תרגול מס' 3: פקודות ל-Preprocessor, כלי לבניית תוכנות ב-Unix (Make)

1 פקודות Preprocessor

כפי שראינו בתרגול הקודם, בשלב הקומפילציה כל קובץ c עובר פעולה מקדימה הנקראת Preprocessing. בפעולה זו מתבצעות הפקודות המתחילות ב-#, הגורמות לשינויים טקסטואליים בקבצים פעולות גזירה והדבקה פשוטות (שינויים אילו הם זמניים: הם תקפים רק במהלך הקומפילציה, אך אינם נשמרים בקבצים שלכם).
ניתן לקבל קובץ C לאחר עיבוד ע"י ה-preprocessor (למשל: כדי לבדוק נכונות פקודות macro), ע"י שימוש בתוכנית הקומפיילר (gcc) עם הדגל E- (אשר קוראת לתוכנית ה-preprocessor):

```
gcc -E source.c > source.i
```

ל-preprocessor ישנם 4 תפקידים עיקריים:

1.1 הכלת קבצי include (פקודת #include)

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

פקודות אילו 'שותלות' את תוכן הקובץ המצוין. תהליך זה יכול להתבצע גם באופן רקורסיבי. (למשל בקובץ mine.h יש include לקובץ header אחר)
ההבדל בין שתי השורות הוא רק בדרך החיפוש: בצורה הראשונה מחפשים קובץ include במקומות סטנדרטיים (כגון /usr/include) ובצורה השנייה ביחס למדריך הנוכחי.

1.2 הגדרות מקרו (macro) (פקודת #define)

ניתן להגדיר שורות macro בלי פרמטרים:

```
#define DAYS_IN_WEEK 7
```

או עם פרמטרים:

```
#define SQUARE(X) ((X)*(X))
#define LEAP(Y) ((Y)%4==0&&(Y)%100!=0 || \
            (Y)%400==0)
```

הערה – התו \ מסמן שהגדרת המקרו נמשכת בשורה הבאה.

כל שימוש במקרו יוחלף ע"י ה-Preprocessor בטקסט המגדיר אותו. למשל השורה:

```
printf("Days: %d Num:%d\n",
        DAYS_IN_WEEK, SQUARE(i));
```

תוחלף בשורה:

```
printf("Days: %d\n", 7, ((i)*(i)));
```

אזהרה: שימו לב לתופעות לוואי של שימוש במקרואים – הביטוי SQUARE(i++) אינו מחזיר את ריבוע הערך הנוכחי של i ומעלה את i ב-1! כדי לראות זאת, "נפתח" את המקרו ונקבל: ((i++)*(i++)). כלומר הערך שיוחזר יהיה i*(i+1), ו-i עצמו יקודם פעמיים!!!

ישנם 2 סימנים מיוחדים שניתן להוסיף בגוף ה-macro מייד לפני שם הפרמטר:
א) סימן # מוסיף גרשיים לערך, למשל:

```
void DoInsert();
void DoRemove();

void func(char* pStr) {
#define CHECK(p, cmd, func) \
    if ( !(strcmp(p, #cmd) ) func()

    CHECK(pStr, Insert, DoInsert);
    CHECK(pStr, Remove, DoRemove);
}
```

ה-Preprocessor יהפוך את שורות ה-CHECK ל:

```
if ( !(strcmp(pStr, "Insert") ) DoInsert();
if ( !(strcmp(pStr, "Remove") ) DoRemove();
```

אי אפשר פשוט לשים סוגריים, כי למשל:

```
#define PRINT(x)\
    printf("x");
```

ידפיס x.

ב) סימן ## מבצע שרשור (concatenation), למשל-

```
#define CAT(X,Y)  X##Y
```

הקריאה:

```
CAT(abc,def)
```

מתורגמת ל:

abcdef

דוגמא

```
#define CAT(x,y)\
        x##y;
int xy = CAT(1,2);
int xy = 12;;

#define CAT(x,y)\
        xy;
int xy = CAT(1,2);
int xy = xy;;
```

אסור להגדיר מחדש macro אשר כבר הוגדר.
יש לבטל קודם את ההגדרה הקודמת ע"י **#undef**.

