סיכום לשפת C

הקדמה

יעילות ופשטות:

- שליטה הדוקה בשימוש בזיכרון וב-CPU
- ניתן להבין ולהשתמש **בכל** השפה (שפה מצומצמת יחסית)
- מתעסקת רק באובייקטים שמחשבים משתמשים בהם (למשל יש משתנים אבל אין רשימות)

בדיקת $Types$:

- הבדיקה היא באחריות המתכנת "המתכנת יודע מה הוא עושה",
  לכן יש אחריות גדולה על המתכנת.

חסרונות:

- משימות שלוקח הרבה זמן לעשות ב-C כמו למשל $Parsing$ של טקסט.
- מכיוון שאנחנו עובדים עם שפה "נמוכה" עם גישה לזיכרון וניהול שלו,
  יש הרבה מקום לביצוע תקלות (מצביעים למיקומי זיכרון לא קיימים וכו')
- יכולות להיווצר בעיות בהרצת הקוד על מחשב אחר.

שימושים נפוצים לשפה:

- מערכות הפעלה כמו למשל $Linux$
- דרייברים, מעבדים, טלוויזיות
- במקרים בהם יש מגבלות על הזיכרון וה-CPU

ממה צריך להיזהר:

- C לא בודק שגיאות זמן ריצה כמו למשל חריגה מגבולות מערך, מצביעים לא חוקיים ועוד.
- אין ניהול זיכרון, זו אחריות של המתכנת להקצות ולשחרר זיכרון.

$C++$:

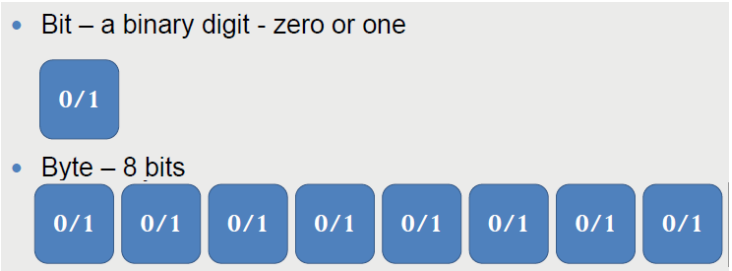
הרחבת $Object\ Oriented$ של C, יש בה מחלקות ומתודות.

בנוסף, יש דרך יותר יעילה לניהול זיכרון (או לא לנהל בכלל)

ביט/בית:

ביט – ספרה בינארית (0 או 1), יחידת הנתונים הקטנה ביותר שבה משתמש המחשב.

בית – יחידת נתונים המכילה 8 ביטים.



סינטקס בסיסי

הערות:

- שורה שמתחילה ב-// היא שורת הערה (רק השורה הנוכחית)
- כל השורות שנמצאות בין */ ל-/* הן שורות הערה.

יבוא ספריה סטנדרטית:

$\#include < stdio.h >$ - ייבוא ספרייה שמכילה כלים של $Input/Output/Print$

פונקציית $Main$:

בכל תוכנית של C יש פונקציית $main$, למשל:

```
int main() {
        printf("Hello class!/n");
        return 0;
}
```

- במקרה זה התוכנית מחזירה משתנה מסוג int.
- יכלנו להשתמש ב-$printf$ כי ייבאנו את הספרייה $stdio.h$.

קימפול והרצה:

- קוד ב-C נשמר בקובץ עם סיומת c (למשל $hello.c$)
- כדי לקמפל את הקובץ, ננווט לתיקייה הרצויה דרך הטרמינל ונרשום $gcc\ hello.c$
- מכיוון שלא ציינו שם לקובץ המקומפל, ברירת המחדל הינו $a.out$
- כדי לקבוע שם לקובץ המקומפל, נוכל להשתמש ב-$gcc\ hello.c\ \ -o\ \ hello$

בקורס נדרש מאתנו לקמפל באופן הבא:

$$gcc\ \ -std = c99\ \ -Wall\ \ hello.c\ \ -o\ \ hello$$

($-std = c99$ כדי לציין שמדובר בסטנדרט $c99$, $-Wall$ כדי לאפשר שגיאות קומפיילר)

דיבוג:

- אם עובדים ב-$cLion$, יש לו דיבאגר מובנה.
- אם עובדים ב-$Command\ Line$ יש דיבאגר בשם gdb,

כדי להשתמש בו נוסיף בזמן הקמפול $-g$ כלומר נקמפל בעזרת הקוד הבא:

$$gcc\ \ -g\ \ -std = c99\ \ -Wall\ \ hello.c\ \ -o\ \ hello$$

משתנים

ב-C, בדומה ל-$Java$, נחליט על טיפוס של משתנים בזמן קומפילציה ($int, char, \dots$)
ניתן לאתחל את המשתנה בזמן ההכרזה אך לא חייב. במקרה שלא נאתחל,
אין ערך ברירת מחדל והמשתנה יכיל "שאריות זיכרון",
לכן נוודא שהשמנו לו ערך בהמשך לפני שימוש בו.

<u>סוגי טיפוסים</u>:

$char\ c = 'A'$; - טיפוס מספרי שלרוב מיועד לאחסון תו בודד (פירוט בהערה).

$short\ s = 0$; - מספרים שלמים קטנים.

$int\ x = 1$; - מספרים שלמים.

$long\ y = 9$; - מספרים שלמים גדולים.

$float\ x = 0.0$; - מספרים לא שלמים, לרוב קטן יותר מ-$double$.

$double\ y = 1.0$; - מספרים לא שלמים.

<u>הערה 1</u>: ב-C הטיפוס $char$ הוא טיפוס מספרי בדומה ל-int. הוא מכיל מספר.
בזמן קימפול, כל עוד אנחנו מתייחסים אליו בתור טיפוס $char$,
מתבצעת המרה של המספר לתו לפי טבלת $ASCII$, אך בפועל מדובר בטיפוס מספרי.
<u>הערה 2</u>: כחלק ממוסכמות השפה, גודל משתנה $char$ הינו בית אחד,
אך על כל שאר המשתנים לא נוכל להניח הנחות מכיוון שזה תלוי מחשב/קומפיילר.

<u>טיפוסים חיוביים ($Unsigned$)</u>:

$unsigned\ char\ c = '\ A'$; $unsigned\ short\ s = 0$;

$unsigned\ int\ x = 1$; $unsigned\ long\ y = 9$;

טיפוסים מספריים ב-C מכילים bit אחד שמציין האם המספר השמור בהם חיובי או שלילי.
במידה ונרצה רק מספרים חיוביים, נוכל לוותר על ה-bit הזה לטובת הגדלת טווח המספרים.
דוגמה שניתנה בכיתה: אם נבחר למשל ב-$unsigned\ char$,
המשתנה יכיל רק מספרים חיוביים אך בתמורה ניתן יהיה לשמור בו מספרים גדולים יותר.
טיפוס $char$ ($signed$) יכול להכיל מספרים בין מינוס 127 ל-128,
לעומת זאת משתנה מסוג $unsigned\ char$ יכול להכיל מספרים בין 0 ל-255.

<u>אתחול משתנים</u>:

מעבר לאתחול הרגיל של משתנה כמו למשל $int\ x = 1$;,

- ניתן לאתחל מספר משתנים באותה שורה באופן הבא: $char\ a = 'A', b = 'B'$;
- ניתן לאתחל מספר משתנים עם אותו הערך: $int\ x = y = 1$;
  בפועל, y מקבל את הערך 1 ולאחר מכן x מקבל את הערך של y (1).

הערות:

- אם נצהיר על משתנים בתוך $Scope$ מסוים הוא יהיה רלוונטי רק לסקופ הספציפי.
- אם נצהיר עליו בתחילת המסמך, מחוץ ל-$main$, הוא יהיה משתנה גלובלי.
  (ללא סינטקס מיוחד, פשוט $int\ x = 0;$ כרגיל)
- באופן כללי, הצהרה/השמה של משתנים בסקופים שונים דומה להתנהגות של $Java$.

משתנים בוליאניים:

משתנים בוליאניים **אינם** קיימים ב-C! נשתמש ב-$char/int$ במקום.

0 שווה ל-$false$, וכל מספר אחר שאינו 0 שווה ל-$true$. דוגמאות:

$while(1)\ \{\dots\}$ – לולאה אין סופית, שווה ערך ל-$while(true)$ ב- $Java$.

$if(-1974)\{\dots\}$ – תנאי אמת, שווה ערך ל-$if(true)$ של $Java$.

$i = (3 == 4);$ - במקרה זה i יהיה שווה 0, מכיוון ש-3 לא שווה ל-4,

וכפי שכבר אמרנו המקביל ל-$false$ ב-C הוא 0.

$i = (4 == 4);$ - במקרה זה i יהיה שווה 1.

$if(!\,(a-3))\ \{\dots\}$ – יהיה תנאי אמת אם ורק אם $a = 3$,

כי אז התנאי הוא $if(!\,0)$ שזה שווה ערך ל-$if(true)$ ב-$Java$.

בסטנדרט $C99$ נוספו משתנים בוליאנים. כדי להשתמש בהם, נייבא את $\langle stdbool.h\rangle$, ונוסיף משתנים מסוג $bool$ שיקבלו $true$ או $false$:

```
#include ⟨stdbool.h⟩
#include ⟨stdio.h⟩
int main() {
        bool t = true;
        bool f = false;
        if (t != f) {
                printf("t = %d, f = %d\n", t, f); // t = 1, f = 0
                printf("It is %s that 3 is greater than 4.\n",
                                (3 > 4) ? "true" : "false");
        }
        return 0;
}
```

הערה 1: הגודל של משתנה מסוג $bool$ הינו 1 בית (8 ביטים)

הערה 2: השתמשנו בקוד הנ"ל באופרטור טרינארי, הוא פועל באופן הבא:

$$bool\ ?\ if\_true\ :\ if\_false;$$

כלומר מצד שמאל יש תנאי בוליאני, למשל $3 > 4$ (אחריו יש סימן שאלה כך שזה אינטואיטיבי)

לאחר מכן נבחר מה לעשות במקרה שהתנאי הוא תנאי אמת, ולבסוף מה לעשות בתנאי שקר.

במילים: "3 גדול מ-4? אם כן תדפיס true : אחרת תדפיס false"

sizeof

למדנו שגודל טיפוס מסוג $char$ הינו בית אחד (בכל מחשב/קומפיילר), אך לא נוכל להניח הנחות דומות לגבי שאר המשתנים. לכן נוכל להשתמש באופרטור $sizeof$ שתפקידו להחזיר את מספר הביתים שטיפוס מסוים תופס בזיכרון. למשל:

```
unsigned long numberOfBytes = sizeof(unsigned char);  // 1
```

באותו אופן נוכל לבדוק לגבי כל שאר הטיפוסים. כחלק ממוסכמות השפה, מתקיים היחס הבא:

$$sizeof(char) < sizeof(short) \leq sizeof(int) \leq sizeof(long)$$

וגם

$$sizeof(char) < sizeof(short) \leq sizeof(float) \leq sizeof(double)$$

<u>הערה</u> 1: לרוב, $short$ יהיה 2 או 4 ביתים, int יהיה 4 או 8 ו-$long$ 8 או 16.

<u>הערה 2</u>: ב-C (להבדיל מ-$C++$) תווים שלא מוגדרים באופן מפורש כ-$char$ יישמרו כ-int.

כלומר אם נעשה $sizeof('A')$ נקבל 4 או 8 (הגודל של טיפוס int) ולא 1 כפי שהיינו מצפים.

Casting

בדומה ל-$Java$, ניתן לעשות $Casting$ מטיפוס מסוים לטיפוס אחר. למשל:

```
int x = 75;
int y = 100;
float z = x / y;  // = 0.0
float z = (float) x / y;  // = 0.75
int z = (float) x / y;  // = 0
```

printf

כדי להדפיס טקסט, נשתמש באופרטור $printf$ של הספרייה $stdio.h$. למשל:

$printf("Size\ of\ char\ \%d\backslash nSize\ of\ short\ \%d\backslash n", sizeof(char), sizeof(short));$

בשונה מ-$Java$, כדי להוסיף משתנים למחרוזת הרצויה נצטרך להוסיף את המשתנים לאחר המחרוזת כאשר הם מופרדים ע"י פסיקים. כדי לציין את מיקום ההדפסה שלהם במחרוזת, נשתמש בסינטקס מיוחד של האופרטור (% ולאחריו תו מסוים שיציין את טיפוס המשתנה).

למשל במחרוזת הנ"ל השתמשנו ב-$\%d$ פעמיים, וכך יידענו את $printf$ היכן להדפיס את המשתנים הרצויים וגם שהם מספר שלם. סדר המשתנים יהיה לפי סדר ה-$\%d$ שבמחרוזת.

נשים לב שהשתמשנו גם ב-$\backslash n$ שזה אנחנו מכירים מ-$Java$ בתור ירידת שורה.

כדי להדפיס טיפוסים שונים או תווים נוספים, נשתמש בסינטקס שמפורט בטבלאות הבאות:

Printf Character	Description
$\backslash n$	$new\ line$
$\backslash b$	$backspace$
$\backslash t$	tab
$\backslash\backslash$	$backslash$
$\backslash"$	$double\ quote$
$\backslash'$	$single\ quote$

Printf Specifier	Type
$\%c$	$char$
	$unsigned\ char$
$\%d\ or\ \%i$	int
	$short$
	$long$
$\%u$	$unsigned\ int$
	$unsigned\ short$
	$unsigned\ long$
$\%f$	$double$
	$float$
$\%s$	$String$
$\%\%$	$Prints\ \%\ character$
$\%p$	$Pointer\ address$

סוגי אופרטורים

אריתמטיים: $+ \quad - \quad / \quad * \quad \%$

הוספה/הסרה: $++ \quad -- $

יחס: $< \quad <= \quad > \quad >= \quad == \quad !=$

לוגיים: $\&\& \quad || \quad !$

$Bitwise$: $\& \quad | \quad ^\wedge \quad << \quad >> \quad \sim$

השמה: $+= \quad -= \quad *= \quad /= \quad \%=$

<u>הערה לגבי אופרטור החילוק</u>:

```
float x = 3 / 2;  //1
float y = 3.0 / 2;  //1.5
int z = 3.0 / 2;  //1
```

שורה 1 – למרות שתוצאת החילוק נשמרת במשתנה $float$,

פעולת החילוק מתבצעת על 2 intים ולכן גם התוצאה שלה היא int.

שורה 2 –לפחות אחד מבין המחולקים מסוג $float$ לכן גם תוצאת החילוק כזו.

שורה 3 – תוצאת החילוק אומנם $float$ אך המשתנה בו נשמרת התוצאה מסוג int,

לכן הוא נחתך ונשאר רק החלק השלם שלו.

<u>הערה לגבי הוספה/הסרה</u>:

נזכור שיש הבדל בין $x++$ ל-$++x$, כלומר:

$y = x++;$ - קודם כל הערך של x מועתק ל-y, ולאחר מכן הערך של x גדל ב-1.

פעולה זו פועלת באותו אופן כמו: $y = x; \quad x = x + 1;$

(כלומר בסוף הערך של x ו-y יהיה **שונה**)

$y = ++x;$ - קודם כל הערך של x גדל ב-1, ולאחר מכן הוא מועתק ל-y.

פעולה זו פועלת באותו אופן כמו: $x = x + 1; \quad y = x;$

(כלומר בסוף הערך של x ו-y יהיה **שווה**)

ביטויים

תנאים:

```
if (expression) {
        // ... (single statement or block)
} else if (expression) {
        // ...
} else {
        // ...
}

switch (integer value) { ... }
```

לולאות:

```
// for( initial ; test condition ; update step )
int i, j; // in ANSI C you can't declare inside the for loop!
for (i = 0, j = 0; (i < 10 && j < 5); i++, j+= 2) {
        // ...
}
```

הערה: נשים לב שב-C לא היה ניתן להצהיר על משתנים בתוך לולאה,

אך החל מסטנדרט 99 (הסטנדרט שאיתו אנחנו עובדים בקורס) כן אפשר.

בדוגמה זו הצהרנו על i ו-j מחוץ ללולאה, אך יכלנו גם אחרת.

```
while (condition) {
        // ...
}
```

```
do {
        // ...
} while (condition);
```

הערה: להבדיל מ-$while$ שתפעל רק במידה והתנאי הוא אמת,

בלולאת $do - while$ קודם כל הקוד שבתוך הלולאה ירוץ פעם אחת,

ואחר-כך אם התנאי הוא אמת, הלולאה תמשיך לפעול.

```
break; // exit loop
continue; // begin next iteration
```

getchar / putchar

נשתמש ב-$getchar$ כדי לקבל קלט מהמשתמש וב-$putchar$ כדי להדפיס אותו למסך. למשל:

```
#include < stdio.h >
int main() {
        int c;
        while( (c = getchar()) != EOF) {
                putchar(c)
        }
        return 0;
}
```

הקוד הנ"ל מבקש קלט מהמשתמש ולאחר מכן מדפיס אותו.

במקרה זה הוא ימשיך כך בלולאה עד שנלחץ $Ctrl + Z$ כדי לציין שסיימנו.

EOF קבוע שמסמן את סוף הקובץ - "$End\ of\ File$".

#define macro

בעזרת המילה השמורה $\#define$ נוכל לשמור קבועים (בהמשך הסיכום נרחיב יותר)

מתחת ל-$include$ נוסיף את מה שנרצה , למשל:

```
#include < stdio.h >
#define NUM_OF_LINES 10
int main() {...}
```

באופן הזה הגדרנו את NUM_OF_LINES כסוג של קבוע עם הערך 10.

scanf

"המשלים" של $printf$, בעזרת $scanf$ נוכל לקרוא מידע מהמשתמש באופן הבא:

```
int n; float q; double w;
printf("Please enter an int, a float and a double\n");
scanf("%d %f %lf", &n, &q, &w);
printf("I got: n = %d, q = %f, w = %lf", n, q, w);
```

<u>הערה</u>: במידה ו-$scanf$ לא מצליח לקרוא את התו, הוא משאיר אותו ב-$Standart\ Input$ (קובץ שמכיל את המידע שהגיע מהמשתמש) ובלולאה אינסופית הוא ממשיך לקרוא שוב ושוב את התו.

לכן כדי להשתמש ב-$scanf$ נדאג לטפל במקרה קיצון הנ"ל בדרכים שונות, למשל:

http://www.giannistsakiris.com/2008/02/07/scanf-and-why-you-should-avoid-using-it/

פונקציות

C מאפשרת להגדיר פונקציות. הסינטקס הינו:

ערך החזרה ↓ הכרזת משתנים ↓

```
int power(int a, int b){
        // ...
        return 7;
}
```

פונקציות $void$:

```
void power(int a, int b){
        // ...
        return;
}
```

* בפונקציית $void$ ניתן לרשום $return$ אך לא חייב.

פונקציות מכירות רק בפונקציות שהוגדרו מעליהן, למשל:

```
void funcA() {
        ...
}
void funcB() {
        funcA();
}
```

הקוד הנ"ל יתקמפל וירוץ בהצלחה מכיוון שמתוך הפונקציה $funcB$

קראנו לפונקציה $funcA$ שהוצהרה מעליו. אך הקוד הבא יזרוק שגיאה:

```
void funcA() {
        funcB();
}
void funcB() {
        ...
}
```

כדי לעקוף את זה, נוכל להצהיר על המתודות הרצויות בתחילת הקובץ, למשל:

```
void funcB();
void funcA() {
        funcB();
}
void funcB() {
        ...
}
```

כלומר הוספנו רק את חתימת השיטה $funcB$ בתחילת הקובץ,

ובכך אנחנו מבטיחים לקומפיילר שנגדיר פונקציה בשם הזה בהמשך.

<u>הערה</u>: בחתימת השיטה ניתן להצהיר רק על סוג הפרמטר מבלי לציין שם, למשל: $int\ foo(int);$

לפונקציות יכולות להיות מספר חתימות שיטה, אך רק מימוש אחד. (אין $overloading$)

```
int foo(int a);
int foo(int);
int foo();
int foo(int a) {
        return a;
}
```

הקוד הנ"ל תקין, אך לא ניתן להוסיף את הקוד הבא:

```
int foo() {
 return a;
}
```

כי מדובר במימוש נוסף של השיטה foo (אפילו שהפרמטרים שונים / ערך ההחזרה שונה).

