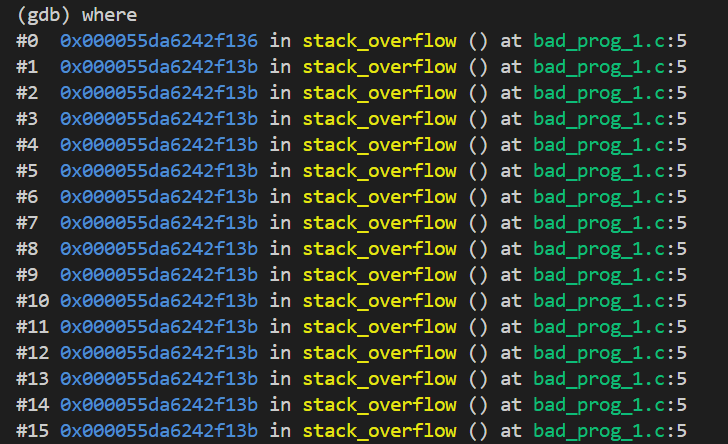


כאשר הרצנו את הקובץ השני (עם הדגל -g שנותן לנו מידע עבור דיאבג)



ההבדל העיקרי הוא בעצם השורה האחרונה (וגם בסוך השורה לפניה), שמדפיסה לנו את השורה שממנה השגיאה נזרקת.

כאשר נדפיס את מחסנית הקריאות עם הפקודה where נקבל:



זה ממשיך עוד אחרי (בסוף המחסנית מלאה) אז צילמתי רק את ההתחלה

אין משתנים אז אין טעם להשתמש בprint.

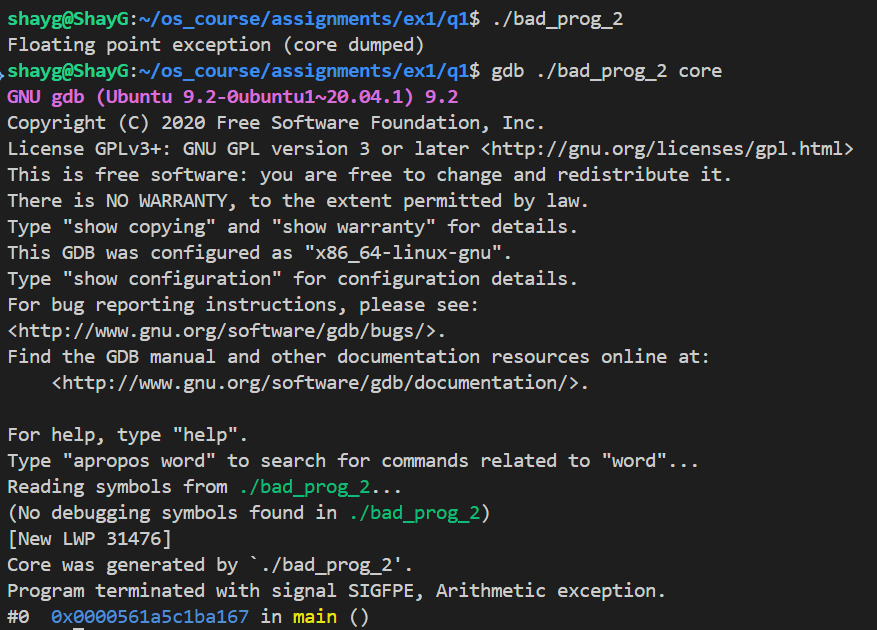
**התוכנית השנייה:**

חלוקה ב0

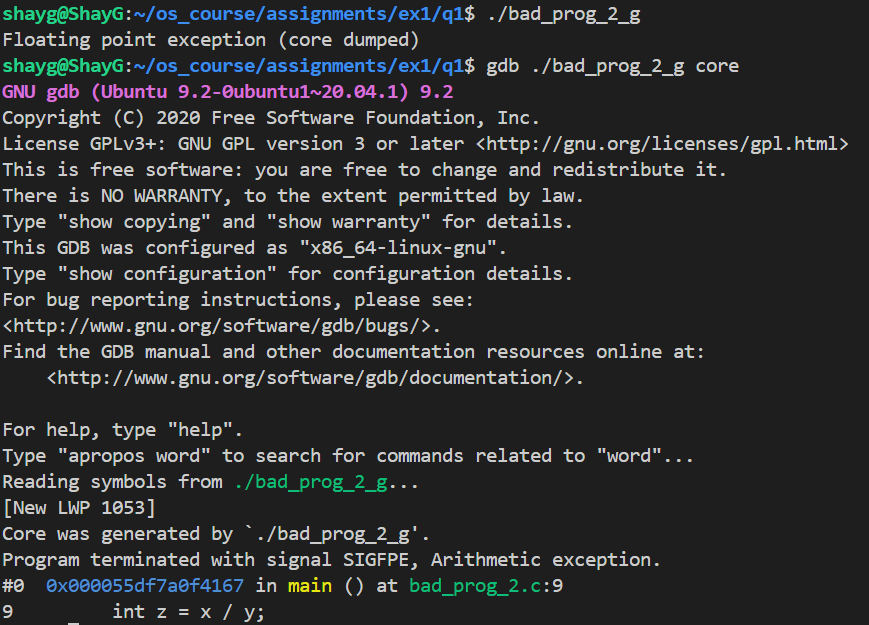
תמונה שמכילה טקסט, צילום מסך, גופן, מספר

התיאור נוצר באופן אוטומטי

התוכנית כאשר קמפלנו אותה ללא דגל g:

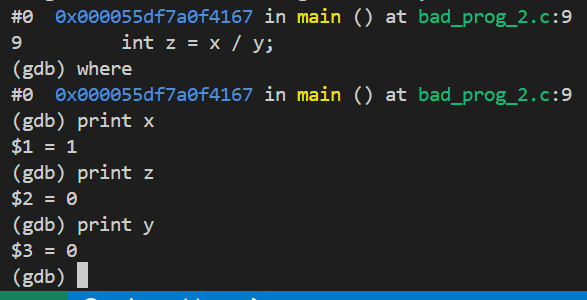


כשהרצתי את התוכנית שהיא מקומפלת עם -g



ניתן לראות שהתוכנית נפלה בשורה 9 בגלל בעיה אריתמטית.