1.3 קומפילציה מותנית

פקודות המכניסות מוציאות קטעי טקסט:

- **#if** <expression> - אם הביטוי שונה מאפס בצע (תכלול).
- **#ifdef** <identifier> - אם המזהה (מאקרו) מוגדר בצע.
- **#ifndef** <identifier> - אם המזהה לא מוגדר בצע.
- כל אחת מהפקודות לעיל חייבת להיגמר בפקודת **#endif**. אם יש התניות מקוננות, ה-**#endif** תמיד יסגור את ההתניה הפנימית ביותר.
- בתוך פקודה כנ"ל, יכולים להופיע פקודת **#elif** (קיצור של else if) אחת או יותר, או פקודת **#else** אחת.

דוגמא 1:

```
#ifndef _UTIL_H_
#define _UTIL_H_

#if DLEVEL > 3
extern int g_DebugLevel;
#define MESSAGE(x) printf(#x)
#else
#define MESSAGE(x)
```

אם הDLEVEL>3 אנחנו נכנסים למצב debug ורוצים להדפיס את ההודעה שרשומה בMESSAGE (לשים לב ל# שיוסיף "" סביב הביטוי.

אחרת, הpreprocessor יחליף מופע של MESSAGE בשום דבר.

```
#endif  /* DLEVEL */

#endif  /* _UTIL_H_ */
```

דוגמא 2 :

```
int myfunc (int arg1, int arg2)
{
#ifdef DEBUG
    printf("function myfunc started\n");
    printf("arguments:arg1=%d,arg2=%d\n",
        arg1, arg2);
#endif
    ...
```

אם הגדרנו במקום כלשהו לפני ש
#define DEBUG
אז כאשר נכנס לפונקציה, יבוצעו השורות הללו

1.4 מתן מקרואים סטנדרטיים בעלי ערך מיוחד

המקרואים הבאים מוגדרים באופן אוטומטי ע"י ה-Preprocessor :

- __FILE__ שם הקובץ הנוכחי כמחרוזת ("main.c")
- __LINE__ מספר השורה הנוכחית בקובץ (15)
- __DATE__ תאריך הקומפילציה ("Apr 18 1998")
- __TIME__ זמן הקומפילציה ("20:12:34")

דוגמאות :

```
void PrintVersion() {
    printf("Compiled at: %s, %s\n",
                __DATE__, __TIME__);
}
```

```
#define ASSERT(x) if (!(x)) \
    printf("Assertion failed at file:%s line: %d\n", \
    __FILE__, __LINE__)
```

Make 2

make 2.1

מוטיבציה: יש תוכנית בעלת 200 קבצים, שכבר קומפלה, אבל עכשיו השתנה קובץ c מסויים. אם נקמפל את כל הקבצים מחדש ואז נבצע קישור זה יקח זמן רב. לכן נרצה לקמפל מחדש רק את קבצי האובייקט שתלויים בשינויים שבוצעו ואז לבצע קישור.

make הינה תוכנית סטנדרטית בסביבת Unix המשמשת ככלי עזר לבניית פרויקטים של תוכנה. הכלי מאפשר אוטומציה בבניה רב-שלבית של תוכנה (לאו דווקא תוכניות C!).

make היא כלי פשוט ויעיל המאפשר קומפילציה הדרגתית של תוכנה בלי חזרה על חלקים שלא עודכנו.

למשל בשפת C: בהינתן תוכנית הבנויה ממספר רב של קבצי-c, ניתן להיעזר ב-make כדי לבצע קומפילציה רק לקבצים ששונו מאז הבניה האחרונה וישר לעבור לשלב ה-Link.

makefile 2.2

<u>להרצת make יש לספק לה קובץ המתאר את התלויות בין קבצי הפרויקט, ואת הצורה בה יש לבנות את הקבצים המבוקשים. התוכנית make מחפשת בצורה אוטומטית קובץ בשם **makefile** או **Makefile** במדריך הנוכחי ממנו הורצה make.</u> אך ניתן לספק ל make קובץ עם שם אחר ע"י שימוש בדגל f–, לדוגמא:

eesoft:t2> **make –f my_file [target]**

ה make תשתמש בקובץ my_file במקום הקובץ makefile.

