

תרגול מס' 1

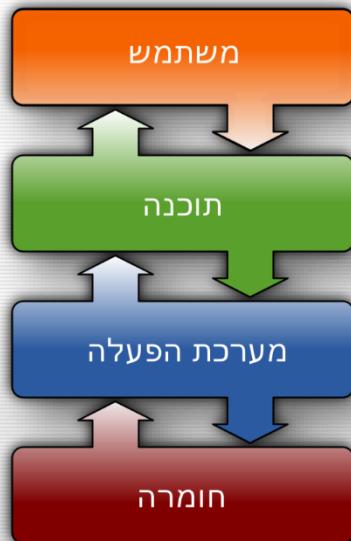
- ❖ מערכת הפעלה אינטגרטיבית
- ❖ תכניות סטנדרטיות בAINI

עבודה בסביבת UNIX

- ❖ מהי מערכת הפעלה?
- ❖ מערכת הקבצים ב-UNIX
- ❖ עבודה בחילון Shell
- ❖ הרשאות והרכבת קבצים

מהי מערכת הפעלה?

- מערכת הפעלה מהוּה **שכבה ביןיהים** בין התוכנה
- מאפשרת **גישה נוחה** למחשב עבור משתמשים וმתכנים
 - עבור המשתמש: מספקת משקל נוח למחשב ושירותים בסיסיים
 - עבור המתכנת: מספקת ממשקים לניהול זיכרון, קבצים והתקני חומרה נוספים



- דוגמאות למערכות הפעלה:
 - Windows –
 - Ubuntu –
 - OSX –
 - Android –
 - iOS –

מהי UNIX?

- UNIX הינה משפחה של מערכות הפעלה
- יתרון מערכות הפעלה ממשפחה זו הוא בקיומו של סטנדרט נוח עבור השירותים אותן היא מספקת (הקרויה POSIX)
- רוב מערכות הפעלה המוכרות לכם עומדות בסטנדרט של UNIX
 - המפורסמת ביותר כיום היא Linux, לה יש גרסאות רבות: RedHat, Debian וngzrootihen.
 - OS מבוססת על Unix
- בפרט - מערכת הפעלה על שרת ה-3אCs עליו נבדקים תרגילי הבית היא (RedHat 7.9).Linux



Shell

- Shell (קליפה) הוא כינוי לתוכנה המ קישרת בין המשתמש לבין מערכת הפעלה
 - בדרך כלל המונח בשימוש עבור shell טקסטואלי בטרמינל (חלון שחזור)
 - העבודה ה-Shell מתבצעת על ידי קבלת פקודה מהמשתמש, ביצוע החלטות טקסט בפקודה בהתאם לתוכנות ה-Shell ולבסוף שליחת הפקודה המעובדת למערכת הפעלה
- דוגמאות ל-Shell:
 - Bash (ברירת המחדל ברוב הפצצות הלינוקס)
 - Powershell (shell מתקדם עבור windows)
 - Zsh (ברירת מחדל מחשבי Apple)
- בקורס זה נלמד Bash
- מלבד ממשק בסיסי לביצוע פקודות ה-Shell **מקל עליינו את העבודה** בעזרת תוכנות מתקדמות, לדוגמה ההשלמה האוטומטית המתבצעת ע"י לחיצה על Tab ה-Shell יאפשר לנו להגדיר **קיצורים, חוזרים, לוחzas** על פקודות, להתייחס **למספר קבועים בנותות** ולהגדיר **משתנים** אשר נוכל להשתמש בהם בפקודות

Shell

rzf הפעולה עם:

1. ה All-Shell-אפשר או "בקשת" מאיתנו להכניס פקודה (prompt)
2. נכניס פקודה ונלחץ enter
3. ה All-Shell-יעביר למערכת הפעלה את הפקודה ויבצע אותה.
4. יודפס הפלט של הפקודה (במידה ויש)
5. ה All-Shell-בקשת להכניס פקודה חדשה.

```
mtm@cs13.cs.technion.ac.il:22 - Bitvise xterm - mtm@cs13:~/public/2122b
[mtm@cs13 2122b]$ pwd
/home/mtm/public/2122b
[mtm@cs13 2122b]$ ls
[mtm@cs13 2122b]$ mkdir ex0
[mtm@cs13 2122b]$ ls
ex0
[mtm@cs13 2122b]$ ls -l
total 4
drwxr-xr-x 2 mtm users 4096 Jan 13 18:28 ex0
[mtm@cs13 2122b]$
```

מערכת הקבצים ב-UNIX

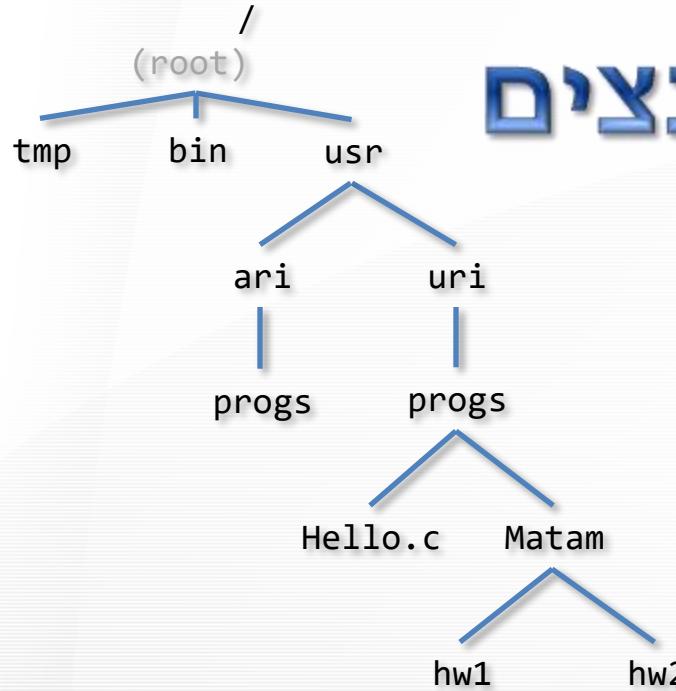
- מערכת הפעלה אחראית בין השאר על **ארגון הקבצים במחשב**
- מערכת הקבצים ב-UNIX מורכבת מקבצים ותיקיות – בדומה ל-Windows
 - **קובץ** הוא אוסף סדר של נתונים
 - **תיקיה** (Directory/Folder) מאפשרת שמירת מספר קבצים ותיקיות נוספות בצורה מסודרת



מערכת הקבצים

- לכל קובץ או תיקיה יש שם
 - אין מוגבלות על מבנה השם (לא חיבתלהיות סימנת)
 - מקובל שם קובץ הוא מהצורה <name.><extension>
 - למשל c. test עברו קובץ מקור בשפת C
 - אין ב-UNIX סימנת מיוחדת לקובץ הריצה (במו exe ב-Windows)
- כל קובץ או תיקיה נמצאים בתוך תיקיה כלשהו
 - מלבד התיקיה הראשית

התיחסות לקבצים

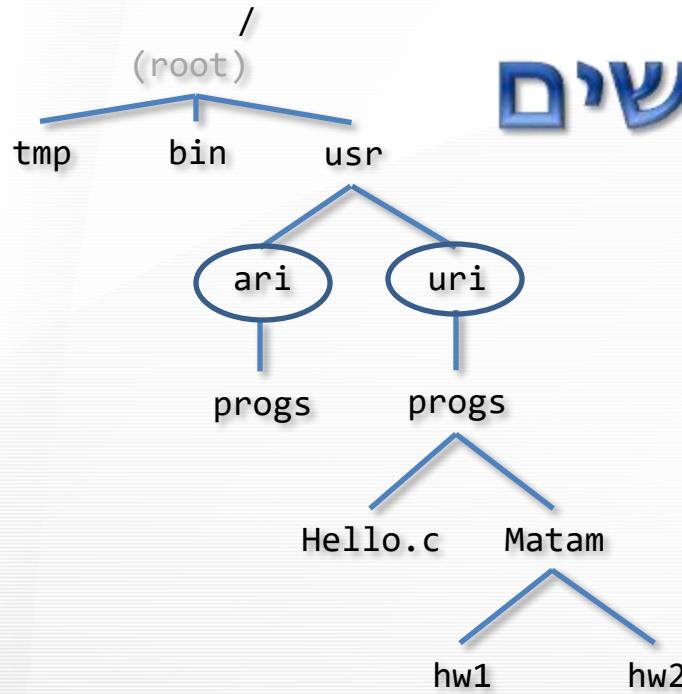


- כדי להתייחס לקובץ יש להשתמש בשמו
- **שורש מערכת הקבצים נקרא "/"**
- ניתן להתייחס לקובץ ע"י **שם המוחלט**:
 - **/usr/ari/progs**
 - התו "/" משמש כסימן מפריד להגדלת מסלול במערכת הקבצים

```
[mtm@cs13 ~]$ pwd  
/home/mtm
```