נדפיס את המשתנים כדי לראות מה קרה:



ואפשר לראות שחילקנו ב0.

תוכנה 3

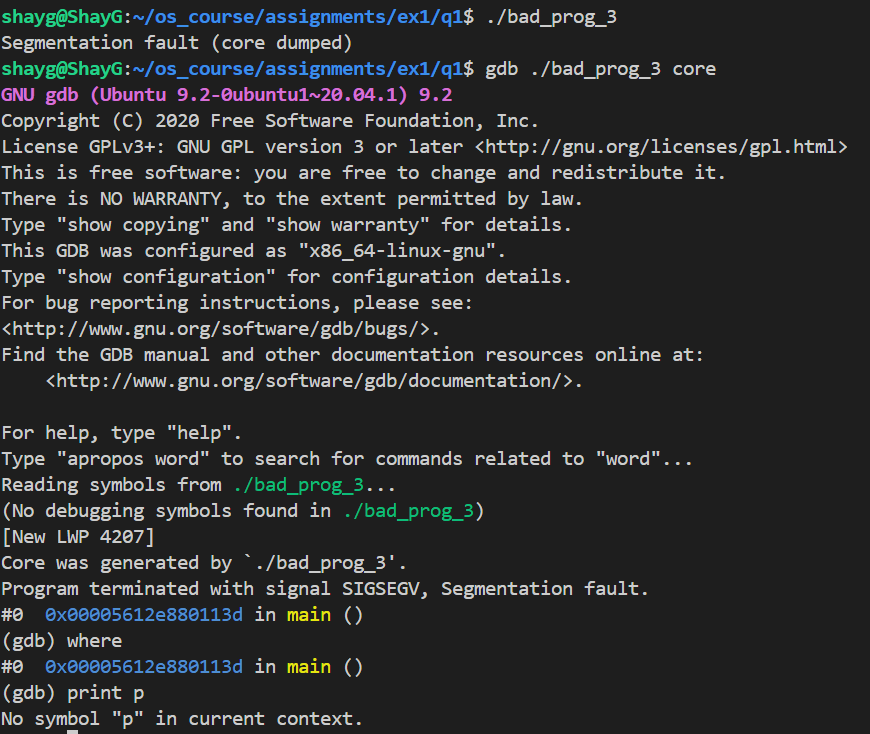
שימוש בזכרון לא מוגדר

הגדרנו מצביע ל NULL (הוא יצביע לכתובת 0), ואי אפשר לגשת לכתובת הזאת.

תמונה שמכילה טקסט, צילום מסך, גופן, קו

התיאור נוצר באופן אוטומטי

כאשר נריץ את gdb כשדיבגנו בלי דגל הg נקבל:



וכאשר קמפלנו עם -g

תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי

לאחר מכן הדפסתי את המשתנה שלנו (המצביע p) ואפשר לראות שהסוג שלו הוא מצביע ל int והערך שלו הוא 0.

