רשימת נושאים:

היפר קישור לקפיצה לנושא: ללחוץ ctrl ועל הכותרת

אם יש הערות להוסיף/ לתקן אשמח לקבל עדכון במייל: mayha3@mta.ac.il

01 - קלט פלט

רשימת המרות: (%=פקודת המרה)

Character	Argument type; Printed As
d,i	int; decimal number
o	int; unsigned octal number (without a leading zero)
x,X	int; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ...,15.
u	int; unsigned decimal number
c	int; single character
s	char *; print characters from the string until a '\0' or the number of characters given by the precision.
f	double; [-]*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
e,E	double; [-]*m.dddddd*e+/-*xx* or [-]*m.dddddd*E+/-*xx*, where the number of *d*'s is given by the precision (default 6).
g,G	double; use %e or %E if the exponent is less than -4 or greater than or equal to the precision; otherwise use %f. Trailing zeros and a trailing decimal point are not printed.
p	void *; pointer (implementation-dependent representation).
%	no argument is converted; print a %

עיצוב תצוגה: % [-][x][.y] z

z	y	x	-
תו המרה	מס' ספרות אחרי נקודה	מס' מקומות להצגה	יישור לשמאל

- מס' מקומות **משתנה** לתצוגה: % [*][x][.y] z

נגדיר משתנה נוסף int name1 עבור מס' מקומות שנרצה לשמור לתצוגה, ונדפיס: printf("%*[z]", name1, [ ]);.

ניתן לשנות את המשתנה name1 במהלך התוכנית (לקלוט ממשתמש או השמה חדשה).

תווים מיוחדים:

\t	\n	\b	\\	\a
הזחה TAB	שורה חדשה	רווח	הצגת התו \	פעמון

EOF – תו קבוע (-1) המציין סוף קלט, מוגדר בספריה stdio.h.

פונק' קלט חשובות נוספות: (stdio.h)

קולטת תו	int getchar(void)	ניתן לבצע השמה ל-int או ל-char (ASCII).
מדפיסה תו	int putchar(int c)	
קולטת שורה (עד \n)	char* gets(char *s)	עד לירידת רווח. יש לוודא שגודל המחרוזת s* מתאים לקלט
מדפיסה מחרוזת, ויורדת שורה אחרי	int puts(cont char *s)	
קולטת תו/מחרוזת עד לרווח	int scanf()	
מדפיסה קלט מבוקש	int printf(char *format, arg1,...)	מבצעת המרה לארגומנטים, מחזירה מס' תווים שהודפס

הערה: לאחר הדפסת הכוללת \n- אם נרצה לקלוט תו ממשתמש, נציין לפני getchar() (ללא שימוש), כדי להימנע מקליטת התו \n.

פונקציות פלט-קלט שימושיות:

Int Sprint(char *dest, char *format, arg1, ...)- פלט למחרוזת. מחזירה מס' תווים שעדכנה (ניתן "לשרשר" עם indx וחשבון מצביעים).

int Sscanf(char *src, char *format, &arg1, ...)- קלט ממחרוזת. מחזירה מס' ערכים שקלטה בהצלחה לכתובות משתנים.

Char* strtok(char *s, const char *delim)- "חותכת" מחרוזת עד למפריד delim*. הפונק' בעצם מכניסה '\0' במיקום בו מצאה delim ראשון, ומחזירה מצביע למיקום תחילת המחרוזת, אחרת מחזירה NULL (ניתן לשרשר בלולאה המשך pToken = strtok(NULL, delim).

02 – מצביעים ורשומות (כולל מערכים והקצאות דינמיות)

מצביעים:

- מצביע הוא משתנה המחזיק כתובת, בדר"כ גודלו 4 בתים (תלוי במחשב). בהגדרתו יש לציין את סוג הטיפוס אליו הוא מצביע.
- **אופרטורים של מצביעים:**

1. **אופרטור &:** נשתמש כאשר נרצה להשיג כתובת של משתנה. למשל השמה למצביע: <type>* ptr = &i.
2. **אופרטור *:** גישה לזיכרון- לערך השמור בכתובת משתנה. למשל: גישה לכתובת שמחזיק מצביע (אם ptr=&i אז: ptr* = i).

דוגמה: ptr* ו-i משתנים מטיפוס זהה. נגדיר את ptr להצביע על כתובת i כך: ptr = &i. מתקיים: ptr* ≡ i.

- **NULL**: מצביע שאינו מחזיק אף כתובת (ptr* = 0 ≡). לא ניתן לגשת למצביע NULL עד אשר ביצוע השמה שאינה NULL.
- **הדפסת מצביע:** ניתן להדפיס ערך כתובת מצביע ע"י %u ושם המשתנה printf("%u", ptr).
- **השמה למצביע:** משתנה מצביע מוגדר עבור טיפוס מסוים, לכן לא ניתן לבצע השמה עבור כתובת של משתנה מטיפוס אחר.
- **מצביע גנרי:** מצביע חסר טיפוס void*. ניתן לבצע לו השמה לכל טיפוס אחר של מצביע, לא ניתן לעבוד עליו עם חשבון מצביעים.
- **חשבון מצביעים:**

1. **ptr ± k**: עבור ptr *(type) , k מס' שלם, הפעולה תבצע קידום ב-k אלמנטים ((sizeOf(type בתים) מהכתובת שהוא מצביע עליה.
2. **ptr2 ± ptr1**: הפעולה תחזיר את מס' האלמנטים <u>בהתאם לטיפוס</u> (זהה!) שקיימים בין המצביעים (מס' שלם).

הערה: חיסור מצביעים מטיפוסים שונים "רץ", אבל אין בכך שימוש.

רשומות:

- **מבנה רשומה:** רשומה היא בעצם "תבנית" לטיפוס חדש. רק לאחר הגדרת משתנה מטיפוס רשומה יוקצה זיכרון.

בדר"כ נשתמש ב-typedef להגדיר שם חדש לטיפוס הרשומה (מוסכמה: אות גדולה בהתחלה).

- **אתחול רשומה:** ניתן לאתחל בעת הגדרת המשתנה והשמה כמו מערך { כל השדות }, או פניה לשדה-שדה וביצוע השמה בהתאם.
- **אופרטורים של רשומות:**

1. **אופרטור נקודה**: פניה ממשתנה רשומה לשדה מסוים ברשומה: structName.member.
2. **אופרטור חץ:** פניה ממצביע מטיפוס רשומה p לשדה מסוים ברשומה: p->member (שקול ל- (*p).member).