התנהגות מוזרה בעקבות הפתרון הנ"ל:

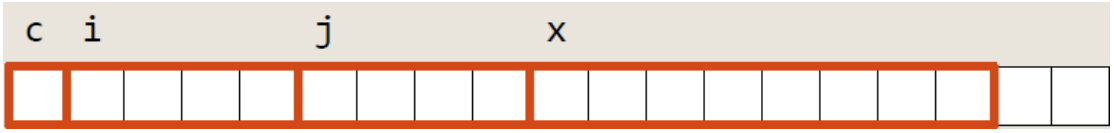
```
int foo();
int main() {
        foo(5,6,7);
        return 0;
}
int foo(int a) {
        return a;
}
```

הקוד הנ"ל יתקמפל וירוץ (אך יכול לגרור התנהגות לא רצויה),
למרות ששלחנו 3 משתנים לשיטה שמקבלת רק משתנה 1.
הסיבה לכך היא שבחתימת השיטה (בתחילת הקובץ),
לא הגדרנו פרמטרים ולכן הקומפיילר לא יודע מה מספר/סוג הפרמטרים הנכון,
ולכן הוא מאפשר זאת בתוך ה-$main$.

זיכרון ומערכים

זיכרון המחשב מקצה תאים (ביטים) לכל משתנה שאנחנו מייצרים. למשל:

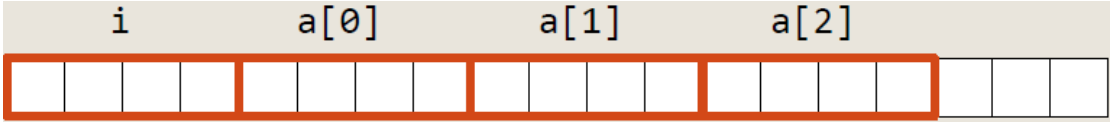
```
int main() {
        char c;
        int i, j;
        double x;
```



בדומה ל-$Java$, גם ב-C ניתן להגדיר מערך – בלוק של תאים עוקבים. למשל:

```
int main() {
        int i;
        int a[3];
```

המערך מיוצג בזיכרון באופן הבא:

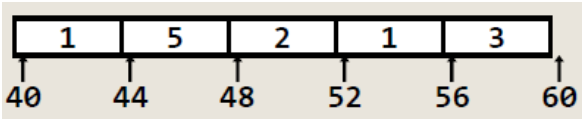


<u>האופרטור [ ]</u>

כדי לגשת לאיברים במערך, אנחנו משתמשים באופרטור [ ] , למשל:

$int\ arr[5] = \{1,5,2,1,3\};$

נניח שהמערך מתחיל בכתובת 40 בזיכרון:



האופרטור [ ] מחשב את מיקום האיברים במערך באופן הבא:

$$arr[i] \ = \ 40 + i * sizeof(int) = 40 + 4 * i$$

(במקרה הזה מדובר במערך של intים לכן גודל תא במערך הוא 4)

הערה: בעבודה עם מערכים צריך להיזהר מאוד מגלישה למקומות אחרים בזיכרון. למשל:

$int\ a[4];$
$a[-1] = 0;$
$a[4] = 0;$

הקוד הנ"ל יתקמפל וירוץ ללא שגיאה, אך בפועל פנינו למקומות אחרים בזיכרון ושינינו את ערכם.

אתחול מערכים:

```
int arr[3] = {3,4,5}; // Good
int arr[] = {3,4,5}; // Good
int arr[3] = {0}; //Init all items to 0, takes O(n)
int arr[4] = {3,4,5}; // Bad style – the last is 0
int arr[2] = {3,4,5}; // Bad
```

מערך דו ממדי:

נגדיר מערך דו ממדי כך: $int\ a[ROWS][COLS]$

```
int arr[2][3] = {{2,5,7},{4,6,7}}; // Good
int arr[2][3] = {2,5,7,4,6,7}; // Good: the same
int arr[3][2] = {{2,5,7},{4,6,7}}; // Bad
int arr[3]; // Uninitialized values
arr = {2,5,7}; Bad (compilation): array assignment only in initialization
```

argc/argv

$Argv\ (Argument\ Vector)$ – מערך של **מחרוזות** שנשלחו בזמן הרצת התוכנית.

נניח שיש לנו קובץ $hello.c$, נקמפל אותו בעזרת $gcc\ hello.c - o\ hello$

ולאחר מכן נריץ את התוכנית על ידי $./hello$.

לעיתים נרצה לשלוח פרמטרים נוספים כמו למשל $./hello\ \ name\ \ age$.

והטקסט הזה מועבר ל-$Argv$ כך ש:

```
argv[0] = ./hello
argv[1] = name
argv[2] = age
```

$Argc\ (Argument\ Count)$ – הגודל של $Argv$, כלומר כמה מחרוזות נשלחו בהרצה.

למשל בדוגמה הנ"ל, $Argc = 3$. שימושי למשל כדי לרוץ בלולאה על כל $Argv$.

- זה הרגל טוב להדפיס את הארגומנטים שנשלחו בתחילת התוכנית. למשל:

```
int main(int argc,  char *argv[]) {
        int i;
        for(i = 0; i < argc; i++) {
                printf("%s ", argv[i]);
        }
}
```

- אם מספר הארגומנטים שונה ממה שצריך, מומלץ להדפיס תיאור לתפעול נכון של התוכנית.

```
if (argc < 2) { // no arguments given
        printf("Usage: myprog < num1 >< num2 >\n");
}
```

מצביעים

לכל משתנה בזיכרון יש כתובת ייחודית. מצביע הוא סוג מיוחד של משתנה שהערך הנמצא בתוכו הינו הכתובת בזיכרון של משתנה או קבוע אחר. במילים אחרות מצביע הוא משתנה מטיפוס $long$ המכיל (במחשבים של 64 ביט) כתובת של 64 ביט (8 בתים) למקום אחר בזיכרון. (אם מדובר במחשבים של 32 ביט אז 4 בתים בהתאם) הכתובת מאפשרת גישה ושינוי של משתנים מכל מקום.
שימושי מאוד אך מסבך את הקוד מכיוון שיותר אחריות נמצאת אצל המתכנת.
כדי לדעת מה הכתובת בזיכרון של משתנה אחר, נשתמש באופרטור &, למשל:

```
int var;
int arr[10];
printf("Address of var: %p\n", &var);
printf("Address of arr: %p\n", &arr);
```

<u>הכרזת מצביע</u> - $\langle type \rangle\ ^*p;$ למשל $int\ ^*p$. בדוגמה הזו p הוא מצביע לאובייקט מסוג int.

<u>השמה</u> – כפי שאמרנו, מצביע מכיל כתובת בזיכרון של משתנה אחר לכן נשתמש ב-& להשמה.

למשל הפקודה $p = \&y$ תאחסן ב-p את הכתובת בזיכרון של y.

<u>גישה</u> – כדי לגשת לערך שיושב בתא בזיכרון נשתמש גם ב-$*$:

- למשל $y = {}^*p;$ יבצע השמה של **הערך** ש-p מצביע אליו בתוך y.
- בנוסף $^*p = x;$ יבצע השמה של הערך של x במקום הערך ש-p מצביע אליו.

הערה: האופרטור $*$ משמש גם כדי להכריז על משתנה מסוג מצביע וגם כדי לגשת לערך שלו.

האופרטור & משמש כדי לקבל את הכתובת של המשתנה ולא את ערכו.

דוגמא 1: $int\ ^*p, q;$ באופן הזה p הוא מצביע לטיפוס מסוג int ו-q הוא משתנה מטיפוס int.

דוגמא 2: נראה מספר פעולות רצופות ונבין מה כל אחת עושה:

```
int main() {
int i, j;
int *x; \\ x points to an integer
i = 1; \\ i value is 1
x = &i; \\ x now points to i (holds the address of i)
j = *p; \\ j value is now 1 (same as i)
x = &j; \\ x now points to j (holds the address of j)
(*x) = 3; \\ j value is now 3
}
```

מי שעדיין לא הבין לגמרי את הנושא, הסבר טוב בעברית עם דוגמאות:
https://he.wikibooks.org/wiki/שפת_C/מצביעים

מצביע NULL

ב-C (להבדיל מ-$Java$ למשל) אין ערכים דיפולטיביים. למשל, כשאנחנו יוצרים מצביע חדש, הוא מצביע לתא בזיכרון שמכיל "שאריות זיכרון" ולא מאותחל לערך דיפולטיבי. כדי להימנע משגיאות זמן ריצה של גישה לערכים לא רצויים כמו שאריות זיכרון, נשתמש ב-$NULL$ כערך דיפולטיבי.

כלומר זו אחריות של המתכנת לבצע השמה של $NULL$ לתוך מצביע חדש במקרים בהם לא ידוע הערך בזמן ההכרזה. בפועל $NULL$ הוא קבוע עם הערך 0 שמוגדר ב-$\langle stdlib.h \rangle$, ולכן כדי להשתמש בו נצטרך לבצע $\#include$ ל-$\langle stdlib.h \rangle$.

```
int main() {
        int *p = NULL;
        printf("The value of p is: %p\n", p);  \\ 0
        if (p! = NULL) {
                ...
        }
}
```

<u>הערה חשובה</u>: במקרה של שינוי הערך שיושב במצביע, נבדוק לפני זה שהוא לא מצביע ל-$NULL$.

```
int *p = NULL;
*p = 1;
```

בקוד הנ"ל ביצענו השמה של הערך 1 למיקום "לא ידוע" בזיכרון,

זה יתקמפל אבל ככל הנראה יגרום לשגיאת זמן ריצה.

לכן נרצה לוודא תמיד שהמצביע הוא לא $NULL$ לפני השמה. גם הקוד הבא יגרום לאותה בעיה:

```
int *p;
*p = 1;
```

מצביעים ומערכים

כשאנחנו מגדירים למשל $a[3]$ אנחנו יוצרים 3 תאים צמודים בזיכרון שמייצגים מערך.
המשתנה a מצביע לתא הראשון במערך, ולכן נוכל לבצע בו מניפולציות של מצביעים.

```
int foo( int *p );
int foo( int a[] );
int foo( int a[NUM] );
```

כל הפקודות הנ"ל מבצעות את אותה הפעולה:

כולן מעבירות לשיטה foo מצביע (כתובת) למשתנה מסוג int.

```
int *p;
int a[3];
p = &a[0]; // same as p = a
*(p + 1) = 1; // same as p[1] = 1
```

בקוד הנ"ל יצרנו מצביע ומערך, וביקשנו מ-p להצביע על המקום הראשון במערך.

בשורה הרביעית ניגשנו למקום השני במערך ושינינו את ערכו ל-1.

```
p = a;  // same as p = &a[0]
```

כעת p מצביע גם הוא למקום הראשון במערך. אך נשים לב שהפעולה הבאה לא חוקית!:

```
a = p;
```

כי a הוא לא משתנה שמוגדר להיות מצביע.

```
p++; // p = a + 1 = &a[1],    p+= sizeof(int)
```

כלומר $++$ על מצביע למערך יגרום לו להצביע על התא הבא. אך הפעולה הבאה לא חוקית!:

```
a++;
```

<u>הערות</u>:

- גודל מערך ידוע בזמן קומפילציה.
- גודל מצביע הוא קבוע לא משנה על מה הוא מצביע.

נקודה חשובה: כדי לחשב את הגודל של מערך מסוים, נוכל לעשות:

```
int main() {
        int arr[4] = {1,3,5,4};
        int i, sum = 0;
        for (i = 0; i < sizeof(arr)/sizeof(arr[0]); ++i) {
                sum += arr[i];
        }
}
```

הסבר: מכיוון שהמערך הוגדר באותו $Scope$ שבו עשינו $sizeof(arr)$, התוצאה תהיה 16 כי הגודל ידוע בזמן קומפילציה. אך אם היה מדובר במערך שהוגדר ב-$Scope$ אחר (למשל היה עובר כארגומנט), אז $sizeof(arr)$ היה שווה לגודל של משתנה מסוג מצביע, שכן arr הוא מצביע ובזמן קומפילציה לא ידוע איזה מערך ישלח כארגומנט או כמה תאים הוא מכיל. מה אפשר לעשות? לשלוח משתנה נוסף מסוג int שיציין את גודל המערך.

VOID

$void\ ^*p;$ יגדיר מצביע p שיכול להצביע על כל סוג של טיפוס.

```
int j;
void *p = &j;
int k = *p; //illegal
int k = (int)*p; //illegal
int k = *(int *)p; //legal
```

נשים לב שמצביע מסוג $void$ לא צריך לבצע $cast$, אך מ-$void$ למצביע מסוג אחר צריך.

<u>הערה</u>: ניתן לבצע $cast$ מכל סוג של מצביע לכל סוג אחר, זה שימוש לפעמים אך יש להיזהר.

<u>הערה 2</u>: אין "אריתמטיקת מצביעים" על $void$.

<u>הערה 3</u>: gcc מסמן את הגודל של משתנה $void$ כ-1. $(sizeof(^*p) == 1)$.

<u>הערה 4</u>: לא ניתן לגשת לערך של מצביע מסוג $void$. הפקודה $int\ k = ^*p;$ אינה חוקית.

<u>הערה 5</u>: ניתן לבצע $cast$ לסוג הרצוי ואז יהיה ניתן לגשת, למשל $int\ k = ^*(int\ ^*)p$ חוקי !

מצביעים למצביעים

מצביע הוא משתנה, לכן אפשר להצביע על מצביע וכן הלאה.. זה יכול להיות מבלבל:

```
int main() {
int n = 17;
int *p = &n;
int **p2 = &p;
printf("the address of p2 is %p \n",&p2);
printf("the address of p is %p \n",p2);
printf("the address of n is %p \n", *p2);
printf("the value of n is %d \n", **p2 );
return 0;
}
```

חשוב להבין למה בחרנו ב- $&p2, p2, ^*p2, ^{**}p2$ בכל הדפסה !

ניהול זיכרון ($malloc/calloc/free/realloc$)

ניהול זיכרון הוא יתרון בולט של C. כשאנחנו כותבים קטע קוד, אנחנו משתמשים במקטעי זיכרון ($Memory\ Segments$) , שהם חלקים שונים בזיכרון המחשב.

$Code\ Segmant$ – מקטע המכיל העתק של אוסף הפקודות של התוכנית (הקוד שכתבנו).

$Data\ Segmant$ – מורכב מ-3 חלקים:

- $Stack$ – מחסנית, מקטע זיכרון המיועד למשתנים לוקאליים וקריאות לפונקציות.

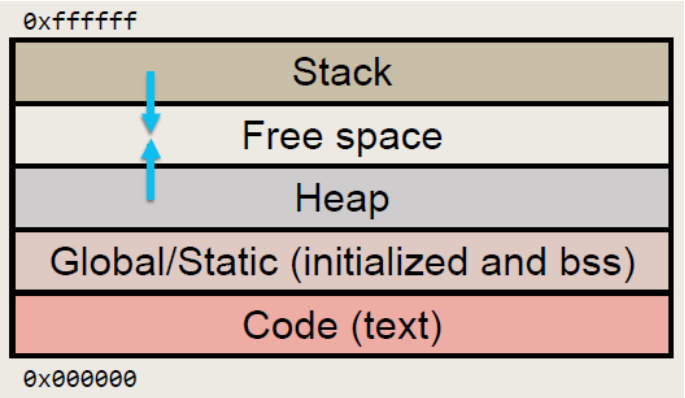
הגודל של המשתנים מוגדר בזמן קומפילציה, והם "חיים" בזמן מוגבל כל עוד ה-$Scope$ שלהם פעיל. עד לחלק זה של הקורס, כל מה שעשינו נשמר במחסנית.

<u>הערה 1</u>: $StackOverflow$ זו שגיאה המציינת שלא נשאר מקום פנוי במחסנית.

<u>הערה 2:</u> עד לסטנדרט $C99$ לא היה אפשר להקצות מקום במחסנית בגודל שנקבע בזמן ריצה למשל לפי קלט מהמשתמש, אך ב-$C99$ הוסיפו $VLA's$ שפותרים את הבעיה הזו.

בקורס שלנו אסור להשתמש ב-$VLA's$, לא בתרגילים ולא במבחן.

- $Global\ /\ Static\ Area$ – מקטע המכיל משתנים גלובליים וסטטיים שמוגדרים בהרצה.
- $Dynamic\ Heap$ – ערימה, מקטע זיכרון המיועד להקצאה דינמית, כלומר הקצאת זיכרון בזמן ריצה לפי הצורך.



Malloc היא פונקציה המאפשרת לנו לבקש/להקצות זיכרון בערימה ($heap$):

$void\ ^*malloc(size\_t\ Size);$

השיטה מקבלת משתנה מטיפוס $unsigned\ int$ שמציין את מספר הבתים שאנחנו צריכים.

השיטה מחזירה מצביע מסוג $void\ ^*$ עבור המקום שהוקצה בזיכרון.

זו אחריות שלנו לבצע $casting$ לטיפוס הנכון. במידה ואין מספיק מקום פנוי בזיכרון , השיטה תחזיר $NULL$, לכן תמיד בקריאה לשיטה נוודא שאכן הוקצה הזיכרון ולא קיבלנו בחזרה $NULL$.

$char\ ^*str\ =\ (char *)\ malloc(5 * sizeof(char));$

$if\ (str\ ==\ NULL)\ \ //\ print\ error\ message\ or\ perform\ other\ relevant\ operation$

ביקשנו מהשיטה $malloc$ $5 \cdot sizeof(char)$ בתים בזיכרון, וביצענו $down-casting$ ל-$char *$.

<u>הערה</u>: פונקציה שמקצה זיכרון דינמי צריכה גם לשחרר אותו, או לציין בדוק' שהיא לא עושה זאת.

Calloc היא פונקציה נוספת שמקצה זיכרון דינמי, אך יש לה 2 הבדלים עיקריים:

1. היא מקבלת 2 פרמטרים – אורך המערך הרצוי בזיכרון וגודל כל תא במערך. למשל:

$int\ ^*p\ =\ (int\ ^*)\ calloc\ (length, sizeof(int));$

מספר הבתים שיוקצו יהיה $length \cdot sizeof(int)$.

2. היא מאתחלת את כל הערכים בזיכרון להיות 0. ($O(n)$)

FREE היא פונקציה שמשחררת את המקום שהוקצה בזיכרון הדינמי. על כל קריאה ל-$malloc$ בקוד צריכה להיות קריאה ל-$free$ (השיטה מקבלת את המצביע לזיכרון שהוקצה ע"י $malloc$):

$int\ ^*iptr\ =\ (int *)\ malloc(sizeof(int));$

...

$free(iptr);$

$iptr = NULL;$

<u>הערה 1</u>: אם p לא ניתן ע"י $malloc$ או שהוא שוחרר כבר, תהיה התנהגות בלתי צפויה.

<u>הערה 2</u>: $free(NULL)$ לא יעשה כלום.

<u>הערה 3</u>: השיטה $free$ יודעת כמה מקום לפנות בזיכרון מכיוון שהמערכת מבצעת מעקב על גודל הזיכרון המוקצה (שומרת מידע נוסף לפני הזיכרון המוגדר) ולכן אין צורך להגיד לשיטה כמה לשחרר.

<u>הערה 4</u>: הקצאה דינאמית של זיכרון מבלי לשחררו ($memory\ leak$) עלולה לפגוע בביצועים.

<u>הערה 5</u>: אחרי שחרור המקום, זה הרגל טוב לעדכן את המצביע עם $NULL$ (כפי שעשינו בדוגמה)

Realloc היא פונקציה שתפקידה להגדיל/להקטין את גודל המערך שהוקצה לנו בזיכרון.

היא תנסה להגדיל את רצף הזיכרון הנוכחי. אם הדבר אפשרי, הפונקציה תחזיר את כתובת הזיכרון של הרצף הנוכחי כ- $void\ ^*$. אם הדבר אינו אפשרי, היא תבדוק האם יש רצף אחר מתאים בזיכרון. אם היא הצליחה, היא תעתיק את תוכן הרצף הנוכחי לרצף החדש, תשחרר את הרצף הנוכחי, ותחזיר את כתובת הרצף החדש. אם אין רצף אחר מתאים בזיכרון, היא לא תשנה כלום בזיכרון, לא תשחרר את הרצף הנוכחי, ותחזיר $NULL$ $realloc(old\_p,\ total\_size);$

כאשר old_p הוא מצביע למקטע הזיכרון הנוכחי, ו-$total\_size$ הוא הגודל החדש המבוקש (בבתים).

<u>הערה 1</u>: בדומה ל-$malloc$, תוכן התאים החדשים לא ידוע.

<u>הערה 2</u>: אם $arr == NULL$, $realloc$ יתנהג כמו $malloc$.