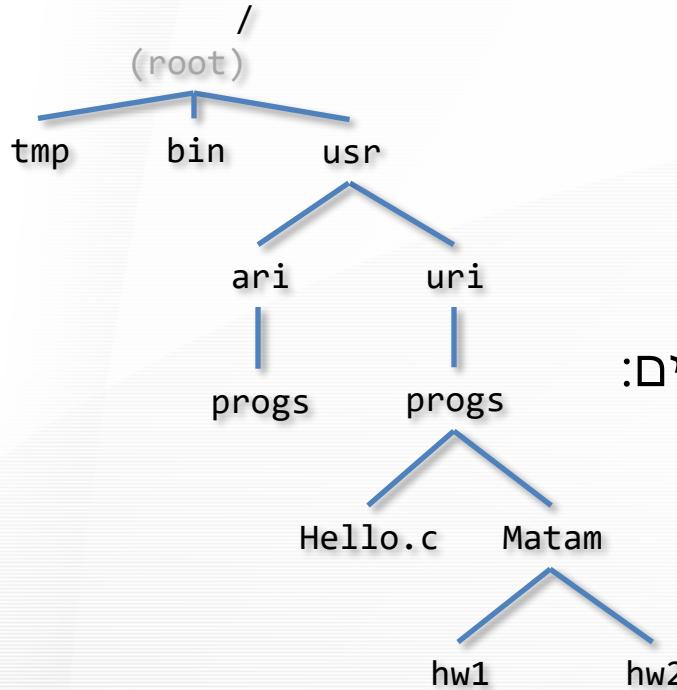
- בכל חיבור ובכל זמן קיימת תיקיה המוגדרת **תיקיה הנוכחיית**
- ניתן להתייחס לקובץ על ידי **שם ייחסי** מהתיקיה הנוכחיית:
 - **Hello.c** אם התיקיה הנוכחיית היא **/usr/uri/progs**
 - **/usr/uri/progs>Hello.c** אם התיקיה הנוכחיית היא **/usr/uri/progs**

ריבוי משתמשים



- UNIX היא מערכת הפעלה **מרובת משתמשים**
- לכל משתמש במערכת קיימת **תיקית בית**
- בהתחברות למערכת התקינה הנוכחות היא **תיקית הבית** של המשתמש

קיצורים

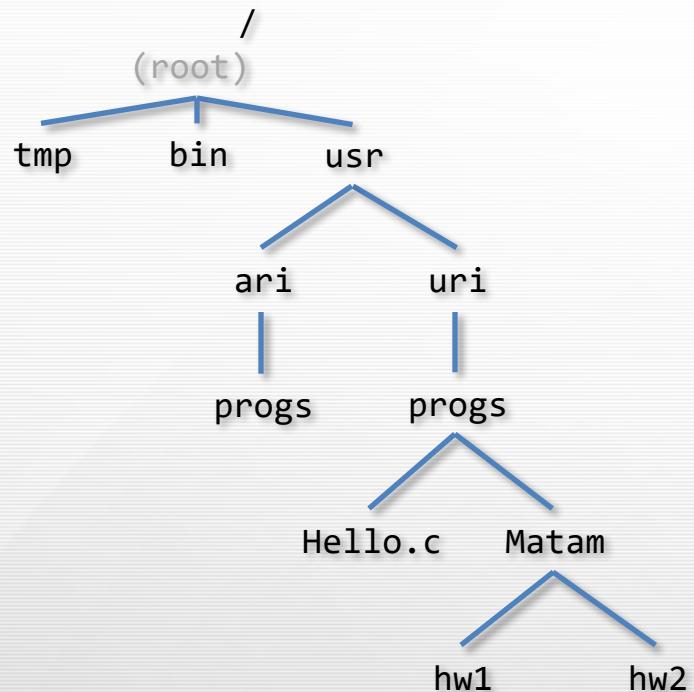


- בנוספּ מוגדרים הקיצורים הבאים עבור שמות קבצים:
 - נקודת אחת מייצגת את **התיקיה הנוכחית**
 - .. שתי נקודות מייצגות את **תיקיות האב**
 - התיקיה בה נמצאת התיקיה הנוכחית
 - ~ - קיצור **תיקיות הבית** של המשתמש
 - user~ - מייצגת קיצור לתיקיות הבית של המשתמש user
 - למשל mtm~ היא תיקית הבית של הקורס בשורת 3als

- ניתן להשתמש בקיצור * כדי להתייחס למספר קבצים בחת אחת ע"י שימוש בתבניות, למשל:
 - * מתייחס לכל הקבצים
 - *.txt מותאם לכל הקבצים ששמות מסתיים ב-.txt.

דוגמה

- נתון מבנה של מערכת קבצים לדוגמה ב-XINU:
למה מתייחסים paths הבאים?



/usr/uri/progs/Matam
progs
~/progs
. ./hw1
.. ../hw2
~ari/progs
~mtm/public/1011a/ex1
~/progs/*

הרשאות קבצים

- לכל קובץ יש הרשאות הקובעות למי מותרת הגישה לקובץ ולצורך אילו פעולה.
- ישנו **3 סוגי משתמשים**:
 - .1. **User** - בעל הקובץ.
 - בעל הקובץ הוא המשתמש שיצר את הקובץ במערכת.
 - .2. **Group** - משתמש אשר שייך לקובוצה של בעל קובץ
 - הגדרת קבוצות משתמשים הינה מחוץ לחומר הקורס
 - .3. **Other** - שאר המשתמשים במערכת
- לכל סוג משתמש יש **3 הרשאות שונות**:
 - .1. **Read** - מאפשרת קריאת תוכן הקובץ (כולל העתקתו)
 - .2. **Write** - מאפשרת שינוי או מחיקת הקובץ הקיימים
 - .3. **Execute** - מאפשרת להריץ את הקובץ

```
drwxr-xr-x 2 mtm users 4096 Mar 17 16:06 ex0
```

הרכבת קובץ

- הרכבת קובץ מתרבצת על ידי הוספת "./". לפני שם הקובץ. על מנת להריץ הקובץ צריך להיות בעל הרשאות הרצה, אחריה תודפס שגיאה.

```
> ./main.c
```

```
-bash: ./main.c: Permission denied
```

```
> gcc main.c -o hello_world
```

```
> ./hello_world
```

```
Hello World!
```

- שינוי הרשאות קובץ אפשרי בעזרת הפקודה **chmod**. לדוגמה הפקודה:

```
> chmod u+x <file>
```

```
drwxr-xr-x 2 mtm users 4096 Mar 17 16:06 ex0
```

תכניות בסיסיות ב- UNIX

לניהול קבצים ותיקות

rm ♦	ls ♦
cd ♦	pwd ♦
cat ♦	mkdir ♦
less ♦	rmdir ♦
diff ♦	cp ♦
	mv ♦

תכניות סטנדרטיות ב-UNIX

"Make each program do one thing well. To do a new job, build a fresh rather than complicate old programs by adding new "features". Expect the output of every program to become the input to another, as yet unknown, program."

(Doug McIlroy, 1978 Bell System Technical Journal)

- קיימ מוגון בסיסי של תכניות הזמיןות בכל מערכת הפעלה משפחת UNIX
- צורת הפעלה של התכניות היא בד"כ מהצורה הבאה:
`> <command> [options] [input file]`
 - לרוב הפקודות, אם לא יצוין קובץ קלט, הקלט ילקח מהקלט הסטנדרטי על מנת לאפשר הרכבת פקודות.
 - בד"כ ניתן להחליף את הסדר בין שם קובץ הקלט לדגלים
- זכרו כי מלבד הדגמים שילמדו כאן ניתן למצוא רשימה מפורטת שלהם ע"י שימוש בפקודה `man` או בחיפוש באינטרנט

פקודות בסיסיות ב-UNIX

- כדי להדפיס תוכן של תיקיה נשתמש בפקודה **ls**:
 > **ls [flags] [files]**
 - ניתן לרשום פשוט **ls** עבור הדפסת תוכן התיקיה הנוכחית
 - ניתן להוסיף את שם התיקיות שאות תוכנן נרצה להדפיס
- לרוב הפקודות שנלמד אפשר להוסיף פרמטרים נוספים הנקראים **דגלים**:
 - למשל הדגל **-l** - עבור הפקודה **ls** יגרום להדפסת הפלט כך שכל קובץ מופיע בשורת פלט נפרדת עם מידע נוסף
 - הדגל **-a** - יגרום לפקודה להדפיס גם פרטיים נסתרים
 - פרטי נסתר ב-UNIX הוא כל קובץ או תיקיה ששמות מתחלilib. (נקודה)
 - בהרבה מהפקודות ניתן **לשלב מספר דגלים בחת- אחת**
 - כאשר הדגלים מיוצגים ע"י אות יחידה
 - לדוגמה **ls -a -l** זהה לפקודה **ls -al**

ls - דוגמאות

```
>ls  
hello.c  private/  story.txt  
>ls private  
example.txt  
>ls -l  
total 12  
-rw-r--r--  1 mtm users 3047 Jun 20 10:35 hello.c  
drwx----- 2 mtm users 4096 Sep  1 14:28 private/  
-rw------- 1 mtm users 1081 Sep  1 14:28 story.txt  
>ls -al  
total 28  
drwx----- 3 mtm users 4096 Sep  1 14:29 ./  
drwx--x--x 50 mtm users 12288 Sep  1 10:52 ../  
-rw-r--r--  1 mtm users 3047 Jun 20 10:35 hello.c  
drwx----- 2 mtm users 4096 Sep  1 14:28 private/  
-rw------- 1 mtm users 1081 Sep  1 14:28 story.txt
```

ניהול תיקיות

- הדפסת שם התיקיה הנוכחי ע"י הפקודה **pwd**:
 > **pwd**
- ניתן להחליף את התיקיה הנוכחי ע"י הפקודה **cd**:
 > **cd <directory>**
- ניתן ליצור תיקיה חדשה ע"י הפקודה **mkdir**:
 > **mkdir <name>**
- ניתן למחוק תיקיה ריקה ע"י הפקודה **rmdir**:
 > **rmdir <directory>**

ניהול תיקיות - דוגמאות

```
>cd ~
```

```
>pwd
```

```
/usr/030/moshe
```

```
>cd matam
```

```
matam: No such file or directory.
```

```
>mkdir matam
```

```
>cd matam
```

```
>pwd
```

```
/usr/030/moshe/matam
```

```
>cd ..
```

```
>rmdir matam
```

מעבר לתיקית הבית
בעזרת הקיצור ~

בכל הפקודות, אם אחד
הפרמטרים אינו תקין
תודפס הודעה שגיאה
מתאימה והפקודה תופסק

מעבר לתיקית האב
בעזרת הסימן ..