- **רשומה המכילה מערך:** תחביר גישה לאיבר במערך: (structName.arr)[i] או (structName.arr + i)*.
- **העברת רשומה לפונק':** מועברת <u>כהעתק</u>. כדי לעבוד על הרשומה המקורית יש לשלוח את מצביע לכתובת הרשומה כפרמטר לפונק'.

לצורכי <u>יעילות</u> נעביר רשומה כמצביע לפונק'. במידה ולא נרצה לבצע שינויים על הרשימה המקורית נגדיר אותה כמשתנה const.

- **העתקת רשומות:** ניתן להעתיק רשומות זהות ע"י שורת השמה (s1=s2). כל השדות יועתקו, כאשר עבור שדה מצביע תועתק הכתובת.

כדי להעתיק ערך מצביע לשדה מטיפוס מצביע- יש להקצות לו זיכרון דינמי ואז לבצע העתקה בהתאם לטיפוס (למשל strcpy למחרוזת).

- **אופרטורים לוגיים:** אופרטורים לוגיים לא עובדים על רשומות. יש לפנות לכל שדה בנפרד ולהשתמש באופרטור בהתאם לטיפוס שלו.
- **רשומה בתוך רשומה:** אם lst1 מכילה שדה lst2- ניגש לאיבר (member) בשדה lst2 ע"י אופרטור נקודה: lst1.lst2.member.

אם נרצה לגשת ממצביע לרשומה נשתמש באופרטור חץ: pLst1->lst2.member.

```
struct rect r, *rp = &r;
```

then these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

מערכים:

- **מערכים**: שם המערך הוא כתובת תחילת המערך, כלומר מצביע constant. מתקיים: arr[i] ≡ *(arr + i) וגם: &(arr[i]) ≡ (arr+i) .
- **סדר איברי מערך**: איברי המערך נשמרים ברצף בזיכרון, החל מכתובת תחילת המערך (כתובות עוקבות בקפיצות sizeOf(type) בתים).

1. **כתובת מערך:** &arr = כתובת המערך כולו, מטיפוס [SIZE](*)<type> (לעומת arr = כתובת תחילת המערך מטיפוס (*)<type>).
2. **גודל מצביע מערך:** sizeof(arr) = מס' הבתים הכולל של כל איברי המערך (כלומר הגודל הוא: SIZE * (sizeof(<type>).
3. **קפיצה במערך עם מצביע:** ניתן להגדיר מצביע לכתובת תחילת המערך (שם המערך) ולרוץ בעזרתו על איברי המערך.

דוגמה: for(p = arr ; p < &(arr[SIZE]) ; p++) ופנייה לאיבר ע"י *p ≡ for(i = 0 ; i < SIZE ; i++) ופנייה לאיבר ע"י arr[i] או *(arr+i).

- **מערך דו-ממדי arr[ROWS][COLS] :** נתייחס אליו כאל "רצף" של ROWS תתי מערכים, כך שכל שורה מהווה מערך בגודל COLS.

בעצם, נוכל להגדיר מערך דו-ממדי כמערך של מצביעים, כאשר כל מצביע הוא מערך בפני עצמו (מערך של מערכים).

ניתן להתייחס לשורה מסוימת (תת מערך) בלבד ע"י arr[row] – גישה למערך בגודל COLS של שורה row.

1. **גישה לאיבר במערך:** arr[row][col] ≡ *(arr + COLS *col + row) ≡ *(&arr[0][0] + COLS*col + row).
2. **גודל קפיצה של מצביע:** כל שורה היא בעצם "מערך של COLS" מאותו טיפוס, לכן תחביר מצביע לקפיצה: [COLS](*p)<type>.

דוגמה: עבור: for(p = arr ; p < (arr + ROWS) ; p++) (ריצה על שורות), נפנה לאיבר ה-i בשורה ע"י [i](p*).

3. **טווח מערך דו-ממדי:** כולל את כל ROWS תתי המערכים. כלומר טווח המערך הוא COLS*ROWS איברים (ניתן לרוץ ב-for אחד).

דוגמה: for(int i=0 ; i < ROW*COLS ; i++) ניגש לכל איבר במערך הדו-ממדי ע"י arr + i**.

דוגמה שקולה: for(type* ptr = arr ; ptr < (arr + ROWS) ; ptr++) ניגש לכל איבר במערך הדו-ממדי ע"י (ptr*).

כלומר, כל עוד ניסינו לגשת לאיבר בטווח מ-0 עד COLS*ROWS נקבל ערך מוחזר שהוא איבר במערך הדו-ממדי (אחרת שגיאה).

- **מערך לפונק':** מועבר מצביע (כתובת), לכן כל שינוי שיבוצע על איברי המערך יישמר בסיום הפונק' (לא נוצר העתק של המערך).
- **מחרוזת**: מערך של תווים, כאשר התו האחרון הוא '\0' (=NULL). כלומר, מחרוזת היא מצביע constant מסוג תו.
- **מערך מחרוזות:** מערך דו-ממדי של תווים. כל שורה מהווה מחרוזת, שניתנת להדפסה ע"י פנייה לשורה בלבד ("%s", arr[row]).

דוגמה: for (char** p = arr ; p < (arr + ROWS) ; p++) הדפסת כל שורה (=מחרוזת) ע"י פנייה לשורה כך: p*.

דוגמה שקולה: for (int i = 0 ; i < ROWS ; i++) הדפסת כל שורה (=מחרוזת) ע"י פנייה לשורה: arr[i] או *(arr + i).

- **מערך מצביעים- מחרוזות**: ניתן להגדיר מערך מצביעים מטיפוס תו עבור ROWS מחרוזות בגדלים שונים: char* arr[ROWS].

הקצאות דינמיות:

- הקצאה דינמית היא הקצאת זיכרון תוך כדי ריצה של התוכנית (גודל שלא ידוע מראש). ספריה: stdlib.h.
- ערך ההחזר עבור בקשת הקצאה הוא *void, לכן יש לבצע המרה מפורשת בהתאם לטיפוס המבוקש: (<type>)(*ptr) = f().
- **סוגי ההקצאות הדינמיות:**

1. Void* malloc(size_t size) O(1)

מקבלת: size_t size = מס' **הבתים** מבוקש להקצאה.

מחזירה: מצביע *void לתחילת שטח הזיכרון המכיל זבל. אם נכשלה ההקצאה יוחזר NULL.

2. Void* calloc(size_t n, size_t size_el) O(n)

מקבלת: size_t n = מס' האלמנטים שנרצה להקצות, size_t size_el = גודל **הבתים** של כל אלמנט ((sizeof(element).