מבנה makefile סטנדרטי 2.3

makefile סטנדרטי מכיל:

1. הערות (כל מה שמופיע אחרי הסימן # נחשב כהערה). ניתן וכדאי לרשום הערות בכל מקום בקובץ.
2. אחרי ההערות מופיעות בד"כ הגדרות של משתנים, לדוגמא: הגדרת שם הקומפיילר, הגדרת שם ה- linker , הגדרת הדגלים שיש להשתמש בהם בפעולת הקומפילציה.
3. לאחר מכן, יופיעו ה"מטרות", במה תלויה כל מטרה , ואיך לבנות אותה בפורמט הבא:

target : dependencies
<TAB>command to create target

כאשר **target** מהווה שם של מטרה.

dependencies הינה רשימה של **קבצים/מטרות אחרות** בהם תלויה ה- target הזה. **משמע: כל שינוי באחד מה dependencies הנ"ל, יגרור את בניית ה target הזה מחדש.**

השורה השנייה מתארת את **הפקודה** לבניית ה target.

שימו לב חייבים לרשום **TAB** בתחילת השורה השנייה.

פקודת המטרה מתבצעת באופן הבא:

- תחילה נסרקת רשימת התלויות. אם לקובץ תלוי כלשהו קיימת שורת מטרה, תוכנית ה-make תנסה לבנות מטרה זו תחילה (באופן רקורסיבי).
- נבדקים זמני היצירה (תאריך ושעה) של קבצי התלויות וקובץ המטרה. אם לפחות אחד מקבצי התלות יותר חדש מקובץ המטרה, מורצת הפקודה לבניית המטרה (השורה השניה).

כאשר make מורצת ללא פרמטרים, היא תמיד תתחיל לבצע את הבניה **מהמטרה הראשונה** (המטרה שבתחילת הקובץ) בקובץ ה-makefile. אחרת – הפרמטרים מציינים את שמות המטרות שיש לבנות (נראה בהמשך).

בשורת target עבורה עדיין לא קיימת גרסה ישנה של קובץ המטרה (כלומר קומפילציה או קישור שלא בוצעו עדיין), השורה השנייה תתבצע **תמיד**.

2.4 דוגמא פשוטה לשימוש ב-make

התוכנית prog מורכבת משלושה קבצים: main.c, f.c ו-f.h.

main.c מבצע #include ל-f.h, ולכן נרצה לבנות את main.o מחדש כל פעם ש-f.h משתנה. נניח ששאר קבצי המקור אינם תלויים אחד בשני. באותו אופן, נרצה לבנות את prog מחדש (לבצע link), כל פעם שאחד מהקבצים main.o ו-f.o משתנים.

להלן קובץ ה-makefile עבור prog:

```
# Makefile for creating the 'prog' program

# Link the final executable
prog: main.o f.o
     gcc -o prog main.o f.o

# Compile main.c to create main.o
main.o: main.c f.h
     gcc -c main.c

# Compile f.c to create f.o
f.o: f.c f.h
     gcc -c f.c
```

נשים לב שאם יבקשו לבצע make main.o אז לא יבוצע קישור מחדש, ולכל כיוון שאין תלות בין קבצי המקור אז רק main.c יקומפל שוב

עבור קובץ Makefile זה ביצוע הפקודה make או make prog שקול.

לפני שבנינו את הפרויקט בפעם הראשונה, קיימים רק קבצי המקור, ולכן הרצת make תבנה את כל אחת מהתלויות של prog (main.o ו-f.o), ולבסוף תבנה את prog עצמה:

```
> ls -l
total 16
-rw-rw-r--   1 user     user          132 Mar 21 12:59 f.c
-rw-rw-r--   1 user     user           10 Mar 21 14:57 f.h
-rw-rw-r--   1 user     user          304 Mar 21 13:34 main.c
-rw-rw-r--   1 user     user          404 Mar 21 14:45 makefile

> make
gcc -c main.c
gcc -c f.c
gcc -o prog main.o f.o

> ls -l
total 40
-rw-rw-r--   1 user     user          132 Mar 21 12:59 f.c
-rw-rw-r--   1 user     user           10 Mar 21 14:57 f.h
-rw-rw-r--   1 user     user         1000 Mar 21 15:27 f.o      <--
-rw-rw-r--   1 user     user          304 Mar 21 13:34 main.c
-rw-rw-r--   1 user     user         1096 Mar 21 15:27 main.o   <--
-rw-rw-r--   1 user     user          404 Mar 21 14:45 makefile
-rwxrwxr-x   1 user     user        12058 Mar 21 15:27 prog     <--
```