פקודות לניהול קבצים

-
-

הפקודה **cp** משמשת להעתיקת קבצים או תיקיות
ניתן להשתמש בפקודה בשתי דרכים:

>cp [options] <file1> <file2>

– יוצרת העתק חדש של file1 בשם file2

>cp [options] <file1> ... <filen> <directory>

– יוצרת העתקים חדשים של כל הקבצים בתיקייה קיימת

-

אם קיים כבר קובץ בעל השם המבוקש הוא נמחק ומוחלף (overwritten)

▪ דגלים שימושיים:

- **-r**: מבצע העתקה רקורסיבית של תיקיות ותוכנן
 - ללא דגל זה תיקיות אינן מועתקות
- **-i**: מבקש אישור לפני מחיקת קובץ קיים

פקודות לניהול קבצים

- הפקודה **mv** משמשת **להעברה** קבצים או תיקיות או שינוי שם ניתן להשתמש בפקודה בשתי דרכים:
 - > **mv [options] <file1> <file2>**
 - משנה את שמו של file1 ל file2
 - עובדת גם עבור תיקיות
 - > **mv [options] <file1> ... <filen> <directory>**
 - אם הparameter האחרון הוא שם של תיקייה קיימת הפקודה מעבירה את הקבצים לתוכה
- אם קיים כבר קובץ בעל השם המבוקש הוא נמחק ומוחלף (overwritten)
 - דגלים שימושיים:
 - **-i**: מבקש אישור לפני מחיקת קובץ קיים

פקודות לניהול קבצים

- הפקודה **rm** מאפשרת מחיקת קבצים או תיקיות:
 - > **rm [options] <files>**
 - דגלים שימושיים:
 - **r-**: מבצע מחיקה רקורסיבית של תיקיות
 - ללא דגל זה ספריות אינן נמחקoot
 - **i-**: מבקש אישור לפני מחיקת קובץ
 - **f-**: לא מבקש אישור בשום מקרה
 - **שים לב: לא ניתן לשחזר קבצים שנמחקו**
 - הפקודה הבאה מוחקת את כל הקבצים והספריות במקומות הנוכחי ועלולה להיות הרסנית למדדי:
 - > **rm -rf ***

פקודות לניהול קבצים - דוגמאות

```
>ls  
hello.c private/ story.txt  
>cp hello.c prog.c  
>ls  
hello.c private/ prog.c story.txt  
>mv prog.c private/program.c  
>ls  
hello.c private/ story.txt  
>ls private  
example.txt program.c  
>rm -i private/example.txt  
rm: remove regular file `private/example.txt'? y  
>rm private  
rm: cannot remove directory `private': Is a directory  
>rm -rf private  
>ls  
hello.c story.txt
```

הדפסת תוכן של קבצים



- הפקודה **cat** מדפיסה למסך את תוכן הקובץ

> **cat <file>**

- אם הקובץ גדול המסר יכולול ולא נוכן לראות בנוחות את התחלתו
 - אם לא מופיע שם קובץ הפקודה מדפיסה את הקלט הסטנדרטי (בד"כ קלט מהמקלדת)
- בහמשך נראה למה זה מועיל



- הפקודה **less** מאפשרת גלילה בשני הכוונים ועוד אפשרויות

> **less <file>**

השוואת תוכן של קבצים

- ניתן להשוות בין תוכן של 2 קבצים בעזרת הפקודה **diff**
> **diff [options] <file1> <file2>**
 - הפקודה משווה בין תוכן 2 הקבצים שורה אחר שורה. אם הקבצים זהים לא מודפס דבר.
 - אם הקבצים אינם זהים מודפסות השורות השונות ומספרי השורות הרלוונטיים.

```
> diff farm1 farm2
0a1
> cow Leni
2d2
< slim cow Dazy
4d3
< sheep Brook
```

farm1

goat Upton
slim cow Dazy
cow Betsy
sheep Brook

farm2

cow Leni
goat Upton
cow Betsy

הפקודה echo

- במקירים רבים נרצה להדפיס לפלט מחרוזת, לשם כך ניתן להשתמש בפקודה echo
- הפקודה echo מדפיסה את הפרמטרים שקיבלה
 - > echo [-n] [words]
 - **-n**: מדפיס ללא ירידת שורה

```
> echo Hello world!  
Hello world!  
>
```

```
> echo -n Hello world!  
Hello world!  
>
```

טבלת פקודות - סיכום

ls	מדפיס רשימה של שמות קבצים
cd	משנה את המיקום הנוכחי
pwd	מדפיס את מיקום התקינה הנוכחי
mkdir	يוצר תיקייה חדשה
rmdir	מוחק תיקייה ריקה
cp	מעתיק קבצים
mv	מעביר קבצים
rm	מוחק קבצים
cat	מדפיס תוכן של קובץ
less	מדפיס תוכן של קובץ בצורה נוחה יותר
diff	משווה בין תוכן של קבצים

קיימות פקודות רבות נוספות שבחלקן תתקלו תוך כדי העבודה על תרגילי הבית.

הפניית קלט/פלט

❖ ערכאים סטנדרטיים

❖ הפניה

עורך קלט/פלט

- לכל תכנית יש עורך קלט ועורך פלט – אלו הם הדרכים בהם היא מקבלת קלט ומציגה את הפלט שלה
- לכל תכנית קיימים ב-AIOS 3 עורך ברירת מחדל:
 - **עורך הקלט הסטנדרטי:** ממנו מקבלת התכנית את הקלט
 - ברירת המחדל היא המקלדת (מה שМОוקלד בחלוון הטרמינל)
- **עורך הפלט הסטנדרטי:** אליו מודפס הפלט
 - ברירת המחדל היא הדפסה לחלוון הטרמינל
- **עורך השגיאות הסטנדרטי:** עורך פלט נוסף המשמש להדפסת הודעות שגיאה
 - ברירת המחדל היא הדפסה לחלוון הטרמינל
- עורך קלט/פלט נוספים יכולים להיות למשל קבצים, נתיחס למקרים אלו בהמשך הקורס

הפנียת קלט/פלט

- בעת קרייה לפקודה בטרמינל ניתן להפנות את ערוצי הקלט והפלט הסטנדרטיים לקבצים
 - הפנียת ערוץ הפלט הסטנדרטי נעשית באמצעות האופרטור "<":
> command [arguments] > <output_file>
 - הפנียת ערוץ הקלט הסטנדרטי נעשית באמצעות האופרטור ">":
> command [arguments] < <input_file>
 - הפנียת ערוץ השגיאות הסטנדרטי נעשית באמצעות האופרטור ">2":
> command [arguments] 2> <errors_file>
- אם ננסה לכתוב לתוך קובץ קיים, נדרוס את תוכן הקובץ הקיימים.
 - אם ברצוננו להוסיף את הפלט למשכו של קובץ קיים ניתן להשתמש ב- ">>" או ">>2" (בהתאם)

הפנויות קלט/פלט

- ניתן להפנות מספר ערכאים בבאת אחת
- הפנויות הקלט והפלט ייחדי נועשית כך:
 - > command [arguments] < input > output
- כדי להפנות את ערז השגיאות ואת ערז הפלט ניתן לרשום:
 - > command [arguments] > output 2> errors
- כדי להפנות את ערז השגיאות לערז הפלט יש לכתוב:
 - > command [arguments] 2>&1
- כדי להדפיס את ערז הפלט וערז השגיאות לאותו הקובץ יש לכתוב:
 - > command [arguments] > output 2>&1

הפנית קלט/פלט - דוגמאות

- בעזרת הפנית קלט/פלט ניתן לבצע פעולות שונות עם הפקודה cat, למשל:

```
>cat > file.txt
```

```
Hello world!
```

```
>ls
```

```
file.txt
```

```
>cat file.txt
```

```
Hello world!
```

```
>cat file.txt > file2.txt
```

```
>cat file2.txt
```

```
Hello world!
```

```
>cat file.txt >> file2.txt
```

```
>cat file2.txt
```

```
Hello world!
```

```
Hello world!
```

קלט שהוקלד במקלדת

הפנית קלט/פלט - דוגמאות

```
>cat < file2.txt > file.txt
```

```
>cat file.txt
```

Hello world!

Hello world!

```
>rmdir foo
```

```
rmdir: `foo': No such file or directory
```

```
>rmdir foo > output.txt
```

```
rmdir: `foo': No such file or directory
```

```
>rmdir foo 2> error.txt
```

```
>cat error.txt
```

```
rmdir: `foo': No such file or directory
```

הודעת השגיאה נשלחת
לערוץ השגיאות ולכן אינה
מודפנת לקובץ

הרכבת פקודות

```
stud.technion.ac.il - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
drwx----- 2 root root 16384 Jul 22 2007 lost+found
drwxr-xr-x 5 root root 4096 Dec  2 2007 media
drwxr-xr-x 2 root root 4096 Oct 19 2010 misc
drwxr-xr-x 3 root root 4096 Nov  2 2009 mnt
drwxr-xr-x 24 root root 4096 Mar 27 16:06 opt
dr-xr-xr-x 863 root root     0 Feb 19 11:07 proc
drwxr-x--- 18 root root 4096 Apr 16 08:53 root
drwxr-xr-x 2 root root 12288 Dec 14 04:11 sbin
drwxr-xr-x 2 root root 4096 Jul 22 2007 selinux
drwxr-xr-x 2 root root 4096 Aug 12 2004 srv
drwxr-xr-x 3 root root 4096 Aug 17 2008 ssl
drwxr-xr-x 4 root root 4096 Apr  3 14:33 st
lrwxrwxrwx 1 root root      5 Oct 15 2007 staff -> /tx/u
-rwxrwxrwx 1 root root      0 Nov 14 03:00 state
drwxr-xr-x 9 root root      0 Feb 19 11:07 sys
lrwxrwxrwx 1 root root      7 Jul 23 2007 tcc -> /u1/tcc
-rw-r--r-- 1 root root    143 Nov 18 2009 temp
drwxr-xr-x 3 root root 4096 Feb 21 2008 tftpboot
drwxrwxrwt 97 root users 61440 Apr 16 10:42 tmp
drwxr-xr-x 3 root root      0 Apr 16 09:11 tx
drwxr-xr-x 106 root users 8192 Feb  1 01:48 u1
drwxr-xr-x 20 root root 4096 Nov  2 2009 usr
drwxr-xr-x 33 root root 4096 Aug 31 2011 var
-bash-3.00$
```