מחזירה: מצביע *void לתחילת שטח הזיכרון המאותחל ל-0. אם נכשלה ההקצאה יוחזר NULL.

3. Void* realloc(void* ptr, size_t size) O(n)

מקבלת: ptr*= כתובת תחילת המערך שנרצה לעדכן את גודלו (להגדיל/להקטין), size_t size= גודל השטח החדש שנרצה **בבתים**.

מחזירה: *void לתחילת הזיכרון החדש, מעתיקה את ערכי ptr* (הנוספים יהיו זבל), ומשחררת את ptr* (אחרת NULL**.**

הערה: במידה ויש מקום ב-heap הפונק' תחזיר את אותו מצביע, אחרת תשחרר את זיכרון המצביע ותחזיר מצביע לכתובת חדשה.

הערה: ניתן בעזרתה ל"הקטין" גודל של מערך, במידה והוקצה גודל MAX מראש וידוע לנו שנעשה שימוש רק ב-k בתים (=size_t).

- בכל הקצאה **יש לבדוק האם נכשלה** (if(ptr==NULL, ולשחרר את הזיכרון השמור במצביע בסיום השימוש בה ע"י free(ptr).
- **מערך דינמי ופונק':** ניתן להקצות מערך דינמי ב-2 דרכים ע"י פונק' שאינן main:

1. **ערך מוחזר**: נגדיר החזר מסוג מצביע לטיפוס המתאים, כאשר בפונק' נגדיר אותו כמשתנה מקומי, נקצה לו זיכרון דינמי ונעבוד עליו. יש לזכור בסיום השימוש במצביע שהוחזר מחוץ לפונק' לשחרר את הזיכרון (במקום בו נעשה בו שימוש אחרון).
2. **משתנה קלט-פלט:** נגדיר משתנה פלט- כתובת למצביע (**), ונעבוד עם כתובת המשתנה ע"י גישה לערך שלו ע"י אופרטור *. בפונק' נקצה למצביע זיכרון דינמי ונעבוד על כתובת המצביע (ע"י אופרטור *), בסיום נשחרר את המצביע במקום המתאים.

03 – סוגי משתנים

משתנה לוקאלי / אוטומטי

- משתנה פנימי "חי" בתוך הפונק' שבה הוא הוגדר. בקריאה לפונק' מוקצה עבורו זיכרון ב- stack, אשר מוחזר בסיום הריצה שלה ("מת").
- ערך משתנה פנימי בפונק' אינו נשמר בין קריאות שונות לפונק'. בכל פעם שנקרא לפונק' המשתנה יאותחל מחדש בהתאם.

משתנה גלובלי / חיצוני

- משתנה חיצוני "חי" מחוץ לכל פונק' מתחילת התוכנית ועד לסיומה. extern <type> <name> (מילה שמורה).
- הגדרת משתנה חיצוני מתבצעת פעם אחת מחוץ לכל פונק', בדר"כ בתחילת הקובץ (בהגדרתו מתבצעת הקצאת זיכרון).
- ניתן לקרוא או לשנות ערך של משתנה חיצוני בכל פונק', כל עוד הוצהר עליו כמשתנה חיצוני (כמו בהגדרה, ללא הקצאת זיכרון נוספת).
- ערך משתנה חיצוני נשמר גם לאחר סיום ריצה של פונק' שקבעה את ערכם (כל עדכון נשמר במקור).

משתנה סטטי

- משתנה סטטי "חי" לאורך כל התוכנית ומוכר בתוך הפונק'/ הבלוק שבו הוא הוגדר.
- משתנה סטטי חיצוני: מוגדר בתוך הקובץ בו הוא הוגדר בלבד (אחרים לא יכולים לגשת אליו). Static <type> <name> (מילה שמורה).
- משתנה סטטי לוקאלי: שומר על ערכו בין קריאות שונות לפונק', ומוכר רק בתוכה. אם לא אותחל בשורת ההצהרה עליו, יאותחל ל-0.

משתנה בבלוק { }

- משתנה המוגדר בתוך בלוק "חי" בתוך הבלוק בלבד, ומאותחל מחדש בכל פעם שהבלוק מתבצע.

אתחול משתנים (אם לא אותחלו מפורשות לערך מסוים)

- משתנה לוקאלי מאותחל להיות "מזובל" (ערך לא מוגדר), לעומת משתנים סטטיים/ גלובליים המאותחלים להיות 0.

סוג משתנה	מקום הגדרה	תחום הכרה scope	אורך חיים	אתחול	הערות
לוקאלי/ אוטומטי	בתוך פונק' / בלוק	בתוך פונק' / בלוק	חיי הפונק'	זבל	זיכרון מוקצה ב-stack ומשוחרר בסיום ריצת הפונק' בכל קריאה לפונק' המשתנה מאותחל מחדש
חיצוני/ גלובלי	מחוץ לכל פונק'	החל ממקום ההגדרה	חיי התוכנית	0	extern <type> <name> זיכרון מוקצה פעם אחת בהגדרתו ניתן לקריאה/שינוי בכל פונק' בה הוצהר עליו ערכו נשמר גם לאחר סיום ריצת הפונק'
סטטי - חיצוני	בתוך פונק' / בלוק	בתוך פונק' / בלוק	חיי התוכנית	0	static <type> <name> קבצים אחרים לא יכולים לגשת אליו
סטטי - לוקאלי	בתוך פונק' / בלוק	בתוך פונק' / בלוק	חיי התוכנית	0	static <type> <name> שומר על ערכו בין קריאות שונות לפונק'

04 – רשימות מקושרות

- מבנה נתונים דינמי המאפשר שמירת נתונים בזיכרון שאינו נשמר ברצף (יש להקצות זיכרון בהתאם ולשחרר בסיום השימוש).
- כל איבר ברשימה מקושרת נקרא צומת (מתחילה בצומת- head). רשימה מקושרת מורכבת מ-2 רשומות:

1. רשומה עבור צומת: שדה עבור ערך data (אולי יותר מ-1), ושדה מצביע לצומת הבא (מטיפוס רשומה צומת).
2. רשומה עבור רשימה: בעלת שדה head עבור צומת תחילת הרשימה, ולעיתים גם שדה tail עבור צומת סוף הרשימה.

תחביר

- רשימה מקושרת היא בעצם מבנה דינמי המתבסס על מצביעים לרשומות, כך שכל צומת (רשומה) מאפשרת גישה לצומת הבא.

כדי לעבור על איברי רשימה, יש לעבור איבר-איבר ע"י מצביע שיקודם בכל איטרציה לצומת הבא, כאשר סוף הרשימה יצביע ל-NULL.