Valgrind

הקדמה - חבילת כלי $Valgrind$ מספקת מספר כלים לאיתור באגים ולשיפור מהירות התוכנית. הכלי הפופולרי ביותר הינו $Memcheck$ שתפקידו למצוא שגיאות זיכרון (כמו דליפת זיכרון) ולמנוע התנהגויות בלתי רצויות. התכנה מסוגלת לזהות שגיאות רק בזיכרון שקבלנו באופן דינמי במהלך התכנית ע"י ארבעת הפונקציות המתוארות בעמוד קודם. התוכנה מותאמת ל-$Linux$ בלבד.

הכנה – כדי לאפשר ל-$Memcheck$ לתת מידע אינפורמטיבי לגבי השגיאות שלנו, נקמפל את התוכנית עם הדגל g-, כלומר:

```
gcc -g -std=c99 -Wall myprog.c -o myprog
```

הרצה – כדי להשתמש ב-$Memcheck$, נריץ את התוכנית שלנו באופן הבא:

```
valgrind --leak -check=yes myprog arg1 arg2
```

כעת התוכנית תרוץ הרבה יותר לאט מהזמן הרגיל ותשתמש בהרבה זיכרון. לאחר מכן יודפסו הודעות אינפורמטיביות לגבי שגיאות זיכרון.

פענוח – נסתכל על קטע קוד המכיל שגיאות זיכרון, ועל הפלט של $Memcheck$:

```
#include <stdlib.h>
void f(void) {
        int * x = malloc(10 * sizeof(int));
        x[10] = 0;      // problem 1: heap block overrun
}                       // problem 2: memory leak --x not freed
int main(void) {
        f();
        return 0;
}
```

<u>הפלט (המשך בעמוד הבא)</u>:

```
== 19182 == Invalid write of size 4
== 19182 ==    at 0x804838F: f (example.c: 6)
== 19182 ==    by 0x80483AB: main (example.c: 11)
== 19182 == Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
== 19182 ==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c: 130)
== 19182 ==    by 0x8048385: f (example.c: 5)
== 19182 ==    by 0x80483AB: main (example.c: 11)
```

<u>הערות</u>:

- השורה הראשונה מציינת את סוג השגיאה. במקרה זה מדובר על כתיבה של ערך בגודל 4 במיקום לא תקין בזיכרון (המערך בגודל 10 וניגשנו למקום ה-11)
- השורות הבאות מתארות את היסטוריית המחסנית, מומלץ לעקוב מלמטה למעלה. אם נרצה היסטוריה מפורטת יותר נשתמש ב- num-callers--
- חלק מהשגיאות יכילו מידע נוסף כמו השורה הרביעית בדוגמה הנ"ל, המציינת שהתא הבעייתי נמצא אחרי מערך באורך 40 שהוקצה עבורנו.
- מומלץ לסדר תקלות לפי סדר הופעתן שכן חלק מהתקלות יכולות לנבוע מתקלות קודמות.

$== 19182 ==\ 40\ bytes\ in\ 1\ blocks\ are\ definitely\ lost\ in\ loss\ record\ 1\ of\ 1$
$== 19182 ==\quad at\ 0x1B8FF5CD: malloc\ (vg\_replace\_malloc.c: 130)$
$== 19182 ==\quad by\ 0x8048385: f\ (a.c: 5)$
$== 19182 ==\quad by\ 0x80483AB: main\ (a.c: 11)$

<u>הערות</u>:

- השורה הראשונה מתריעה על מיקום בזיכרון שהוקצה עבורנו ולא שחררנו אותו. (אכן לא השתמשנו בדוגמה ב-$free$)
- יש 2 סוגים עיקריים של שגיאות זיכרון:

"$definitely\ lost$": יש דליפת זיכרון בוודאות.

"$probably\ lost$": יש דליפת זיכרון, אלא אם כן זה בוצע במכוון כמו למשל לשנות מיקום של מצביע מתחילת מערך לאמצע שלו.

הערות כלליות

- $Memcheck$ מתריעה גם על שימוש במשתנים שלא אותחלו, בעיקר בעזרת ההודעה:

  $Conditional\ jump\ or\ move\ depends\ on\ uninitialised\ value(s)$

  מציאת המקור לשגיאה יכולה להיות משימה לא פשוטה.

  אפשר להשתמש בתוספת --track-origins=yes כדי לקבל מידע נוסף שלרוב יועיל,

  אך הוא יגרום לתוכנית לרוץ לאט יותר.
- $Memcheck$ צודק ב-99% מהמקרים, לכן צריך להתייחס לשגיאות ברצינות ולפתור אותן.
- במידה ויש שגיאה לא ברורה, ניתן לקרוא עליה כאן:

  http://valgrind.org/docs/manual/mc-manual.html#mc-manual.errormsgs

מחרוזת $Null - terminated\ String$

מחרוזת ב-C מוגדרת על ידי מערך של משתנים מסוג $char$. כמוסכמה, כל מחרוזת צריכה להסתיים ב-$\backslash 0$, תו שמציין את סוף המחרוזת. במידה והגדרנו את המחרוזת בעצמנו, למשל:

$char\ my\_string[]\ =\ "wiki";$

הקומפיילר יקצה בזיכרון 5 תאים (לפחות): $w, i, k, i, \backslash 0$, אך חשוב לזכור שבמקרים אחרים אחריותנו לוודא שהמחרוזת מסתיימת ב-$\backslash 0$. יש שיטות כמו למשל $strncpy$ (יצירת $substring$ למחרוזת) שעובדות באופן הבא: אם ביקשנו להעתיק מחרוזת בגודל 8 בתים למשתנה מסוג 8 בתים, **לא** יתווסף $\backslash 0$ בסוף המחרוזת. אחרת, אם גודל המחרוזת קטן מ-8 בתים, השיטה תוסיף $\backslash 0$.

אחריות המתכנת לוודא שהמחרוזות חוקיות (מסתיימות ב-$\backslash 0$) לכל אורך התוכנית.

מקרה קיצון מיוחד של מחרוזות

```
char* msg = "text";
msg[0] =' w'; \\seg fault!
```

המשתנה msg הוא מצביע שנשמר ב-$Stack$ ומכיל כתובת לתא הראשון במערך $\{'t','e','x','t',\backslash 0\}$

המערך עצמו נשמר ב-$Code\ Segment$ שזה מקטע בזיכרון שהוא לקריאה בלבד.

כלומר **אי אפשר** לשנות את המערך הזה במהלך התוכנית אלא רק לקרוא אותו/ממנו.

המקרה הזה נכון רק ל-$char\ *$ **ולא** לכל מצביע (כמו למשל $int\ *$)

שיטות נפוצות למחרוזות מתוך $\langle string.h \rangle$

$Function$	$Syntax$	$Example$
$strcpy$	$char\ *strcpy(char\ *dest, const\ char\ *src)$	$Copies\ the\ string\ pointed\ to,$ $by\ src\ to\ dest.$
$strncpy$	$char\ *strncpy(char\ *dest, const\ char\ *src, size\_t\ n)$	$Copies\ up\ to\ n\ characters\ from$ $the\ string\ pointed\ to, by\ src\ to\ dest.$
$strcat$	$char\ *strcat(char\ *dest, const\ char\ *src)$	$Appends\ the\ string\ pointed\ to,$ $by\ src\ to\ the\ end\ of\ the\ string$ $pointed\ to\ by\ dest.$
$strncat$	$char\ *strncat(char\ *dest, const\ char\ *src, size\_t\ n)$	$Appends\ the\ string\ pointed\ to,$ $by\ src\ to\ the\ end\ of\ the\ string\ pointed\ to,$ $by\ dest\ up\ to\ n\ characters\ long.$
$strcmp$	$int\ strcmp(const\ char\ *str1, const\ char\ *str2)$	$Compares\ the\ string\ pointed\ to,$ $by\ str1\ to\ the\ string\ pointed\ to\ by\ str2.$
$strncmp$	$int\ strncmp(const\ char\ *str1, const\ char\ *str2, size\_t\ n)$	$Compares\ at\ most\ the\ first\ n\ bytes$ $of\ str1\ and\ str2.$
$strlen$	$size\_t\ strlen(const\ char\ *str)$	$Computes\ the\ length\ of\ the\ string\ str$ $up\ to\ but\ not\ including\ the$ $terminating\ null\ character.$

CONST

המילה השמורה $Const$ מציינת שמשתנה מסוים לא הולך להשתנות לאורך ריצת התוכנית.

```
const double E = 2.71828;
E = 3.14; \\ Compiliation Error!
```

$Const$ מגן על כל מה שמשמאלו, אם אין שום דבר משמאלו רק אז הוא יגן על מה שמימינו. למשל:

```
const int arr[] = {1,2};
arr[0] = 1; \\ Compiliation Error!
```

בדוגמה זו אין שום דבר משמאל ל-$Const$ ולכן הוא יגן על ה-intים, כלומר על ערכי המערך.

```
int arr[] = {1,2,3};
int const *p = arr;
p[1] = 1; //illegal
*(p + 1) = 1; //illegal
p = NULL; //legal
```

בדוגמה הנ"ל int נמצא משמאלו של $const$ לכן הוא יגן על ערכי המערך,

ולא יגן כלל על המצביע p, ולכן יכלנו לשנות אותו ל-$NULL$.

```
int arr[] = {1,2,3};
int * const const_p = arr;
const_p[1] = 0; //legal
const_p = NULL; //illegal
```

בדוגמה הנ"ל אנחנו נגן על המצביע $int *$ כלומר על $const\_p$ ולא על הערכים עצמם.

<u>למה נשתמש ב-$Const$?</u>

1. בחתימת שיטה נצהיר על משתנים שלא הולכים להשתנות בה כדי ליידע אנשים אחרים שמשתמשים בקוד שלנו שאנחנו לא מתכוונים לשנות את תוכן המשתנה. (חלק מה-API)
2. קוד קריא יותר, כדי להימנע משגיאות, חלק מהממשקים שאנחנו מגדירים.

File I/O

fopen זו שיטה שמחזירה מצביע מסוג $FILE$ כדי לקרוא מידע מקובץ טקסטואלי.

אם הייתה שגיאה בניסיון פתיחת הקובץ, השיטה תחזיר $NULL$

והמשתנה הגלובלי $errno$ יאותחל עם מספר שונה מ-0 כדי לציין את השגיאה.

```
FILE *fp = fopen(filename,"r");
```

"r" מציין שאנחנו מעוניינים באפשרות קריאה בלבד ($read$)

בכל פעם שנפתח קובץ, נוודא שאכן הצלחנו לפתוח אותו בהצלחה:

```
if(fp == NULL) {
fprintf(stderr,"Cannot open the file");
}
```

<u>הערה</u>: נשים לב שהשתמשנו בשיטה $fprintf$, שהיא המקבילה ל-$printf$.

$printf$ מדפיסה באופן אוטומטי את המידע לקובץ ($FILE$) בשם $stdout$ ($Standart\ Output$),

אך במקרה של שגיאה אנחנו נדפיס את השגיאה הרלוונטית לקובץ $stderr$ עם $fprintf$.

fclose זו שיטה שסוגרת את הקובץ הפתוח.

```
fclose(fp);
```

במידה וסיימנו עם הקובץ, נזכור **תמיד** לסגור אותו !

<u>חשוב</u>: גם במקרה של שגיאה, נסגור את הקובץ לפני שנבצע $exit(EXIT\_FAILURE)$.

Struct

מבנה ($Struct$) הוא גרסה ראשונית של "מחלקה". $Struct$ הוא קטע זיכרון רציף אך בשונה ממערך, הוא יכול להכיל טיפוסים שונים של משתנים, והגישה אליהם אינה מספרית אלא שמית. כמו כל טיפוס אחר ניתן להעביר אותו בין פונקציות, להחזיר אותו כערך החזרה,

לייצר משתנה מסוגו - $struct\ MyStruct\ varName;$

להצביע עליו - $struct\ MyStruct\ {}^{*}p = \&varName;$

לייצר מערכים שלו - $struct\ MyStruct\ arrName[10];$

דוגמה ליצירת מבנה חדש:

```
struct MyStruct {
        int i;
        int a[3];
        int *p;
};
```

אתחול המבנה עם ערכים:

```
int k = 0;
struct MyStruct s;
s.i  =  5;
s.a  = {0,1,2};
s.p  = &k;
```

בצורה מקוצרת ניתן להשתמש ב:

```
struct MyStruct s  =  { 5, 0,1,2, &k };
```

אתחול מבנה ממבנה אחר:

```
struct MyStruct d  =  s;
```

השורה הזו שקולה ל:

```
struct MyStruct d  =  {s.i, s.a, s.p};
```

גישה לערכים ושינוי הערך שלהם בעזרת האופרטור '.' :

```
int n = 1;
s.i  =  1;
s.a[0]  =  5;
s.p  =  &n;
```

גישה למשתנים דרך מצביעים:

```
main() {
        struct MyStruct x;
        struct MyStruct *p_x  =  &x;
        x._a[0]  =  3;
        (*p_x)._a[1]  =  5; // Same
        p_x-> _a[2]  =  6; // Same
}
```

<u>הערה</u>: האופרטור $->$ מחליף את הצורך ב-$dereference$ (*) וגם בגישה למשתנה a (.)

מבנה יכול להכיל בתוכו מבנים נוספים, למשל:

```
struct Point {
        int x;
        int y;
};
struct Triangle {
        struct Point a;
        struct Point b;
        struct Point c;
};
```

הגישה אליהם תהיה בעזרת שימוש נוסף באופרטור '.':

```
struct Triangle t;
int ax = t.a.x;
int by = t.b.y;
```

Typedef

מילה שמורה שמאפשרת לתת שם קצר יותר \ מובן יותר למשתנים קיימים. משפר את הקריאות ומקצר את הקוד. אפשר להשתמש בו גם ל-$Struct$ וגם לטיפוסים רגילים.

נגדיר את ה-$typedef$ של מבנה בזמן האתחול שלו:

```
typedef struct Complex {
        double _real, _imag;
} Complex;
```

כעת במקום להשתמש ב-$struct\ Complex$ כדי להצהיר על מבנה חדש מסוג $Complex$,

נוכל לרשום ישר $Complex$ כמו בדוגמה הבאה:

```
Complex name;
```

או כערך החזרה של פונקציות למשל:

```
Complex addComplex(Complex, Complex);
Complex subComplex(Complex, Complex);
```

בזמן הצהרה על משתנה מטיפוס $struct$, נאתחל אותו בעזרת פונקציית אתחול.

(אין ב-C בנאי כמו ב-$Java$, לכן נבצע זאת באופן ידני) למשל:

```
struct Complex *complex_ptr = initComplex(0.3,0.8);
```

העתקת מבנים

נשתמש באופרטור = כדי לבצע העתקה של כל הערכים ממבנה אחד למבנה אחר:

```
Complex a, b;
a._real = 5;
a_image = 3;
b = a;
```

נקודה חשובה לגבי העתקת מצביעים ממבנים:

```
Complex a, b;
a._real = 5;
a_image = 3;
a._p_arr = a._arr;
b = a;
```

במקרה זה מה שיועתק ל-b הם הערכים של כל המשתנים. במקרה של משתנה מסוג מצביע, מה שיועתק זה המצביע. הבעיה בדוגמה הנ"ל היא שנרצה ש-$b.\_p\_arr$ יצביע למערך $b.\_arr$ ולא ל-$a.\_arr$. לכן במקרה זה נצטרך לבנות פונקציית $Clone$ שתבצע העתקה של כל הערכים הרגילים, ונשנה את המצביע של b באופן ידני להצביע על המערך הנכון:

```
void cloneComplex (Complex *a, Complex *b) {
        int i = 0;
        for (i = 0; i < ARRAY_SIZE; i++) {
                b-> _arr[i] = a-> _arr[i];
        }
        b-> _p_arr = b-> _arr;
}
```

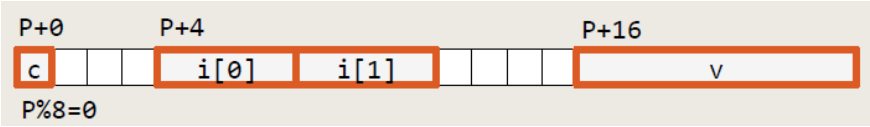
Data Alignment

אמרנו ש-$Struct$ הוא קטע רציף בזיכרון, אך בפועל הוא לא לגמרי רציף. מה הכוונה?

גודל המבנה $\leq$ לסכום הגדלים של האיברים בתוכו.

נסתכל על קטע הקוד הבא ועל אופן הייצוג שלו בזיכרון:

```
struct s {
        char c;
        int i[2];
        double v;
}
```



מצד אחד המבנה אכן נמצא בקטע רציף בזיכרון כי מובטח לנו שהבתים שבין המשתנים יישארו ריקים, מצד שני הוא לא באמת רציף שכן יש בתים ריקים בתוכו.

תהליך הקצאת הזיכרון פועל לפי 2 העקרונות הבאים:

- כל משתנה נמצא בזיכרון לפי סדר הגדרתו משמאל לימין.
- כל משתנה נמצא בתא הכי צמוד למשתנה שלפניו המקיים בנוסף:

$$mod(address/sizeof(member)) == 0$$

(כלומר המיקום המספרי של התא מתחלק במספר הבתים שהמשתנה תופס)

למשל משתנה מסוג int הוא בגודל 4 בתים (לרוב), ולכן הוא נמצא במקום ה-4 וה-8

בדוגמה הנ"ל. בנוסף המשתנה v מסוג $double$ נמצא במקום ה-16 ומתקיים:

$$4\%4 = 0 \quad \&\& \quad 8\%4 = 0 \quad \&\& \quad 16\%8 = 0$$

כנדרש.

הפעולה הזו מובטחת ומבוצעת על ידי C ולא על ידינו, אך אנחנו צריכים להכיר זאת למקרים בהם נרצה לכתוב קוד לחומרה עם מינימום זיכרון פנוי. במקרה ונרצה למנוע את הפעולה המתוארת לעיל ו"לבטל" את הבתים הריקים, נוכל להשתמש בקטע הקוד הבא:

```
struct s {
        char c;
        int i[2];
        double v;
} __attribute__((packed));
```

אז למה זה טוב?

1. הגישה של חומרה למשתנים מהירה יותר.
2. פלטפורמות מסוימות ($CPU's$) לא תומכים בגישה לזיכרון לא מיושר באופן שתואר.

Arrays & Structs

כשאנחנו מעבירים מערך בין פונקציות, אנחנו מעבירים בעצם מצביע לתא הראשון במערך.

זה לא נכון עבור $Structs$, במקרה זה נוצר עותק חדש ב-$Stack$ ושינויים בו ישנו את העותק הלוקאלי. (כלומר העותק המקורי יישאר ללא שינוי)

```
typedef struct MyStr {
        int _a[10];
} MyStr;
void f(int a[]) {
        a[7] = 89;
}
void g(MyStr s) {
        s._a[7] = 84;
}
main() {
        MyStr x;
        x._a[7] = 0;
        f(x._a);
        printf("%d\n", x._a[7]); //89
        g(x);
        printf("%d\n", x._a[7]); //89
}
```

חשוב להבין למה הפלט של הקוד הנ"ל הוא 89 פעמיים.

הפעלת הפונקציה f על $x._a$ אכן שינתה את הערך של $a[7]$ ל-89,

כי העברנו עותק של מצביע למערך a של המבנה x. (מצביע למקום הראשון במערך של x)

אבל הפעלת הפונקציה g על x יצרה עותק לוקאלי של כל x (כל מקבץ הזיכרון שמייצג את x הועתק במלואו למקבץ חדש בזיכרון) ולכן השינוי לא בוצע על x אלא על עותק שלו.

משתנים סטטיים וחיצוניים

בשפת C קיימים שלושה סוגים עיקריים של משתנים:

- **משתנים מקומיים (אוטומטיים)** – המשתנים שהשתמשנו בהם עד עכשיו. קיומם וטווח הכרתם מוגבל לבלוק שבו הם מוצהרים, כולל הבלוקים המקוננים (הפנימיים) בתנאי לא הוצהר בהם משתנה נוסף עם אותו שם.
- **משתנים סטאטיים** – ההבדל היחיד בינם לבין משתנים מקומיים הוא אורך החיים שלהם. הם "חיים" לאורך כל התוכנית ! משתנים סטטיים בשיטות שומרים על ערכם מהקריאה הקודמת לשיטה. המשתנים נמצאים בזיכרון גלובלי הנקרא $static\ heap$. למשל:

```
int getUniqueID() {
        static int id = 0;
        id++;
        return id;
}
int main() {
        int i = getUniqueID(); // i = 1
        int j = getUniqueID(); // j = 2
}
```

נשים לב שהמשתנה הסטטי id אותחל פעם אחת בלבד ל-0. בכל קריאה לפונקציה, הוא שמר על ערכו מהקריאה הקודמת. שימושי למשל ליצירת $counter$ או למשל לדיבוג מספר הפעמים שנכנסו לפונקציה מסוימת.