Connected to stud.technion.ac.il SSH2 - aes128-cbc - hmac-md5 - nc | 80x24

- נניח שברצוננו להדפיס את תוכן של תיקיה המכילה הרבה קבצים לא נוכל במצב זה לראות את כל הקבצים נוכל להשתמש בהפניית פלט הפקודה זו לקובץ זמני ולאחר מכן שימוש ב-more:
 - > ls -l > tmp
 - > more < tmp
 - > rm tmp

משתבי סביבה

- ❖ משתנים ב**bash**
- ❖ פקודות פנימיות וחיצוניות
- ❖ המשתנה **PATH**

משתנים

- ניתן להציב ערך למשתנה ב-bash על ידי השמה ישירה
 - > `<varname>=<value>`
 - אין צורך להזכיר על המשתנים ב-bash, לאחר ביצוע השמה למשתנה הוא מוגדר אוטומטית
 - המשתנה נגיש רק ב-shell הנוכחי.
 - ניתן להחליף ערך של המשתנה על ידי השמה נוספת
 - אסור לשים רווח בין סימן ההשמה לשם המשתנה והערך
 - אם יהיה זהה רווח ה-Shell ינסה להריץ פקודה בשם המשתנה
- ניתן לקרוא המשתנים על ידי שימוש באופרטור `$`, למשל:
 - > `echo $<varname>`
 - תוכנת ה-shell מחליפה את המשתנה בערכו (מחזרה)

משתני סביבה

- **משתני סביבה** הם משתנים שנגישים לכל התוכנות שרצות על המערכת.
- מחזיקים ערכים המשפיעים על הדרך בה יתנהגו תהליכי בסביבה הנתונה.
- קיימים משתני סביבה המוגדרים אוטומטית ע"י מערכת הפעלה, לדוגמה .HOME, PATH
- מנגנון המשתני הסביבה דומה מאד בין כל מערכות הפעלה - **הם משמשים בכל מערכות הפעלה (לקבוע ולשמור ערכים עבור תהליכי**.

```
[mtm@cs13 ~]$ echo $HOME  
/home/mtm
```

ביצוע פקודות shell

- הפקודות בbash מתחולקות ל 2 סוגים:
 - פקודות פנימיות של ה-Shell. דוגמא הפקודה `.history`.
 - פקודות חיצונית – שמות קבצי הרצה, למשל `ls`, `cd`, `gcc`, `zip`.
- אם שם הפקודה מתאים לפקודה פנימית אז היא תבוצע, אחרת `shell` יחפש בפקודות החיצונית הזמיןות לו.
- הפקודות החיצונית הזמיןות לו הן קבצי הרצת שנמצאות בכל התקיות המופיעות במשתנה הסביבה `PATH`.
- ניתן להשתמש בפקודה `which` כדי לגלות את מיקום התוכנית המתאימה לפקודה החיצונית.

```
[mtm@cs13 ~]$ which cd  
/usr/bin/cd
```

משתנה הסביבה PATH

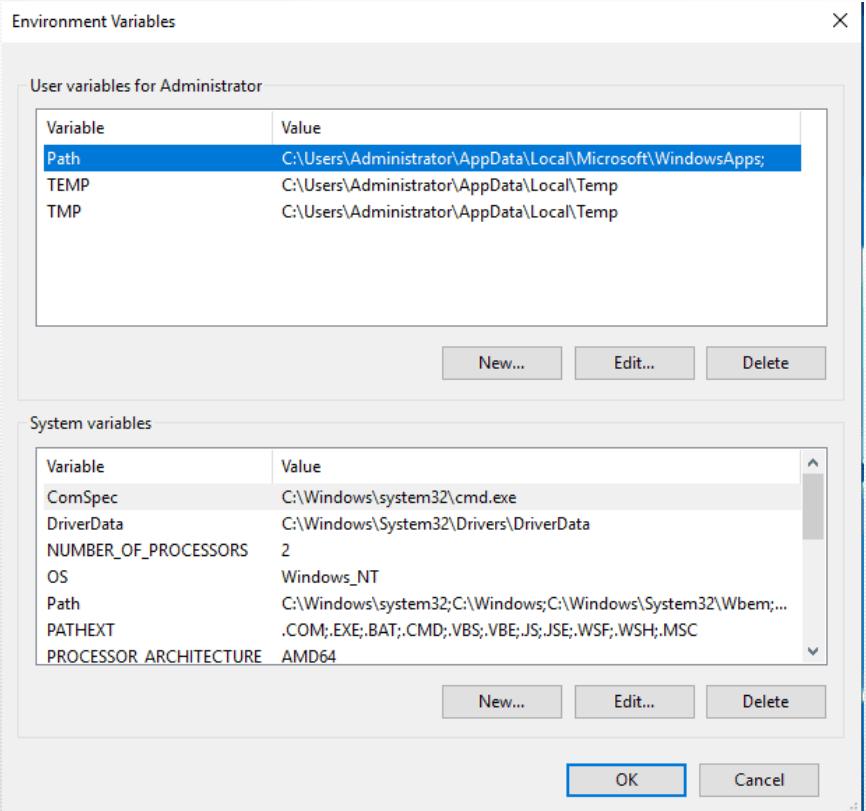
- משתנה הסביבה **PATH** מציין את התקיימותם של programs shell יחפש אותם בפתקודות.
- והוא מחזמת כל המסלולים הרלוונטיים, מופרדים על ידי ":".

```
[mtm@cs13 ~]$ echo $PATH  
/usr/lib64/qt-3.3/bin:/usr/bin:/usr/local/bin:  
in:/home/mtm/bin
```

- את התוכנות הזמיןות בתקיינות הנמצאות במשתנה **PATH** ניתן להריץ בתוכנת shell מבלי להשתמש ב"/".

PATH בוינטוס

▪ משטנה הסביבה PATH קיימ גם בwindows.



```
C:\WINDOWS\system32\cmd.exe
C:\Users\Flo>path
PATH=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows
(x86)\Skype\Phone\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Co
Files\Common Files\Intel\WirelessCommon\;C:\WINDOWS\system32;C:\WINDOWS
PowerShell\v1.0\;C:\Program Files\Microsoft SQL Server\110\Tools\B;
\Users\Flo\AppData\Local\Microsoft\WindowsApps;C:\Program Files\Intel\I
lessCommon\;
```

תרגול מס' 2

- ❖ מצביעים
- ❖ הקצאת זיכרון דינמית
- ❖ העברת פרמטרים ל-main
- ❖ מבנים – Structures
- ❖ רשימות מקשורות

מצבייעים

- ❖ שימוש בסיסי
- ❖ אריתמטיקת מצבייעים
- ❖ `void*`
- ❖ מצביע לצביע

מבנה הזיכרון - תזכורת

ערך הבית

כתובת

int הטופס
4 בתים

7	0x0200
0	0x0201
0	0x0202
0	0x0203
0	0x0204
0	0x0205
0	0x0206
0	0x0207
104	0x0208
101	0x0209
108	0x020A
108	0x020B
111	0x020C
0	0x020D

- הזיכרון מורכב מתחאים הקוראים בתים (bytes)
 - כל בית מכיל ערך **מספרי**
 - פירוש הערכים כערך לא מספרי הוא ע"י התכנית
 - לכל בית יש שם – כתובת
 - הכתובת היא מיקומו בזיכרון
 - בד"כ כתובות בזיכרון נרשומות בסיס הקסדצימלי (16)

משתנים מאוחסנים בתים:

- טיפוסים שונים דורשים מספר שונה של בתים, למשל:
 - int צריך 4 או 8 בתים
 - char צריך בית יחיד.

מצביים - תזכורת

0x208	0x0204
34	0x0205
37	0x0206
0	0x0207
8	0x0208
20	0x020A
11	0x020B
9	0x020C

- עבור טיפוס `D` נקרא ל-`*D` **מצביע ל-D**
 - למשל `*int` הוא מצביע ל-`int`

- מצביע מסוג `*D` הוא משתנה אשר שומר **כתובת** של משתנה מטיפוס `D`.

- ניתן לקבל את כתובתו של משתנה ע"י שימוש **באופרטור &**
 - לא ניתן להשתמש ב-`&` על ביטויים או קבועים (מדוע?)