הרצה נוספת של make לא תעשה דבר, כי בכל שורת מטרה קובץ המטרה יותר חדש מקבצי התלויות שלו:

```
> make
make: `prog' is up to date.
```

אם נשנה את f.h (הפקודה touch רק משנה את חותמת הזמן של הקובץ), הרצה של make הפעם תבנה את main.o ולאחר מכן תיצור מחדש את prog בגלל ש-main.o יותר חדש ממנו:

```
> touch f.h
> ls -l
total 40
-rw-rw-r--   1 user     user          132 Mar 21 12:59 f.c
-rw-rw-r--   1 user     user           10 Mar 21 15:28 f.h      <--
-rw-rw-r--   1 user     user         1000 Mar 21 15:27 f.o
-rw-rw-r--   1 user     user          304 Mar 21 13:34 main.c
-rw-rw-r--   1 user     user         1096 Mar 21 15:27 main.o
-rw-rw-r--   1 user     user          404 Mar 21 14:45 makefile
-rwxrwxr-x   1 user     user        12058 Mar 21 15:27 prog

> make
gcc -c main.c
gcc -o prog main.o f.o

> ls -l
total 40
-rw-rw-r--   1 user     user          132 Mar 21 12:59 f.c
-rw-rw-r--   1 user     user           10 Mar 21 15:28 f.h
-rw-rw-r--   1 user     user         1000 Mar 21 15:27 f.o
-rw-rw-r--   1 user     user          304 Mar 21 13:34 main.c
-rw-rw-r--   1 user     user         1096 Mar 21 15:29 main.o   <--
-rw-rw-r--   1 user     user          404 Mar 21 14:45 makefile
-rwxrwxr-x   1 user     user        12058 Mar 21 15:29 prog     <--
```

2.5 makefile מלא

כדי להגדיל את השמישות ולחסוך בכתיבה, בד"כ משתמשים במשתנים. מגדירים משתנה ע"י הצבה: VAR=abc, וניגשים למשתנה ע"י שימוש בתו $: $(VAR).
נהוג להגדיר מספר משתנים סטנדרטיים:

CC: הוא משתנה שאנו נותנים לו את הערך שמהווה את שם הקומפיילר של C. לו היינו מחליטים לעבור לקומפיילר אחר (למשל koko) כל שעלינו לעשות הוא לשנות רק את ההגדרה למעלה. **CC** הוא משתנה סטנדרטי שמוגדר בכל makefile וברירת המחדל שלו היא cc.

CFLAGS: מכיל את כל הדגלים לפעולות הקומפילציה, משתנה סטנדרטי ל-make וברירת המחדל שלו היא שורה ריקה.

CCLINK: מכיל את שם ה-linker, במקרה שלנו (ובדרך כלל) זה שם הקומפיילר.

LIBS: מכיל את הספריות שיש לקשר אותן בשלב ה-linking. למשל -l**m** אומר שיש לקשר ספריה סטנדרטית ששמה *lib**m**.a* (הספריה המתמטית).

OBJS: מכיל את שמות קבצי ה-object. משתמשים בו בשורת ה-linker.

RM: מכיל את שם פקודת המחיקה. משתמשים בו בפעולת ה-clean בכדי למחוק קבצים ישנים.