- **משתנים חיצוניים (גלובליים)** - משתנים גלובליים הינם משתנים המוצהרים (באופן רגיל) מחוץ לבלוקים ופונקציות, ב-$Scope$ החיצוני ביותר. אורך חייהם הוא כאורך חיי התוכנית וטווח הכרתם הוא מהמקום שבו הם מוצהרים ו"מטה", גם כאן אם יש משתנה מקומי ש"מסתיר" את הגלובלי, ההתייחסות תהיה אליו. הצהרת משתנה יכולה להיעשות מספר פעמים אך הגדרתו פעם אחת בלבד. נרחיב קצת על ההבדל בין הצהרה להגדרה:

<u>הצהרה</u> על משתנה גלובלי ב-C מתבצעת בעזרת "$extern$", למשל: $extern\ int\ var;$.
לא הכנסנו ערך למשתנה ולכן בפועל לא הוקצה זיכרון עבורו. לעומת זאת, אם היינו מכניסים גם ערך, היה מוקצה זיכרון, וזו הייתה ההצהרה + הגדרה. למשל: $extern\ int\ var = 5;$

<u>הגדרה</u> של משתנה ב-C מתבצעת במקרה בו לא נשתמש במילה $extern$, למשל: $int\ var;$ במקרה זה, למרות שלא הכנסנו ערך, כן הוקצה מקום בזיכרון.

<u>הערה 1:</u> בשונה ממשתנים רגילים, משתנים סטטיים ומשתנים גלובליים מאותחלים להיות 0 במידה ולא בחרנו ערך ספציפי בזמן ההכרזה.

<u>הערה 2</u>: ניתן להצהיר על משתנים סטטיים רק בעזרת ערכים שידועים עוד בזמן קומפילציה.
(לא ניתן למשל להצהיר על משתנה סטטי בעזרת ערך החזרה של פונקציה אחרת)

<u>הערה 3</u>: פונקציות גם יכולות להיות סטטיות. פונקציה או משתנה שמוגדרים עם "$static$" יהיו נגישים רק לפונקציות שהוגדרו באותו הקובץ. (וזה הכל. בשונה מ-$Java$ **לא** מדובר על שיטה או משתנה ששייכים למחלקה ולא לאובייקט כי אין מחלקה/אובייקט ב-C)

רשימות מקושרות

ב-C ניתן לממש רשימות מקושרות בעזרת $Struct$:

```
typedef struct Node {
....
}
```

נייצר בנוסף משתנה גלובלי שיהיה ראש הרשימה:

```
Node *head = NULL;
```

כדי לתפעל את הרשימה נצטרך את השיטות הבאות:

$Push$ - הוספת איבר חדש בתחילת הרשימה.

```
void push(int new_data) {
        Node *new_node = (Node*) malloc(sizeof(Node));
        new_node-> data = new_data;
        new_node-> next = head;
        head = new_node;
}
```

$PrintList$ – הדפסת הערכים של איברי הרשימה.

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

$DeleteList$ – מחיקת הרשימה ע"י שחרור הזיכרון.

```
void deleteList() {
        Node *temp =  head;
        while(temp ! =  NULL) {
                Node *next  =  temp-> next;
                free(temp);
                temp  =  next;
        }
        head  =  NULL;
}
```

<u>הערה</u>: במקרה בו לא נוכל להגדיר את $head$ כמשתנה גלובלי, נשנה את השיטות הנ"ל כך שיקבלו כפרמטר את ראש הרשימה. **חשוב לזכור** שהמבנה שנשלח כפרמטר לשיטה מועתק במלואו ל-$Scope$ של השיטה, וכל שינוי בו יהיה לוקאלי בלבד. לכן נזכור לשלוח לשיטות את הכתובת של $head$ ולא עותק שלו (בעזרת &). באותו אופן, כל חתימת שיטה תקבל כפרמטר $head^{**}$ ולא $head^{*}$.

מערכים דו-ממדיים

מערך דו-ממדי סטטי

$int\ arr[5][7];\ \backslash\backslash\ 5\ rows, 7\ columns$

- זיכרון רציף: "מערך של מערכים" (מחולק ל-5 בלוקים של 7 int-ים)
- מספר השורות ומספר העמודות ידוע בזמן קומפילציה
- יעילות: גישה אחת לזיכרון כדי להגיע לערך ספציפי

<u>הערה</u>: VLA הוא מערך שמספר השורות ומספר העמודות שלו לא ידוע בזמן קומפילציה **וגם** אנחנו מגדירים אותו על ה-$Stack$. מה הכוונה? נניח שיש לנו 2 משתנים n ו-m, הגדרת מערך בצורה הזו: $int\ arr[n][m];$ נחשב ליצירת VLA, מכיוון שנייצר מערך בגודל $n \cdot m$ ב-$Stack$. לעומת זאת, נוכל להגדיר מערך בגודל לא ידוע ב-$heap$ בעזרת $malloc$, וזו התנהגות כן רצויה.

מערך דו-ממדי חצי-דינמי

$int\ ^*arr[5];\ \backslash\backslash\ array\ of\ 5\ pointers\ to\ int$

- כל שורה נמצאת במקום אחר בזיכרון
- מספר השורות ידוע בזמן קומפילציה
- אורך השורות יכול להשתנות בין שורה לשורה
- פחות יעיל: 2 גישות זיכרון כדי להגיע לערך ספציפי
- כדי לאתחל אותו:

```
int *pa[5]; // allocates memory for 5 pointers
for (i = 0; i < 5; i++)
{
        pa[i] = (int *) malloc( 7 · sizeof(int) );
        // pa[i] now points to a memory of 7 ints
}
```

- כדי לבצע השמה של המספר 5 במערך הנ"ל נוכל לבצע את אחד מהבאים:

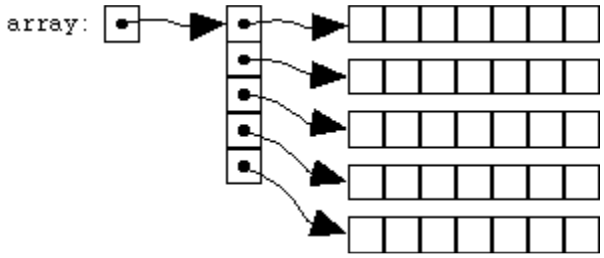
```
pa[i][j] = 5;
*(pa[i] + j) = 5; \\ same
(*(pa + i))[j] = 5; \\ same
*(*(pa + i) + j) = 5; \\ same
```

מערך דו-ממדי דינמי (שלם)

$int\ ^{**}array;\ \backslash\backslash\ pointer\ to\ pointer\ to\ int$

- כל שורה נמצאת במקום אחר בזיכרון
- גודל המערך לא חייב להיות ידוע בזמן קומפילציה
- יעיל אפילו פחות: 3 גישות זיכרון כדי להגיע לערך ספציפי
- כדי לאתחל אותו:

```
array = (int **) malloc(5 * sizeof(int *));
for (i = 0; i < 5; i++)
{
        array[i] = (int *) malloc(7 * sizeof(int));
}
```



- נבצע השמה של המספר 5 במערך הנ"ל באותו אופן כמו מערך חצי-דינמי.

מערך חד-ממדי המייצג מערך דו-ממדי

$int\ ^{*}arr = (int\ ^{*})\ malloc\big(n \cdot m \cdot sizeof(int)\big);$

באופן הזה הקצנו $n \cdot m$ תאים רצופים בזיכרון מטיפוס int , ובעזרת חישוב קל נוכל להתייחס אליו כמערך דו ממדי מהצורה $int\ arr[n][m];$. כדי לגשת ל-$arr[i][j]$, עבור מערך עם m עמודות, נפעל לפי החישוב הבא:

$$arr + (i \cdot m + j)\ \backslash\backslash\ same\ as\ arr[i][j]$$

באופן הזאת אנחנו ניגשים בצורה מהירה יותר לערכים ובנוסף ניתן לממש איטרטור בצורה קלה ויעילה יותר. החיסרון הוא שהקוד פחות קריא.

<u>הערה</u>: בדוגמאות הנ"ל השתמשנו ב-$malloc$ מספר פעמים עבור מערך בודד.
חשוב לזכור לשחרר את כולם בסיום העבודה עם המערך (ולא רק את המערך הראשי).
למשל במקרה של מערך דינמי, יש לנו מערך של מצביעים, וכל תא בו מצביע למערך אחר שהוקצה ע"י $malloc$, לכן נעבור על כל התאים שלו ונשחרר אותם (לפני שנשחרר את הראשי).

מערך של מערכים דו ממדיים

כן, גם את זה אפשר לעשות...

```
double ** mat1 = getMatrix();
double ** mat2 = getMatrix();
//allocate an array of matrices
double *** matrices =
(double ***) malloc(n * sizeof(double**));
matrices[0] = mat1;
matrices[1] = mat2;
```

איך נשלח מערכים דו ממדיים לפונקציות

```
void func( int x[5][7] ) //ok
void func( int x[][7] ) //ok
void func( int x[][] ) //error
void func( int *x[] ) //something else
void func( int **x) //same something else

int foo (char arr_a[][20]); \\ arr_a is a pointer to an array of 20 chars
int bar (char arr_b[20]); \\arr_b is a pointer to a char

Therefore:
sizeof (arr_a) = sizeof (void *);
sizeof (* arr_a) = 20 * sizeof (char);
sizeof (arr_b) = sizeof (void *);
sizeof (* arr_b) = sizeof (char);
```

Compilation and Linkage

כשאנחנו משתמשים בפקודה gcc כדי לקמפל את הקובץ, מתבצעות 3 פעולות (המסומנות באדום):

$hello.c \rightarrow Preprocessor \rightarrow tmpXQ.i \rightarrow Compiler \rightarrow hello.o \rightarrow Linker \rightarrow a.out$

<u>הערה</u>: נשים לב שאחת מהפעולות היא קימפול,

כלומר שימוש ב-gcc הוא לא רק קימפול אלא ביצוע שלושת השלבים הנ"ל.

הקדם מעבד ($Preprocessor$) הוא יישום הפועל על קבצי התוכנית לפני פעולתו של המהדר.

פקודות רלוונטיות עבורו מתחילות בסימן #, כמו למשל $\#include, \#define, \#if, \#else, \#endif$

- $\#include$

הפקודה הזו גורמת לקדם המהדר **להעתיק** תוכן של קובץ מסוים (קובץ מסוג $*.h$ לרוב) לתוך הקובץ שלנו. למשל $\#include\ "foo.h"$ או $\#include\ \langle stdio.h \rangle$.

קובץ מסוג $*.h$ נקרא $header\ file$, ולרוב הוא מכיל הצהרות על פונקציות ומבנים ($Structs, typedef$). במילים אחרות, משמש תיאור למהדר לגבי פונקציות שמוגדרות במקום אחר. (נרחיב על כך בעמודים הבאים)

- $\#define$

קדם המהדר מבצע "העתק-הדבק" לערך של המאקרו, בכל מקום בו רשום השם שלו. למשל $\#define\ FOO\ 1$ **יחליף** כל חלק בקוד שרשום בו FOO ב-1. לכן $int\ x = FOO$ אחרי הקדם מעבד שווה ל-$int\ x = 1$. בנוסף בעזרת $\#define$ ניתן להגדיר פונקציות פשוטות:

$\#define\ SQUARE(x)\ \ x * x$ יחליף את $b = SQUARE(a);$ ב- $b = a * a$.

כדי לייצר $\#define$ עם יותר משורה אחת, נוסיף \ בסוף כל שורה:

```
#define x (5 +\
5)
```

ובצורה הזאת הקדם מעבר ידע שהשורה הבאה היא המשך של השורה הנוכחית.

- $\#if$

נשתמש בו להוספת חלקים מסוימים בקוד לשלב הקומפילציה רק אם מוגדר $define$ ספציפי

```
#define DEBUG
...
#if defined(DEBUG)
// compiles only when DEBUG exists (defined)
printf("X = %d\n",X);
#endif
```

במקרה הנ"ל אם נמחק את $\#define\ DEBUG$,

הקוד שבין $\#if$ ל-$\#endif$ פשוט **ימחק** בשלב של קדם המהדר.

לבסוף נשאר קוד שמוכן לעבור לשלב הקומפילציה.

הערה: נזכור שהקדם מעבד מבצע ממש "העתק-הדבק" ולכן נשתמש בו בחוכמה.

```
#define SQUARE(x) x * x
#define PLUS(x) x + x
b = SQUARE(a + 1);
c = PLUS(a) * 5;
b = a + 1 * a + 1; // b = 2 * a + 1
c = a + a * 5; // c = 6 * a
```

בשני המקרים לא קרה מה שציפינו. נוכל לפתור זאת ע"י הוספה של סוגריים מסביב לערכים.

מהדר ($Compiler$) הוא יישום הפועל על קבצי התוכנית ומתרגם אותם לשפת מכונה. הוא "פולט" קבצים מסוג $object$ (סיומת $*.o$) כל קובץ מסוג $object$ מכיל קוד $opcode$ (שפת מכונה).
לא ניתן להריץ קובץ מסוג $object$ כי נותר תהליך הקישור.

מקשר ($Linker$) מחבר תוכניות מחשב שעברו הידור לשפת מכונה לכדי תוכנית אחת שניתנת להרצה על ידי מערכת הפעלה מסוימת. המקשר יתלונן על שימוש בפונקציות לא מוגדרות, להבדיל מהמהדר שיניח במקרה כזה שהפונקציה הוגדרה בקובץ אחר. במידה ואכן היא הוגדרה בקובץ אחר הוא לא יתלונן. פעולת המקשר היא לקשר בין קריאה של פונקציה לפונקציה עצמה ובנוסף הגדרת סמלים לכתובות זיכרון במחשב (הרחבנו על כך בנאנד).

שגיאות שהמקשר יכול לציין:

1. $Missing\ Implementation$

```
> gcc -Wall -o Main Main.c
Main.o(.text + 0x2c): Main.c: undefined reference to 'foo'
```

2. $Duplicate\ implementation\ (in\ separate\ modules)$

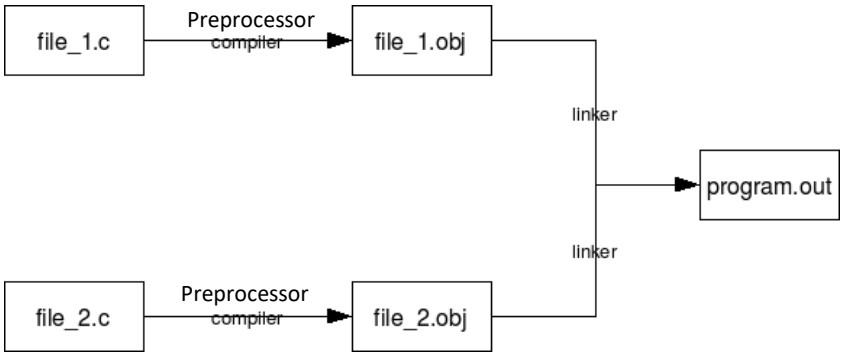
```
> gcc -Wall -o Main Main.o foo.o
foo.o(.text + 0x0): foo.c: multiple definition of 'foo'
Main.o(.text + 0x38): Main.c: first defined here
```

Modules & Header files

שפת C משמשת לכתיבת תוכנות מסובכות מאד. ברור לנו שלא ניתן לתחזק באופן ראוי קובץ אחד המכיל עשרות אלפי שורות קוד, לכן נפצל את הקוד שלנו למודולים.

נניח למשל שאנחנו עובדים על קובץ בשם $file.c$ ואנחנו מעוניינים לפצל אותו למודולים נפרדים.

נחלק את הקוד באופן לוגי לקבצים $file\_1.c$ ו-$file\_2.c$ ותהליך הקמפול יפעל כעת באופן הבא:



<u>הערות</u>:

- אם משנים את אחד הקבצים, נניח $file\_2.c$, אז אין צורך להדר מחדש את $file\_1.c$, מספיק להדר מחדש את הקובץ שהשתנה, ולקשר את קבצי הביניים.
- פעולות ההידור תמיד נעשית לפני פעולת הקישור.

קבצי כותרת $(header\ files)$

אז איך בפועל המהדר והמקשר יודעים איזה קבצים קשורים לתוכנית? איך הם יודעים איזה פונקציות הוגדרו בקובץ אחר ואיזה צריכות לזרוק שגיאת קומפילציה? לשם כך נשתמש בקבצי כותרת.

מבנה של קובץ כותרת $(file\_1.h)$ הינו:

```
#ifndef FILE_1_H
#define FILE_1_H
struct Complex
{
        double _real, _imag;
};
struct Complex addComplex(struct Complex, struct Complex);
#endif /* #ifndef FILE_1_H */
```

כלומר הוספנו הכרזות לפונקציות שנממש בקובץ $file\_1.c$.

בתחילת הקורס למדנו שכדי להשתמש בפונקציה בשלב מסוים של התוכנית,

נצטרך להכריז עליה או להגדיר אותה מעל לשורה הנוכחית. באותו אופן קובץ הכותרת מוסיף את ההכרזות הרצויות לתחילת הקובץ (כי קדם המעבד מעתיק אותן בפועל לתוך הקובץ) ולכן לא תתקבל שגיאה בניסיון להשתמש בפונקציות מקובץ אחר. כעת לאחר שהגדרנו את קובץ הכותרת, נוסיף אותו לקובץ $file\_2.c$ כדי שנוכל להשתמש בפונקציות מתוכו ללא שגיאות קומפילציה:

```
#include "file_1.h"
```

איך נקמפל את התוכנית שלנו במקרה הזה?

ראשית, נהדר את הקבצים:

$gcc\ -Wall\ -c\ file1.c$

$gcc\ -Wall\ -c\ file2.c$

הפעולה תיצור לנו את הקבצים $file1.o$ ו-$file2.o$, שהם קבצים מהודרים,

אך עדיין אינם קבצי הרצה. כעת נבצע את פעולת הקישור:

$gcc\ -Wall\ file1.o\ file2.o\ -o\ program$

פקודה זו תיצור קובץ הרצה בשם $program$.

לחילופין, ניתן לבצע את כל הפעולות האלה בבת אחת:

$gcc\ -Wall\ file1.c\ file2.c\ -o\ program$

הערה חשובה לגבי קבצי כותרת

```
Complex.h:
struct Complex { ... };

MyStuff.h:
#include "Complex.h"

Main.c:
#include "MyStuff.h"
#include "Complex.h"
```

נשים לב שבמקרה הנ"ל, הקדם מעבד יעתיק את כל התוכן מ-$Complex.h$ לתוך $MyStuff.h$,

ולאחר מכן יעתיק את כל התוכן של $MyStuff.h$ לתוך ה-$Main$. לכן במקרה זה התוכן של

$Complex.h$ כבר הועתק ל-$main$ אך אנחנו ביקשנו מקדם המעבד להעתיק אותו שוב.

במקרה זה תהיה שגיאה: $Error: Complex.h: 1: redefinition\ of\ `struct\ Complex'$

לכן יש את המילה השמורה $\#ifndef$ ($if\ not\ defined$) ונשתמש בה כך:

```
Complex.h (revised):
#ifndef COMPLEX_H
#define COMPLEX_H
struct Complex {
...
#endif
```

ובכך הקדם מעבד יעתיק את התוכן רק אם הוא לא הועתק כבר.

Assert

המאקרו $assert$ הוא כלי מועיל לניפוי שגיאות. כדי להשתמש בו, נוסיף $\#include\ \langle assert.h \rangle$ לתחילת הקובץ. בעזרתו נוכל לתפוס בעיות במצב $debug$, בזמן ריצה (ולא קומפילציה)!

נשתמש ב-$Assert$ באופן הבא $assert(< cond >);$, כאשר $cond$ הוא ערך בוליאני.