- ניתן לקרוא את ערכו של המצביע ע"י **אופרטור ***
dereferencing
 - פעולה זו קרויה

```
int n = 8;  
int* ptr = &n; // ptr now points to n  
printf("%d", *ptr); // dereferencing ptr
```

הכתבת 0x0 - NULL

- הכתובת 0 הינה כתובת לא חוקית:
 - אף עצם אינו יכול להיות בעל כתובת זו
- ניתן לעשות שימוש בכתובת זו כדי לציין שמצבי מסויים אינו מצביע לאף עצם כרגע
- **השתמשו ב-NULL** (שהוא `#define # 0`) ולא בקבוע 0 מפורשöt כאשר אתם מתייחסים לכתובת זו
 - שימוש בקבועים כאלה משפר את קריאות הקוד
- נסיון לקרוא מצביע המכיל את הכתובת NULL יגרום **לקriseת התוכנה**.
 - ב-AION תתקבל הודעה:
“segmentation fault”
 - גישה למצביע המכיל “זבל” תגרום לתכנית להתנהג בצורה לא צפואה
 - אסור להשאיר **מצביעים לא מאוחדים** בקוד!
- ניתן להריץ על המשתנה מאוחר יותר (C99)
 - במקרה ולא ניתן יש לאותל אותו ל-NULL



אריתמטיקת מצביעים

- ניתן בנוספַּח לבצע **פעולות חשבוניות** על מצביעים
 - **חיבור עם מספר שלם:**

```
int n = ...;
```

```
int* ptr2 = ptr + n;
```

- התוצאה היא כתובתו של המשתנה מהטיפוס המתאים או תאים קדימה/אחריה

- **חיסור שני מצביעים:**

```
int diff = ptr2 - ptr;
```

- התוצאה היא מספר שלם (int)

- ניתן לחסר רק מצביעים מאותו טיפוס

- פעולות אלו מאפשרות להסתכל על המשתנה הבא/הקדם בזיכרון
 - הכרחי לשימוש במערכות ומחרחות
 - **מסוכן** –TeVויות חשבוניות עלולות לגרום לחריגה ולקראאת "זבל" מהזיכרון
- **שאלה:** מדוע לא ניתן לחבר שני מצביעים?

מצביים – גישה למערך

- ניתן להשתמש בזיכרון כדי לגשת למשתנים הנמצאים בהמשך בזכרון, למשל כרך:

```
int* ptr = ...;  
int n = *(ptr + 5);
```

- האופרטור [] משמש כקיצור לפעולה זו:
`int n = ptr[5];`
- כלומר הפעולות הבאות שקולות:
$$*(\text{ptr} + \text{n}) \equiv \text{ptr}[\text{n}]$$

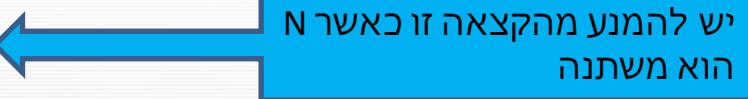
מצביים ומערכות

- מערכים ומצביעים מתנהגים בצורה דומה
 - ניתן להשתמש בשם המערך כמצביע לאיבר הראשון בו
 - כאשר שולחים מערך לפונקציה ניתן לשלוח אותו כמצביע:

```
void sort(int* array, int size);
```

- מצביע יכול לשמש **כ迭代器** עבור מערך

```
int array[N];  
//...  
for(int* ptr = array; ptr < array+N; ptr++) {  
    printf("%d ", *ptr);  
}
```



- הבדלים:

- הקריאה על מערך מקצת זיכרון בגודל המערך, **הקריאה על מצביע אינה מקצת זיכרון לאחסן המשתנים!**
- ניתן לשנות את ערכו של מצביע, אך לא ניתן לשנות את "ערכו" של תחילת המערך

void*

- ניתן להגדיר מצביעים מטיפוס `void*`. מצביעים אלו יכולים לקבל את כתובתו של כל משתנה
- לא ניתן לקרוא את התוכן של מצביע מטיפוס `void*`, יש להמירו קודם לכן

```
int n = 5;  
double d = 3.14;
```

```
void* ptr = &n;  
ptr = &d;
```

```
double d2 = *ptr;           // Error: cannot dereference void*  
double d3 = *(double*)ptr; // O.K. - option 1  
double* dptr = ptr;        // Implicit cast from void* to double*  
double d4 = *dptr;          // O.K. - option 2
```

מצביע למאבוי

- ניתן ליצור מצביע לכל טיפוס, בפרט עבור טיפוס `*T` ניתן ליצור מצביע מטיפוס `**T`
 - מתקבל **מצביע למאבוי של T**
 - אפשר להמשיך לכל מספר של *

▪ דוגמאות:

- שליחת מערך של מצביעים לפונקציה:

```
void sort_pointers(int** array, int size);
```

- כתיבת פונקציית swap עבור מחרוזות:

```
void swap_strings(char** str1, char** str2) {  
    char* temp = *str1;  
    *str1 = *str2;  
    *str2 = temp;  
}
```

- מדוע יש כאן צורך במאבוי למאבוי?

מצביעים - סיכום

- מצביעים משמשים להתייחסות לתאי זיכרון
- ניתן לקבל את כתובתו של משתנה ע"י אופרטור **&**
- ניתן לקרוא ממצביע ולקבל את הערך המוצבע ע"י *
- הערך **NULL** מציין שאין עצם מוצבע ואסור לקרוא אותו
- ניתן לבצע **פעולות חשבוניות** על מצביעים
 - אפשר התייחסות למצביעים בדומה למערכות
- חשוב **לאთחל** במצבים
 - הרצת קוד הנגיש למצביעים המכילים ערך לא תקין תגרום להתנהגות לא מוגדרת
 - הכמה על מצביע אינה מתחילה זיכרון עבור המשתנה המוצבע!
- מצביע מטיפוס ***void** יכול להצביע לעצם מכל סוג ומשמש לכיתבת קוד גנרי

הקצתת זיכרון דיבאמתית

- ❖ סוגים משתנים
- ❖ הקצתת זיכרון
- ❖ שחרור זיכרון
- ❖ נזילות זיכרון

סוגי משתנים

את המשתנים השונים בקוד ניתן לסוג לפי טווח ההכרה ואורך חייהם:

- **משתנים מקומיים**: משתנים פנימיים של פונקציות. נגישים רק בבלוק בו הם הוגדרו. משתנים אלו מוקצים בכל פעם שהבלוק מורץ ומשוחררים בסופו אוטומטית.
- **משתנים גלובליים**: משתנים אשר קיימים לכל אורך התכנית וניתן לגשת אליהם מכל מקום. המשתנים מוקצים כאשר התכנית מתחילה ונשמרים לכל אורך הריצה
- **משתנים סטטיים של פונקציה**: משתנים פנימיים של פונקציה. המשתנים אלו נשמרים על ערכם בין הקריאה השונות לפונקציה. מוקצים בתחילת ריצת התוכנית ומשוחררים בסיוםה.

```
//global variable
int global = 1;
int main()
{
    //local variable
    int local = 0;
    return local;
}
```

הروع של משתנים גלובליים

- משתנים גלובליים, משתנים סטטיים של קובץ (נלמד בהמשך) ומשתנים סטטיים של פונקציה נחכבים **לתוכנות רע**
- הסיבה העיקרית לכך - שימוש במשתנים אלו מקשה על הבנת וDİבוג הקוד:
 - כדי להבין פונקציה המשמשת במשתנה גלובלי יש להסתכל בקוד נוסף
 - קשה לצפות את תוצאה הפונקציה כי היא אינה תלואה רק בפרמטרים שלה
 - קשה לצפות השלכות של שינויים על ערך המשתנה
 - שימוש במשתנה סטטי של פונקציה - בשביל לצפות את תוצאה הפונקציה צריך לדעת מה קרה בהרצות קודמות
- **אין להשתמש במשתנים גלובליים במת"מ**

- בקורסים מתקדמים בהמשך התואר תראו מקרים בהם חובה או מומלץ להשתמש במשתנים כאלה

```
//global variable
int global = 1;
int main()
{
    //Local variable
    int local = 0;
    return local;
}
```

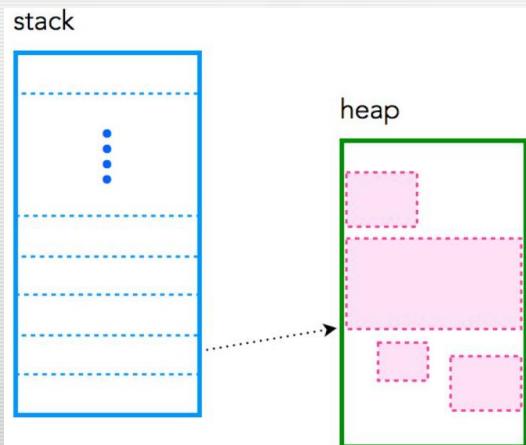
זיכרון שמווקצתה דינמית

- **זיכרון שמווקצתה דינמית** הוא זיכרון שזמן החיים שלו הוא בשליטת המתכנת
 - קוד מפורש מקצה אותו וקוד מפורש דרוש לשחררו
 - הזיכרון מוקצה באזור שקרוי ה-**heap**
 - בנגד למשתנים מקומיים שמווקצים על מחסנית הקראות (ה-stack)

▪ mol stack :heap

- זיכרון heap גדול משמעותית ממחסנית הקראות. מחסנית הקראות איןנה מתוכננת להכיל כמויות נתוניות גדולות
- במקרה של חוסר זיכרון ב-heap, התוכנית מקבלת על כר התראה מסודרת.
- **לעומת זאת, לא ניתן להתאושש מכישלון הקצאה על המחסנית!**

▪ נשתמש בזכרון דינامي כאשר:



- נדרש לאחסן כמות גדולה של נתונים, או להקנות זיכרון בגודל שאינו ידוע מראש
- יש צורך לשמור את הנתונים בזכרון גם לאחר יציאה מהפונקציה

▪ הגישה לזכרון דינامي נעשית תמיד בעזרת מצביעים

הקצת זיכרון באמצעות malloc

- כדי להקצת זיכרון דינامي משתמש בפונקציה malloc:

```
void* malloc(size_t bytes);
```

size_t הוא טיפוס
יעודי עבור גודלים
של זיכרון.