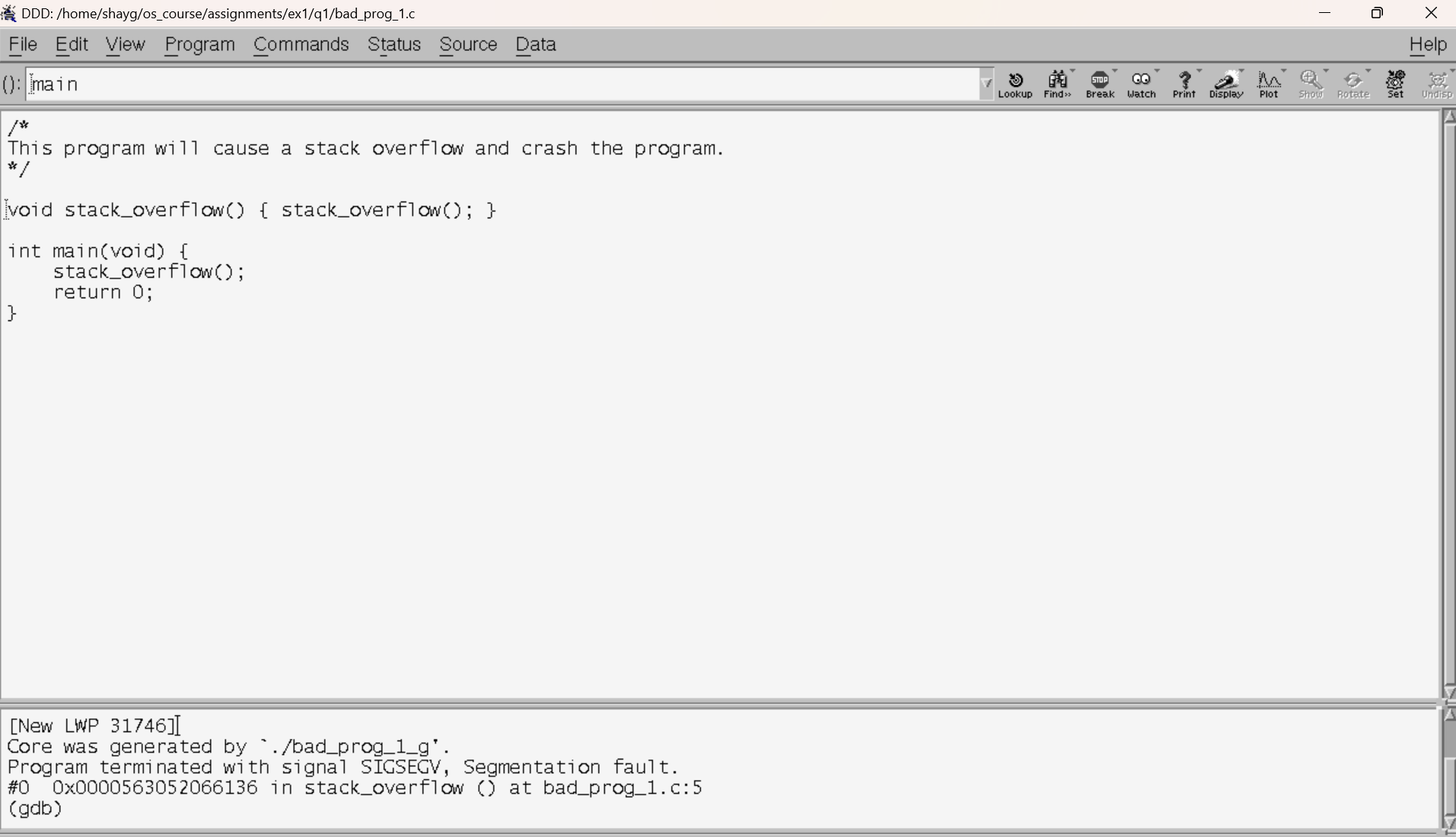
DDD

כדי לדאבג עם ddd הרצנו את הפקודה: ddd ./bad\_prog1

נשים לב שהתוכנית מקומפלת עם דגל g שנוכל לדאבג.

תוכנית 1

(אין מה לדאבג כאן יותר מידי, התוכנית די פשוטה)



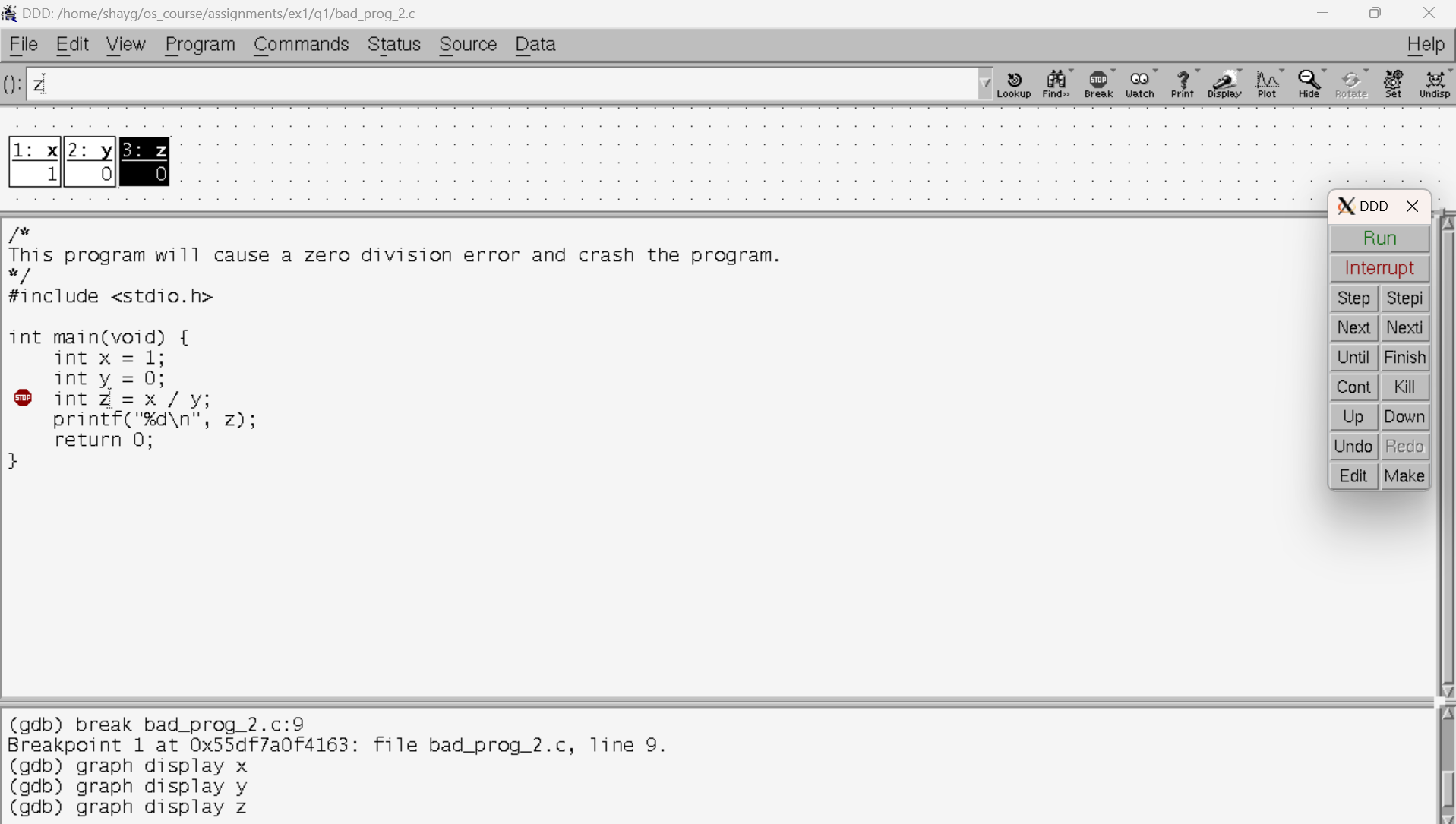
תוכנית 2

הפעם יש באמת מה לדאבג,

שמתי break point בשורה שזה אמור ליפול, והצגתי למעלה את המשתנים שלנו (מסומן בצהוב)