לדוגמה, אפשר לכתוב כך: $assert(x\ !=\ 0);$. אם התנאי לא מתקיים, התוכנית תדפיס:

$$Assertion\ failed: 'x\ !=\ 0'\ in\ line\ 80\ test.c$$

ותעצור. אם יש שורה שלעולם אין להגיע אליה, אפשר לכתוב: $assert(0);$

אם בכל אופן התוכנית תגיע לשורה אי פעם (כלומר - יש טעות בקוד), התוכנית תדפיס הודעה כזו:

$$Assertion\ failed: '0'\ in\ line\ 100\ foo.c$$

ותעצור. נסתכל על קטע הקוד הבא:

```
void foo(int x)
{
        float y = 1.0 / x;
        ...
}
```

הפונקציה הנ"ל מניחה ש-$x! = 0$, למשל במקרה בו אנחנו בטוחים שלעולם לא נשלח אליה ערך כזה. אך מה קורה אם יש לנו בעיה בקוד ובטעות כן שלחנו את הערך? נוכל לרשום:

```
void foo(int x)
{
        assert(x != 0);
        float y = 1.0 / x;
        ...
}
```

ובמקרה של שגיאה בקוד יהיה לנו קל יותר לדבג אותה. שימוש ב-$Assert$ זו מיומנות חשובה בתכנות, היא מצהירה על הנחות "נסתרות" בקוד ועוזרת בדיבוג.

NDEBUG

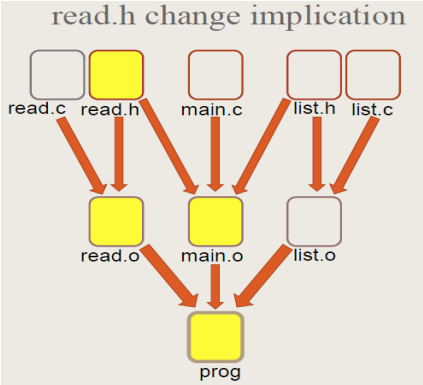
נוכל להשתמש בדגל $NDEBUG$ כדי למחוק שאריות $Assert$ לאחר קומפילציה. איך נעשה זאת?

אם בתחילת הקובץ, לפני ה-$include$ של $Assert$ אנחנו נכתוב $\#define\ NDEBUG$,

כל קריאה ל-$assert$ לא תיכלל בקובץ המקומפל. הקדם מעבד פשוט יתעלם מהם. אופציה אחרת היא לקמפל את הקבצים עם הדגל $DNDEBUG$ ולחסוך הוספה של $\#define\ NDEBUG$ בכל קובץ:

$gcc\ my\_program.c - DNDEBUG - o\ my\_exe$

Make



מה זה? כלי אוטומטי לניהול פרויקטים (לא רק ב-$C/C++$)
מבצע פעולות קומפילציה וקישור בצורה מהירה יותר ומפחית בשגיאות.
התפקיד המרכזי של $make$ לעדכן תוכניות אחרות בצורה חכמה,
בעיקר ע"י בנייה מחדש של קבצים שעברו שינוי בלבד (לפי חותמות זמן)
נניח שביצענו שינוי בקובץ $read.h$, נרצה לעדכן רק את הקבצים
שמשתמשים בקובץ שלנו ותלויים בו. לכן $make$ עובד בעזרת עץ
תלויות, כפי שניתן לראות בתמונה משמאל.
תפקידנו להגדיר ל-$make$ מה לעשות בעזרת קובץ $makefile$:

$\#comment$
$target: dependencies$
$[tab]\ system\ command$
$[tab]\ system\ command$
$\ldots$

נשים לב שחייבת להיות הזחה (tab) לפני כל שורת פקודה ! הרצת $make$ על התוכנית שלנו:

$> make\ prog$
$> make$

$make$ מחפש באופן אוטומטי קובץ בשם $makefile$ או $Makefile$.

אם בחרנו לקרוא לו בשם אחר, נוכל לשנות את ברירת המחדל בצורה הבאה:

$> make\ -f\ myMakefile$

דוגמה לקובץ $makefile$ בסיסי:

$prog:$
$\quad gcc\ -Wall\ square.c\ main.c\ -o\ prog$

(שימו לב להזחה !) ניתן להוסיף משתנים לקובץ ולהשתמש בהם בפקודות, למשל:

$CC = gcc$
$CCFLAGS = -Wall$
$prog:$
$\quad \$(CC)\ \$(CCFLAGS)\ square.c\ main.c\ -o\ prog$

כדי לבנות עץ תלויות, נוכל לרשום לאחר $prog$: את הקבצים שהוא תלוי בהם.

נראה דוגמה לקובץ $makefile$ עם עץ תלויות מלא:

$CC = gcc$
$CCFLAGS = -Wall$
$prog: square.o\ main.o$
$\quad \$(CC)\ square.o\ main.o - o\ prog$
$main.o: main.c\ square.c\ square.h$
$\quad \$(CC)\ \$(CCFLAGS)\ -c\ main.c$
$square.o: square.c\ square.h$
$\quad \$(CC)\ \$(CCFLAGS)\ -c\ square.c$

במקום לרשום ידנית $main.o: main.c\ square.c\ square.h$ וכן הלאה,

נוכל להשתמש בהגדרה כללית יותר, למשל:

```
%.o : %.c
        gcc -c $< -o $@
```

במקרה זה אם קובץ מסוג $object$ (סיומת $.o$) משתנה,

קובץ עם אותו שם ועם סיומת c יתקמפל מחדש גם הוא.

משתנים מיוחדים

$@ - מציין את שם הקובץ שהוגדר כהתאמה. למשל בקוד הנ"ל קובץ בשם $list.o$

$< - מציין את שם הקובץ שתלוי בקובץ הנוכחי. למשל בקוד הנ"ל קובץ בשם $list.c$

$^ - כל שמות הקבצים שתלויים בקובץ הנוכחי, מופרדים באמצעות רווחים.

$? - כל שמות הקבצים שתלויים בקובץ הנוכחי וגם חדשים יותר ממנו, מופרדים באמצעות רווחים.

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

Libraries

ספרייה היא קובץ המורכב מאוסף של קבצי $object$. זה אוסף של פונקציות שימושיות עבורנו שנכתב על ידינו או על ידי מישהו אחר.

למשל ספריית הסטנדרט של C, ספריית ה-$Math$, ספריות גרפיקה וכו'.

ספריות סטטיות מקושרות עם התוכנית שלנו בזמן קומפילציה. כל אפליקציה שמשתמשת בה צריכה להחזיק עותק נפרד של הקוד שלה מה שגורם לבזבוז מקום במערכת הקבצים ובזיכרון המחשב.

סיומת קובץ מסוג ספרייה סטטית ב-$unix$ הינו "$.a$" וב-$Windows$ הינו "$.lib$".

ספריות שיתופיות נטענות בזמן ריצה לתוכנית שלנו. בזמן הפעלת התוכנית שלנו נטענת הספרייה הרצויה לזיכרון. במידה ותוכנית נוספת מעוניינת להשתמש בספרייה, היא משתמשת באותו עותק המצוי כבר בזיכרון ולכן השם "ספריה שיתופית".

סיומת קובץ מסוג ספרייה משותפת ב-$unix$ הינו "$.so$" וב-$Windows$ הינו "$.dll$".

יתרונות ספריה סטטית לעומת שיתופית:

- בלתי תלוי בקיום/במיקום הספריות (מוטמע בפנים)
- בלתי תלוי בגרסאות הספריות השונות
- פחות קישורים בזמן ריצה

יתרונות ספריה שיתופית לעומת סטטית:

- קבצי exe קטנים יותר (לא כוללים את הספריות)
- תהליכים מרובים חולקים את הקוד
- אין צורך בהידור מחדש במקרה של שינוי בספרייה
- אותו קובץ exe יכול לרוץ עם ספריות שונות

ספריות סטטיות

הכלי שבו משתמשים ליצירת קובץ ארכיון המורכב מקבצי $Object$ שונים נקרא 'ar' עבור $archiver$ והוא יכול להיות מופעל באופן הבא:

$ar\ rcs\ libutils.a\ data.o\ stack.o\ list.o$

קבצי ה-$Object$ הנ"ל הם קבצים רגילים שאינם כוללים $main$. הקובץ $libutils.a$ הוא קובץ הארכיון שיווצר. שם ספרייה יתחיל תמיד ב-'lib' ולאחר מכן שם הספרייה שבחרנו. ($utils$ בדוגמה זו)

rcs אלו 3 דגלים:

c - צור ($create$) ארכיון חדש אם הוא עדיין אינו קיים.

r - החלף ($replace$) קבצי $object$ קיימים בקובץ ארכיון קיים בקבצים חדשים בעלי אותו שם.

s - צור טבלת סמלים ($symbol-table$) עבור המקשר ($linker$)

שימוש בספרייה סטטית

לאחר שיצרנו את הספרייה, נצטרך לציין בפני המהדר (קומפיילר) את המסלול ($Path$) לספרייה הסטטית שלנו. למשל עבור קובץ c המציין $\#include\ \langle utils.h \rangle$, נשתמש בפקודה:

```
gcc -Wall -c -I /usr/lib/include/ main.c -o main.o
```

כדי לציין שהכתובת $/usr/lib/include/$ היא מיקום נוסף בו הוא צריך לחפש קבצי ספריות.

במידה ואכן הקובץ $utils.h$ נמצא שם, תהליך ההידור יתבצע ללא שגיאות וייווצר הקובץ $main.o$.

כדי לבצע את תהליך הקישור ($linking$) עם הספרייה שיצרנו $libutils.a$ נשתמש ב:

```
gcc main.o -L /usr/lib/include/ -lutils -o app
```

הדגם L מאפשר להוסיף מסלול נוסף לספרייה. הדגל l (שנמצא לפני שם הספרייה $utils$) מבטא בקשה לקישור לספרייה ספציפית. נשים לב שבזמן בקשת הקישור אנחנו משתמשים בשם הספרייה $utils$ ולא בשם הקובץ $libutils.a$.

ספריות סטטיות ב-$Makefile$

$LIBOBJECTS\ =\ data.o\ stack.o\ list.o$
$libutils.a$: ${$$LIBOBJECTS$}
$ar\ rcs\ libutils.a$ ${$$LIBOBJECTS$}
...
$OBJECTS\ =\ main.o\ another.o$
$CC\ =\ gcc$
$prog$: ${$$OBJECTS$} $libutils.a$
${$$CC$} ${$$OBJECTS$} $-L. -lutils -o\ prog$

הסדר משנה. נוסיף ספריות בסוף, מה הכוונה?

```
gcc main.o -L. -lutils driver.o -o app
```

המקשר מקשר את קבצי הספריות משמאל לימין. אם $driver$ ינסה להשתמש בדברים שהוגדרו ב-$libutils.a$, ככל הנראה תהיה שגיאה. זה מכיוון שקישורים לספריות לא שימושיות "יזרקו" בתהליך הקישור.

ספריות שיתופיות

לעובדה שקיים רק עותק אחד של ספרייה שיתופית בזיכרון יש השלכות לגבי האופן שבו הקוד שלה צריך להיות מקומפל. קוד של ספרייה שיתופית לא יכול להניח הנחות לגבי מיקומו בזיכרון, הוא אמור לפעול כאשר הוא ממוקם בכל אזור בזיכרון. לקוד בעל תכונה כזו קוראים PIC ($position\ independent\ code$) בכדי לייצר קוד כזה מוסיפים לקומפיילר את הדגל $fPIC$:

```
gcc -Wall -fPIC -c file1.c
gcc -Wall -fPIC -c file2.c
gcc -Wall -fPIC -c file3.c
```

בניגוד לספרייה סטטית, ספריה דינמית איננה קובץ ארכיון, האופן בו היא בנויה תלוי בארכיטקטורה הספציפית עליה עובדים ולכן מטלת הרכבת הספרייה מוטלת על ה-$linker$:

```
gcc -shared file1.o file2.o file3.o -o libutils.so
```

הדגל $-shared$ מסמן ל-$linker$ שעליו ליצור ספרייה שיתופית.

שם קובץ הספרייה מתחיל ב-lib כמו ספרייה סטטית, אך טיפוס הקובץ הוא so ($shared\ object$)

<u>הערה</u>: ב-$Windows$ נצטרך להשתמש ב-$\_\_declspec(dllimport)$ ו-$\_\_declspec(dllexport)$

ניתן לקרוא על זה כאן: https://stackoverflow.com/questions/8863193/what-does-declspecdllimport-really-mean

שימוש בספרייה שיתופית

עבור קובץ c המציין $\#include\ \langle utils.h \rangle$, נשתמש בפקודה:

```
gcc -Wall -c -I /usr/lib/include/sharedLibInclude main.c
```

כפי שניתן לראות, זו אותה פקודת קומפילציה בה השתמשנו קודם, רק המסלול ל-$include$ השתנה בכדי שימצא ה-$header$ המתאים. אין צורך ליצור קוד PIC עבור האפליקציה עצמה מכיוון שהיא איננה קוד שיתופי. כעת ניתן לקשר את האפליקציה לספריה השיתופית:

```
gcc main.o -L /usr/lib/mySharedLib/ -lutil -o app
```

אמרנו קודם שה-$linking$ לספרייה שיתופית מתבצע בזמן ריצה. מדוע אם כן אנו מבצעים את ה-$linking$ בשלב הקומפילציה? התשובה היא שלא באמת ביצענו כאן $linking$. רק אפשרנו למקשר לבדוק שקיימת התאמה בין האפליקציה והספרייה מבחינת שמות פונקציות ומשתנים. הקישור עצמו יתבצע בזמן הריצה, כאשר app תטען לזיכרון. בכדי שהמנגנון הטוען את התוכנית לזיכרון ידע למצוא את הספרייה השיתופית שיצרנו, הספרייה צריכה להיות בתיקיה $c:\backslash windows\backslash system$ (בווינדוס) או שנצטרך להוסיף את המסלול אליה למשתנה סביבה מיוחד המכיל מסלולים לספריות הנקרא $LD\_LIBRARY\_PATH$.

הבדלים בין קבצי $Object$ לספריות

קבצי $Object$:

- כל הקובץ מקושר לקובץ ה-exe אפילו אם הפונקציות שבתוכו לא בשימוש כלל.
- 2 מימושים לאותה פונקציה (אותו שם) יגרור שגיאה.

ספריות:

- רק פונקציות שלא נמצאות בקבצי ה-Obj מקושרות.
- במידה והמקשר נתקל ב-2 מימושים לאותה פונקציה (אותו שם) כאשר הראשון בקובץ Obj והשני בספרייה, הראשון יקושר בלבד.
- הסדר חשוב – המהדר עלול לבטל הפניות שלא בשימוש בעת קישור הספריה.

משתנים סטטיים

למילה השמורה $static$ יש 2 משמעויות , בהתאם למיקום שלה.

מחוץ לפונקציה, משתנים/פונקציות שהוגדרו בעזרת המילה $static$ יהיו גלויים בתוך הקובץ בלבד ולא מחוץ לו. $static$ מגביל את ה-$Scope$ לקובץ הנוכחי.

בתוך פונקציה, משתנים סטטיים הם:

- לוקאליים לפונקציה הנוכחית
- מאותחלים **פעם אחת בלבד** לאורך התוכנית
- מספקים אחסון פרטי וקבוע בתוך הפונקציה

(למשל במקרה בו נרצה לבצע מעקב של מספר הקריאות לפונקציה בעזרת $counter$ סטטי)

- אורך החיים שהם הוא לכל זמן הריצה של התוכנית
- כל מי שנמצא ב-$Scope$ בו הוגדר המשתנה, יכול לגשת אליו
- הזיכרון המוקצה להם הוא ב-$Global\ Space$ (גם למשתנים המאותחלים וגם לאלה שלא)
- מאותחלים עם 0 כערך ברירת מחדל
- ניתנים לאתחול רק בעזרת משתנים קבועים

Inter Module Scope Rules

למדנו שתוכניות C לא חייבות להיות בקובץ אחד, להפך רצוי לפצל את התוכנית למספר קבצים (מודולים) כדי לשמור על מודולריות ונוחות עבודה. בנוסף, למדנו שהכרזות על פונקציות ישמרו בקובץ $header$, ואת המימושים שלהן נכניס לקובץ c. כשאנחנו מכריזים על פונקציה, הקומפיילר מבין שהפונקציה קיימת איפה שהוא במהלך התוכנית, אבל עדיין לא מוקצה זיכרון עבורה. התוכנית יודעת מה הארגומנטים של השיטה, סוגי המשתנים שלה, סדר המשתנים וערך ההחזרה. כאשר נגדיר את הפונקציה, רק אז יוקצה זיכרון עבורה.

עבור משתנים, הסברנו על המילה השמורה $extern$ בעמודים קודמים ושם אמרנו ששימוש במילה $extern$ על משתנים מכריז עליהם אך לא מקצה זיכרון עבורם. נסתכל על התוכנית הבאה:

```
extern int var;
int main(void)
{
        var = 10;
        return 0;
}
```

הקוד הנ"ל יגרור שגיאת $linker$, מכיוון שניסינו לבצע השמה של 10 למשתנה var, אך בפועל לא הוקצה שום מקום בזיכרון עבורו. הקומפיילר לא יתריע על כך כי ההכרזה על המשתנה העבירה את האחריות אלינו המתכנתים.

<u>הערה</u>: $extern$ נמצאת כברירת מחדל בהכרזה על פונקציה, בניגוד להכרזה על משתנה.

static vs extern

משתנה סטטי – זמין למודול הנוכחי בלבד

משתנה $extern$ – משתנה גלובלי, ניתן להשתמש בו (ולהגדיר אותו) גם מחוץ למודול

נסתכל על 2 הקבצים הבאים:

```
int y;
static int x;
int z;
int myFunc1() {
        x = 3;
}
```

```
extern int y; // import y (from file1.c )
extern int x; // import x (from file1.c)
int myFunc2() {
extern int z; // import z (from file1.c)
        y = 5;
        x = 3; // linker error
        // x visible only in file1.c (static)
}
```

בקובץ הימני יש שגיאת $linker$ מכיוון שניסינו לבצע השמה למשתנה הסטטי x , שזמין רק למודול הנוכחי בו הוא הוצהר, כלומר רק לקובץ השמאלי. נשים לב גם שבאמצעות $extern\ int\ y$, ייבאנו את המשתנה y מהקובץ השמאלי (שאינו סטטי ולכן זה תקין).

Linkage

תהליך הקישור בין משתנה לבין תא בזיכרון. ה-$Linkage$ מקשר בין שם לבין ישות מחוץ לבלוק הנוכחי. משתנים שאין להם $linkage$ נגישים רק בבלוק הנוכחי.

$External\ Linkage$ – שמות מקושרים לאותו אובייקט **בכל** יחידות התרגום.

לכל הפונקציות והמשתנים הגלובליים יש $linkage$ (אלא אם כן הם סטטיים)

בנוסף הם "חיים" לכל אורך התוכנית.

$Internal\ Linkage$ – שמות מקושרים לאותו אובייקט **באותה** יחידת תרגום.

לאובייקטים המוגדרים כסטטיים יש $internal\ linkage$.

$No\ Linkage$ – האובייקט ייחודי לסקופ שלו.

לכל המשתנים הלוקאליים אין $linkage$. בנוסף הם "חיים" לאורך ה-$Scope$ הנוכחי.

ENUM

מידע ממוספר ($Enumerated$) – קבוצה של קבועים בעלי שם.

שימוש: $enum\ [identifier]\{enumerator\ list\}$ (ה-$identifier$ הוא אופציונלי)

יתרונות: קוד קריא יותר, קוד המועד לפחות שגיאות, עוזר להימנע מ-$Magic\ Numbers$.

```
enum { SUNDAY = 1, MONDAY, TUESDAY, ... };
enum Color {BLACK, RED, GREEN, YELLOW, BLUE, WHITE = 7, GRAY};
enum Seasons
{
E_WINTER,      // = 0 by default
E_SPRING,      // = E_WINTER + 1
E_SUMMER,      // = E_WINTER + 2
E_AUTUMN       // = E_WINTER + 3
};
```

$enum$ הופך מספרים שלמים (int) לקבועים שנוכל להשתמש בהם לכל אורך התוכנית,

$int\ n = RED;$ למשל. אם לא נגדיר ערכים למשתנים ב-$enum$,

הקומפיילר יבצע השמה של ערכים המתחילים מ-0 בסדר עולה.

עד עכשיו יכלנו לבצע פעולה דומה באמצעות $define$, אך ל-$enum$ יש 2 יתרונות:

- הוא פועל לפי חוקי ה-$Scope$ (חי רק איפה שצריך וכו')
- הערכים יכולים למוגדרים לבד על ידי הקומפיילר (אם נרצה).