- malloc מקבל גודל בbytes של זיכרון אותו עליה להקצת
- ערך החזרה מכיל מצביע לתחילת גוש הזיכרון שהוקצת
- התוצאה היא תמיד גוש זיכרון רציף
- במקרה של כשלון מוחזר NULL

- לאחר מכן ניתן להתייחס לשטח המוצבע כאל משתנה או מערך:

```
int* my_array = malloc(sizeof(int) * n);
for (int i=0; i<n; i++) {
    my_array[i] = i;
}
```

קביעת הגודל אותו מזמנים

- כיצר נדע כמה בתים علينا להקצות עבורה משתנה מסוג int?
- נסיוון ראשון:

```
int* ptr = malloc(4);
```
- 4 הוא **מספר קסם** - מספר לא ברור המופיע בקוד שאינו 0 או 1
 - מספרי קסם הם **הרגל תכניות רע**:
 - פוגעים בקריאות הקוד
 - מקשים על שינוים עתידיים בקוד
 - למשל מעבר לסייעתה בה גודלו של int הוא 8
 - ככל שציריך יותר שינויים הסיבו לפסק אחד מהם גדול
- יש להימנע ממספרי קסם:
 - ע"י הגדרת **קבועים** בעזרת #define שיקלו על שינויים ועל קריאת הקוד
 - ע"י שמירת ערכם **במשתנה קבוע** בעל שם ברור

קביעת הגודל - נסיוון שני

- נסיוון שני - נגדיר את הגודל של int כקבוע:

```
#define SIZE_OF_INT 4
```

```
int* ptr = malloc(SIZE_OF_INT);
```

- עכשו הקוד קרייאי וקל לשנות את הערך
- אבל אם נעביר את הקוד לסביבה אחרת עדין נצטרך לעדכן את הערך
 - קוד שדורש שינויים במעבר בין סביבות שונות נקרא **non-portable**

אופרטור sizeof

- נשתמש באופרטור **sizeof** אשר מחזיר את הגודל המתאים:

```
int* ptr = malloc(sizeof(int));
```

- ניתן להפעיל את sizeof על שם טיפוס, על משתנה, או על ביטוי
▪ עבור שם טיפוס יוחזר הגודל בבתים של הטיפוס:

```
int* ptr = malloc(sizeof(int));
```

- עבור הפעלה על משתנה או ביטוי יוחזר הגודל של המשתנה או של ערך הביטוי בבתים:

```
int* ptr = malloc(sizeof(*ptr) // = sizeof(int))
```

למה השיטה זו
עדיפה?

- שימוש לב להבדל בין גודל של מצביע לגודל העצם המוצבע
- מה נעשה אם ברכוננו להקצות זיכרון לעותק של מחרוזת?

```
char* str = "This is a string";
```

```
char* copy = malloc(sizeof(char)*(strlen(str)+1));
```

אפשר להוציא את (sizeof(char)
מובטח שהוא תמיד 1

למה צריך +1 ?

בדיקות ערכי חזרה

- alloc עלולה להיכשל בהקצתה היזכרו - במקרה זה מוחזר NULL
- מה קורה במקרה זה אם alloc נכשלה?

```
int* my_array = malloc (sizeof(int) * n);
for (int i=0; i<n; i++) {
    my_array[i] = i;
}
```

- הפתרון: **בדיקה ערך ההחזרה** של פונקציות העוללות להיכשל וטיפול בו
 - הטיפול צריך להופיע מיד לאחר ההקצתה ולפני השימוש הראשון

```
int* my_array = malloc(sizeof(int) * n);
if (my_array == NULL) { // or !my_array
    handle_memory_error();
}
```

- בהמשך נראה מקרים נוספים של שגיאות יותר שכיחות ו פשוטות להתמודדות

שחרור זיכרון באמצעות free

הפונקציה `free` משמשת לשחרר גוש זיכרון שהוקצה ע"י `malloc`

```
void free(void* ptr);
```

- המצביע שנשלח ל-`free` חייב להצביע לתחילת גוש הזיכרון (אותו ערך שהתקבל מ-
- (`malloc`)
- לאחר שחרור הזיכרון אסור לגשת יותר לערבים בזיכרון ששוחרר
- אם שולחים `NULL` ל-`free` לא מתרחש כלום (זו אינה שגיאה)
- כמובן אין צורך לבדוק את הформטර הנשלח ולודא שאינו `NULL`
- למה זה טוב?
- אסור לשחרר את אותו זיכרון פעמיים או לשלוח ל-`free` מצביע שאינו מצביע לתחילת גוש זיכרון שהוקצה דינמית (או `NULL`)

```
int* my_array = malloc(sizeof(int) * n);
// ... using my_array ...
free(my_array);
```

מקרים קצה

- במקרה ונשלח NULL ל-free לא מתבצע כלום
- ```
if (ptr != NULL) {
 free(ptr);
}
```
- ניתן להחליף את הקוד הקודם בזה:  

```
free(ptr);
```
- NULL הוא **מקרה קצה** עבור free
  - מה היה קורה אם free לא הייתה מתמודדת עם מקרה הקצה זהה?
- **עדיף לטפל במקרים קצה בתוך הפונקציה**
  - מונע מה משתמש בה ליצור **באגים ושכפولي קוד**

# גישה לזכרון אחרי שוחרר

- גישה לכנות זיכרון שאינה מוקצת (או הוקצתה ושותררה) **אינה מוגדרת**
- שחרור כפול של כנות זיכרון **אינו מוגדר**
- קוד שתוצאתו אינה מוגדרת הוא קוד שמתקמל ורץ אך תוצאה ריצתו אינה מוגדרת
  - התוצאה יכולה להשתנות בין מערכות הפעלה, בין קומפיילרים, אפילו בין ריצות
  - התוכנית יכולה לקרוא (למה זה עדיף?), או להמשיך לרוץ ולתת תוצאה שגוייה
  - בחלק מהמקרים התוצאה אף עשויה להתאים לציפיות, ולכון קשה לאתר שגיאה זו
- קוד שתוצאתו אינה מוגדרת הוא באג קשה לטיפול
  - קשה לצפות את התנהגותו והשלכותיו
  - יכול להשפיע על משתנים באזור אחר לחלוטין בקוד
- חשוב להקפיד על **שימוש נכון בשפה** כדי להימנע ממקרים אלו

# דליפות זיכרון

- **דליפת זיכרון** מתרחשת כאשר שוכחים לשחרר זיכרון שהוקצהה:

```
void sort(int* array, int n) {
 int* copy = malloc(sizeof(int) * n);
 // ... some code without free(copy)
 return;
}
```

- דליפת זיכרון אינה גורמת לשירوت לשגיאות בהתקנות התוכנה
- דליפת זיכרון יגרמו לצריכת זיכרון גדולה של התוכנה ככל הזמן ריצתה גדל **ולהאטת התוכנה** ומערכת הפעלה יכולה
- תחת אונס ניתן להשתמש בכללי **valgrind** לאיתור דליפות זיכרון
  - valgrind מרייצ' את התכנית שלכם ומ Chapman גושי זיכרון שהוקצטו אך לא שוחררו
  - ניתן למצוא מידע נוספת על השימוש ב-valgrind בתרגול עזר 3

# בעלות על זיכרון

- על מנת למנוע **דליפות זיכרון** חשוב לשים לב לנושא **הבעלויות** על הזיכרון הדינامي.
- **הבעלים** של הזיכרון הוא מי שאחראי **לפנות** אותו.
- פונקציה אשר מקצתה זיכרון, אחראית לפנותו. אלא אם היא מחזירה את הזיכרון שהקצתה. במקרה זה **הבעלויות עוברת** למי שקרה לפונקציה.
- בקוד חייב להיות ברור מי אחראי על הפינוי של כל זיכרון שמוקצתה. בפונקציה המחזירה זיכרון דינامي שהקצתה (כלומר, מעבירה את האחריות למי שקרה לה) – **מידע זה חייב להיות מתועד היטב**.

//This function returns the allocated array. Note that the caller is responsible to free it.

```
int* CreateIntArray(int size) {
 int* arr = malloc(sizeof(int) * size);
 return arr;
}
```

- הבעלים של זיכרון יכול להיות גם משהו מורכב יותר כמו אובייקט או מודול, כפי שנזכיר בהמשך הקורס..

# AIR מתמודדים עם כל הקשיים?

- כדי להימנע מכל הבעיות שתוארו כאשר עובדים עם הקצאות דינמיות קיים רק פתרון אחד עיל - **עבודה מסודרת**
- בעזרת עבודה מסודרת ניתן לשמור על הקוד פשוט יותר
- קוד מסובך מקל על הכנסת באגים בטעות
- הטיפול בבאגים קשה יותר אם הקוד מסובך
- בשפות מודרניות כמו **++** יש כלים מתקדמים יותר לניהול זיכרון דינامي ומניעת זליגת זיכרון. נלמד על כך בהמשך.

# הקצתת זיכרון דינמית - סיכום

- מומלץ לא להשתמש במשתנים גלובליים וסטטיים
- משתנים דינמיים משמשים בכל מקרה שצריך כמות זיכרון שאינה קטנה מאוד
- ניתן להשתמש ב-**malloc** ו-**free** כדי להקצת ולחזר זיכרון בצורה מפורשת
  - עבור ייצירת מערכיים גדולים או בגודל לא ידוע
  - עבור שימירת ערכים לאורך התכנית
- ניהול הזיכרון מתבצע ע"י מנגנונים לתחילת גוש הזיכרון שהוקצה
- יש לבדוק הצלחת הקצת זיכרון
- יש לזכור לשחרר את הזיכרון המוקצה כאשר אין בו צורך יותר
- ניתן להשתמש ב-**valgrind** כדי למצוא בעיות גישות לזכרון לא מוקצה ודליפות זיכרון

# העברה פרמטרים ל-`main`

- ❖ הפרמטרים `argc` ו-`argv`
- ❖ תכנית לדוגמה

# העברה פרמטרים ל-main

- את הפונקציה `main` המתחילה את ריצת התכנית ניתן להגדיר גם כך:

```
int main(int argc, char** argv)
```

- במקרה זה יילקחו הארגומנטים משורת ההרצתה של התכנית ויושמו לתוך המשתנים `argc` ו-`argv` ע"י **מערכת הפעלה**
  - `argc` יאותחל למספר הארגומנטים בשורת הפקודה (כולל שם הפקודה)
  - `argv` הוא מערך של מחרוזות כאשר התא ה-`a` בו יכול את הארגומנט ה-`a` בשורת הפקודה. תא 0 יכול את שם התוכנה שהורצתה.
  - בנוסף, קיימ איבר אחרון נוסף במערך המאותחל ל-NULL