כמו כן, לעיתים מסתמכים על תכונות נוספות של make:

- שורת target שרשימת התלויות בה **ריקה** – תבצע את השורה השניה **תמיד**.
- target עם השם xxx.o אך **בלי שורת בנייה** (השורה השניה), יפורש **אוטומטית** ע"י ה make בצורה:

$(CC) $(CFLAGS) –c xxx.c

נניח שאנו כותבים משחק גרפי בשם doom ואנו מחליטים לחלק את הפרוייקט למספר קבצים:

1. screen.h מכיל את **ההצהרות** של **הפונקציות** לטיפול במסך והגרפיקה.
2. mouse.h מכיל את **ההצהרות** של **הפונקציות** לטיפול בעכבר.
3. screen.c מכיל את **מימוש הפונקציות** של screen.h.
4. mouse.c מכיל את **מימוש הפונקציות** של mouse.h.
5. game.c מכיל את **מימוש המשחק** (כולל פונקצית main()), ונעזר בפונקציות של screen ו- mouse.

כל פעם שאנו רוצים ליצור את קובץ הריצה העדכני עלינו להעביר קומפילציה את שלושת קבצי ה C ולעשות link בין ה objects שנוצרו.

אך אם משתנה **מימוש** של פונקציה בקובץ C **אחד** בלבד, אנו רוצים לעשות קומפילציה **רק לקובץ זה**. ה make מאפשר לנו לעשות קומפילציה רק לקבצים שהשתנו ולעשות link לאחר מכן.

ה-**Makefile** לפרוייקט זה ייראה כך:

```
# This is a Makefile for the doom project
CC = gcc
CFLAGS = -g -Wall
CCLINK = $(CC)
LIBS =
OBJS = game.o screen.o mouse.o
RM = rm -f

# Creating the executable (doom)
doom: $(OBJS)
	$(CCLINK) -o doom $(OBJS) $(LIBS)

# Creating object files using default rules
game.o: game.c mouse.h screen.h
mouse.o: mouse.c mouse.h
screen.o: screen.c screen.h

# Cleaning old files before new make
clean:
	$(RM) doom  screen_test *.o *.bak *~ "#"* core
```

CC - משתנה שאנו נותנים לו את הערך שמהווה את שם הקומפיילר של C.
CFLAGS – מכיל דגלים לפעולת קומפילציה.
CCLINK – מכיל את שם הלינקר
LIBS – מכיל את שמות הספריות אותן יש לקשר בזמן הlinkage
OBJS – מכיל את שמות קבצי האובייקט, משתמשים בשורת הלינקר
RM – מכיל את שם פקודת המחיקה.

target עם השם xxx.o אך **בלי שורת בנייה** (השורה השניה), יפורש **אוטומטית** ע"י ה make בצורה:

$(CC) $(CFLAGS) –c xxx.c

הרצת הפקודה **make** תגרום לביצוע הפעולות הבאות:

1. make תתחיל ב-target הראשון: doom. היות ו-doom תלוי במטרות game.o, screen.o ו-mouse.o, make תנסה לבנות מטרות אילו תחילה.

2. המטרה game.o תלויה ב-game.c, mouse.h ו-screen.h, make תחפש שורת מטרה עבורם. מכיוון שאין כזו (אילו הם קבצי המקור – אין מה "לבנות" אותם), make מניחה שקבצים אילו קיימים ומעודכנים.

3. מכיוון שבפעם הראשונה game.o אינו קיים, תורץ ה-"שורה השניה" של המטרה שבמקרה זה היא ברירת המחדל:

   $(CC) $(CFLAGS) –c game.c

   המטרות screen.o ו-mouse.o יבנו באופן דומה.

4. לבסוף make תחזור למטרה doom, וחותמות הזמן של התלויות יושוו לחותמת של הקובץ doom. מכיוון ש-doom עדיין לא קיים, תורץ השורה ששניה המפעילה את ה-Linker ובונה את התוכנית הסופית!

מאידך הרצת **make clean**, תגרום ל make לחפש את המטרה clean ולבנות אותה – למטרה clean אין תלויות, ובנייתה משמעה ניקוי המדריך מקבצי o ומקובץ ההרצה doom, ומשאר הקבצים הפרזיטיים.
אופציה זו שימושית מאוד כאשר רוצים **לחייב** את make לבנות את כל הפרויקט ללא קשר לחותמות הזמן (למשל במקרה שהעתקנו גרסאות ישנות של חלק מקבצי המקור ממקום אחר): תחילה נריץ make clean ואחר כך make.