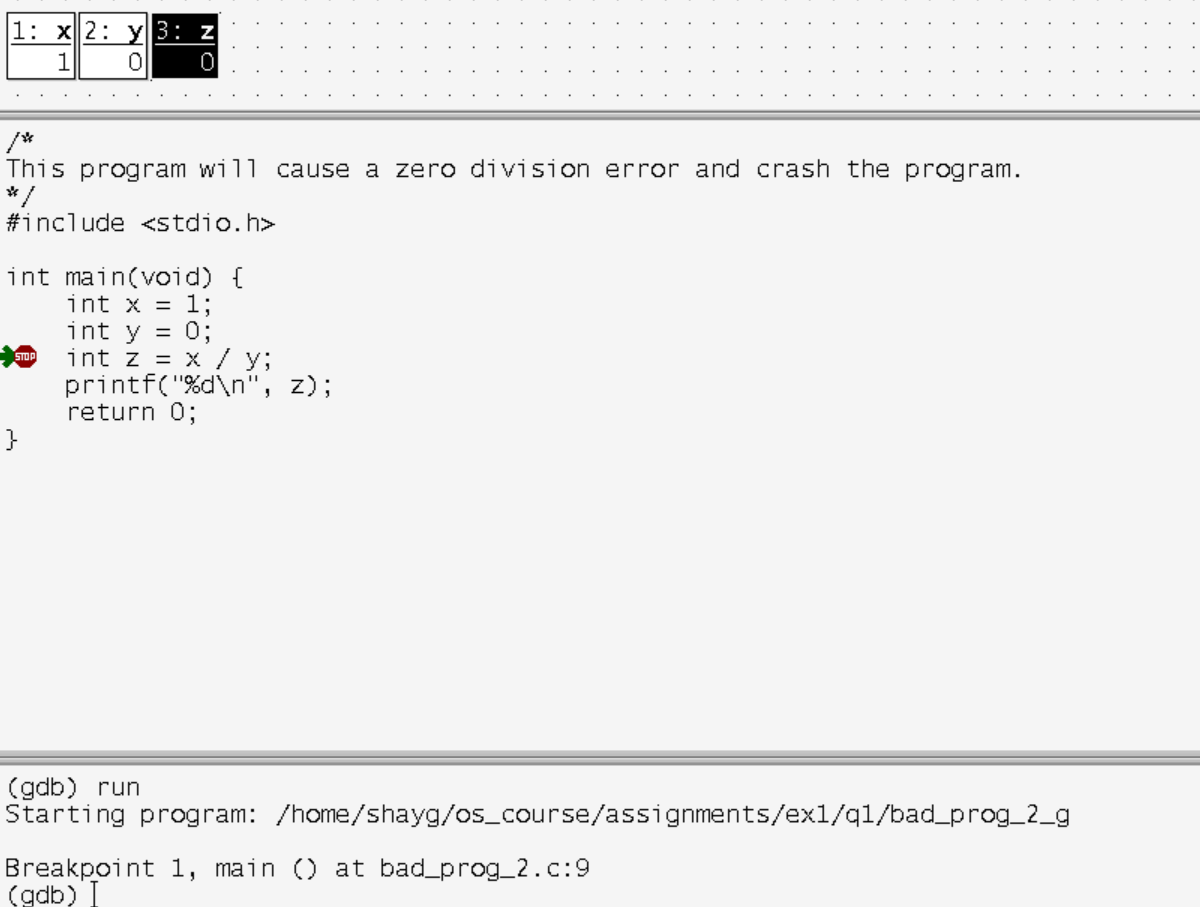
הרצתי את התוכנית והיא עכשיו עצרה

כשלחצתי על step הוא ביצע את השורה, וזרק את השגיאה



לאחר מכן חלצתי על run והוא הגיע ב break point ששמתי (מסומן בכחול)