# דוגמה - תכנית echo

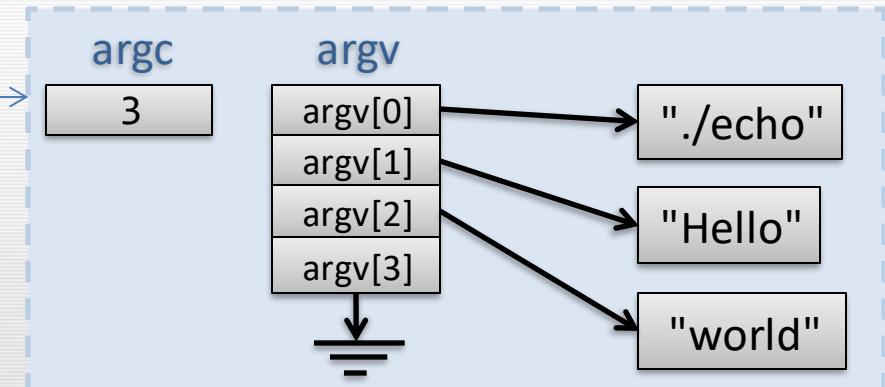
```
#include <stdio.h>

int main(int argc, char** argv) {
 for(int i = 1; i < argc; i++) {
 printf("%s ", argv[i]);
 }
 return 0;
}
```

כיצד ניתן לכתוב את  
הקוד זהה ללא שימוש  
במשתנה argc?

```
> ./echo Hello world
Hello world
> ./echo Hello > world
> cat world
Hello
```

לאן נעלמה  
המילה world?



# מבנים

- ❖ הגדרת מבנה
- ❖ פעולות על מבנים
- ❖ `typedef`

# הטיפוסים הקיימים אינם מספיקים

- נניח שברצוננו לכתב תוכנה לניהול אנשי קשר, לכל איש קשר נשמר: שם פרטי, שם משפחה, מספר טלפון, כתובת e-mail וכתובת מגורים.
- לשם כך נדרש לשמר **5 מערכים שונים!**
- כל פונקציה שתצרוך לקבל את פרטיו של איש קשר כלשהו צריכה לקבל **5 פרמטרים שונים לפחות!**

```
void someFunction(char* firstname, char* lastname, char*
 address, char* email, int number, ... more?);
```

- מה יהיה אם נרצה **בעתיד** להוסיף עוד פרט על איש קשר?
- כדי להימנע מריבוי משתנים ניתן להגדיר **טיפוסים חדשים** המהווים הרכבה של מספר טיפוסים קיימים

```
void someFunction(Contact contact, ...);
```

# מבנהים - Structures

- ניתן להגדיר טיפוסים חדשים המהווים הרכבה של מספר טיפוסים קיימים בעזרת המילה השמורה :**struct**

```
struct <name> {
 <typename 1> <field name 1>;
 <typename 2> <field name 2>;
 ...
 <typename n> <field name n>;
} <declarations>;
```

- הטיפוס החדש מורכב מ**שדות**:
  - **כל שדה יש שם**
  - טיפוס השדה נקבע לפי הגדרת המבנה
- השדות של מבנה נשמרים בזיכרון בראץ'
- ניתן להשתמש במערכות בעלי גודל קבוע כשדות - כל המערך נשמר במבנה
- ניתן להשתמש במצביים כשדות - במקרה זה הערך המוצבע אינו חלק מהמבנה

# מבנים - דוגמאות

```
struct point {
 double x;
 double y;
};
```

**point**

|       |
|-------|
| x=3.0 |
| y=2.5 |

```
struct date {
 int day;
 char month[4];
 int year;
};
```

למה 4?

**date**

|             |
|-------------|
| day=31      |
| month="NOV" |
| year=1971   |

כל המערך  
נשמר בתוך  
המבנה

```
struct person {
 char* name;
 struct date birth;
};
```

**person**

|               |              |
|---------------|--------------|
| name=0x0ffff6 |              |
| birth         | day=31       |
|               | month="MAR " |
|               | year=1953    |

"Ehud Banai"

המחetta  
נשמרת מחוץ  
למבנה

# שימוש במבנים

- הטיפוס החדש מוגדר בשם **struct <name>**
- כדי לגשת לשדות של משתנה מטיפוס המבנה נשתמש באופרטור **.** (נקודה)

```
struct point p;
p.x = 3.0;
p.y = 2.5;
double distance = sqrt(p.x * p.x + p.y * p.y);
```

- עבור מצביע למבנה ניתן להשתמש באופרטור החץ **->**

```
struct point* p = malloc(sizeof(*p));
(*p).x = 3.0; // Must use parentheses, annoying
p->y = 2.5; // Same thing, only clearer
double distance = sqrt(p->x * p->x + p->y * p->y);
```

מה חסר?

# פעולות על מבנים

- ניתן לאותחל מבנים בעזרת התחביר הבא:

```
struct date d = { 31, "NOV", 1970 };
```

- ניתן לבצע השמה בין מבנים מאותו הטיפוס:

```
struct date d1,d2;
// ...
d1 = d2;
```

– במקרה זה מתבצעת השמה בין כל שני שדות توأمים

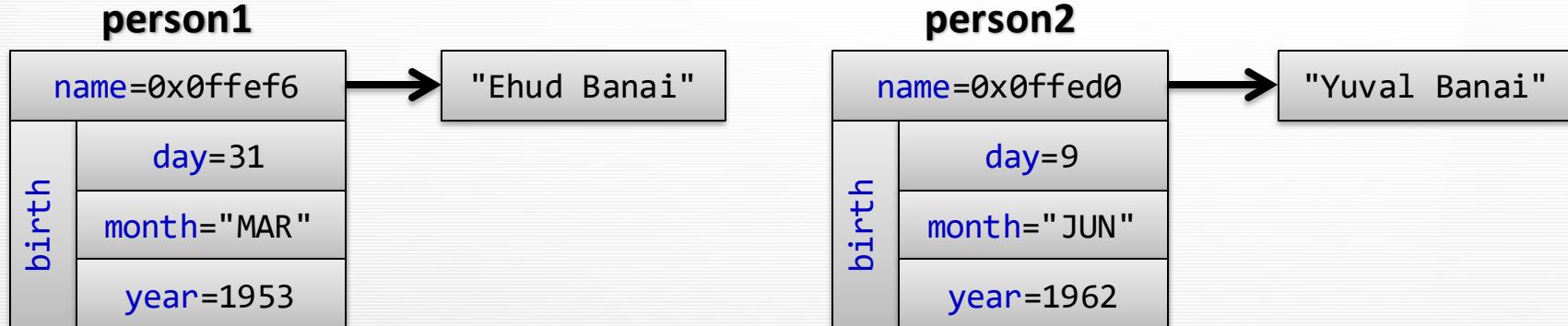
- מבנים מועברים ומוחזרים מפונקציות `value by` – קלומר מועתקים

– גם במקרה זה מתבצעת העתקה שדה-שדה

- הפעולות הללו אינן מתאימות למבנים מסובכים יותר (בד"כ בגלל מצביעים)

# מבנים עם מצביעים

- מבנים המכילים מצביעים אינם מתאימים בדרך כלל לביצוע השמות והעתקות
- מה יקרה אם נבצע השמה בין שני המבנים בדוגמה זו?



- מסיבה זו ו כדי למנוע העתקות כבאות ומירות של מבנים בדרך כלל **נשתמש** ב**מבנה אמצעות מצביעים**

- נשלח לפונקציות (וונקבל כערך חזרה) מצביעים למבנה
- יוצא הדופן הוא מבנים קטנים ופשוטים כמו point

# הגדרת טיפוסים בעזרת `typedef`

- המילה השמורה `typedef` משמשת להגדרת טיפוסים חדשים ע"י נתינת שם חדש לטיפוס קיים

```
typedef int length;
```

- פקודת `typedef` עובדת על שורת הקריאה של משתנה – אך מגדירה טיפוס חדש במקום משתנה.
- נשתמש בפקודת `typedef` כדי לתת שמות נוחים לטיפוסים:

```
typedef struct point Point;
```

- במקרה זה נוכל להתייחס למבנה מעכשו C-`Point` (ללא המילה השמורה `struct`)
- נוח לתת שם גם לטיפוס המצביע למבנה:

```
typedef struct date Date, *pDate;
```

- עבור מבנים מסובכים נשמר את השם ה"נוח" לטיפוס המצביע:

```
typedef struct person *Person;
```

# הגדרת טיפוסים בעזרת typedef

- ניתן להוסיף `typedef` יישורת על הגדרת המבנה:

```
typedef struct point {
 double x;
 double y;
} Point;
```

- ניתן להשמיט את שם הטיפוס בהגדרה ולהשאיר רק את השם החדש:

```
typedef enum { RED, GREEN, BLUE } Color;
typedef struct {
 double x;
 double y;
} Point;
```

# **מבנים - סיכום**

- מבנים מאפשרים לאחד מספר טיפוסים לטיפוס חדש
- מבנה מורכב משדות בעלי שם
  - ניתן לגשת לשדות ע"י האופרטורים . ו->.
- העתקה והשמה של מבנים בטוחה כל עוד אין בהם מצביעים
- מומלץ להשתמש ב-typedef כדי לתת שם נוח לטיפוס החדש