הערה חשובה: כדי לקבל שורת התלויות (dependencies) של קבצי *.c, ניתן (ורצוי) להשתמש בקריאה: gcc –MM *.c
מקבלים כתוצאה שורות שניתן להכניסן ישירות ל-Makefile:

```
game.o: game.c mouse.h screen.h
mouse.o: mouse.c mouse.h
screen.o: screen.c screen.h
```

2.6 makefile ליצירת מספר קבצי הרצה

נניח כי ברצוננו לבנות תוכנית קטנה לבדיקת המימוש של screen.c לפני שילובה ב doom. לצורך זה כתבנו קובץ הנקרא test1.c.
אנו נרצה להגדיר מטרה חדשה בשם screen_test , שתיצור קובץ הרצה בשם screen_test לבדיקת ה screen בלבד.

ה- makefile החדש יראה בצורה:

```
# This is the enhanced Makefile for the doom project
CC = gcc
CFLAGS = -g -Wall
CCLINK = $(CC)
LIBS =
OBJS = game.o screen.o mouse.o
RM = rm -f

# Creating the executable (doom)
doom: $(OBJS)
	$(CCLINK) -o doom $(OBJS) $(LIBS)

screen_test: test1.o screen.o
	$(CCLINK) -o screen_test test1.o screen.o

# Creating object files using default rules
game.o: game.c mouse.h screen.h
mouse.o: mouse.c mouse.h
screen.o: screen.c screen.h
test1.o: test1.c screen.h

# Cleaning old files before new make
clean:
	$(RM) doom  screen_test *.o *.bak *~ "#"* core
```

והרצת הפקודה make screen_test תגרום ליצירת קובץ ההרצה screen_test.

סיכום Makefile:

- ניתן להגדיר קבצי makefile כדי להקל ולזרז את בניית התוכנה
- הפקודה make משתמשת ב-makefile כדי לבנות את התוכנה בצורה אוטומטית
- ניתן להגדיר מאקרו בתוך makefile כדי להקל על שינויו בעתיד
- ניתן להשתמש ב-gcc כדי ליצור שלד של makefile

▪ ***ללימוד עצמי: מנפה השגיאות - gdb***

gdb הוא מנפה שגיאות אינטראקטיבי הפועל בסביבת UNIX המספק שירותי ניפוי שגיאות **בזמן ריצה**. מנפה שגיאות **ddd** שלמדתם בסדנא הינו בסה"כ מעטפת גרפית שמפעילה (בפועל) את מנפה השגיאות הבסיסי gdb !
ה- gdb מאפשר בין היתר :

1. קביעת נקודות עצירה בתוכניות.
2. הרצה מבוקרת של התוכנית.
3. בדיקת ערכי משתנים.
4. שינוי ערכי המשתנים.

שימוש ב gdb מקל באיתור ה- bugs.
כדי לעבוד עם gdb (ולכן – גם עם ddd) יש להדר את קבצי המקור **עם הדגל g-** .

הרצת gdb:

1. הרצת התוכנית מתוך gdb בצורה מבוקרת.

```
eesoft:t2> gdb  executable
```

2. ניתוח שלאחר המוות (תוך שימוש בקובץ ה core) :
נניח , כי הרצנו תוכנית בשם executable

```
eesoft:t2> executable
```

וקבלנו הודעה כנ"ל :

Segmentation fault (core dumped)

ואז כדי להריץ את ה- gdb , ולנתח אחר המוות נריץ :

```
eesoft:t2> gdb executable core
(gdb)
```

הפקודות החשובות ביותר:

1. **run** - הרצת תוכנית (עם פרמטרים)
2. **help** - הסברים על הפקודות.
3. **quit** - יציאה מ gdb.
4. **break** - מציבה נקודת עצירה.
5. **print** - מדפיסה ערך של משתנה.
6. **step/next** - הרצה מבוקרת.

עבודה עם משתנים:

בדיקת משתנים:
ניתן לבצע בדיקה של ערכם וטיפוסם של משתנים כאשר התוכנית במצב עצירה (למשל לאחר נקודת עצירה). הפקודות הקיימות הן :

1. **whatis** - פקודה לזיהוי סוג משתנה.
2. **print** - מדפיסה את ערך המשתנה (או ערך ביטוי).