אפשר לראות את החץ הירוק שמצביע על איזה שורה אנחנו נמצאים



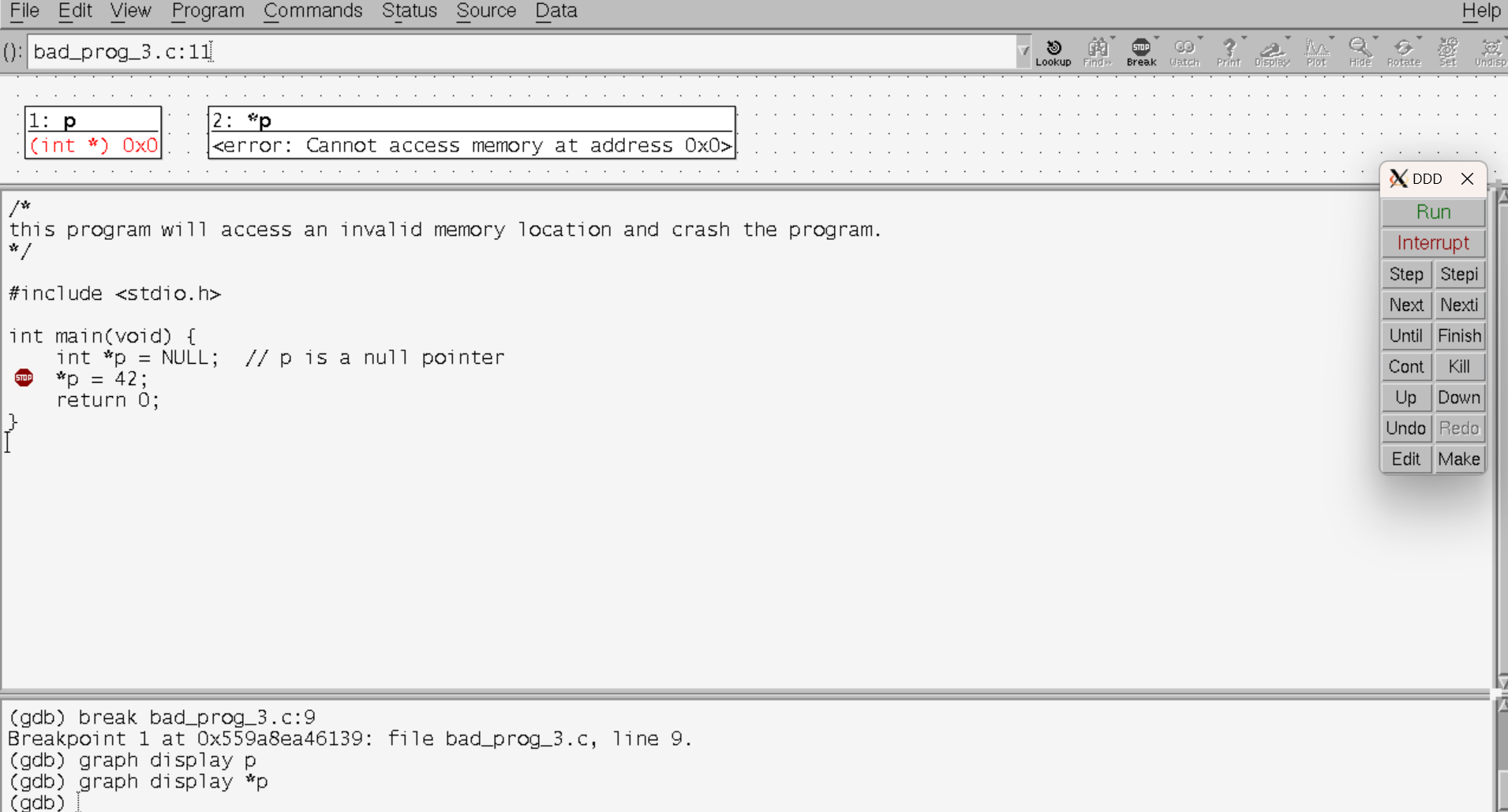
כשלחצתי על step הוא ביצע את השורה והוא נפל.

אפשר לראות את החץ בצבע אדום שמסמן את הנפילה.

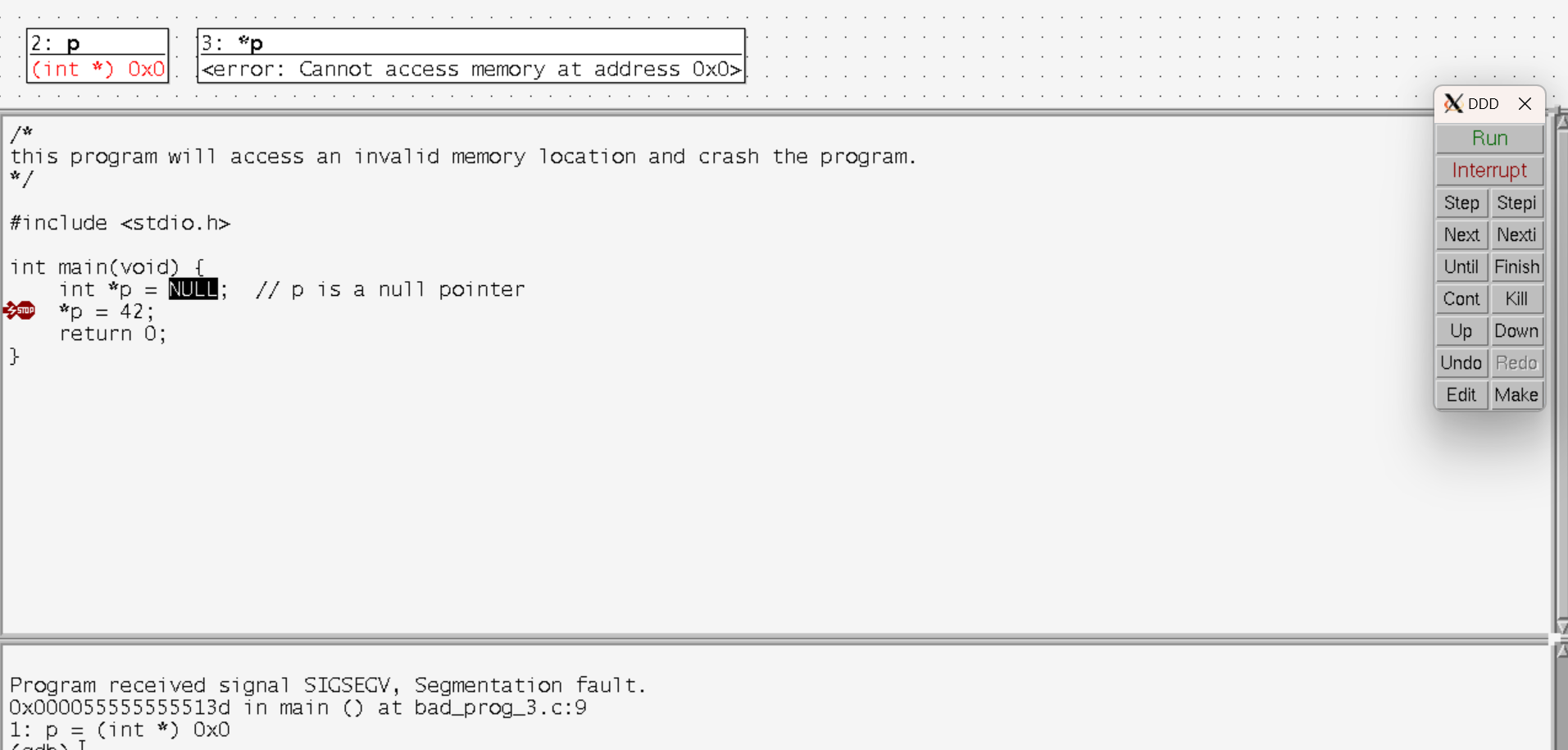
תמונה שמכילה טקסט, צילום מסך, תוכנה, תצוגה