- **כל איבר ברשימה (=צומת) הוא מצביע מסוג רשומה-צומת**, המכיל שדה מצביע של רשומה-צומת. לכן:

1. כדי לייצר צומת חדש יש להקצות זיכרון למצביע מטיפוס רשומה-צומת זיכרון בגודל של רשומה-צומת 1, ולבדוק הקצאה (NULL).
2. כדי לפנות ממצביע של רשומה לשדה בה נשתמש באופרטור חץ (למשל בקידום מצביע "נשרשר" curr = curr->next->next->...).

סוגי רשימות מקושרות:

- **רשימה דו-כיוונית:** לכל צומת 2 שדות מצביע-צומת next ו-prev (לצומת שלפניה ושאחריה), כאשר: head->prev=NULL.

הערה: יש לשים לב עבור הכנסת איבר לאמצע רשימה- לעדכן את כל ה-prev וה-next של 3 צמתים (curr->prev->next=curr).

- **רשימה מעגלית:** האיבר האחרון ברשימה מצביע לאיבר הראשון ברשימה (בעצם אין ברשימה כזו חשיבות רבה למי נקבע -head).
- **רשימה עם dummy head:** איבר דמה, שתמיד יהיה קיים ברשימה אחרי head, בעזרתו נוכל לבצע שינויים בתחילת הרשימה בקלות.

ריצה על רשימה

- **אלגוריתם פונק' עזר**: ניתן לרוץ על איברי רשימה ע"י מצביע באופן איטרטיבי או רקורסיבי.

כל איבר ברשימה הוא רשומה-צומת בעלת שדה מצביע לצומת הבא, עד למצביע NULL (מהווה תנאי עצירה לאיטרציה/ רקורסיה).

- **שחרור זיכרון:** יש לגשת לכל איבר ברשימה ו"למחוק אותו" ע"י שחרור הזיכרון שהוקצה עבורו (מ-head עד NULL).
- **עדכון/ שינוי רשימה עבור צומת מסוים:**

כדי לשנות/ לעדכן רשימה נצטרך לעבוד על המקור, לכן נעביר את הרשימה כמצביע רשומה. בכל פעולה נבדוק עם הרשימה ריקה.

בכל פעולה נעדכן את המצביעים בהתאם לסוג הרשימה וסוג הפעולה (החלפת מיקומי איברים, הוספת איבר חדש, מחיקת איבר).

נשים לב למיקום הצומת בו יש לבצע שינוי ולשים לב לעדכון המצביעים בהתאם:

1. **תחילת רשימה:** יש לעדכן את ה-head החדש, ואת ה-next של החדש להיות ה-head הקודם (או אחרת בהתאם למבוקש).
2. **סוף רשימה:** יש לעדכן את ה-tail הנוכחי להצביע על ה-tail החדש, לעדכן את ה- tail לחדש ואת ה-next שלו להיות NULL.
3. **אמצע רשימה:** יש להגדיר מצביע זמני (או יותר) שישמור את ה-*next ולעדכן בהתאם את המצביעים של כל צומת מחדש.

רשימות מקושרות

מבנה נתונים דינמי, השומר נתונים בזיכרון בלי רצף

תחביר רשימה מקושרת (ראש + זנב):

```
typedef struct listNode {
    int data;
    struct listNode* next;
} ListNode;

typedef struct list {
    ListNode* head;
    ListNode* tail;
} List;
```

האם הרשימה ריקה:

```
bool isEmptyList(List* lst) {
    return (lst->head == NULL);
}
```

יצירת רשימה ריקה:

```
void makeEmptyList(List* lst) {
    lst->head = lst->tail = NULL;
}
```

שחרור רשימה מהזיכרון:

```
void freeList(List lst) {
    ListNode* next, *curr = lst.head;
    while (curr != NULL) {
        next = curr->next;
        free(curr);
        curr = next;
    }
}
```

הדפסת רשימה:

```
void printList(List lst) {
   LNODE* curr = lst->head;
   while (curr != NULL) {
       printf("%d ", curr->data);
       curr = curr->next;
   }
   printf("\n");
}
```

יצירת צומת חדש

```
ListNode* createListNode(type data,ListNode* next) {
    ListNode* node = (ListNode*)malloc(sizeof(ListNode));
    node->data = data;
    node->next = next;
    return node;
}
```

מחיקת צומת ברשימה:

```
void removeNode(ListNode* after) {
    ListNode* toDelete;

    toDelete = after->next;
    after->next = toDelete->next;
    free(toDelete);
}
```

הוספת צומת קיים לתחילת הרשימה (ראש חדש):

```
void insertNodeToStart(List* lst, ListNode* newHead){
     if (isEmptyList(lst))
        lst->head = lst->tail = newHead;
     else {
        newHead->next = lst->head;
        lst->head = newHead;
     }
}
```

החזרת הצומת ה-i ברשימה:

```
ListNode* getListNode(List* lst, int i) {
    ListNode* curr = lst->head;

    while ((curr != NULL) && (i > 0)) {
        curr = curr->next;
        --i;
    }
    return curr;
}
```

הוספת צומת קיים לסוף הרשימה (זנב חדש):

```
void insertNodeToEnd (List* lst, ListNode* newTail){
     if (isEmptyList(lst))
        lst->head = lst->tail = newTail;
     else {
        lst->tail->next = newTail;
        lst->tail = newTail;
     }
}
```

הוספת צומת באמצע רשימה:

```
void insertNodeToList(ListNode* after, ListNode* newNode){
    newNode->next = after->next;
    after->next = newNode;
}
```

05 – עצים בינאריים

עץ: מבנה נתונים המורכב מצמתים וקשתות מכוונות (חצים). לכל צומת מצביעה קשת 1 בדיוק, מלבד השורש (לא מכוונות אליו קשתות).

עץ בינארי: עץ שבו לכל צומת יש לכל היותר 2 בנים (ימין ושמאל).

מושגים

- **הורה**: צומת אשר ממנו יוצאת קשת מכוונת לצומת אחר, הנקרא בן (לאותו הורה). צומת ללא בנים יקרא עלה.
- **רמה/ עומק**: המרחק של צומת מהשורש (שורש מתחיל מרמה 0).
- **מסלול** הוא רצף של צמתים (או קשתות) החל מהשורש. אורך המסלול הוא מס' הקשתות, וגובה עץ הוא המסלול הארוך ביותר.
- **תת עץ:** עץ הנפרש מצומת מסוים. עץ ייקרא אב קדמון של עץ אחר אם קיים מסלול העובר דרכו (הצומת שייך לעץ אב קדמון).