דוגמא : בתוכנית שבה יש משתנה a מסוג int.

```
(gdb) whatis a
type = int
(gdb) print a
$1 = 2
```

ניתן לגשת רק למשתנים שבתחום ההכרה של הפונקציה שבתוכה עצרנו, לקבלת רשימת כל המשתנים המקומיים ניתן להשתמש בפקודה **info locals**.
כדי לעלות לאזור הנתונים של הפונקציה הקוראת ניתן להשתמש בפקודה **up**, כדי לרדת בחזרה ניתן להשתמש ב **down**.
ניתן גם להריץ פונקציה על ידי **print** למשל :

```
(gdb) print is_prime(6)
```

תריץ את הפונקציה is_prime ותדפיס את התוצאה.

שינוי ערכי משתנים:

set var ***varname = value***

למשל

(gdb) set var a = 5

סקירת התוכנית:

list ***file:function***

למשל

(gdb) list file.c:foo

list ***file:line_number***

למשל

(gdb) list file.c:100

עבודה עם נקודות עצירה:

קביעת נקודות עצירה:

עצירה עם **הכניסה לפונקציה** :

break ***file:function***

למשל

(gdb) break file.c:foo

עצירה **בשורה מסויימת** :

break ***file:line_number***

למשל

(gdb) break file.c:35

לסקירת נקודות העצירה ניתן להשתמש ב **info breakpoints**.

לביטול כל נקודות העצירה ניתן להשתמש ב **delete**.

לביטול נקודת עצירה מסויימת ניתן להשתמש ב **clear** באותו אופן שבו משתמשים ב **break**,למשל

(gdb) clear file.c:35

להשעיית נקודת עצירה מסוימת ניתן להשתמש בפקודה **disable** *break_num*, כאשר *break_num* הוא מספרה הסידורי של נקודת העצירה ב **info breakpoints**.

לביטול ההשעיה ניתן להשתמש ב **enable break_num**.

נקודת עצירה מותנית:

זהו כלי שימושי מאוד לבדיקת הנחות לוגיות לגבי נכונות התוכנית (האם הפונקציה עושה את מה שהיא באמת אמורה לעשות ?)
ניתן לקבוע תנאי על נקודת עצירה מסוימת ע"י :

```
(gdb) cond <break_num> (a==5)
```

הרצה מבוקרת של תוכנית:

run - מריצה את התוכנית בתוך ה gdb עד לנקודת עצירה או לסוף התוכנית, ניתן גם להריץ עם פרמטרים למשל :

ללא פרמטרים	(gdb) **run**
עם פרמטרים	(gdb) **run param1 param2 .....**

step - הרצת שורה אחת של התוכנית, אם השורה היא קריאה לפונקציה תבוצע רק השורה הראשונה של הפונקציה.

next - הרצת שורה אחת של התוכנית, אם השורה היא קריאה לפונקציה תבוצע כל הפונקציה כיחידה אחת.

ניתן בכניסה לפונקציה לסקור את מחסנית הקריאות בעזרת הפקודה **where** למשל :

```
(gdb) where
#0 0x229c in a (p = 0x0) at cr.c:3
#1 0x229c0 in c (x = 6) at cr.c:10
#2 0x229dc in b (x = 6) at cr.c:15
#3 0x229fc in main () at cr.c:21
```

דוגמא למציאת bug:

נתונה התוכנית הבאה :

```
#define boolean int
#define TRUE 1
#define FALSE 0
#include <stdio.h>

boolean is_prime(int x)
{
  int limit, checked;
  boolean divided = FALSE;
  limit = x/2;
```

```
  checked = 2;
  while(!divided && (checked < limit))
    {
      if(0 == (x % checked))
        divided = TRUE;
      else
        checked++;
    }
  return (!divided);
}

void main(void)
{
  int num;
  for(num = 2 ; num < 10 ; num++)
    if(is_prime(num))
      printf("%d is prime \n", num);
}
```

ההידור יתבצע על ידי הפקודה :

```
eesoft:t2> gcc -g bug1.c -o bug1
```

פלט :

```
2 is prime
3 is prime
4 is prime          /* THIS IS AN ERROR !!! */
5 is prime
7 is prime
```