התיאור נוצר באופן אוטומטי

תוכנית 3:

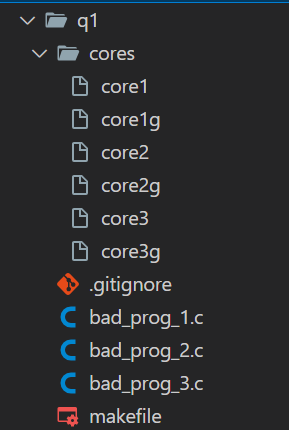


גם כאן הרצתי, הוא נכנס ל break point וכשהמשכתי את הריצה הוא נפל, וddd מסמן לי איפה הוא נפל.

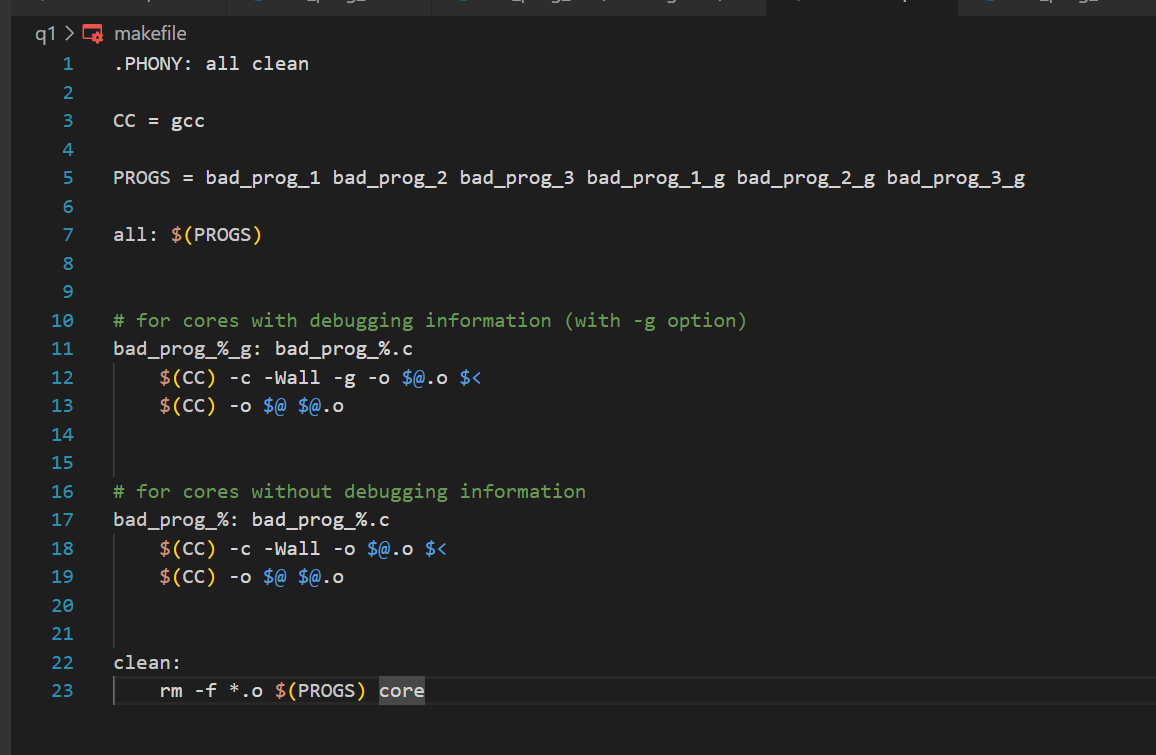


המבנה של הקבצים הוא:

בתיקייה הראשית יש את שלושת התוכניות, makefile וקובץ .gitignore שעוזר לי לעבוד עם git, ועוד תיקייה שבה יש את כל הcore שנוצרו בריצה של התוכניות. קובץ שנגמר בg מסמן שהתוכנית קומפלה עם הדגל -g.



הקובץ make נראה ככה:



הוא יוצר 6 תוכניות – לכל אחת משלושת התכוניות שכתבנו הוא יוצר 2, אחת מקומפלת עם הדגל g ואחת לא.