תחביר

- מבנה נתונים המורכב מ-2 רשומות (זיכרון לא שמור ב'רצף' כמו רשימות מקושרות):
  1. **עץ**: רשומה בעלת שדה מצביע מסוג רשומה לצומת בעץ.
  2. **צומת עץ**: רשומה בעלת שדות ערכים, ו-2 שדות מסוג מצביע רשומה- לצמתי העץ הבאים אם קיימים (בן ימני ובן שמאלי).

ריצה על עץ בינארי

- **פונק'**: רוב הפונק' בעצים יהיו מעטפת (מקבלת עץ) לפונק' עזר שתרוץ רקורסיבית על צמתי העץ, החל מהשורש ועד לעלה (NULL).
- **הערה**: יש להבדיל בין עץ "ריק" (שאינו מוגדר, שורש NULL) לבין עץ שמכיל רק שורש (שגובהו 0). לכל אופציה נגדיר תנאי מתאים.
- **יצירת צומת**: כל צומת היא מצביע מטיפוס רשומה. לכן נקצה זיכרון דינמי, נעדכן שדות ומצביעים לבנים, כאשר עלה מצביע ל-NULL.
- **הדפסת ערכי עץ**: קיימות 3 אפשרויות לסדר ההדפסה רקורסיבית (D= שורש): DLR, LDR, LRD (לפי מיקום הקריאה להדפיס ברקורסיה).
- **שחרור זיכרון**: רקורסיבית נשחרר כל צומת ע"י קריאה לתתי העצים שמאל וימין עד לעלה (NULL), ונשחרר שורש.

עץ חיפוש בינארי

עץ בינארי כאשר בכל צומת הבן הימני וכל צאצאיו גדולים מערך הצומת, והשמאלי וכל צאצאיו קטנים מערך הצומת.

כלומר, בכל תת עץ נשמרת התכונה החל משורש העץ (כל הבנים בתת עץ יהיו קטנים/ גדולים מהשורש בהתאם לכיוון).

למשל: שורש = 8, בן הימני = 10, וממנו 2 בנים נוספים: הבן הימני יהיה גדול מ-10 (8<10), והבן השמאלי יהיה קטן מ-10 **וגם** גדול מ-8.

עצים בינאריים

תחביר (רשומת עץ):

```
typedef struct treeNode {
    <type> data;
    struct treeNode* left;
    struct treeNode* right;
} TreeNode;

typedef struct tree {
    TreeNode* root;
} Tree;
```

שחרור זיכרון של עץ:

```
void freeTree(Tree tr) {
    freeTreeRec(tr.root);
}

void freeTreeRec(TreeNode* root) {
    if (root == NULL)
        return;
    else {
        freeTreeRec(root->left);
        freeTreeRec(root->right);
        free(root);
    }
}
```

הדפסת ערכי עץ (D שורש, R/L צידי תת עץ):

```
void printTree(Tree t) {
    printRec(t.root);
}

void printRec(TreeNode* tn) {
    if (tn == NULL)
        return;

  //החלפת השורות הבאות תשנה סדר הדפסה
D     printf("%[redacted] ", tn->data);
L     printRec(tn->left);
R     printRec(tn->right);
}
```

יצירת צומת:

```
TreeNode* createNewTreeNode(int data, TreeNode* left, TreeNode* right) {
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    node->data = data;
    node->left = left;
    node->right = right;
    return node;
}
```

ספירת צמתים בעץ:

```
int numNodes(Tree tr) {
    return numNodesRec(tr.root);
}

int numNodesRec(TreeNode* root) {
    if (root == NULL)
        return 0;
    else {
        int numLeft, numRight;
        numLeft = numNodesRec(root->left);
        numRight = numNodesRec(root->right);
    return numLeft + numRight + 1;
   }
}
```

ספירת עלים בצומת:

```
int numLeaves(Tree tr) {
    return numLeavesRec(tr.root);
}

int numLeavesRec(TreeNode* root) {
    if (root == NULL)
        return 0;
    else if (root->left == NULL && root->right == NULL)
        return 1;
    else {
        int leavesLeft, leavesRight;
        leavesLeft = numLeavesRec(root->left);
        leavesRight = numLeavesRec(root->right);
        return leavesLeft + leavesRight + 1;
    }
}
```

06 – קדם מעבד, קומפילציה

חלוקה לקבצים

- חלוקת פרויקט ליחידות (מודולים) שניתן לאפיין, לתכנת ולבדוק באופן עצמאי. כל יחידה יכולה להכיל הגדרות, קבועים ופונקציות.
- מודול מורכב מ-2 קבצים:
  1. **ממשק** file.h **(**header**):** מכיל את ההגדרות (רשומות, קבועים וכו'), חתימות הפונקציות, וספריות שנעשה בהן שימוש.
  2. **מימוש** file.c **():** מכיל את מימוש הפונקציות השונות. יש לכלול בתחילת הקובץ את קובץ הממשק "#include "file.h.
- בחלוקה לקבצים יהיה קובץ מימוש הקוד main.c, הכולל את כל הספריות הרלוונטיות וקבצי הממשק הקיימים (headers).

תהליך קומפילציה

1. **Editor:** התוכנית נכתבת בעורך טקסטואלי. זהו בעצם המסך בו אנחנו כותבים את הקוד (הטקסט).
2. **Preprocessor:** עיבוד ראשוני של קדם מהדר. מבצע את כל פקודות # בתוכנית (include- הכללת קבצים, define- החלפת משתנים).
3. **Compiler:** בדיקת תקינות התחביר בפונק'. הקומפיילר (מהדר) מייצר קובץ obj לכל קובץ c בשפת מכונה ושומר על הדיסק.
4. **Linker:** מקשר בין קריאה לפונק' והמימוש שלה. מייצר קובץ exe שהוא איחוד כל קבצי ה-obj שנוצרו, הנשמר בדיסק ומוכן לשימוש.
5. **Loader:** בזמן ההרצה התוכנית נטענת מהדיסק לזיכרון הראשי.

הערה: אם קיימת שגיאה בשלב הקומפילציה אין התקדמות לשלב הלינקר (תיתכן שגיאת לינקר ותקינות בשלב הקומפילציה- נקבל אזהרה).