דוגמא לעבודה עם gdb :

```
eesoft:t2> gdb bug1
(gdb) list bug1.c
2  #define TRUE 1
3  #define FALSE 0
4  #include <stdio.h>
5
6  boolean is_prime(int x)
7  {
8     int limit, checked;
9     boolean divided = FALSE;
10    limit = x/2;
(gdb) list 6, 20
6  boolean is_prime(int x)
7  {
8     int limit, checked;
9     boolean divided = FALSE;
10    limit = x/2;
11    checked = 2;
12    while(!divided && (checked < limit))
13    {
14          if(0 == (x % checked))
15                divide = TRUE;
16          else
17                checked++;
18    }
19    return (!divided);
20 }
(gdb) break 11 if (x == 4)
Breakpoint 1 at 0x22bc: file bug1.c , line 11
(gdb) run
Starting program: /home/boris/os-course/bug1
2 is prime
```

```
3 is prime

Breakpoint 1, is_prime (x=4) at bug1.c:11
11   checked = 2;
(gdb) info locals
limit = 2
checked = -134218760
divided = 0
(gdb) next
12   while(!divided && (checked < limit))
(gdb) next
19   return (!divided);
(gdb) quit
The program is running. Quit anyway? (y or n) y
```

שאלה: מה הייתה השגיאה בתוכנית? איך לתקנה?

שימו לב: ניתן למצוא מידע על כל פקודה ב-UNIX ע"י ביצוע man, למשל:

1. man man - מידע על פקודת man עצמה
2. man make - מידע ופרמטרים של make (ניתן לתת שם אחר)
3. man -k dir - רשימת הפקודות הקשורות למדריכים

* **שימו לב** כי כאשר קיימים מספר "דפי" מידע לאותה פקודה, יש חשיבות לסוג המידע אותו מחפשים (האם זו פקודת שורה - 1, פונקציה - 2 או 3, וכו'). את "סוג" המידע ניתן להעביר כפרמטר לפקודה man. למשל:

1. man 1 date - נותנת מידע על הפקודה date
2. man 3 date - נותנת מידע על הפונקציה date (אם קיימת)

יש לקרוא man לפני שפונים למתרגלים!!!

ללימוד עצמי: הכלים tar/gzip/gunzip

כמו ב DOS כך גם ב UNIX קיימות תוכנות לדחיסה ופריסה של קבצים.
gzip דוחס קבצים , לדוגמא :

```
% gzip file.c
```

ייצר את הקובץ הדחוס file.c.gz.

```
% gunzip file.c.gz
```

ישחזר את הקובץ file.c.

התוכנית tar מייצרת ארכיבים מקבצים/מדריכים בלי לדחוס אותם.

למשל אם directory הוא שם של מדריך במדריך הנוכחי

```
% tar cvf dir.tar directory
```

ייצר את הארכיב dir.tar, המכיל את תת-העץ תחת directory.

ניתן לפרוש ארכיב זה על ידי הפקודה:

```
% tar xvf dir.tar
```

ניתן לקבל את רשימת הקבצים בארכיב זה על ידי הפקודה:

```
% tar tf dir.tar
```

שימוש: להעברת פרויקט גדול, בדרך כלל , משתמשים תחילה בפקודת tar למדריך הראשי של הפרויקט (פקודת ה- tar עוברת על כל הקבצים ותת-המדריכים בצורה רקורסיבית) אח"כ משתמשים ב-gzip כדי לדחוס את קובץ ה- tar.

הגשת תרגילי הבית: במהלך הקורס תצטרכו להגיש את התרגילים בפורמט tar. בהתאם להנחיות ההגשה, יהיה עליכם להכין מדריך המכיל את כל הקבצים להגשה, כולל קובץ בשם readme המכיל את שמות ומספרי ת"ז של המגישים, לעבור למדריך זה (cd), ולהכניס את כל הקבצים לקובץ tar.

הפקודה שעליכם להריץ היא:

```
% tar cvf 12345678.tar *
```

כאשר 12345678 הוא מספר ת"ז של אחד המגישים. במקום * ניתן להשתמש ברשימת כל הקבצים להגשה:

```
% tar cvf 12345678.tar readme file1.c file2.c file3.h
```