שאלה 2

הפונקציה שמחשבת את ההסתברות:

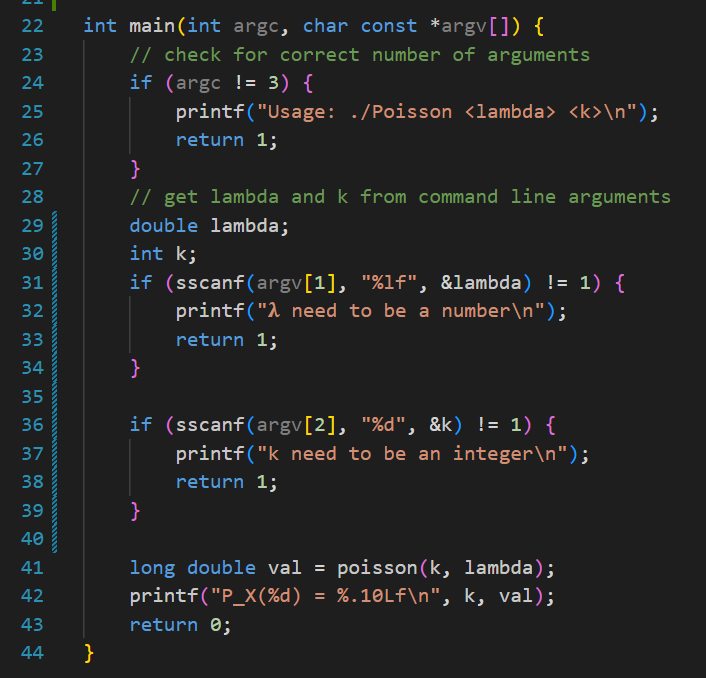
תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי

בעזרת הפונקציה expl אחנו משיגים דיוק של long double

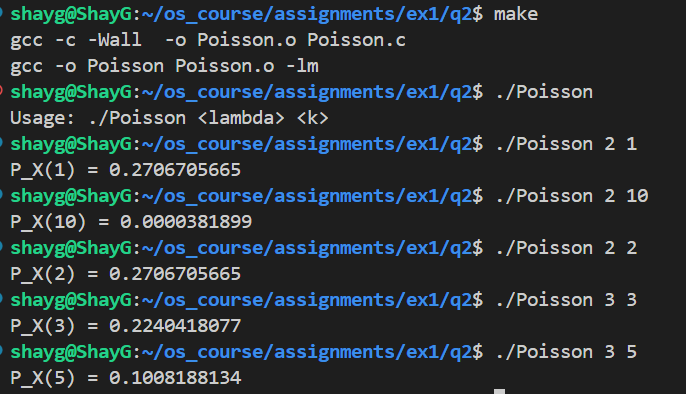
פונקציית הmain:

אנחנו בודקים שקיבלנו ביוק 3 פרמטרים, ואז בודקים את תקינות הקלט, לאחר מכן מחשבים את ההסתברות ומדפיסים.

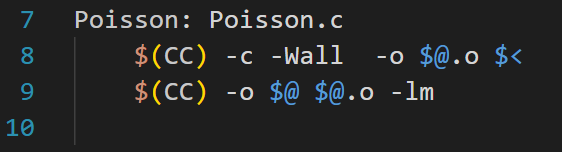


הרצת התוכנית:

* הרצה של make
* הרצה ללא ארגומנטים
* הרצה עם ארגומנטים שונים



הקימפול והלינקוג' עם הדלג lm



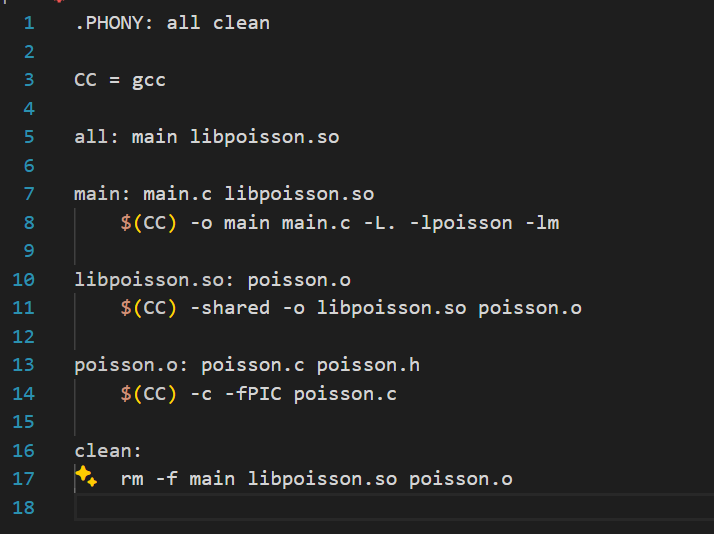
שאלה 3

הקבצים שיש לנו:



נשתמש ב poisson.c כדי ליצור את הספרייה, ובpoisson.h בקובץ main.c כדי להשתמש בה.

הקובץ make:



כדי ליצור את הספרייה נשתמש בדגל -shared ואת לקובץ שיווצר נקרא libpoisson.so.

בלינקוג' נשתמש ב-L. בשביל להגיד ללינקר להסתכל בתיקייה הנוכחית, ב -lpoisson שזה השם של הספרייה ללא ההתחלה של lib ולא הסיומת של .so , ובדגל -lm כדי להשתמש בספרייה math.

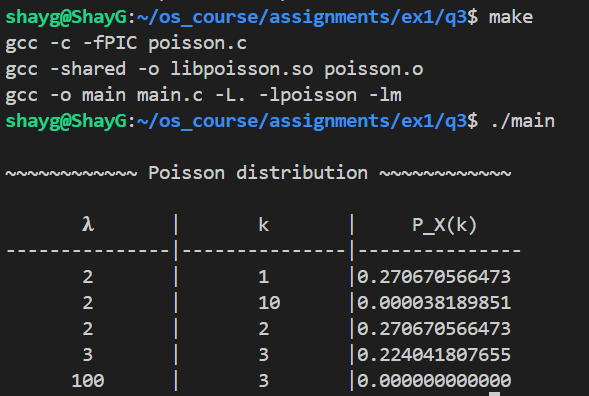
בנוסף נצטרך להגיד לdynamic linker לחפש סיפריות דינמיות גם בתיקייה הזאת לכן חשוב להריץ את הפקודה

export LD\_LIBRARY\_PATH=.