הבדלים עיקריים בין $macro, const$ ו-$enum$

- מערכים יכולים להיות מאותחלים רק ב-$macro\ defined\ constants$
- $consts$ הם בעלי טיפוס מוגדר, יכולים להחזיק כתובת והם עומדים בחוקי ה-$Scope$.
- $define$ נמצא בסקופ הגלובלי, מועד להתנגשויות.
- $enums$ הם בעלי טיפוס מוגדר (int), מאותחלים לבד ועומדים בחוקי ה-$Scope$.

SWITCH

תחליף לתנאים בוליאניים ארוכים המשווים משתנה למשתנה אחר מסוג int

(נזכור שב-C גם $char$ הוא int).

```
switch (n) {
        case 1: // code to be executed if n = 1;
                break;
        case 2: // code to be executed if n = 2;
                break;
        default: // code to be executed if n doesn't match any cases
}
```

הערך של כל $case$ חייב להיות קבוע. למשל: $case(1+2+3)$ זה בסדר אבל $case(a+b)$ לא.

- אסור שיהיו 2 תנאים ($cases$) אותו דבר.
- השימוש ב-$break$ הוא אופציונלי, אך כל התנאים שיבואו אחרי תנאי אמת (למשל אם $case\ i = True$) יבוצעו אף הם עד שנגיע ל-$break$ כלשהו.
- ביטוי ה-$default$ אופציונלי גם הוא ויכול להופיע בכל מקום. הוא יבוצע כל עוד לא נתקל ב-$break$ לפניו.
- מותר לבצע קינון תנאים.

Enum & Switch

ניתן לשלב את $Switch$ עם $Enum$ מכיוון ש-$Enum$ הוא מסוג int.

```
int main() {
        enum Color {BLACK, RED, GREEN, BLUE};
        enum Color my_color;
        scanf("%d", &my_color);
        switch (my_color) {
                case RED: printf("your favorite color is Red");
                        break;
                case GREEN: printf("your favorite color is Green");
                        break;
                default: printf("you did not choose any color");
        }
        return 0;
}
```

Program Design

$Interfaces$ – הגדרה ($definition$) של קבוצת פונקציות המגדירות ביחד מודול או ספריה.
המשתמש לא צריך לדעת את אופן מימוש המודול ("how") אלא רק מה שמימשנו ("$what$").

$information\ hiding$ – זו אחריות שלנו למנוע גישה מבחוץ למידע פרטי שהוא לא חלק מה-API,
לכן נמנע גישה למשתנים פנימיים של המודול עבור תוכניות חיצוניות. דרך פעולה זו הכרחית
למודולריות. בנוסף, ב-C חשוב להגדיר מי מקצה זיכרון ומי משחרר אותו. אם יש לנו שיטה שמקצה
זיכרון אבל לא משחררת אותו, נציין זאת בדוקומנטציה. נדגים זאת על מחסנית שמחזיקה מחרוזות.
הפונקציונליות של המחסנית תהיה יצירת מחסנית חדשה, הכנסה ($Push$), הוצאה (Pop), לבדוק
האם המחסנית ריקה ($IsEmpty$), ולפנות את כל הזיכרון שהוקצה עבור המחסנית.
ב-C, נגדיר אינטרפייס בעזרת קובץ $header$. נייצר קובץ חדש ונוסיף בו את כל הפונקציות הרצויות:

```
#ifndef _STRSTACK_H
#define _STRSTACK_H
typedef struct StrStack StrStack;
StrStack * StrStackNew();
void StrStackFree(StrStack **stack);
// This procedure * does not * duplicate s
void StrStackPush(StrStack *stack, char *s);
// return NULL if the stack is empty
char *StrStackPop(StrStack *stack);
// Check if the stack is empty
int StrStackIsEmpty(StrStack const *stack);
#endif // _STRSTACK_H
```

אופן המימוש:

- נשתמש ברשימה מקושרת כי הוספה והסרה של ראש הרשימה מתבצע ב-$O(1)$.
- לא נעתיק מחרוזות למחסנית אלא נצביע עליהן (כלומר לא נקצה זיכרון דינמי למחרוזות)

(מימוש השיטות נמצא במצגת של הרצאה מס' 9 או בקבצים המצורפים להרצאה זו)

איך נשתמש ב-$StrStack$ שבנינו?

- נוסיף "$\#include\ "StrStack.h$ בראש הקובץ ,
- לאחר מכן נוכל לייצר משתנה מסוג $StrStack$ באופן הבא:

```
StrStack *stack = StrStackNew();
```

- לאחר מכן גישה לשיטות שמימשנו היא ישירה:

```
StrStackPush(stack, line);
StrStackPop(stack);
StrStackFree(&stack);
```

עקרונות ה-$Interface$

נסתיר את אופן המימוש:

- כימוס מבנה הנתונים בו השתמשנו. המשתמש לא צריך לדעת איך מימשנו אלא איזה פונקציונליות מימשנו. לכן $Pop, Push$ וכו' מגדירים את הפונקציונליות אך לא "מסגירים" את מבנה הנתונים בו השתמשנו.
- לא נספק גישה למבני נתונים שעשויים להשתנות במימוש חלופי.
- מידע גלוי לא יכול להשתנות בהמשך מבלי לאלץ את המשתמש לשנות את הקוד שלו.

נמזער את ה-API שלנו:

- נמקסם את יכולות המודול תוך מזעור האפשרויות המוצעות למשתמש.
- לא נספק פונקציות שאין בהן צורך ממשי רק כי אנחנו יכולים.

נקודות נוספות:

- לא נשתמש במשתנים גלובליים אלא אם כן אנחנו חייבים.
- נמנע מהתנהגויות בלתי רצויות בתוכנית שלנו.
- נשתמש בהערות במקרים בהם אנחנו מניחים משהו כמו למשל הנחות על הקלט (ונכריח זאת אם אנחנו יכולים)

מימוש עקבי:

- נבצע פעולות דומות באותו האופן.

ניהול משאבים:

- נשחרר משאבים באותה "רמה" בה הם הוקצאו עבורנו.
- מי שהקצה את הזיכרון אחראי לשחרר אותו.

Generic Programming in C

מבני נתונים גנריים הם מבני נתונים שיכולים להחזיק משתנים מכל טיפוס (או לפחות מספר טיפוסים)

הטיפוס הספציפי שהמופע של מבנה הנתונים יחזיק, נקבע בזמן ריצה

הכלי העיקרי ש-C מספר עבור מימוש מבני נתונים גנריים הוא $void\ *$

לפני שנתחיל לדבר על המימוש של מבנה נתונים גנרי, נסתכל על הפונקציה $memcpy$.

```
void *memcpy(void *destination, const void *source, size_t num);
```

הפונקציה מעתיקה בלוק בזיכרון בגודל ספציפי מכתובת אחת לכתובת אחרת.

הפונקציה לא מודעת לטיפוס שמועתק. האתגרים העיקריים במימוש זה הם:

- איך לבצע איטרציה על $void\ *$ (אין אריתמטיקת מצביעים עבור $void\ *$)
- איך לגשת לערך שנמצא בתא בזיכרון ($dereference$)

נראה מימוש אפשרי לפונקציה:

```
void *memcpy(void *destination, const void *source, size_t num) {
        char *d = (char *) destination;
        char *s = (char *) source;
        for (int i = 0; i < num; ++i) {
                // pointer arithmetics for char * is done
                // with units of sizeof(char) == 1 byte
                d[i] = s[i];
        }
}
```

נרצה לבנות כעת מחסנית גנרית באמצעות $void\ *$

כל האיברים במחסנית יהיו מאותו טיפוס, אך סוג הטיפוס יקבע בזמן ריצה.

רעיון המימוש הוא להחזיק את המשתנים הרצויים בתוך מבנה בעזרת מצביע מסוג $void\ *$

כדי להוסיף איברים למחסנית, נשתמש ב-$memcpy$ כדי להעתיק מידע מכתובת אחת לאחרת

כאשר 2 הכתובות הוקצאו עם $malloc$.

כדי למחוק איברים מהמחסנית, נבצע $free$ על המצביע $void\ *$ הרצוי.

(גם כאן מימוש השיטות נמצא במצגת של הרצאה מס' 9 או בקבצים המצורפים להרצאה זו)

איך נשתמש במימוש הגנרי?

- נייצר מחסנית חדשה בעזרת:

```
Stack *stack = stackAlloc(sizeof(int));
```

- נשתמש בשיטות שכתבנו כך:

```
push(stack, &i);
pop(stack, &headData);
freeStack(&stack);
```

ניהול שגיאות

ב-$Java$ ובאינטרו למדנו לטפל בשגיאות בעזרת $Try - Catch$ ו-$Exceptions$.

ב-C אין מכניזם לניהול שגיאות. נוכל להפריד את השגיאות שלנו ל-2 סוגים עיקריים:

- <u>באגים</u> – טעויות של המתכנת, כמו למשל חילוק ב-0, שורש לערך שלילי וכו'.

  הדרך הנכונה לטיפול ומניעה בבאגים היא שימוש ב-$Assert$ (שהסברנו בעמודים קודמים)

  אופציה זו **לא** מאפשרת למי שמשתמש בספרייה שלנו לטפל בבעיה כרצונו.

  לא נשתמש ב-$Assert$ כדי לטפל באקספשנים אלא רק בבאגים.

  נזכור שלאחר קימפול עם הדגל $NDEBUG$, כל ה-$Assert$ים נמחקים, לכן נבנה את הקוד כך ששום קריאה לפונקציה / שינוי מהותי יקרה בתוך ה-$Assert$. (כי הוא ימחק גם כן)
- <u>אקספשנים</u> – שגיאות שאינן תלויות במתכנת אלא בסביבה. למשל קובץ שלא יכול להיפתח, שגיאה בניסיון הקצאת זיכרון ועוד.

כיצד נטפל בשגיאות ב-C?

<u>זיהוי:</u>

1. נתפוס את השגיאה לפני שהיא מתרחשת.
2. נשתמש בערכי החזרה של פונקציות כדי לציין שגיאה. (למשל 0 שגיאה ו-1 הצלחה)
3. נשתמש במשתנים גלובליים כדי לציין אם הייתה בעיה ומידע עליה (כמו למשל $errno$)
4. נבנה מכניזם שדומה לאופן הפעולה של תפיסת אקספשנים (לא נרחיב בקורס הזה)

<u>טיפול:</u>

1. נדפיס הודעה שגיאה אינפורמטיבית (אופציונלי)

   ההדפסה תתבצע ל-$stderr$ בעזרת $fprintf$.

   בהרצה נוכל להפריד בין הפלט שמודפס ל-$stdout$ לבין זה שמודפס ל$stderr$ ע"י

   $(myProg > outputFile) > \& \, errorFile$

   באופן הזה הפלט של $stdout$ ישמר בקובץ $outputFile$ (ולא יודפס בטרמינל),

   ובאותו אופן בדיוק $stderr$ ישמר ב-$errorFile$ (ולא יודפס בטרמינל).
2. נעצור את התוכנית (אופציונלי)

   ה-$main$ של התוכנית שלנו צריך להחזיר 0 במידה והתוכנית רצה בצורה תקינה,

   ו-1 (או כל מספר שונה מ-0) במקרה והייתה שגיאה. נמנע משימוש ב-$exit()$ כי זה מעיד שמשהו ממש לא טוב קרה במהלך הריצה. ברוב המקרים בהם ניתקל בבעיה, נשחרר משאבים שהקצנו ונחזיר 1 ב-$main$. זו הדרך הנכונה לסיים תוכנית.

   הספרייה $stdlib.h$ מכילה 2 קבועים $EXIT\_SUCCESS$ ו-$EXIT\_FAILURE$ עם הערכים 0 ו-1 בהתאמה, ומומלץ להשתמש בהם.

איך נתריע על שגיאה?

במקרה של שגיאה, נצטרך לחשוב על דרך יצירתית לדווח על שגיאה ובו זמנית להחזיר פלט.

מכיוון שכל פונקציה בעלת ערך החזרה בודד, נראה 2 פתרונות שונים לבעיה:

- נעדכן על שגיאה באמצעות שינוי משתנה גלובלי ייעודי. למשל משתנה גלובלי בשם err שיעודכן למספר שונה מ-0 במקרה שהייתה שגיאה, ומשתמש שהריץ את הפונקציה שלנו יבדוק את הערך של המשתנה הזה לאחר הקריאה.
- נשתמש בשילוב של ערך החזרה כדי לעדכן על שגיאה, ובנוסף בארגומנט למיקום בזיכרון כדי לבצע השמה של ערך ההחזרה הרצוי. למשל עבור שיטה בשם $divide$ שמקבלת 2 מספרים ומחלקת ביניהם, נשלח גם מצביע למיקום בזיכרון שבו נשמור את הפלט.

משתמש שהריץ את הפונקציה שלנו יבדוק את ערך ההחזרה כדי לדעת אם קרתה שגיאה, ואם לא, הפלט הרצוי נמצא במיקום שהוא שלח לשיטה.

בקורס שלנו אנחנו נעדיף את הפתרון השני על הפתרון הראשון, לעומת זאת הספרייה הסטנדרטית בחרה בפתרון הראשון, לכן פונקציות של הספרייה הסטנדרטית יחזירו 0 במקרה של הצלחה או -1 במקרה של שגיאה. משתנה גלובלי בשם $errno$ מטיפוס int יציין את סוג השגיאה, ונשתמש בשיטה ייעודית כדי להדפיס הודעה אינפורמטיבית מתאימה בהתאם לערך של $errno$.

נשים לב שכדי להשתמש ב-$errno$ נייבא את הספרייה $\langle errno.h \rangle$ עם $\#include$.

```
int main(int argc, char **argv) {
        int fd = 0;
        fd = open( FILE_NAME, O_RDONLY, 0644);
        if(fd < 0) { // Error, as expected.
                perror("Error opening file");
                printf("Error opening file: %s\n", strerror(errno));
        }
        return EXIT_SUCCESS;
}
```

השיטה $perror$ מקבלת מחרוזת, מוסיפה לה נקודותיים בסוף ולאחר מכן מדפיסה את סיבת השגיאה של $errno$. בדוגמה הנ"ל יודפס: $Error\ opening\ file: No\ such\ file\ or\ directory$

בנוסף, אנחנו משתמשים בשיטה $strerror$ שמקבלת את המשתנה $errno$ כארגומנט, ומדפיסה הודעה תואמת בהתאם לטבלה ידועה מראש של שגיאות ממוספרות והסיבה שלהן.

באגים נפוצים

1. מספר ה-$malloc$-ים גדול ממספר ה-$free$-ים.
2. ניסיון להחזיר כתובת למשתנה לוקאלי כערך החזרה של פונקציה.
   ניזכר שמשתנים לוקאליים חיים הפונקציה ונמחקים כשהיא מסיימת,
   ולכן לשלוח מצביע למשתנה לוקאלי זו בעיה.
3. ניסיון לעשות $free$ על מיקום שלא הוקצה ע"י $malloc$.
4. מצביעים שלא אתחלנו ל-$NULL$ וניסינו לגשת לערכים שלהם.
5. שינוי ערך משתנה בתוך פונקציה שיצרה עותק לוקאלי של המשתנה.
   (למשל הדוגמה של $SWAP$ שראינו) נרצה לשלוח כתובת למשתנה.

מצביעים לפונקציות

כפי שהסברנו בעמודים קודמים, כל פונקציה מקבלת מקטע זיכרון ייעודי עבורה בו נשמר הקוד שלה.
ב-C ניתן להגדיר מצביע לפונקציה, כלומר לתא בזיכרון המכיל את תחילת הקוד של הפונקציה.
שם של פונקציה ב-C הוא מצביע לתא הזה בזיכרון, וניתן "לשכפל" אותו למצביעים נוספים.
מכיוון שמדובר במצביע, ניתן לייצר מערך של מצביעים לפונקציות,
להעביר מצביעים בין שיטות או להחזיר אותן כערך החזרה.

מתי זה שימושי

למשל במקרה בו נרצה לממש פונקציה שעוברת על כל האיברים במערך מסוים ומפעילה עליהם פונקציה כלשהי. אם נשלח לפונקציה מצביע לפונקציה הרצויה, נוכל להשתמש באותה שיטה עבור מספר פונקציות שונות בהתאם לבחירה שלנו (למשל מערך של int-ים ופונקציה שמפעילה ערך מוחלט על כולם, או לחילופין פונקציה שמוסיפה לכל איבר במערך ערך כלשהו).
אחרת, היינו צריכים לבנות שיטות ייעודיות או לחילופין להוסיף תנאים בתוך השיטה.

איך נעשה זאת

```
void fun(int a, int b) {
        printf("Value of a is %d\n", a);
        printf("Value of b is %d\n", b);
}
int main() {
        // fun_ptr is a pointer to function fun()
        void (*fun_ptr)(int, int) = &fun;
        // Invoking fun() using fun_ptr
        (*fun_ptr)(10,6);
        return 0;
}
```

בדוגמה זו הגדרנו פונקציה בשם fun ללא ערך החזרה ($void$) המקבלת 2 משתנים מסוג int.
ב-$main$ הגדרנו מצביע חדש לפונקציה בשם fun_ptr , והסינטקס הינו:

ערך_החזרה (*שם_המצביע)(טיפוסי_הארגומנטים) = &שם_הפונקציה;

לאחר מכן כדי לקרוא לפונקציה השתמשנו בסינטקס:

(*שם_המצביע)(הארגומנטים);

<u>הערות</u>:

- בהגדרת המצביע **חובה** להוסיף כוכבית וסוגריים מסביב לשם המצביע (במקרה שלנו $(*fun\_ptr)$)
- אפשר גם לא לכתוב בכלל את טיפוסי הארגומנטים (כלומר סוגריים ריקים).
- לעומת זאת כדי לקרוא לפונקציה נוכל לעשות $(*fun\_ptr)(10,6)$ או $fun\_ptr(10,6)$.
- בהשמה, נוכל לעשות $fun\_ptr = &fun;$ או $fun\_ptr = fun;$

(שני המקרים תקינים, מכיוון ששם הפונקציה היא כתובת למקטע שלה בזיכרון)

מכיוון שפונקציה מצביעה לעצמה, הסינטקס הבא תקני: $(****fun)(1,2);$ (או כל מספר אחר של כוכביות, כולל 0 כוכביות) כל כוכבית "תפנה" לאותו מקום בזיכרון שוב ושוב.

מנגד, הסינטקס הבא שגוי: $(\&fun)(1,2);$ (כי $\&fun$ זה תא בזיכרון שמכיל כתובת לתא של תחילת הפונקציה, אבל זה לא התא עצמו. כלומר $(*(\&fun))(1,2);$ כן יעבוד)

ההבדלים בין מצביעים לפונקציות לבין מצביעים לערכים

- מצביע לפונקציה מצביע לקוד ולא לערך.
- אין צורך בהקצאת זיכרון ($allocate$ או $deallocate$) עבור מצביעים לפונקציות
- שם הפונקציה הוא מצביע לקוד שלה.

איך נשלח מצביע לפונקציה כארגומנט לפונקציה אחרת?

```
int someFunc(int a, int b, void (*func)(int, int)) {
        ...
}
```

הפונקציה $qsort$

נסתכל לדוגמה על הפונקציה $qsort$ שמממשת מיון $QuickSort$.