מאקרו - פקודת #define:

- ניתן להגדירו בכל שורה בתוכנית (מחוץ או בתוך ה-main, בכל פונק' בתוכנית), והוא מוכר לכל אורך התוכנית (כל עוד לא בוצע #undef).
- **הגדרת קבוע #define id <token>:** מתבצעת החלפה של כל מופע בשלב פרה-קומפילציה (מלבד מופע עם מירכאות).
- **מאקרו עם ארגומנטים #define id(arg,..) <token>:** מתבצעת החלפה של הארגומנטים בהתאם ל"תבנית" שהוגדרה (token).
- **מאקרו** VS **פונק':** פיענוח מאקרו מתבצע לפני קריאת פונק', לא ניתן לדבאג מאקרו, הוא פחות מובן, והוא מנפתח את קובץ ה-EXE.
- **אופרטורים וארגומנטים במאקרו:** שימוש בהם יכול להוביל לערך שלא נרצה. עדיף להשתמש במאקרו להחלפת תבניות פשוטות.

דוגמאות לשגיאות תהליך קומפילציה

1. קומפילציה: קריאה או שימוש בפרמטר שלא הוגדר בפונק'.
2. לינקר: קריאה לפונק' שלא מוגדרת בתוכנית (למשל קריאה לשם שגוי לפונק'- השם אינו מוכר בתוכנית).

דוגמאות שגיאות זמן ריצה

- לא בוצע שחרור של זיכרון.
- גישה לכתובת שערך הזיכרון בה הוא NULL או מזובל, גישה לזיכרון ששוחרר או עודכן מחדש לכתובת חדשה.

למשל: realloc בפונק' פנימית לפרמטר מצביע יכולה לשנות את כתובתו. לכן כאשר ננסה לגשת שוב למצביע בסיומה תיתכן שגיאה.

- ניסיון לגשת לזיכרון של מצביע NULL. גם אם בוצעה לו הקצאה דינמית , אין אפשרות לעדכן את הערך ישירות (פנייה לשדה *).

יש לבצע השמה של המצביע למצביע אחר, כדי לעדכן את הערך שנרצה לשמור בו (בהתאם לטיפוס המצביע).

07 – פרמטרים ל-main

שימוש: הפעלת תוכנית עם ערכי התחלה שונים שאינם מהמשתמש, למשל: העברת פרמטרים מתוכניות אחרות או ממערכת ההפעלה.

פורמט הצהרה (קבוע): (main(int argc , char*argv[] <>

- Argv- מערך מחרוזות המכיל את הפרמטרים שהתקבלו, אשר יכולים להיות מכל סוג (תו/ מס'), ומשוחרר ע"י מערכת ההפעלה.
  נזכור, שכאשר בתוכנית נשתמש בפרמטרים מסוימים ייתכן ונצטרך לבצע המרה בהתאם (למשל: ממחרוזת למספר).
- Argc- מס' הפרמטרים שהתקבלו. ערכו תמיד לפחות 1, כאשר הראשון (argv[0]) הוא שם התוכנית.

הערות:

- נזכור ש-argv הוא מערך מחרוזות, לכן נשים לב שאנחנו משתמשים בערכים בהתאם לטיפוס *char (אופרטורים, העתקות, המרה).

08 – קבצים

- **שימוש**: שמירת מידע ב-hard disk, קריאת נתונים מקובץ מוכן מראש, כתיבת נתונים לקובץ, שמירת נתונים לשימוש לתוכניות אחרות.
- **קובץ טקסט**: ניתן לקרוא בקלות את המידע ממנו. קובץ בינארי מצריך המרה לצורך קריאה, הוא יותר חסכוני בזמן ובמקום בזיכרון.
- **Stdout**- קובץ טקסט של הדפסות למסך. הקובץ אליו כותבת ()printf ולאחר מכן מדפיסה ממנו.
- **Buffer**- אחסון זמני בזיכרון של נתוני מידע (זהו בעצם מערך תווים).
- כתיבת משתנים מטיפוס int נכתבת בסדר הפוך של ביטים (נושא הבא).
- **סמן**: בכל קריאה של תו קיים סמן שזז עד לתו האחרון שנקרא. כלומר בכל קריאה של n תווים יש קידום של הסמן לתו ה-(n+1) בקובץ.

מבנה עבודה כללי עם קבצים:

- יש להגדיר משתנה מטיפוס קובץ: *FILE, המוגדר בספריית stdio.h.
- לפני תחילת העבודה, יש לפתוח אותו לקריאה/ כתיבה כרצוננו כך (**השמה**): FILE* f = fopen(<file-name>, <open-mode>).
- יש לבדוק האם הקובץ נפתח בהצלחה (כמו בדיקת הקצאה), ולסגור קובץ בסיום העבודה עליו ע"י fclose(f) (כמו שחרור זיכרון).

הערה: אם הקובץ נמצא בתיקיית הפרויקט נפתח לפי שמו: ("name.txt", "")", אחרת יש לציין כתובת מיקום הקובץ + שמו + סיומת.

- Assert(void* ptr)- תבדוק האם NULL, ואם כן תדפיס שגיאה ותסיים תוכנית עבור assert(f) (כלומר אם הקובץ לא נפתח כראוי).

סוגי פתיחת קובץ: (עבור קובץ בינארי יש לציין b מימין (למשל: "rb"), ועבור קובץ טקסט יש לציין מימין t או לא לציין אות נוספת בכלל)

1. **קריאה**: "r"- הקובץ חייב להיות קיים, אחרת יוחזר NULL.
2. **כתיבה**: "w"- במידה ויש נתונים בקובץ, הם יידרסו בעת הכתיבה. אם קובץ לא קיים, יפתח חדש.
3. **כתיבה**: "a"- כתיבה לסוף הקובץ (לא דורס נתונים). אם קובץ לא קיים, יפתח חדש.
4. **פתיחה לקריאה וכתיבה**: נוסיף + מימין. עבור "r+" אם לא קיים יוחזר NULL, עבור w+/a+ יפתחו לעדכון בהתאם למצב המבוקש.

פונק' ספריית stdio.h עבור קבצים:

1. Int fclose(FILE* f)- מקבלת מצביע לקובץ. מחזירה 0 אם הצליחה לסגור אותו, אחרת מחזירה EOF (=תו קבוע שערכו 1-).
2. Int fcloseall()- סוגרת את כל הקבצים שנפתחו, ומחזירה את מס' הקבצים שסגרה.
3. Int fprintf(FILE* f, char* format, arg1, …) – כתיבה לקובץ.
4. Int fputs(char* s, FILE* f)- כותבת את המחרוזת s לתוך קובץ **ללא ירידת שורה**.
5. Int fputsc(char c, FILE* f)- כותבת את התו c לתוך קובץ.
6. Int fscanf(FILE* f, char* format, arg1,…)- קריאה מקובץ.
7. Char* fgets(char* dst, int n, FILE* f)- קוראת n תווים מקובץ או עד \n. אם הצליחה תוחזר המחרוזת שנקלטה, אחרת יוחזר NULL.
8. Int fgets(FILE* f)- קוראת תו אחד מהקובץ. אם הצליחה, יוחזר הערך האקסי של התו, אחרת יוחזר EOF.
9. Int feof(FILE* f)- מקבלת קובץ **פתוח** ומחזירה EOF אם קריאה נכשלה, אחרת תחזיר 0. שימושית כתנאי עצירה: while(feof(f)==0).
10. Int fwrite(const void* src, int size, int count, FILE* f)- כתיבת count איברים ממערך src, שכל איבר בגודל size, לתוך קובץ f.
11. Int fread(void* dst, int size, int count, FILE f)- קריאת count איברים, כאשר גודל כל איבר size, מקובץ f אל תוך מערך dst.
12. Int fseek(FILE* f, long offset, int origin)- הוזזת הסמן מ-origin שנבחר קדימה offset בתים (תו = בית):

    SEEK_SET- מתחילת הקובץ, SEEK_END- מסוף הקובץ, SEEK_CUR- מהמיקום הנוכחי. אם הצליחה לעדכן סמן יוחזר 0.
13. Long ftell(FILE* f)- מחזירה את מיקום הסמן בקובץ.

14. Void rewind(FILE* f) - מחזירה את הסמן לתחילת הקובץ. שקול ל: fseek(f,0,SEEK_SET).
15. Int fflush(FILE* f) - מרוקנת את ה-buffer של הקובץ, בעצם מבצעת עדכון של מה שנכתב עד כה בקובץ.

השורה fflush() תבצע ריקון (עדכון) של כל ה-buffer-ים הפתוחים (קבצים, _std וכו').

שגיאות אפשריות הקשורות לקבצים:

1. ניסיון לפתיחה לכתיבה של קובץ המוגדר לקריאה בלבד.
2. ניסיון לפתיחת קובץ ע"י "r" שאינו קיים.
3. ניסיון לסגירת קובץ לא פתוח.
4. ניסיון לקריאת מקובץ בסוף, לאחר שנגמרו התווים (בפונק' מסוימות השגיאה תופיע כהחזר של התו EOF או NULL).

הערות לקריאה/כתיבה - קובץ בינארי:

1. **מצביעים**: בכתיבה לקובץ בינארי מתבצעת המרה של הערך השמור בכתובת המשתנה המבוקש (לא נכתוב כתובות- מצביעים).
2. ניתן לכתוב לקובץ בינארי ע"י **fwrite בלבד**, ורצוי להשתמש ב-fread לקריאה. בקובץ בינארי אנחנו קוראים לפי בתים ולא תווים.
3. תמיד נצטרך לדעת כמה מה **הטיפוס** של האיברים שאנחנו קוראים/כותבים, **וכמה** כאלו יש.
4. **מערך:** כתיבה ממערך: fwrite(arr, sizeOf(ArrType), sizeOfArr, f), קריאה לתוך מערך: fread(arr, sizeOf(ArrType), sizeOfArr, f).
5. **מחרוזת:** בקריאת מחרוזת בודדת מקובץ בינארי ע"י **fread** יש לזכור **לשים תו סיום '\0'** בעצמנו.
6. **כתיבת רשומות המכילות מצביעים**: יש לכתוב שדה-שדה מהרשומה לקובץ כדי לרשום את הערך המתאים (ולא בטעות את כתובתו).
7. **מערך היסט:** מערך אינדקסים לחישוב מיקום של כל אובייקט שונה (למשל שמות, רשימות). שימושי לגישה ישירה לאובייקטים בקובץ.

09 – ביטים

- אחסון מידע בזיכרון בצורה חסכונית ודחוסה. כל בית 1 = 8 ביטים, כאשר כל ביט מקבל ערך 1/0, ללא גישה ישירה אליו (לביט).
- פעולות ביטים מוגדרות עבור **טיפוסים שלמים בלבד**: char, int, long, short, מסוג unsigned (פעולות על signed לא זהות לכל מכונה).
- **גודל בתים למשתנים**: long int/ int = 4, short int = 2, char = 1 (unsigned משמעותו טווח ערכים אי שליליים בלבד למשתנה).
- הביט הכי ימני נקרא "הפחות משמעותי" / "least significant", בעל הערך הקטן ביותר (הכי שמאלי נקרא most significant).
- **קבצים וביטים:** בקבצים משתנה int ייכתב בסדר הפוך בביטים. כלומר, כל המרה של hex "נכתבת" הפוך בסדר הכתובות.
- **הגדרת משתנה לבית:** ניתן לבצע השמה למשתנה BYTE לפי הערך העשרוני של הבית הרצוי, או לפי בסיס HEX (כל ספרה = 4 ביטים).

אופרטורי ביטים (BitWise):

לכל X,Y משתנים הביטוי: X (אופרטור) Y = מחזיר משתנה חדש Z, כאשר כל ביט שלו נקבע בהתאם לאופרטור והביטים של X ו-Y.

בעצם, לכל ביט X_i של X ולכל ביט Y_i של Y, יוחזר ביט Z_i עבור המשתנה החדש בצורה הבאה:

AND: $(X\&Y) = \forall i: Z_i \begin{pmatrix} 1 & X_i = Y_i = 1 \\ 0 & \text{אחרת} \end{pmatrix}$. **תכונת "כיבוי ביט":** $\begin{cases} ((bit)\ \&\ 1) = (bit) \\ ((bit)\ \&\ 0) = 0 \end{cases}$

OR: $(X|Y) = \forall i: Z_i \begin{pmatrix} 0 & X_i = Y_i = 0 \\ 1 & \text{אחרת} \end{pmatrix}$. **תכונת "הדלקת ביט":** $\begin{cases} ((bit)\ |\ 0) = (bit) \\ ((bit)\ |\ 1) = 1 \end{cases}$

XOR: $(X\hat{}Y) = \forall i: Z_i \begin{pmatrix} 0 & X_i = Y_i \\ 1 & \text{אחרת} \end{pmatrix}$.

NOT: $(\sim X) = \forall i: Z_i \begin{pmatrix} 1 & X_i = 0 \\ 0 & X_i = 1 \end{pmatrix}$.

פעולות שיפט: עבור משתנה X ו-k שלם, הפעולה [X (shift) k]- תבצע הוזזה של k ביטים בהתאם לכיוון השיפט שנבחר.

- **Shift left <<:** unsigned: "דוחף" שמאלה k ביטים ("עפים"), ומוסיף k אפסים מימין.
- **Shift right >>:** unsigned: "דוחף" ימינה k ביטים ("עפים"), ומוסיף k אפסים משמאל (signed ישכפל k פעמים את השמאלי ביותר).

שליפת מידע מביט i:

- הגדרת מסכה-משתנה unsigned char, בעזרתו נבצע "ריצה" על בית-בית ועל ביט-ביט של המשתנה, ונבדוק האם "דולק"/"כבוי".
- **האם ביט i דולק:**

נשתמש בתכונת "הכיבוי" של אופרטור & ← if(byte&mask) (אם true אז הביט דלוק =1).

```
int isBitSet(BYTE b, int i) {
        BYTE mask = 1 << i;
        return (b & mask);
}
```

- **הדלקת ביט i:**

נשתמש בתכונת "ההדלקה" של אופרטור I ← הפעולה: (byte|mask) תדליק את הביט **אם** כבוי.

```
BYTE setBit(BYTE b, int i) {
        BYTE mask = 1 << i;
        return (b | mask);
}
```

- **עדכון k ביטים מתוך 8:**

נכבה את כל k הביטים: נגדיר mask1 בה k הביטים שנרצה לשנות הם 1 והיתר 0, ונכבה אותם ע"י X&(~mask).

נדליק את k הביטים בהתאם: נגדיר mask2 בו k הביטים מעודכנים למידע החדש (8-k האחרים 0), ונדליק ע"י: X|mask2.

- **ריצה על ביטים בבית 1:**

נתייחס לבית כמערך ביטים: For(int i = 7 ; i >= 0 ; i--) ← הגדרת mask = 1 << i ← ביצוע הפעולה על הביט במקום ה-i.

10 – מצביעים לפונקציות גנריות

הגדרת מצביע לפונקציה:

- מצביע לפונק' הוא חתימה של פונק'. הוא מכיל כתובת לזיכרון, ומצביע לטיפוס ספציפי- ערך ההחזר של הפונק' עליה הוא מצביע.
- יש **להכריז** על מצביע לפונק' f כמשתנה בתוך הפונק' כך: <type-f-returned-value> (*<Ptr-Name>)(<type-param1>, …).
- מצביע לפונק' **ניתן להעביר כפרמטר לפונק'**, כאשר בשימוש בו הוא בעצם מתבצעת קריאה לפונק' שהוא מצביע עליה.
- עבור המצביע: <type> (*funcP)(par1, par2), נבצע **השמה** באחת הדרכים: funcP = functionName / funcP = &functionName.
- **הגדרת מערך של מצביעים לפונק'**: יש להגדיר עבור מצביעים לפונק' שחתימתן זהה (ערך החזר, ופרמטרים שמקבלת).

  עבור N פונק' שמחזירות טיפוס t1 ומקבלות k פרמטרים מטיפוס tK נגדיר: t1(*ptrName[N])(t1,…,tK) = { f1, … , fN }.
- **Typefef:** typedef <returnedValue> (*pName)(<arg1>, …)- כך נוכל להגדיר מס' מצביעים לפונק' עם חתימה זהה ע"י pName.

  למשל: typedef int (*pStr)(char*, char*) ← נגדיר ; pStr f1, f2, … / חתימת פונק' עם משתנה pStr ושליחת strcmp.
- **מצביע *Void-** יכול להכיל כתובת של משתנה מכל טיפוס. לא ניתן להשתמש בחשבון מצביעים עליו, או לגשת לערכו ע"י אופרטור *.
- **המרת *char ל- *int:** מתבצע עדכון **בסדר הפוך** של הבתים- הבית הכי ימני של *int מתעדכן מהבית הכי שמאלי של *char והלאה.

מימוש גנרי לפונק' – שימוש במשתנה *void (חסר טיפוס):

- בדר"כ נצטרך להשתמש במשתנה מטיפוס (unsigned char), לכן נגדיר טיפוס חדש לצורך נוחות: typedef (unsigned char) BYTE.
- על הפונק' לקבל בנוסף למשתנה החסר טיפוס- void* ptr, משתנה נוסף- int elemSize, עבור גודל הטיפוס בבתים.
- כדי לבצע פעולה על המשתנה *void יש להגדיר משתנה BYTE* x = (BYTE*)ptr, בעזרתו נבצע פעולות עליו.
- **ריצה על מערך מטיפוס לא ידוע *void:**

  עלינו לקבל את כתובת תחילת המערך (void*), גודל טיפוס בבתים (elemSize=sizeOf(type)), ומס' האיברים במערך (ArrSize).

  כדי להתקדם במערך **ניעזר בשקילות**: arr[i] ≡ (BYTE*)arr + i * elemSize)* (רצוי להגדיר משתנה BYTE* p = (BYTE*)arr ולרוץ ע"י).
- **אופרטור []** מחשב את מס' הבתים שיש לעבור מכתובת תחילת מערך עד לאיבר ה-i, לכן לא **עבור מערך *void לא נוכל להשתמש בו**.

הפונק' memcpy:

Void* memcpy(void* dst, cont void* src, int num)- מעתיקה num **בתים** החל מכתובת קבועה src אל כתובת dst (ערך מוחזר).

- העתקת מערכים מטיפוס type בגודל SIZE: memcpy(arr2, arr1, size*sizeOf(type)).
- Swap(void* a, void* b, int elemiSize)- הגדרת tmp מתאים, ושימוש בפונק' 3 פעמים לביצוע העתקות לפי elemSize.

הפונק' qsort: יעילות n*log(n)

Void qsort(void* base, size_t elemNum, size_t elemSize, int(*comparator)(cont void*, const void*)

פונק' המבצעת מיון של מערך base לפי קריטריון השוואה מסוים שנקבע ע"י הפונק' comparator (מצביע לפונק' בחתימה), כאשר ערך ההחזר שלה מוגדר לכל מקרה כך: אם 1st elem > 2nd elem- מס' חיובי, אם 1st elem < 2nd elem- מס' שלילי, אם שווים- 0.

הפונק' bsearch: יעילות log(n)

Void* bsearch(const void* key, cont void* base, size_t eNum, size_t eSize, , int(*comparator)(cont void*, const void*)

מבצעת חיפוש עפ"י קריטריון מסוים במערך לפיו הוא **ממוין**, של הערך של key. אם מצאה את האיבר תוחזר כתובתו, אחרת יוחזר NULL.

הפונק' comparator מוגדרת עם ערך החזר כפי שמוגדר ב-qsort.