כדי שהוא יידע לקחת את הסיפרייה מהתיקייה הנוכחית.

כשנקמפל את הקובץ poisson.c נשתמש בדגל -fPIC שזה אומר Position Independent Code זה בשביל שנוכל להשתמש בו בספרייה דינאמית, וזה אומר שהקובץ שהמקופל לא יהיה תלוי במיקום של הקובץ בשביל לעבוד.

הרצה:



שאלה 4:

הקוד שמופיע ב geeksforgeeks מניח שהגודל של הגרף קבוע ל 9 קודקודים, כדי להתגבר על זה אני קיבלתי כקלט את הגודל מהמשתמש, והקצתי את המטריצה בצורה דינמית.

כדי להגיע לכיסוי מלא, כתבתי 10 קבצי קלטים רעים, וקובץ קלט רע אחד, הם מופיעים בתיקייה q4/inputs ויש שם קובץ inputs.txt שמסביר מה כל קלט בודק.

הסבר:

* קלטים רעים:

1. קלט ריק
2. מספר שלילי של קודקודים
3. לא שולחים מטריצה בכלל
4. שולחים צלע שלילית
5. שולחים צלע שהיא לא 0 לאלכסון (הגרף אמור להיות פשוט)
6. לא שולחים מאיזה קודקוד להתחיל את האלגוריתם
7. שולחים קודקוד התחלה שהוא לא בגרף
8. שולחים יותר מידי קלט
9. שולחים מספר גדול מידי של קודקודים שהוא לא יצליח להקצות את המטריצה
10. שולחים מספר גדול מידי של קודקודים, אבל לא גדול מידי, שהוא יצליח להקצות את המטריצה אבל ייכשל להקצות את אחת מהשורות.

* קלט טוב: קלט שיעבוד.

Makefile:

כדי שנוכל לבדוק את הכיסוי נקמפל עם הדגלים -fprofile-arcs -ftest-coverage

ובליקוג' נשתמש בדגל -lgcov

הכנתי גם target שיעזור לי להריץ את כל הקלטים שהכנתי, קראתי לו run.

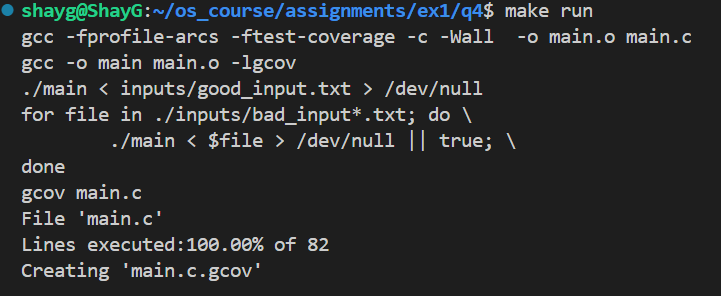
הוא מריץ את הקלט הטוב, ואז הוא עובד על כל הקבצים הרעים ומריץ אותם אחד אחרי השני. את כל הפלט אני רושם ל/dev/null.\

בסוף אני מרית gcov main.c כדי לראות את אחוז הכיסוי.

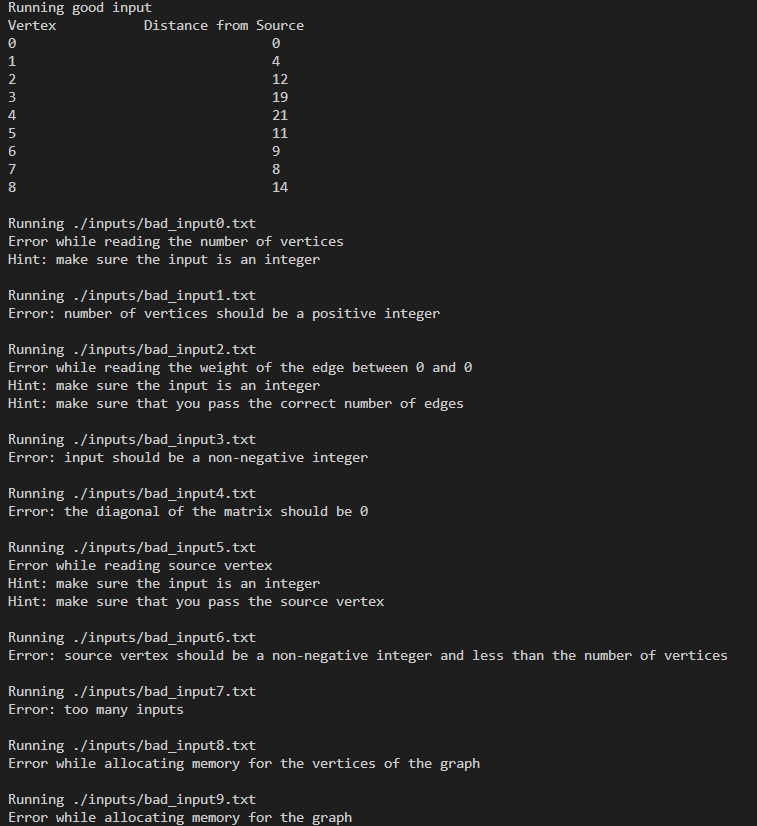
תמונה שמכילה טקסט, צילום מסך, תוכנה, תכונות מולטימדיה

התיאור נוצר באופן אוטומטי

דוגמא של הרצה:



אם אני רוצה לראות את הפלטים:



תרגיל 5: