# סיכומים למבחן בקורס פרוייקט תוכנה
**סמסטר א' 2008-9 (פרופ' רודד שרן)**

## פרק 1:
תהליך יצירת תוכנה ב-C:
Source (.c, code file, high level) → preprocessing → compiling (.o, object file – machine code) → linking → executable

**ספריות:**
כדי להוסיף שימוש בספריות יש להוסיף בכותרת:
- #include <library_name.h> - לספריות סטנדרטיות (stdio.h, stdlib.h ...)
- #include "my_library_name.h" – לספריות שאנחנו כותבים

**הערות:**
- כל תוכנית main ((int main(void){...}) חייבת להחזיר בסוף int, ולכן תמיד נסיים ב-"return 0;".
- הגדרת קבועים (בכותרת או בקובץ h): "#define CONST <value>;".
- הגדרת / אתחול מערך רק למספר או לקבוע, לא ניתן להגדיר מערך בגודל תלוי משתנה שאינו קבוע.
- כתיבת הערות בקוד (יכול להתפרס על כמה שורות): "/* <note> */". בקומפילציה הערות מוחלפות ברווח בודד.

## פרק 2,3: Lexical Elements, Fundamental Data-types:
**Keywords**: מילים שמורות לדקדוק של C. תהליך הקומפילציה: preprocessing→lexical analysis → syntax an. → semantic an.
**Identifiers**: משתנים, case-sensitive, יש לבחור שמות בעלי משמעות.
**Fundamental Data-types (גודל ב-bytes בסוגריים על PC Pentium 3)**:
- Integral: (Signed / unsigned) – int(4), short(2), long(4), char(1), unsigned(4) ברירת המחדל היא signed.
  - תוספת של u / U בצמוד לערך מימין הוא unsigned.
  - תוספת l / L בצמוד לערך מימין הוא long.
  - ל-char אפשר לקרוא גם כערך מספרי וגם כערכו התווי. תווים מיוחדים לדוגמא: '\0', '\t', '\n', '\\'.
  - המרות הצגת int:

| בסיס | Decimal | Hexadecimal | Octal |
|---|---|---|---|
| קריאה ב- scanf, printf | %d | %x | %o |
| אופן כתיבת מספר בקוד | 123456 | **0x**12345 | **0**123456 |

  - בדיקת גודל טיפוס נתונים ב-bytes, תלוי מכונה: sizeof(<data_type>)
- Floating point: float(4), double(8), long double(8).
  - תוספת f בצמוד לערך מימין הוא float.
  - תוספת l – long double.
  - מוקצה מקום קבוע (ביטים) לסימן, חלק שלם וחלק שברי (עשרוני).

**המרות אוטומטיות בעירוב משתנים בביטוי אחד:**
- אם מי מהמשתנים הוא מהטיפוסים הבאים, כל המשתנים יומרו לטיפוס זה (לפי הסדר הבא): long double, לאחר מכן: double, לאחר מכן: float, לאחר מכן: unsigned long.
- בהינתן long ו-unsigned: אם ה-unsigned יכול להישמר ב-long בשלמותו, הוא יומר ל-long. אחרת שניהם יומרו ל-unsigned long.
- אחרת: המרה ל-long, המרה ל-unsigned, המרה ל-int (בסדר זה, בהתאם לגדלים).
- <u>הערה</u>: אם נעשה חישוב למשל של ints והתשובה מושמת ב-float, התשובה תחושב לפי ints (כלומר יושם שלם ב-float). למשל: i=5,j=4 שלמים ו-i/j = x float אז יושם ב-x: 1.0000 ולא 1.25.

**Casting**:
- הוספת (<type_to_cast_to>) לפני הערך (משתנה) המושם.
- השמת double ב-int: יופעל trunc (עיגול לערך שלם תחתון).

**Mathematical functions**:
sqrt,log, exp, pow... - כולם נמצאים בספריה <math.h>.

**הערות**:
- התו הראשון משם משתנה <u>לא יכול</u> להיות ספרה.
- EOF: End Of File, מוגדר ב-stdio.h להיות 1-.
- במקרה של חישוב לא חוקי/מוגדר: יחזיר nan (למשל עבור חלוקה ב-0) או inf (למשל עבור 1/0.0).
- אין טיפוס Boolean ב-C, במקום: $anything\ but\ 0 \equiv True;\ \ 0 \equiv False$.

## פרק 4: Control Flow:
פעולות אוניריות (כגון i++) קודמות לפעולות אריתמטיות (כגון +,-,*) קודמות לפעולות השוואה (כגון <, ==, =>) קודמות לפעולות לוגיות (&&, ||).
**Short circuit-eval**:
בשרשור תנאים (and, or) שערוך התנאי יעצר ברגע שיש תשובה ברורה (T/F). כך, ביטוי שיכול להיות לא חוקי (כמו חלוקה במשתנה שיכול להיות 0) יכול לא להשתערך אם מקדים אותו תנאי הבודק אותו.
**If/else statement**: סטנדרטי, אין מה לכתוב על זה.

**Switch statement**:

```
switch(<integral expression>){
    case <val1>:
        <exp1.1>; <exp1.2>;...
        break;
    case <val2>:  ...
```

חייב להיות ביטוי אינטגרלי, כגון int, char
אין צורך במסלוסליים גם למספר ביטויים

**Conditional operator: exp1?exp2:exp3**:

- אופרטור הבודק תנאי ב-exp1, ומשערך בהתאם את exp2 (אם מתקיים) או exp3 (אחרת) – ומחזיר ערך, למשל: $x = 4 > 5 ? 15 : 17$ - במקרה זה יושם ב-x הערך 17. אופרטור זה בעל קדימות (אסוציטיביות) אחרונה.
- סוג הערך המוחזר נקבע ע"י exp2, exp3 (ה"חמור" מביניהם).

**For, While, do-while**:

- לא חייבים לכתוב ביטויים ב-for (למשל: $for(;;)$ זה ביטוי חוקי – לולאה אינסופית).
- ניתן לכתוב מספר ביטויים מופרדים ע"י ",", למשל: $for(sum = 0, i = 0; i < 10; sum+= i, i++)$.

**Comma operator**:

- הקדימות הנמוכה ביותר מכל האופרטורים.
- משוערך משמאל לימין. אם מופיעים כמה ביטויים המופרדים בפסיק, הערך הכולל של כולם יחד יהיה של הביטוי הימני ביותר.

**break command**: עוצר את הלולאה הנוכחית וקופץ ל-statement הראשון לאחר הלולאה ממנה יצא.

**continue command**: עוצר את האיטרציה הנוכחית של הלולאה בה נמצא ועובר לאיטרציה הבאה.

**goto & labels**: goto <label_name>; מקפיץ לשורה המתחילה ב: label_name:...

**הערות**:

- קריאה למשתנה לא מאותחל אינה טעות קומפילציה (אך יכולה כמובן לגרור שגיאת זמן ריצה).
- דגש על סוג הערך המוחזר ל-exp1?exp2:exp3 (מפורט לעיל).
- בכתיבת תנאים עדיף להשתמש בתנאי השוואה יחסית מאשר השוואה רגילה (למשל >, < על פני =! ,==).

## פרק 5: פונקציות:

**פונקצית main**:

ניתן לקבל ארגומנטים לפונקצית main: int main(int args_num, char ** args){...}:

- args_num מחזיק את מספר המחרוזות (מופרדות ברווחים) שנכתבו בשורת הקריאה לתוכנית, כולל שם התוכנית עצמה (למשל: test.exe 123 ariel ab יחזיר 4).
- args יהיה מערך של strings המחזיק את הארגומנטים עצמם שנשלחו לתוכנית, למשל: "args[0]="test.exe.

כדי להשתמש בפונקציות במהלך הקוד יש להגדיר את החתימה שלהן (prototype) בתחילת הקובץ (או לכלול ספריה מתאימה). הגדרה זו לא חייבת לכלול שמות משתנים לוקליים, למשל: int foo(int, char); אך אפשר גם כפי שתמומש אח"כ: int foo(int i, char c);.

**Call by value**:

קריאות לפונקציה המקבלת משתנים כלשהם לא משנה את המשתנים המקוריים, אלא רק לוקחת את ערכם ושמה אותם במשתנים חדשים מקומיים של הפוקנציה.

**הערות**:

- כל ביטוי הנשלח לפונקציה משוערך בעת הקריאה.
- כל ביטוי שנשלח בקריאה לפונקציה מושם, עם casting אוטומטי במידת הצורך, בפרמטר המיועד לו - משתנה חדש בתחילת גוף הפונקציה (call-by-value).
- return מסיים את ריצת הפונקציה בכל שלב בו מופעל.
- בעת return נעשה שערוך (אם זה ביטוי) וגם casting אוטו' במידת הצורך לטיפוס שהפונקציה מוגדרת להחזיר, והערך מוחזר לסביבה הקוראת.

**כתיבת תוכנית בכמה קבצים**:

כוללים בקומפיליציה קובץ h הכולל את כל חתימות הפונקציות בהן משתמשים (לא כולל main; פונקציות יכולות להיות מוגדרות בכל קובץ שהוא). בכתורות כל הקבצים המשותפים לפרוייקט מוסיפים פקודת #include "lib_name.h" לאותו קובץ h.

**Assert**:

הוספת ביטויי Assert לבדיקת תקינות שלבים בתוכנית, כגון קריאת משתנה (האם נקרא, האם הערך שקיבלנו עומד בתנאים מסויימים), גישה לקובץ. למשל: assert(k > 0); - מוודא שהתקבל k חיובי. אם לא, יוצא מהריצה עם הודעה מתאימה.

**Scope rules, nested blocks, global & local vars**:

- משתנה גלובלי: מוגדר בתחילת הקובץ, לא תחת פונקציה; תמיד מאותחל ל-0; משתנה לוקאלי: מוגדר בתוך פונקציה, לא מאותחל אוטו'.
- static eval: משתנים נקראים לפי הסביבה האחרונה בה הוגדרו. אם לא נמצא משתנה, יעבור לחפש בסביבה בה הוגדרה הפונקציה.

**קדימויות למשתנים ומשמעותם**:

- הוספת תחילית auto בהגדרת משתנה (למשל: auto int i = 0;): סימון למשתנה לוקאלי.
- extern: סימון למשתנה גלובלי, כלומר חפש אותו בסביבה הגלובלית. אם לא קיים, לא יקומפל. יאותחל תמיד ל-0.
- register: משתנה המושם ברגיסטר, לגישה מהירה, למשל עבור לולאות על int. המשתנה ישוחרר מהרגיסטר בעת סיום הבלוק (הלולאה).
- static: לפני פונקציה – לא ניתן להשתמש בפונקציה בקבצים אחרים מלבד הקובץ בו מוגדרת. עבור משתנה – אם מוגדר בתוך פונקציה, ישאר עם ערכו גם לאחר סיום ריצת הפונקציה (כך למשל בודקים כמה פעמים נקראה פונקציה כלשהי במהלך תוכנית). ברגע שאותחל משתנה static, הוא לא יאותחל שנית בריצה הבאה של הפונקציה. דגש: משתנה זה חי רק כאשר הפונקציה נקראת, לא ניתן לגשת אליו מחוץ לפונקציה.

## פרק 6: Arrays, Pointers :

**מערכים** :

- כאמור, לא ניתן לקמפל הגדרת מערך עם משתנה, חייבים להגדירו עם קבוע (#define CONST <value>;) או מספר ממש.
- אתחול : - ניתן לא לאתחל, ואז הערכים יהיו זבל.
  - ניתן לאתחל ע"י השמת (חלק מ) ערכים בתוך מסולסליים, שאר הערכים יאותחלו ל-0 : int a[100] = {1,2,3}

**מצביעים** : - אתחול ללא השמת גודל המערך עם ערכים במסולסליים יגדיר את גודל המערך למס' האיברים במסולסליים : int a[] = {1,2}
לכל משתנה יש ערך וכתובת. למשל int i שהכתובת שלו יושבת ב-p (p הוא מסוג *int) :

- **&i**; : קריאה לכתובת של i בזיכרון (כתובת מוחזקת בדרך כלל ב-unsigned int, שזה גודל מילה = 4 bytes).
- **p*** : קריאה לערך שנמצא בכתובת שמחזיק p, שהוא הערך של i.
- 0 או null הוא סימן לכך שאין מצביע. למשל אתחול של p : $int * p = NULL; \equiv int * p = 0; \equiv int * p = '\0';$.
- השמה של כתובת אבסולוטית : p = (int*)<some_unsigned_int> - ע"י casting.
- הדפסה (printf) : מצביע ע"י p%, ע"י u% (unsigned int) או בהקסדצימלי ע"י x%. p% ידפיס בהקס' עם תוספת "0x" לפני, ו-x% בלי.
- לאופרטורים *, & קדימות על פני כל שאר האופרטורים.
- השמות מצביעים לא חוקיות : - קבועים, ביטויים ורגיסטרים, כגון : 4& – "מצביע" למספר 4. כן אפשר : $4(int *)$.
  - p=q כאשר p מסוג *int, q מסוג *float. כן אפשר עם casting או "דרך" v מסוג *void (גנרי) : p=v=q.

**מצביעים ומערכים** :

נניח והגדרנו מערך a[N] int כלשהו ומצביע p* int כלשהו, אזי :

- $a[i] \equiv *(a+i)$ - ב-a+i : קפוץ i יחידות (במקרה זה של int) קדימה.
- $p[i] \equiv *(p+i)$ - כנ"ל.
- כתובת הבסיס של מערך היא בעצם המצביע ל-$a[0]$. כדי להוציא את הכתובת של $a[i]$ : נקרא ל-$\&a[i]$.
- ניתן לבצע השמה : $p = a + 1$ – שים ב-p את הכתובת של a[1]; או : $p = \&a[1]$ – אותה משמעות.
- מהגיון זה, ניתן למשל לרוץ על כתובות ע"י : $for\ (p = a; p < \&a[N]; p++)$. דגש : ++ מקפיץ את p ב-sizeof(int) קדימה.

**ההבדלים העיקריים בין מערך למצביע:**

- זיכרון : למערך מוקצה זיכרון בהגדרה, למצביע צריך לפנות זיכרון. לכן אתחול מצביע לערך כלשהו אסור (כי לא הוקצה זיכרון) ולמערך מותר. כמו כן, מרגע שהוגדר מערך, לא יכול להשתנות גודלו, לעומת מצביע. שחרור זיכרון : אוטומטי במערכים, לא אוטומטי למצביעים (צריך free).
- מצביע יכול לקבל כתובות שונות, כולל מערך. מערך לא יכול לקבל הצבה של כתובת אחרת.
- מכאן שהמהלכים הבאים לא חוקיים : $a = p$ : ניסיון לשים כתובת ממצביע במערך ; השמה חדשה כלשהי במערך, כמו : $a = \&a[2]$, a++.
- sizeof : למערך מחזיר גודל המערך בזיכרון (מס' אלמנטים כפול גודל כל אלמנט) ; מצביע מחזיר גודל קבוע (כתובת = (sizeof(int)).

**אריתמטיקה של מצביעים** :

- פעולת $p1 - p2$ תחזיר unsigned int שהוא ההפרש ביחידות הטיפוס עליהם מצביעים.
- אבל : $(int)p1 - (int)p2$, אם למשל p1,p2 הם מסוג *int, יחזיר את אותו הפרש כפול 4 (כי int תופס 4 bytes).

**Call by reference** :

- ניתן לקרוא בפונקציה למצביע של משתנה, ובכך לשנות את ערך המשתנה המקורי, למשל : פונקציה המקבלת $int * p, int * q$ ומבצעת פעולות על p*, q* – תשנה את הערך של המשתנים שעליהם נקראה הפונקציה.
- העברת מערכים לפונקציה – המערך לא מועתק, אלא מועברת כתובת הבסיס שלו (כתובת בסיס call by value = מערך call by reference).
- ניתן לקרוא לכתובת עם const לפני – כדי לשמור שלא ישנו את הערך המקורי.

**הקצאת זיכרון** : משתנים מקומיים (stack) משוחררים עם סיום הפונ' בה הוגדרו, אך משתנים שמוקצה להם זיכרון (ב-heap) לא משוחררים.

- void *calloc(<#element>, <size-of-elem>) – מחזיר מצביע המוגדר ע"י מספר איברים×גודל המקום לכל אבר. מאתחל הערכים ל-0.
- void *malloc(<size>) – מחזיר מצביע למקום מוקצה בזיכרון בגודל הנדרש. לא מאתחל ל-0.
- free(<pointer>) – משחרר את המקום בזיכרון אליו מצביע המצביע.
- void *realloc(<pointer>, <new-size>) – משנה את המצביע להצביע לגודל הרצוי החדש, תוך שימור ה-DATA הקיים.

לאחר הקצאת זיכרון תמיד נבדוק האם ההקצאה נכשלה (נשארנו עם null-pointer) ע"י : assert(p!=null).

**Strings** :

- פשוט מערך של characters, המכיל באיבר האחרון תמיד את התו '\0' המסמן את סוף המחרוזת. כך למשל string מאורך i היא למעשה מערך chars מאורך i+1 (האחרון תו הסיום) לכן string-size = string-length+1.
- אתחול לדוגמא : $char * s = abc; \equiv char\ s[\,] =$ "abc"; לא לאתחל עם {'a','b','c','\0'} כי זה מתורגם למערך של int.
- פונקצית isspace (מתוך ctype.h) : בודקת האם תו כלשהו הוא רווח לבן כלשהו.
- ספריית string.h מכילה פונקציות שונות על מחרוזות, כגון (רשימה מלאה בסוף) :
  - unsigned strlen(const char*) : מחזירה אורך מחרוזת.
  - char *strcat(char *s1, char *s2) : משרשרת לסוף s1 את s2.
  - char *strcpy(char *dest, char *src) : מקבלת s1, s2. מעתיקה ל-s1 את s2 (דורסת את $|s2|$ התווים הראשונים ב-s1). אם s2 ארוכה מ-s1, אז פשוט מחליפה את כל s1 (לא מחזיר שגיאת קומפילציה).
  - int strcmp(const char *s1, const char *s2) : משווה לקסיקוגרפית בין s1 ו-s2 ומחזירה 0 בשוויון, מספר שלילי אם s1 קטנה מ-s2 (s2 מאוחרת בסדר הלקסי') ומספר חיובי אם s1 גדולה מ-s2.
  - char *strdup(const char *s) : משכפלת את s (יוצרת מקום חדש בזיכרון עבורו) ומחזירה מצביע לשכפול, או null אם נכשלה.

**מערכים רב-מימדיים** :

- עבור a[3][5], מתקיים : $a[i][j] \equiv *(\&a[0][0] + 5 * i + j)$ - לך לכתובת הבסיס של a, זוז $5i + j$ תאים קדימה (בהתאם לטיפוס), והחזר את הערך הנמצא בכתובת שאליה הגעת.
- $*a[i]$ - יחזיר את הכתובת של המערך (כתובת של כתובת) הנמצא במקום ה-i ב-a.
- $**(a + 1)$ - יחזיר את הערך $a[1][0]$, כי מקדם את a ביחידה אחת קדימה, כאשר יחידה אחת = מערך בגודל 3 $(3 * sizeof(int))$.
- אתחול עם מסולסליים : אותו עקרון של מערך חד מימדי : {... ,{...} ,{...} ,{{...}}.

**מצביעים לפונקציות**:
ניתן לשלוח לפונקציה גם פונקציות ע"י קריאה באופן הבא:
ב-prototype תופיע פונ' הארגומנט עם רשימת הארגומנטים באופן הבא: <f-arg-return-val> (*)(<f-arg_arg1>,...), כיוון שקריאה לפונ' כארגומנט הוא קריאה לכתובת של הפונקציה.
בהגדרת הפונ' עצמה, ארגומנט הפונקציה יהיה (למשל): int f(int, double). כאשר נקרא לפונ' עם פונקצית-ארגומנט (שתשב לוקאלית ב-f), נקרא לכתובת של הפונ' שרוצים כארגומנט. למשל: foo(**&arg-func**, arg2, arg3,...).

**Const**:
- הוספת const לפני הגדרת משתנה מקבעת את ערכו ולא מאפשרת לשנותו: כל ניסיון שינוי, כולל השמת כתובתו במצביע (נותן פתח לשינוי אפשרי) לא מתקמפל.
- הופעת const לפני מצביע לא מאפשר לשנות את המצביע או את תוכנו.

**Typedef**:
הגדרת טיפוסים חדשים עם קידומת typedef, למשל: typedef double vector[3]; - מגדיר טיפוס חדש מסוג vector, שהוא מערך d באורך 3.
לאחר הגדרה ניתן להשתמש בטיפוס החדש כבכל טיפוס אחר (עם התכונות של טיפוס האב שלו. למשל כאן – ניתן להתייחס אליו כאל מערך).
הערה: לצורך נוחות בהגדרת טיפוסים חדשים ע"י struct, משתמשים בו גם בהגדרת טיפוס חדש (פרק 9).

**הערות**:
- יש לבדוק assert(p!=null) לאחר הקצאת זיכרון.
- ניתן לשלוח לפונקציה גם מצביע למצביע – כדי שנשנה כתובת לא באופן לוקאלי (כמו שבשליחת מצביע משנים את הערך המקורי, ולא לוקאלי).
- אם רוצים לקרוא לפונ' f כארגומנט, נקרא לכתובת שלה, כלומר ל-&f.

## פרק 7: bit-wise operators:

**אופרטורים**:
& (and), | (or), ^ (xor), ~ (1's complement). כאשר מפעילים על שני משתנים a,b אופרטור כלשהו כזה, הוא מופעל על כל ביט בייצוג של אותם משתנים, ומתייחס לכל מיקום בנפרד. למשל & יפעל באופן הבא: הביט הראשון יהיה $a_0 \& b_0$, הביט השני יהיה $a_1 \& b_1$ וכו'.

**Shift:**
- Shift-left: a<<=3 (או: a=a<<3) – מזיז את הביטים ב-a שלושה מקומות שמאלה, וממלא את ה-lsb'ים שנותרו ריקים ב-0.
- Shift-right: אותו עיקרון; ב-unsigned ימלא את ה-msb'ים שנוצרו ב-0, וב-signed – ימלא ב-msb לפני ההזזה (0 או 1).

נשים לב: a>>n שקולה להכפלה ב-$2^n$, ו-a>>n שקולה לחלוקה ב-$2^n$ (הכפלה ב-$2^{-n}$), ופעולה זו מהירה יותר מאשר * או /.

## פרק 8: The Preprocessor:

**Define, include**:
על include כבר נאמר: <lib.h> לספריות סטנדרטיות, "lib.h" לספריות שלנו.
Define: macro פשוט המחליף כל מופע בקוד בדיוק בערך שניתן לו בהתחלה (או בספריה כלשהי), למשל: #define PI 3.14.
שימוש ב-#undef: ביטול הגדרה כלשהי המופיעה באחד ה-include'ים כדי שנוכל לדרוס אותו בקובץ זה.

**Macros & Parameters**:
הגדרת מקרו באופן הבא:

```
#define identifier0(identifier1, ... ,identifierk) string
```

מעיין הגדרת פונקציה, כאשר identifier0 הוא שם "הפונקציה", identifier1-identifierk – שמות המשתנים, וה-string הוא הגוף. למשל:

```
#define SQ(X) ((X)*(X));
```

שיטה זו (כשניתן לעשותה) פשוט מחליפה בקומפילציה בקוד כל מופע בדיוק בתבנית המוגדרת, והיא לעתים יעילה יותר מפונקציה אמיתית (שם תקרא פונקציה, תפתח לה סביבה עם משתנים וכו').
- חשוב להשתמש בסוגריים, כי יתכן ונקבל ביטויים מורכבים, שישוערכו אחרת ללא סוגריים (למשל a+b ב-SQ בלי סוגריים: a+b*a+b).
- לא נסיים את שורת הגדרת המקרו עם ';'.

ניתן לקנן macros: נגדיר תחילה מקרו, ובהגדרת מקרו אחריו נוכל כבר להשתמש בזה שהגדרנו.

**Operating the preprocessor only**:
הרצת gcc –E file.c מציגה את הקוד אחרי שלב ה-preprocessor. בשלב זה:
- כל ה-define מוחלפים בקוד ושורת ה...define# מוסרת.
- כל ההערות מוסרות (ומוחלפות ברווח לבן יחיד).

**Conditional Compilation**:
- #if <exp>: אם הביטוי משוערך אמת, השורות שאחריו עד #endif יכנסו לקוד, אחרת יזרקו ב-preprocessor (יש גם #elif ו-#else).
- #ifdef <macro>: אם המקרו macro הוגדר (יש לו #define) אז השורות שאחריו עד #endif/#else יכנסו לקוד. ניתן גם לכתוב באריכות: #if defined (identifier) עבור ביטויים ארוכים.
- #ifndef <macro>: אם המקרו macro לא הוגדר, אז השורות שאחריו עד ה-#endif/#else יכנסו לקוד.

ניתן להשתמש באופרטורים לוגיים, כגון &&, || וכו'.
**Macros קיימים**: קיימים כל מיני פקודות מקרו מוכנות, שלא נפרט כאן (סוף פרק 8).

## פרק 9: הגדרות טיפוסי נתונים חדשים:

השיטה הטובה ביותר להגדיר טיפוס נתונים, ושיהיה ניתן ליצור משתנים מסוגו (ללא צורך ב-typedef נפרד):

```
typedef struct{
        <inner fields, such as: "int number;">
} <struct-name>;
<struct-name> a, b; ...      ניתן מיד ליצור ממנו מופעים
```

ניתן להגדיר גם אחרת (רק עם struct) אבל אז נצטרך להוסיף: typedef struct <name> <name> כדי שנוכל ליצור ממנו מופעים. אם לא נוסיף זאת, כל הגדרת משתנה חדש מטיפוס זה תהיה בצורה: struct <name> a,b;..., ולא פשוט: <name> a,b;....

- אם נבצע כעת a=b, תתבצע השמה של שדות פנימיים.
- נניח יש לנו טיפוס: struct{ int f1; int f2} newType; ,newType tmp =...; ,newType *p = &tmp.
  אז: $tmp.f1 \equiv (*p).f1 \equiv p->f1$
- אם מגדירים מספר structים, ניתן ליצור באחד שדה מסוג struct אחר, כל עוד הוא הוגדר קודם לכן. לעומת זאת, אם נרצה שיהיה בו שדה של מצביע ל-struct אחר, אין חובה שה-struct הזה יוגדר לפני, כיוון שמצביע הוא תמיד גודל קבוע (sizeof(unsigned int)).
- אתחול של struct יתבצע עם מסולסליים. למשל לדוגמא לעיל: newType x = {3, 15};. ניתן גם להשאיר מקומות ריקים, שיאותחלו ל-0.

**הערות:**

- ניתן להגדיר ב-struct שדה מסוג struct אחר בתנאי שהוגדר קודם. ניתן להגדיר שדה מצביע ל-struct אחר, גם אם טרם הוגדר (גודל ידוע).

## פרק 10: רשימות מקושרות:

מבנה ה-linked list:

```
typedef char DATA;          כאן נשתמש בסוג הערך המוחזק באיברי הרשימה כתווים
struct linked_list {
        DATA    d;                  שדה של ערך
        struct linked_list*next;    שדה של מצביע לאותו סוג טיפוס
};
typede  struct linked_list          ELEMENT;        הגדרת שם טיפוס לאיבר ברשימה
typedef struct linked_list          *LINK;          הגדרת שם טיפוס למצביע לאיבר ברשימה
```

דוגמא ליצירת איבר (נניח ראש הרשימה):

```
LINK     head;                                                  יוצרים מצביע לאיבר ברשימה
head = (ELEMENT *) (malloc(sizeof(ELEMENT)));                   הקצאת מקום בזיכרון
                    casting to LINK
assert(head!=null)                                              בדיקה שההקצאה הצליחה
head->d = 'A';                                                  השמת ערך לדטה
head->next = (ELEMENT *) (malloc(sizeof(ELEMENT)));             הקצאת זיכרון למצביע לאיבר הבא
                    casting to LINK
assert(head->nex != null);                                      בדיקה שההקצאה הצליחה
```

**stacks**: רעיון דומה, לא יפורט כאן.

## פרק 11: printf, scanf:

**תווי המרה ודגלים:**

| c | d | u | o | x | e | f | g | s | p | n |
|---|---|---|---|---|---|---|---|---|---|---|
| char | int | unsigned int | octal rep. | hexa rep. | scientific rep. | float | double | string | pointer ("0x...") | כמה תווים הצלחת לקרוא עד כה |

- דגלים: תוספת בצמוד משמאל לתו ההמרה של h – עבור short ושל l – עבור long. למשל: %hd ; %lf – (עבור long double).
- width.precision: למשל 5.3%f יכתוב את ה-float עם שתי ספרות משמאל לנקודה ושלוש ספרות מימין לנקודה. ה-width נותן הזחה של רווחים במידת הצורך. דוגמאות עבור 'c': %5c – "    c" ; %-3c – "c  ". דוגמאות עבור "hello": %2s – "he" ; %.2s – "he". %-8.2s – "he    ".
- הוספת *: מתעלם מההמרה, למשל עבור %*s יתעלם מה-string הבא שיפגוש. דוגמא: "%d %*s %s", &i, &str שיקבל 8 abc def יבצע: i יקבל את הערך 8, str יקבל את הערך "def".

**printf:**

מחזירה את מספר התווים שנכתבו, או מספר שלילי אם נכשלה. הערה: תווים מיוחדים כמו '\n' גם נספרים, למשל printf("123\n"); יחזיר 4.

**scanf:**

- מחזיר EOF אם לא מקבל כלום, אחרת מחזיר את מספר ההמרות המוצלחות.
- ב-scanf תמיד נציב מצביעים, למשל כדי לשים ב-i int נכתוב: scanf("%d", **&**i);.
- כתיבת scanf("%[^x]", str) תקלוט לתוך str מחרוזת ללא הופעות של 'x' בתוכה.

**קבצים:**

- fprintf: הדפסה לתוך קובץ. למשל: fprintf(stdout, ...) (הדפסה מה-standard output) שקולה ל: printf(...). מחזיר את מספר התווים שנכתבו.
- fscanf: קריאה מקובץ. למשל: fscanf(stdin, ...) שקולה ל-scanf(...). מחזיר את מספר ההמרות המוצלחות.
- sprintf(str, ...): תכתוב לתוך המחרוזת str. sscanf(str,...): תקרא מתוך המחרוזת str (במקום מתוך ה-standard input).

פתיחת וסגירת קבצים:

- fopen: מחזיר מצביע לקובץ (FILE) נפתח או null אם לא הצליח לפתוח אותו. אופן שימוש:
  - fopen("filename", "r") – פתיחת הקובץ לקריאה. "r+": גם לכתיבה, מתחיל בתחילת הקובץ.
  - fopen("filename", "w") – פתיחת הקובץ לכתיבה. אם לא קיים, ייצור אותו, אחרת ימחק את הישן. "w+": גם לקריאה.
  - fopen("filename", "a") – פתיחת הקובץ לכתיבה. אם לא קיים ייצור אותו, אחרת ישרשר לסוף. "a+": גם לקריאה. יתחיל בסוף.
  - תוספת "b" (כמו: "wb") – אותו דבר רק לקובץ בינארי.
- fclose: מחזיר 0 אם הצליח לסגור את הקובץ ו-EOF אם הסגירה נכשלה או אם הקובץ כבר סגור.

קריאת/כתיבת תו בודד:

- getc / fgetc(FILE*): בהינתן מצביע לקובץ, מחזיר את התו הבא בקובץ, או EOF בסוף קובץ או שגיאה. למשל getchar() שקול ל-getc(stdin).
- putc / fputc(char, FILE*): בהינתן תו ומצביע לקובץ, שם בסוף הקובץ את התו שניתן לו. למשל: putchar(c) שקול ל-fputc(c, stdout).

קבצים בינאריים:

- size_t fwrite(const void *p, size_t elem_s, size_t n_elem_s, FILE *fp): מקבל מצביע למערך כלשהו, גודל כל איבר במערך, גודל המערך כולו ומצביע לקובץ. **כותבת לקובץ את המערך** שמתקבל. מחזירה את מספר האיברים שהצליחה לכתוב לקובץ.
- size_t fread(void *p, size_t elem_s, size_t n_elem_s, FILE *fp): **כותבת** הפעם את המערך **מהקובץ אל המצביע למערך**, כאשר המערך מגודל n_elem_s, וכל איבר בו מגודל elem_s. מחזירה את מס' האיברים שהצליחה לקרוא. צריך להקצות למערך זיכרון בגודל מתאים.

מיקום בקבצים:

- void rewind(FILE *fp) – מחזיר את המצביע בקובץ לתחילת הקובץ.
- long ftell(FILE *fp) – מחזיר את המיקום הנוכחי בקובץ או 1L- אם נכשל.
- int fseek(FILE *fp, long offset, int place) – קופץ בקובץ למיקום הנמצא ב-offset ממיקום place. place יכול לקבל 0,1,2 (מוגדרים כקבועים SEEK_SET...) המייצגים בהתאמה את תחילת הקובץ, המיקום הנוכחי בקובץ או סופו.

יש עוד כל מיני פעולות I/O (מפורטות בסוף פרק 11), שלא יפורטו כאן.

**דברים שמורידים עליהם:**

- אי בדיקה של: הקצאת זיכרון, הצלחת פתיחת קובץ, הצלחת קריאת N שלמים (file read), הצלחת כתיבת N שלמים.
- בדיקה לא נכונה של argc.
- קריאה לא נכונה של ארגומנטים לתוכנית.
- הקצאה סטטית [<var>]int arr.
- הקצאה בגודל לא מתאים.
- עבודה עם מערך שאינו *int.
- טיפול לא נכון בקבצים בינאריים.
- אלגוריתם לא בסיבוכיות הנדרשת.
- אי מימוש פונקציית compare ב-qsort.
- הגדרת פונקצית compare עם *int במקום *void.
- חוסר ב-casting.
- אי שחרור זיכרון או סגירת קבצים בסוף תוכנית.

**טבלת קדימויות:**

| Operators | Associativity |
|---|---|
| () [] . -> ++(postfix) --(postfix) | left to right |
| +(unary) -(unary) ++(prefix) --(prefix) ! sizeof(type) &(address) *(dereference) ~ | right to left |
| * / % | left to right |
| + - | left to right |
| << >> | left to right |
| < <= > >= | left to right |
| == != | left to right |
| & | left to right |
| ^ | left to right |
| \| | left to right |
| && | left to right |
| \|\| | left to right |
| ?: | right to left |
| = += -= *= /= &= >>= etc | right to left |
| ,(comma operator) | left to right |

**qsort:**

```
qsort's prototype in stdlib.h (using size_t from stddef.h):
void qsort(void *array,
           size_t n_elem,
           size_t elem_size,
           int compare(const void *, const void *));
compare(a,b) returns negative int if a<b
             returns 0            if a=b
             returns positive int if a>b

(int-qsort.c)

// qsort an array of ints

#include <stdio.h>, #include <stdlib.h>, #include <assert.h>, #define   KEYSIZE   16

int compare_int(const void *p1, const void *p2);
int main(void){
   int   key[] = { 4, 3, 1, 67, 55, 8, 0, 4, -5, 37, 7, 4, 2, 9, 1, -1 };
   qsort(key, KEYSIZE, sizeof(int), compare_int);
   return 0;
}
int compare_int(const void *p1, const void *p2){
  const int *q1 = p1, *q2 = p2;
  return ((*q1)-(*q2));
}
```

**string.h**:

char *strcpy( char *s1, const char *s2)

- copies the string **s2** into the character array **s1**. The value of **s1** is returned.

char *strncpy( char *s1, const char *s2, size_t n)

- copies at most **n** characters of the string **s2** into the character array **s1**. The value of **s1** is returned.

char *strcat( char *s1, const char *s2)

- appends the string **s2** to the end of character array **s1**. The first character from **s2** overwrites the '\0' of **s1**. The value of **s1** is returned.

char *strncat( char *s1, const char *s2, size_t n)

- appends at most **n** characters of the string **s2** to the end of character array **s1**. The first character from **s2** overwrites the '\0' of **s1**. The value of **s1** is returned.

char *strchr( const char *s, int c)

- returns a pointer to the first instance of **c** in s. Returns a **NULL** pointer if **c** is not encountered in the string.

char *strrchr( const char *s, int c)

- returns a pointer to the last instance of **c** in s. Returns a **NULL** pointer if **c** is not encountered in the string.

int strcmp( const char *s1, const char *s2)

- compares the string **s1** to the string **s2**. The function returns 0 if they are the same, a number < 0 if **s1** < **s2**, a number > 0 if **s1** > **s2**.

int strncmp( const char *s1, const char *s2, size_t n)

- compares up to **n** characters of the string **s1** to the string **s2**. The function returns **0** if they are the same, a number < 0 if**s1** < **s2**, a number > 0 if **s1** > **s2**.

size_t strspn( char *s1, const char *s2)

- returns the length of the longest substring of **s1** that begins at the start of **s1**and consists ***only*** of the characters found in **s2**.

size_t strcspn( char *s1, const char *s2)

- returns the length of the longest substring of **s1** that begins at the start of **s1**and contains ***none*** of the characters found in **s2**.

size_t strlen( const char *s)

- determines the length of the string **s**. Returns the number of characters in the string ***before*** the '\0'.

char *strpbrk( const char *s1, const char *s2)

- returns a pointer to the first instance in **s1** of any character found in **s2**. Returns a **NULL** pointer if no characters from **s2** are encountered in **s1**.

char *strstr( const char *s1, const char *s2)

- returns a pointer to the first instance of string **s2** in **s1**. Returns a **NULL** pointer if **s2** is not encountered in **s1**.

char *strtok(char *s1, const char *s2)

- repeated calls to this function break string **s1** into "tokens"--that is the string is broken into substrings, each terminating with a '\0', where the '\0' replaces any characters contained in string **s2.** The first call uses the string to be tokenized as **s1**; subsequent calls use **NULL** as the first argument. A pointer to the beginning of the current token is returned; **NULL** is returned if there are no more tokens.

- הסבר על strtok: כל פעם מחזירה מצביע לtoken הבא מתוך s1 שהוא המחרוזת מתחילת s1 ועד הופעה ראשונה של אחד התווים המרכיבים את s2. למשל strtok("hello1there", "1") תחזיר מצביע ל-"hello". דוגמא:

```
/* strtok example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] ="- This, a sample string.";
  char * pch;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok (str," ,.-");
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

ה-tokens שיוחזרו יהיו כל המילים המרכיבות את המשפט.