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))
```

הפונקציה מקבלת מצביע למערך, מספר האיברים בו, גודל כל איבר בודד (למשל $sizeof(int)$) ובנוסף מצביע לפונקציה שיודעת להשוות בין 2 איברים במערך. נראה דוגמה לשימוש:

```
int arr[] = {10, 8, 5, 1, 4, 7};
int asc(void *pa, void *pb) {
        return (*(int *)pa - *(int *)pb);
}
int desc(void *pa, void *pb) {
        return (*(int *)pb - *(int *)pa);
}
/* sort in ascending order */
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), asc);
/* sort in descending order */
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), desc);
```

כלומר הפונקציה $qsort$ מפעילה את הפונקציה הנתונה (במקרה הזה asc או $desc$) כדי לדעת את היחס בין 2 איברים (גדול, קטן, שווה) ובכך למיין אותם מבלי לדעת מה הסוג שלהם או באיזה צורה המשתמש מעוניין למיין את המערך.

מערך של מצביעים לפונקציות

כך נגדיר מערך של מצביעים לפונקציות:

```
void (*fp[3])(struct Shape *ps) = { &draw_square, &draw_rect, &draw_circle};
```

הגדרנו מערך של 3 מצביעים לפונקציות שמקבלות מבנה מסוג $Shape$ ולא מחזירות כלום ($void$)

הסינטקס הזה מסורבל ולכן נעדיף להשתמש ב-$typedef$ כדי להגדיר שם קריא להצהרה:

```
typedef void (*drawfn)(struct Shape *ps);
drawfn fp[3] = {&draw_square, &draw_rect, &draw_circle};
```

הגדרנו שמצביע לפונקציה כזו יקרא בשם $drawfn$, ולאחר מכן יצרנו מערך בגודל 3 מהסוג שלו.

כעת נראה איך ניתן להשתמש במערך בצורה קלה ועמידה לשינויים:

```
void draw(struct Shape *ps) {
        (*fp[ps-> type])(ps); // call the correct function
}
```

נניח שלכל מבנה $Shape$ יש משתנה $type$ המסמל את הצורה שלו.

במקרה שלנו, בהתאם למיקום הפונקציות במערך, 0 יהיה ריבוע, 1 משולש ו-2 עיגול

הפונקציה $draw$ מקבלת מבנה מסוג $Shape$ ומדפיסה אותו על ידי גישה לתא ה-$type$ במערך

והפעלת הפונקציה שנמצאת בתא זה על ה-$Shape$.

מצביע לפונקציה כערך החזרה

כדי להחזיר מצביע לפונקציה דרך פונקציה אחרת, נשתמש בסינטקס הבא:

```
float (*GetPtr1(const char op))(float, float) {
        if(op == '+')
                return &Plus;
        else
                return &Minus;
}
```

$GetPtr1$ היא פונקציה המקבלת $const\ char$ כקלט

ומחזירה מצביע לפונקציה אחרת שמקבלת 2 $floats$ כקלט ומחזירה $float$.

שוב, הסינטקס מאוד מבלבל ולא מובן, לכן נשתמש ב-$typedef$.

```
typedef float (*pt2Func)(float, float);
```

כלומר הגדרנו ש-$pt2Func$ יסמן מצביע לפונקציה שמחזירה $float$ ומקבלת 2 $floats$. כעת:

```
pt2Func GetPtr2(const char op) {
        if (op == '+')
                return &Plus;
        else
                return &Minus;
}
```

הערה חשובה לגבי מספר הארגומנטים

ראינו שאפשר להכריז או להצביע על פונקציה בלי לציין את הארגומנטים שלה: $void\ foo();$

במקרה זה, מבחינת C הפונקציה foo מקבלת מספר לא ידוע של ארגומנטים.

אם נרצה לציין שהפונקציה לא מקבלת ארגומנטים בכלל, נשתמש ב-$void$: $void\ foo(void);$

ננסה לראות דוגמאות למקרים האלה, אך לא ניתן לסגור פה את כל המקרים וההתנהגויות.

מומלץ לנסות מקרים נוספים ולהבין את החוקיות. בגדול, המצביע שאנחנו מגדירים יקבע מתי לזרוק שגיאה או לא. למשל מצביע עם 3 ארגומנטים יזרוק שגיאה אם ננסה להפעיל אותו עם 2 ארגומנטים. אם הפעלת הפונקציה עומדת בתנאים של המצביע, לא תהיה שגיאה אך תהיה התנהגות לא רצויה. למשל אם בפועל נשלחו פחות ארגומנטים, יודפס זבל במקום הארגומנטים החסרים. אם ישלחו יותר מדי ארגומנטים, הפונקציה תתעלם מהם ותשתמש רק במה שהיא צריכה. נראה דוגמאות:

```
void add(int x, int y) { printf("%d + %d = %d\n", x, y, x + y); }
```

- השמה של פונקציה שלא תואמת למצביע (כמו למשל ערך החזרה שונה או מספר ארגומנטים) לא יגרור לשגיאה (יתקמפל וירוץ) אבל הקומפיילר כן יזהה ויזהיר.

```
void (*p_add)(int, int, int) = add; // works
```

- הפעלה של פונקציה עם מספר קטן/גדול של ארגומנטים ממה שהמצביע דורש יגרור שגיאה:

```
void (*p_add)(int, int) = add;
p_add(5); // Error: too few arguments to function 'p_add'
p_add(5, 6, 7); // Error: too many arguments to function 'p_add'
```

- למרות ש-add מקבלת 2 מספרים, הקוד הבא מתקמפל ורץ אבל מדפיס במקום y זבל.

```
void (*p_add)(int) = add;
p_add(5);
```

```
void add1and2() { // undefined number of arguments
        printf("1 + 2 = 3\n");
}
```

- הפונקציה מקבלת מספר לא ידוע של ארגומנטים לכן הקוד הבא תקין (יודפס $1 + 2 = 3$):

```
void (*p_add)(int, int, int) = add1and2;
p_add(5, 6, 7);
p_add(5); // Error: too few arguments to function 'p_add'
```

```
void add2and3(void) { // no arguments
        printf("2 + 3 = 5\n");
}
void (*p_add)() = add2and3;
p_add(5,6,7); // works

void (*p_add)(void) = add2and3;
p_add(5,6,7); // Error: too many arguments to function 'p_add'
```

VLA

VLA $(Variable - Length\ Array)$ זו אופציה נוספת להקצות מערך בגודל משתנה.

החל מ-$C99$ (הסטנדרט איתו אנחנו עובדים בקורס), ניתן להגדיר מערך בגודל שתלוי במשתנה:

$arr[n]$. לפני $C99$ הדרך היחידה לעשות זאת הייתה בעזרת הקצאת זיכרון דינמי $(malloc)$

חסרונות של VLA:

- אין דרך להתמודד עם שגיאות, למשל במקרה בו אין מספיק זיכרון.
  לעומת זאת עם $malloc$ נקבל $NULL$ ונוכל להתמודד עם השגיאה כרצוננו.
- המיקום בו יוקצה הזיכרון לא מוגדר מראש, זה יכול להיות ב-$heap$ ב-$stack$ או בכלל לא.
  - למשל הקומפיילר $GNU\ C$ יקצה את הזיכרון ב-$Stack$.
  - להרבה מערכות הפעלה יש מעט זיכרון $Stack$.
  - לא ניתן לשחרר את הזיכרון.
  - לא ניתן להשתמש ב-$realloc$.
- לא בשימוש ע"י הסטנדרט של $C++$ (שם נשתמש ב-$std::vector$ במקום)
- **אסור לשימוש בקורס שלנו !**

Unions

איגוד $(union)$ הוא טיפוס המאפשר לאחסן סוגים שונים של מידע באותו מיקום בזיכרון.

נגדיר אותו באותו אופן כמו $struct$, אך הוא נבדל ממבנה בנקודה מהותית אחת: בכל נקודת זמן במהלך הריצה, מבנה מאפשר גישה לכל אחד מהשדות שהוגדרו בו. איגוד לעומת זאת מאפשר גישה רק לשדה יחיד מהשדות שהוגדרו בו (האחרון שקיבל ערך בתוכנית). איגודים מספקים דרך יעילה להשתמש באותו מיקום בזיכרון עבור מספר שימושים, וגודל האיגוד גדול מספיק כדי להכיל את הטיפוס הכי גדול מבין המשתנים שלו.

<u>הצהרה</u>:

```
typedef union u_all {
        int i_val;
        double d_val;
} u_all;
```

<u>שימוש</u>:

אם נרצה לבצע השמה של מספר ל-i_val נבצע זאת בדיוק כמו במבנים:

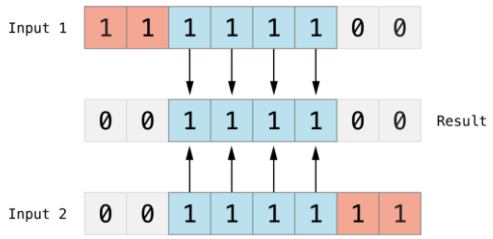
```
u.i_val = 3;
```

כעת נוכל להשתמש בערך i_val לאורך התוכנית. אם הגענו לקטע קוד בו נרצה להשתמש ב-d_val, נבצע השמה רגילה ומאותו רגע נוכל לגשת ל-d_val. בפועל, דרסנו את הערך שהיה ל-i_val ולכן לא נוכל לגשת אליו יותר אלא אם נבצע השמה חוזרת ל-i_val. (ואז d_val לא יהיה נגיש וכו')

Bitwise Operators

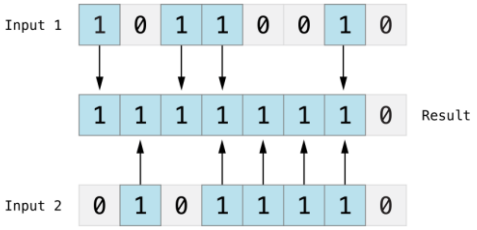
C מאפשרת לנו לבצע מספר פעולות על הייצוג הבינארי של משתנים:

Operator symbol	Operation
&	and
|	or
^	xor
~	One complement (not)
<<	Left shift
>>	Right shift



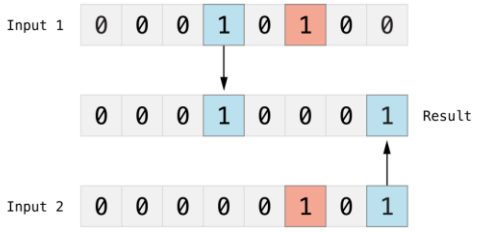
<u>& (Bitwise AND)</u> – מקבל 2 מספרים ומבצע "and" על כל זוג ביטים בנפרד.

```
// a = 4(00000101), b = 9(00001001)
unsigned char a = 5, b = 9;
printf("a & b = %d\n", a & b); // prints 1
```



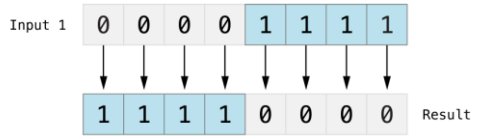
| $(Bitwise\ OR)$ – מקבל 2 מספרים ומבצע "or" על כל זוג ביטים בנפרד.

```
// a = 4(00000101), b = 9(00001001)
unsigned char a = 5, b = 9;
printf("a | b = %d\n", a | b);  // prints 13
```



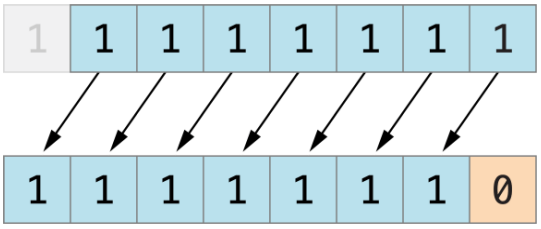
^ $(Bitwise\ XOR)$ – מקבל 2 מספרים ומבצע "xor" על כל זוג ביטים בנפרד.

```
// a = 4(00000101), b = 9(00001001)
unsigned char a = 5, b = 9;
printf("a ^ b = %d\n", a ^ b); // prints 12
```



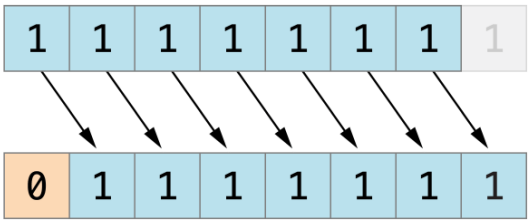
~ $(Bitwise\ NOT)$ – מקבל מספר בודד ומבצע "not" על כל ביט בנפרד.

```
// a = 4(00000101)
unsigned char a = 5;
printf("~a = %d\n", ~a); // prints 250
```



$<<$ $(Bitwisw\ left\ shift)$ – מקבל 2 מספרים ומבצע $left\ shift$ על השמאלי מביניהם. הימני קובע את מספר הביטים להסטה.

```
// b = 9(00001001)
unsigned char b = 9;
printf("b << 1 = %d\n", b << 1); // prints 18
printf("b << 2 = %d\n", b << 2); // prints 36
```



$>>$ $(Bitwise\ right\ shift)$ - מקבל 2 מספרים ומבצע $right\ shift$ על השמאלי מביניהם. הימני קובע את מספר הביטים להסטה.

```
unsigned char b = 8;
printf("b >> 1 = %d\n", b >> 1); // prints 4
printf("b >> 2 = %d\n", b >> 2); // prints 2
```

הערות לגבי פעולות ה-$shift$:

- ההפעלה שלהם חוקית רק על מספרים אי שליליים, עבור מספרים שליליים ההתנהגות לא מוגדרת. כלומר גם $1 << -1$ וגם $-1 << 1$ לא מוגדרים.
- אם המספר מוסט $(shift)$ ביותר מהגודל שלו, ההתנהגות לא מוגדרת. למשל $1 << 33$ לא מוגדר במחשבים בהם מספרים מאוחסנים ב-32 ביט.

Even Number(6)	Odd Number(5)
00000110 (6)	00000101 (5)
00000001 (1)	00000001 (1)
00000000	00000001

איך נבדוק אם מספר הוא זוגי או אי-זוגי? נבדוק את התוצאה של $(x \& 1)$, למה? מספר בייצוג בינארי שנגמר ב-1 הוא אי-זוגי, בנוסף המספר 1 הוא 00000001 בכתיב בינארי. אם התוצאה של & שונה מ-0, אזי שגם במספר הנתון הספרה הימנית ביותר היא 1, ולכן הוא אי-זוגי. אחרת אם הוא זוגי, תוצאה הבדיקה תהיה 0 לכל הספרות.

ראינו שמשתנה מסוג $char$ הוא בגודל $byte$ 1 כלומר 8 ביטים. לכן אם נדע לגשת ולשנות ביט בודד ממנו, נוכל להשתמש ב-$char$ בודד כדי להגדיר 8 משתנים בוליאניים (0 שקר, 1 אמת)

אז איך נבצע מניפולציות על ביטים ספציפיים? נשתמש ב-& ו-> > כדי "לקרוא" ביט:

```
int isZero (int variable, int position) {
        return (variable & (1 << position)) == 0;
}
```

השיטה הנ"ל מקבלת משתנה מסוג int, ומיקום (מ-0 עד 7).

נניח שיש לנו משתנה עם התבנית 00011001 , כדי לדעת אם הביט ה-3 מימין שווה ל-0, נקרא ל: $isZero(variable, 3)$ ונקבל במקרה זה $true$.

נשתמש ב-> > ו-^ (xor) כדי לשנות ביט ספציפי.

```
int toggleBit (int variable, int position) {
        return (variable ^ (1 << position));
}
```

נשים לב ש-$position$ > > 1 יזיז את הספרה הימנית (והיחידה) בייצוג הבינארי של 1 לכיוון שמאל.

אם $position$ הוא 2, הספרה תזוז 2 שמאלה והמספר יהיה 00000100. לאחר מכן נעשה xor בין המספרים ועבור הדוגמה הקודמת נקבל 00011101 (הפכנו את הביט השלישי מימין ל-1)

Variadic functions

פונקציה וריאדית היא פונקציה שיכולה לקבל מספר לא ידוע של משתנים. ראינו את זה למשל ב-$printf$ שמקבלת מחרוזת ומספר לא ידוע של משתנים לאחר מכן. איך נעשה זאת בעצמנו?

נוסיף את $\#includ\ \langle stdarg.h \rangle$ לראש הקובץ שלנו, ואז הצהרת פונקציה וריאדית תראה כך:

```
int printf(const char *format,...)
int countIntegers(int integerNum,...)
```

המשתנה הראשון **חייב** להיות מוגדר, ואחריו נוסיף ... כדי לציין מספר לא ידוע של משתנים.

מכיוון שלא הגדרנו למשתנים האלה שמות, נוכל לגשת אליהם בעזרת משתנה מטיפוס va_list שנמצא ב-$stdarg$, ובעזרת ה-$macros$ הבאים: va_start, va_arg, va_end. נראה דוגמה:

```
#include <stdarg.h>
int sumInts(int argsNum,...) {
        va_list ap;
        int i, sum = 0;
        va_start(ap, argsNum); // now we point to the place
                               // right after the first argument
        for (i = 0; i < argsNum; ++i) {
                sum += va_arg(ap, int);// access current argument and
                                       // move ap to the next argument
        }
        va_end(ap); // free ap
        return sum;
}
//in main:
sumInts(5,1,1,2,3,4); //returns 11
sumInts(2,7,1); //returns 8
```

va_list – טיפוס המאפשר איטרציה על הארגומנטים.

va_start – מתחיל איטרציה על הארגומנטים עם va_list והפרמטר האחרון שהוגדר ידנית עם שם.

va_arg – מחזיר את הארגומנט הנוכחי באיטרציה.

va_end – משחרר את va_list מהזיכרון.

<u>הערות</u>:

- לשיטות וריאדיות חייב להיות לפחות משתנה אחד עם שם.
- אין מנגנון מוגדר כדי לזהות את מספר המשתנים או הטיפוס של המשתנים שנשלחו, לכן אחריותנו לברר את זה מה-$input$ (כדי לדעת כמה לרוץ בלולאה שעוברת על המשתנים) נוכל לעשות זאת על ידי העברת $counter$ כמו בדוגמה, או למשל עבור תבנית נתונה כמו של $printf$, שם אנחנו רושמים במחרוזת "$\%d\ \%s$" ובכך אנחנו מציינים שיש 2 משתנים.

דוגמאות נוספות לפונקציות וריאדיות הן $snprintf$ ו-$vsnprintf$. ניתן לקרוא עליהן כאן:

http://www.cplusplus.com/reference/cstdio/vsnprintf/

Inline Functions

פונקציות $inline$ נוספו ב-$C99$, והן מאפשרות לנו לציין בפני הקומפיילר שמומלץ להעתיק את תוכן הפונקציה לקוד שלנו בזמן קומפילציה כדי לחסוך בקריאות לפונקציה. בין היתר אנחנו חוסכים כך גם בהעברת הארגומנטים ומידע נוסף למחסנית והוצאה של ערך ההחזרה. הקומפיילר יבצע "העתק-הדבק" לקוד במקום הקריאה לפונקציה. מצד אחד אנחנו משפרים את זמן הריצה, אך מצד שני מגדילים את גודל הקובץ הסופי שלנו. איך נגדיר פונקציה להיות $inline$?

```
inline int max(int a, int b) {
        return (a > b) ? a : b;
}
// somewhere in the code ...
a = max(x, y);
// may actually be compiled as ...
a = (x > y) ? x : y;
```

כלומר הוספנו את המילה השמורה $inline$ בתחילת ההכרזה של הפונקציה max, ובכך הודענו למהדר לשקול להעתיק את הקוד שלה לתוך הקוד שלנו.

<u>הערות</u>:

- המילה השמורה $inline$ רק מציינת בפני המהדר לבצע את הפעולה. המהדר יכול להתעלם מנוכחות המילה לטובת אופטימיזציה.
- הפונקציה חייבת להיות מוגדרת באותו מודול, לכן לרוב נגדיר את הפונקציה בקובץ ה-$header$ ולא בקובץ c.

<u>הבדלים בין $macro$ ל-$inline\ functions$</u>:

- פקודת $macro$ לא מבצעת $type\ checking$,
- פקודת $macro$ לא יכולה להשתמש במילה $return$ כי היא תגרום לפונקציה שהשתמשה בה להסתיים (המאקרו מועתק לתוך הקוד, ולכן יהיה פשוט $return$ בשיטה) במילים אחרות פקודת מאקרו לא יכולה להחזיר שום ערך שהוא לא תוצאת החישוב שבוצע על הארגומנטים שנשלחו אליה.
- פקודות $macro$ משמשות כמנגנון "העתק-הדבק" מה שיכול לגרום להתנהגויות בלתי רצויות כמו הדוגמה שראינו בבוחן עם חיבור מספרים כארגומנט.
- שגיאות הקשורות לפקודות $macro$ קשות יותר להבנה.
- מהדרים מסוימים מסוגלים לבצע $inline$ לפונקציות רקורסיביות עד לעומק מסוים. לעומת זאת בפקודות $macro$ לא ניתן להשתמש ברקורסיה.

אופטימיזציה

במצגת 11 שקפים 26-31 ניתן לקרוא על הנושא, הוא הועבר בצורה מאוד מאוד בסיסית בהרצאה.

פתרון Quiz 1

1. התשובה היא $9\backslash n8\backslash n7\backslash n6\backslash n5\backslash n4\backslash n3\backslash n2\backslash n1\backslash n$

2. התשובה היא c
הסיבה לכך היא ש-$(1/2)$ זו פעולת חילוק של 2 משתנים מסוג int,
ולכן התוצאה שלה גם מסוג int, כלומר 0 (מתעגל)
לכן יש $0 \cdot (a+b) + c = c$

3. $Truncation$ – בפעולה הזאת כל מה שנמצא אחרי הנקודה פשוט נחתך.
המספר לא מתעגל למטה / למעלה אלא נשאר רק החלק השלם שלו.

4. התשובה היא $False$
בעליון אנחנו מכניסים לתוך משתנה חיובי ושלם מינוס חצי , ומהפעולה שראינו בסעיף 3 מהמספר -0.5 נשאר רק 0.
בתחתון אנחנו מכניסים -1 למשתנה חיובי בלבד אז זה "ילך אחורה" ויבצע השמה של 4294967295 ואז אנחנו מחלקים ב-2 אז התוצאה היא 2147483647 (ברוב המחשבים)

5. חילוק מספר שלם (int) ב-0 יגרור התנהגות בלתי צפויה.
לכן התשובה היא $Causes\ a\ run\ time\ error$

6. חילוק שבר $(float)$ ב-0 מוגדר וידפיס inf. לכן התשובה היא:
$Prints\ 'SLABC'\ and\ a\ sign\ which\ represents\ infinity$

7. התשובה הראשונה היא $\backslash n$ והשנייה היא $>$
הסיבה לכך היא שלמשל עבור $int\ c = 60$, מה שיושב בפועל בזיכרון זה הייצוג הבינארי של 60. כשאנחנו עושים c% אנחנו בעצם בוחרים לקרוא את הייצוג הבינארי הזה בתור תו לפי טבלת ה-$ASCII$, והתו ה-60 בטבלה זה $>$.. אם היינו עושים $\%d$ זה היה מדפיס 60 כי ככה בחרנו לקרוא את המספר הבינארי שיושב שם. אותה סיבה בדיוק לגבי $int\ c = 10$.

8. הקוד לא יתקמפל וזה כי ה-$define$ צריך להיות:
$\#define\ SWAP(a,b)\ \{int\ tmp\ =\ a;\ a\ =\ b;\ b\ =\ tmp;\}$

9. התשובה היא 9. אם מוגדר המקרו שאחרי $ifdef$, יבוצעו כל ההוראות עד ל-$endif$

10. $Unary\ +$

11. $Will\ be\ stopped\ if\ one\ of\ its\ components\ evaluates\ to\ true$

12. $Prints\ 'Equal'$, הסיבה לכך נעוצה בחוסר היכולת לייצג את כל מספרי ה-$float$.

13. התשובה היא 1485 , דיברנו על כך שכל מה שמוגדר עם $define$ מוחלף לפני קומפילציה באותו אופן כמו "העתק-הדבק". גם במקרה זה מתקיים:
$35 + 67 * 21 + 43$ ומסדר פעולות חשבון נקבל 1485.

14. בגלל שיש סוגריים ב-$define$, ומהסיבה שבקמפול מתבצע העתק הדבק, יתקיים:
$$(2.718 + 1) * 10 = 37.180000$$

15. $Print\ nothings$, יכנס ללולאה אינסופית מאותה סיבה של סעיף 12, לא ניתן לייצג את כל מספרי ה-$float$ ולכן המספר מתעגל למספר הכי קרוב, ומכיוון שזה קורה שוב ושוב אנחנו נתקעים בלולאה.

16. כן, אם נוסיף סוגריים סדר הפעולות ישתנה וקודם כל יחושב החיבור.

17. פתרון ביה"ס:

```
float average(int arr[], int size)
{
  int counter = 0;
  for(int i = 0; i < size; i++)
  {
    counter += arr[i];
  }
  if(size != 0)
  {
    return (float)counter/size;
  }
  return 0.0;
}
```

פתרון Quiz 2

1. בשיטה $printSizeOfArray$, הגודל של a הוא 8 בגלל ש-a הוא משתנה מסוג מצביע, כלומר מסוג $long$ שמכיל כתובת של 64 ביט (8 בתים). לעומת זאת ב-$main$, הגודל של $sizeof(a)$ הוא 16 (4 תאים בגודל 4 בתים כל אחד) בגלל שהמערך הוגדר באותו בלוק ולכן כל שורות הקוד באותו בלוק מודעות לגודל האמיתי של המערך עוד בזמן קומפילציה.
הלולאה תרוץ 4 פעמים כי $sizeof(int) = 4$ ונקבל $\frac{sizeof(a)}{sizeof(int)} = 4$
בנוסף הגדלנו את $a[2]$ ב-1 לכן התשובה היא: $8\backslash n0\backslash n0\backslash n1\backslash n0\backslash n$

2. $sizeof(^*a)$ – הגודל של התא הראשון במערך, 4.
$sizeof(\&a)$ – כתובת בזיכרון מיוצגת על ידי $long$ לכן הגודל הוא 8.
$sizeof(a)$ – 40, 10 מקומות של 4 כל אחד.

3. העברה של $Struct$ כארגומנט לפונקציה יוצר עותק חדש בזיכרון. העותק נוצר ב-$Scope$ של הפונקציה ולכן היא מודעת לגודל האמיתי של המערך – 16. בנוסף, הגדלת הערך $as.a[2]$ ב-1 בוצעה על העותק ולא על המבנה המקורי, ולכן הערכים במבנה המקורי נשארו כולם 0.
לכן התשובה היא: $16\backslash n0\backslash n0\backslash n0\backslash n0\backslash n$

4. הביטוי שקיבלנו מחזיר את הכתובת של התא ה-i במערך x. התשובה הנכונה היא $x + i$ ומה שבפועל קורה כאן זה אריתמטיקת מצביעים. עבור מצביע p ומערך a מתקיים:
$p = a;\ // \ same\ as\ p = \&a[0]$
$p++;\ // \ p = a + 1 = \&a[1]$
כלומר הפעולה $p + 5$ "תקפוץ" 5 תאים במערך באופן הבא: $a + sizeof(a[0]) \cdot 5$.
(נזכור ש-x הוא מצביע לתא הראשון במערך)

5. כן, ה-$main$ כתוב בצורה נכונה, הוא מעביר מצביע למחרוזת שנגמרת ב-$\backslash 0$,
הבעיה היא בשיטה $printLastLetter$. התוכנית תדפיס תו לא ידוע.
ניתן להזיז כוכבית אחת בלבד כדי לתקן את השגיאה, באופן הבא:
$printf("\%c\backslash n", ^*(^*str + strlen(^*str) - 1));$
למה עכשיו זה עובד? נזכור שמחרוזת זה מערך של תווים, לכן המשתנה $buff$ ב-$main$ הוא בעצם מצביע, ולכן שלחנו לשיטה את הכתובת של המצביע. המשתנה str הגדיר עצמו כמצביע בעזרת כוכבית אחת, ובעזרת הכוכבית השנייה הוא ניגש לתא בזיכרון של המצביע, כלומר לתא הראשון במערך. לכן str הוא מצביע לתא הראשון במערך.
מעבר לזה בוצעה שם אריתמטיקת מצביעים + גישה לערך עצמו של המצביע str ושל המצביע $(^*str + strlen(^*str) - 1)$ על ידי כוכביות.

6. ידפיס זבל, התנהגות לא מוגדרת

7. חתימת השיטה $strcat$ עונה על השאלה הזו:

$$char\ ^{*}strcat(char\ ^{*}dest, const\ char\ ^{*}src)$$

המילה השמורה $const$ מבהירה שהמשתנה src נשאר ללא שינוי. אם גם $dest$ היה נשאר ללא שינוי גם הוא היה מתחיל ב-$const$. לכן נסיק כי השיטה מחברת את src ל-$dest$.

התשובות הנכונות הן:

$*\ The\ array\ A\ overflows\ after\ the\ strcat\ operation$

$*\ After\ strcat\ is\ called\ and\ if\ the\ program\ did\ not\ crash,$

$there\ is\ a\ continuous\ place\ in\ memory\ with\ the\ following\ char\ sequence:$

$'A\ studentfail\ the\ exam\backslash 0'$

8. $\boldsymbol{sizeof}(\text{"}\boldsymbol{a}\text{"})$ - "a" יוצר מערך $\{a,\backslash 0\}$ ולכן הגודל הוא 2.

$\boldsymbol{sizeof}(^{*}\boldsymbol{str})$ - ניגשים לערך של המצביע str, כלומר התא הראשון במערך $\{a,\backslash 0\}$ לכן 1.

$\boldsymbol{sizeof}(\boldsymbol{a})$ - המשתנה a הוא מסוג $char$ ולכן הגודל שלו הוא 1.

$\boldsymbol{sizeof}('\boldsymbol{a}')$ - נזכור שב-C , תו הוא int (לפי טבלת $ASCII$). אם נגדיר תו כמשתנה מטיפוס $char$, הגודל שלו יהיה 1 כי ניתן לייצג את כל התווים בעזרת בית אחד. אך אם לא הגדרנו שמדובר בטיפוס $char$, ברירת המחדל int ולכן הגודל הוא 4.

$\boldsymbol{sizeof}(\boldsymbol{str})$ - המשתנה str הוא מצביע, כלומר מטיפוס $long$ ולכן יודפס 8.

$\boldsymbol{sizeof}(\boldsymbol{arr})$ - באותו אופן כמו הראשון, אורך המערך $\{a,\backslash 0\}$ הוא 2.

9. הקוד פועל באופן תקין והפלט הוא:

$$5\ is\ the\ last\ digit\ of\ 345\backslash n3\ is\ the\ last\ digit\ of\ 543\backslash n$$

השינוי המוצע לשורה 6 יגרור התנהגות לא רצויה בשורה 15. הסיבה היא שהמשתנה $retVal$ מאותחל פעם אחת בלבד עם $NULL$ (כי אתחול משתנה סטטי קורה בזמן קומפילציה), ולכן הקריאה השניה ל-$printLastDigit$ דרך השיטה f תהיה כשהמשתנה הסטטי בשיטה שווה ל-345 מהריצה הקודמת. מהסיבה הזו, התנאי $(retVal\ ==\ NULL)$ הוא תנאי שקר וכפועל יוצא המשתנה לא יאותחל עם ה-str הנתון כפי שהיינו מצפים. על כן $retVal$ הוא 345 לאורך כל ריצת התוכנית. (לעומת זאת החישוב של הספרה האחרונה יוצא נכון כי אנחנו עובדים על המשתנה str ולא על $retVal$)

השינוי המוצע לשורה 23 יגרור פלט לא נכון מהסיבה המתוארת לעיל ולא בגלל השינוי.

10. התשובה הנכונה היא:

$$Hello\backslash nclass\backslash nBye\backslash nclass\backslash nclass\backslash n$$

דבר ראשון שצריך להבין זה שבעזרת $fscanf$ אנחנו קוראים מהקובץ עד שנגיע לרווח. כלומר בכל פעם מילה אחת. (כדי לקרוא שורה שלמה נשתמש ב-$\%[^\backslash n]$ במקום ב-$\%s$) בנוסף, תנאי הלולאה שלנו אומר "תרוץ כל עוד לא ניסינו (בעבר) לקרוא את השורה האחרונה בקובץ" ולכן הלולאה רצה סיבוב אחד יותר ממה שהיינו מצפים. במקום לעצור כשהיא מגיעה ל-EOF, היא תמשיך סיבוב אחד נוסף ורק אז יתקיים תנאי הלולאה. לכן אנחנו מדפיסים את המילה האחרונה ($class$) שוב. (כדי להבין למה הדפסנו שוב את המילה $class$ ניתן לקרוא שוב את ההערה בסעיף של $scanf$ בסיכום)

11. השיטה $fseek$ מתקדמת 3 צעדים מהמיקום הנוכחי בקובץ.

השיטה $putc$ כותבת תו בודד ל-$Stream$ הנתון. השיטה $getc$ קוראת תו בודד ממנו. התשובה הנכונה היא $Hobtn$ מכיוון שבכל שימוש ב-$getc$ אנחנו קופצים מקום 1, ובכל שימוש ב-$fseek$ אנחנו קופצים עוד 3. לכן הקפיצות הן של 4 בין אות לאות.

12. $gcc\ -c\ test.c$

13. $gcc\ test.o\ -o\ test$

.14

```
#include < stdio.h >
#include < stdlib.h >
#include < string.h >

int isPalindrom(const char *str) {
  int inputlen = strlen(str);
  int i = 0;
  for( ; i < (inputlen/2); i++) {
    if(str[i] != str[inputlen - i - 1])
    {
      return 0;
    }
  }
  return 1;
}

int main() {
  int n;
  scanf("%d",&n);
  char *str = (char *) malloc(n * sizeof(char) + 1);
  scanf("%s",str);
  if(isPalindrom(str))
  {
    printf("%s",str);
  }
  free(str);
  return 0;
}
```

.15

```
void pairwiseSwap()
{
        Node *current = head;
        while(current != NULL && current-> next != NULL)
        {
                int temp = current-> next-> data;
                current-> next-> data = current-> data;
                current-> data = temp;
                current = current-> next-> next;
        }
}
```

פתרון Quiz 3

1. $stdlib.h$

2. $int\ ^*p\ =\ (int\ ^*)\ malloc(12);$

3. 440 - השורה הראשונה מגדירה מצביע למערך בגודל $MAXCOL$ מסוג int
לכן $sizeof(^*p)$ יהיה $MAXCOL \cdot 2$ (כי 2 זה הגודל של int כפי שהוגדר בשאלה)
לכן התשובה היא $10 \cdot 22 \cdot 2 = 440$

4. $free(p)$

5. $True$ – מכיוון שהזיכרון שמוקצה על ידי $malloc$ נמצא ב-$Heap$ ולא ב-$Stack$.
כשהפונקציה תקרוס, זיכרון המחסנית ($Stack$) ישוחרר לבד,
אך הזיכרון שהוקצה ב-$Heap$ לא וזו אחריות שלנו לשחרר אותו.

6. 3033 – אחרי השימוש ב-$malloc$, נתון ש-p מצביע לתא 3033 בזיכרון, כלומר הערך שיושב בתא הזה הוא 3033. השימוש ב-$free$ לא מאפס או מוחק את הערך שיושב שם, אלא מעדכן את מערכת ההפעלה שהמקום בזיכרון שמתחיל בתא 3033 פנוי להקצאה.

7. $False$
8. $True$
9. $True$
10. $True$
11. $True$

12. $Incorrect\ statement$ – הכוכבית שאחרי $typedef$ לא נמצאת במקום הנכון.

13. $False$

14. $declaration\ of\ the\ function\ f\ which\ returns\ a\ pointer$
$to\ an\ array\ of\ pointers\ to\ functions\ that\ return\ a\ char$

15. $f\ is\ a\ pointer\ to\ a\ function\ which\ returns\ nothing\ and\ receives\ as\ its$
$parameters\ an\ integer\ and\ a\ pointer\ to\ a\ function$
$which\ receives\ nothing\ and\ returns\ nothing$

16. לפי סדר השורות:

$Element\ ^{*}arrToSort$

כי בפונקציה אנחנו משנים את המערך, הוא לא $const$ ולכן לא נצהיר עליו ככזה

(כלומר לא נשתמש ב-$ConstElement$ אלא רק ב-$Element$)

$int\ arrSize$

$int\ (^{*}less)(ConstElement, ConstElement)$

$arrToSort[it + 1], arrToSort[it]$

את שאר השורות מסיקים לפי השמות שהם השתמשו בהם במהלך הקוד.

Practice Exam 1, Version 2 פתרון

שאלה 1

```
int main()
{
  // get user inputs
  int numOfDigits = 0;
  scanf("%d",&numOfDigits);
  char *longNumber = (char *) malloc(numOfDigits * sizeof(char));
  scanf("%s",longNumber);
  // Update sum
  int outputSum = 0;
  char *tempPointer = longNumber;
  for(int i = 0; i < numOfDigits; i++)
  {
    outputSum += (int) tempPointer[0] - '0';
    tempPointer++;
  }
  // Print output and free memory allocation
  printf("%d\n",outputSum);
  free(longNumber);
  return 0;
}
```

שאלה 2

```
void deleteAlt()
{
  Node *current = head;
  while(current != NULL && current->next != NULL)
  {
    Node *toDelete = current->next;
    current->next = toDelete->next;
    free(toDelete);
    current = current->next;
  }
}
```

Practice Exam 1, Version 3 פתרון

שאלה 1

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

int main()
{
  // Read in the length of the input sentence
  int sentenceLength = 0;
  scanf("%d\n",&sentenceLength);
  // Read in the text
  char *sentence = (char *) malloc((sentenceLength + 1) * sizeof(char));
  fgets(sentence,sentenceLength + 1,stdin);
  // Count the number of double letters
  int numOfDouble = 0;
  for(int i = 0; i < sentenceLength - 1; i++)
    if(sentence[i] == sentence[i + 1])
      numOfDouble++;
  // Print the text from the n'th character to the 2n'th character (inclusive)
  if(numOfDouble == 0)
    return 0;
  char *temp = sentence + numOfDouble;
  for(int i = 0; i <= numOfDouble; i++)
    printf("%c",temp[i]);
  free(sentence);
}
```

שאלה 2

```
void decode(char *text) {
  while(text[0] != '\0') {
    // Try to extract a number
      // First, Count number of digits
    int numOfDigits = 0;
    char *temp = text;
    while(0 <= temp[0] - '0' && temp[0] - '0' <= 9) {
      numOfDigits++;
      temp++;
    }
      // Then parse the substring of the number
    char *numberSubstring = (char *) malloc((numOfDigits + 1) * sizeof(char));
    strncpy(numberSubstring, text, numOfDigits);
    (numberSubstring + numOfDigits)[0] = '\0';
    int number = atoi(numberSubstring);
    text += numOfDigits;
    // If extracted number is 0, just print the current char
    if(number == 0)
      printf("%c", text[0]);
    else // Else, print the next char n times
      for(int j = 0; j < number; j++)
        printf("%c", text[0]);
    // Continue to the next char
    text++;
    free(numberSubstring);
  }
}
```

Practice exam 1, version 4 Quiz פתרון

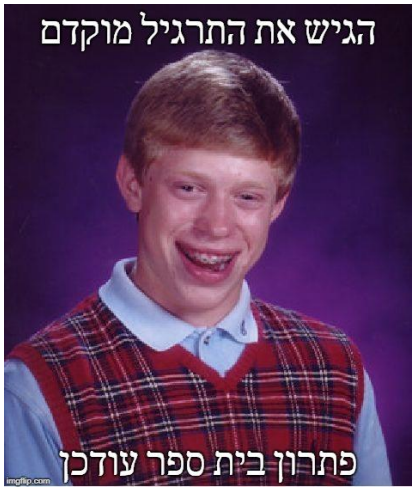
שאלה 1

```
int strLength = 0;
  scanf("%d",&strLength);
  char *str = (char *) malloc((strLength + 1) * sizeof(char));
  scanf("%s",str);
  for(int i = strLength - 1; i >= 0; i--)
  {
    printf("%c",str[i]);
  }
```

שאלה 2

```
void pairwiseSwap()
{
        Node *current = head;
        while(current != NULL && current-> next != NULL)
        {
                int temp = current-> next-> data;
                current-> next-> data = current-> data;
                current-> data = temp;
                current = current-> next-> next;
        }
}
```

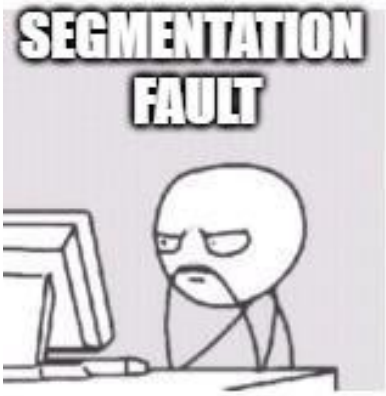
בונוס

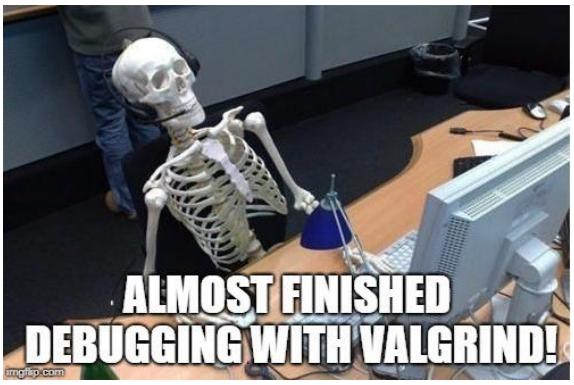

הגיש את התרגיל מוקדם
פתרון בית ספר עודכן


לא שיניתי כלום!
אתמול זה עבד!


stack overflow
bug
me


EVERYONES OUT PARTYING
AND IM JUST SITTING HERE DEBUGGING


SEGMENTATION FAULT


ALMOST FINISHED DEBUGGING WITH VALGRIND!


Dude help me I tried everything and it doesn't work
me adding semicolon to his code
classmate