# רישימות מקושرات

- ❖ מבנים המצביעים לעצם
- ❖ רישימות מקושرات

# מבנה המצביעים לעצם

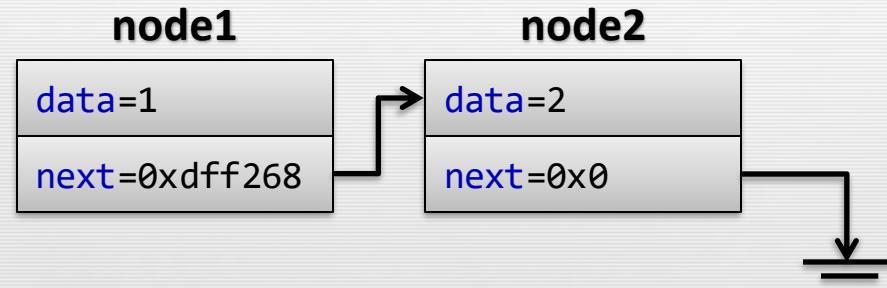
- נסתכל על המבנה הבא:

```
typedef struct node {
 int data;
 struct node* next;
} *Node;
```

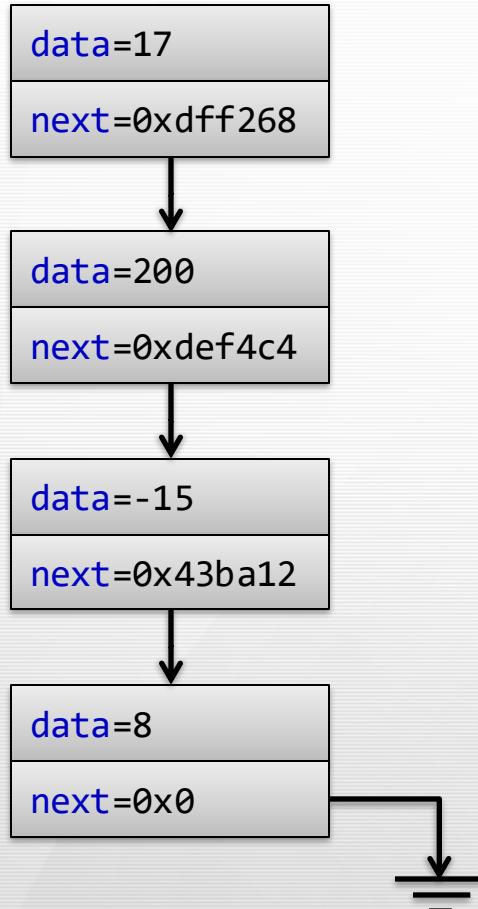
כן, זה מותר!

- איך נראה המבנים בזיכרון לאחר ביצוע הקוד הבא:

```
Node node1 = malloc(sizeof(*node1));
Node node2 = malloc(sizeof(*node2));
node1->data = 1;
node1->next = node2;
node2->data = 2;
node2->next = NULL;
```



# רשימה מקוشرת

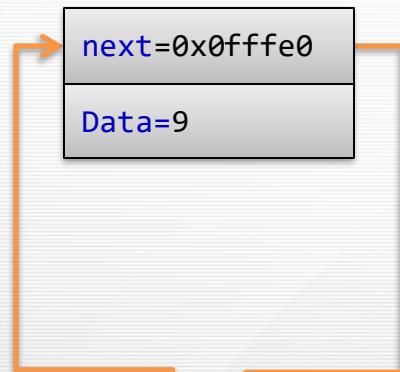


- **רשימה מקוشرת** הינה שרשרת של משתנים בזיכרון כאשר כל משתנה מצביע למשתנה הבא בשרשרת
  - סוף הרשימה מיוצג ע"י מצביע ל-TNULL
- רשימה מקוشرת הינה **מבנה נתונים** המאפשר שיטה מסויימת לשימור ערכים בזיכרון
  - מערך הוא דוגמה נוספת לבנייה נתונים
- רשימה מקוشرת מאפשרת:
  - שימירה של מספר ערכים **שאינו חסום** על ידי קבוע
  - הכנסה והוצאה של משתנים **מאמצע הרשימה** בקלות
  - שרשור פיצול של רשימות בצורה נוחה

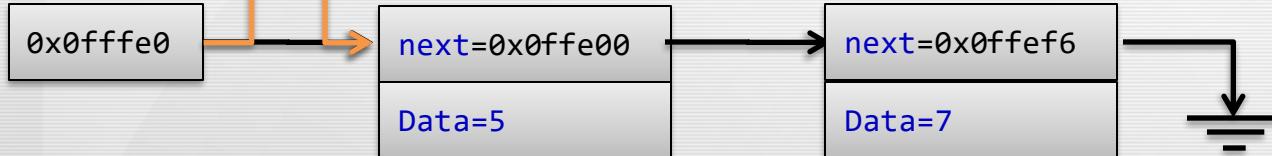
# דוגמה - הדפסה בסדר הפור

- כתבו תוכנית המקבלת רשימת מספרים מהמשתמש ומדפיסה אותם בסדר הפור

**newNode**



**list**



## פתרונות:

- ניצור רשימה מקוורת של מספרים
- לא ניתקל בעיוות בגל חסר הגבלה על גודל הקלט
- בכל פעם שנקלוט תאריך חדש נוכל **להוסיף בклות** לתחילת הרשימה

# פתרון

```
typedef struct node {
 int data;
 struct node* next;
} *Node;
```

```
Node createNode(int d) {
 Node ptr = malloc(sizeof(*ptr));
 if(!ptr) {
 return NULL;
 }
 ptr->data = d;
 ptr->next = NULL;
 return ptr;
}
```

```
void destroyList(Node head) {
 while(head) {
 Node toDelete = head;
 head = head->next;
 free(toDelete);
 }
}
```

למה לא ניתן  
להשתמש ב-  
?Node

למה חייבים  
להקצות דינאמית  
את כל העצמים  
ברשימה?

```
Node insert(Node Head, int d) {
 Node newHead = createNode(d);
 if(!newHead) {
 return NULL;
 }
 newHead->next = head;
 return newHead;
}
```

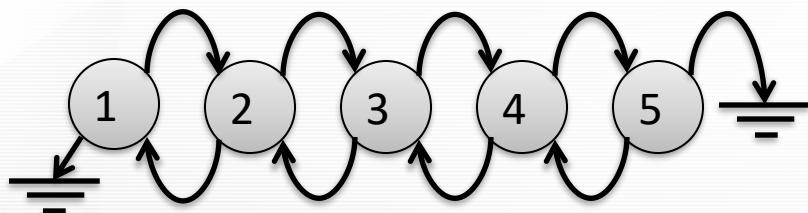
למה צריך את  
?toDelete

כיצד ניתן לכתוב  
קוד זה עם  
רקורסיה?

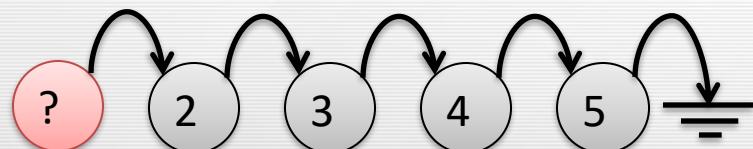
# פתרונות

```
int main() {
 Node head = NULL;
 int input;
 while (scanf("%d",&input)) {
 head = insert(head,input);
 }
 for(Node ptr = head ; ptr != NULL ; ptr=ptr->next) {
 printf("%d",ptr->data);
 }
 destroyList(head);
 return 0;
}
```

# הערות נוספות



- רשימה מקוישת יכולה להיות גם דו-כיוונית.  
במקרה זה לכל צומת שני מצביעים: `next` ו-`previous`
  - יתרונה של רשימה מקוישת דו-כיוונית הוא בסיסיות בלבד - לכן עדיף להתחיל בכתיבה רשימה חד-כיוונית מאחר שהיא פשוטה יותר



- נוח להוסיף **איבר דמה** לתחילת הרשימה כדי **לצמצם את מקרי הקצה** שצרכי לטפל בהם בקוד (למשל, עברור הוצאה של איבר)

# רשימות מקשורות - סיכום

- ניתן ליצור רשימות מקשורות ע"י הוספת מצביע למבנה א בתוך המבנה א
- רשימות מקשורות אינן מוגבלות בגודלן ומאפשרות הכנסה נוחה של איברים באמצעות הרשימה
- את הelts ברשימה יש להקצת דינאמית ולזכור לשחררם
- כאשר מימושים רשימה מקושרת מומלץ להוסיף איבר דמה בתחילתה.
- בדוגמה זו ראיינו רשימת מקשורות של מספרים. כאשר נלמד תכונות גנרי **+c**, נראה כיצד ניתן למשר רשימת מקשורת גנרטית אחת וلتמוך בכל טיפוס של data. ובכך לאפשר שימוש חזר בקוד ולהימנע משכפול קוד.

# תרגול מס' 3

- ❖ מצלעים לפונקציות
- ❖ עבודה עם קבצים
- ❖ וידוא הנחות בזמן ריצה
- ❖ חלוקה למודולים
- ❖ הידור של מספר קבצים

# מצביים לפונקציות

❖ תחביר

❖ דוגמאות

# מצביים לפונקציות

- מה משותף לשתי הפונקציות הבאות?

```
bool isBigger(int a, int b);
```

```
bool isBiggerAbs(int a, int b);
```

- ביצוע קריאה לפונקציות בעלות אותה חתימה נעשה באותה צורה בקוד מוכנה
  1. השמת המשתנים במקום מוגדר (בד"כ על המחסנית)
  2. קפיצה לכתובות תחילת הפונקציה בזיכרון
  3. כתיבת ערך החזרה למקום מוסכם
  4. קפיצה לכתובות ממנה נקראת הפונקציה
- שלבים 1, 3 ו-4 זהים לכל שתי פונקציות בעלות אותה חתימה
  - לכן ניתן לקרוא לפונקציות שונות בעזרת קוד דומה

# מצביים לפונקציות

- ניתן להזכיר על מצביע לפונקציה בעלת חתימה מסוימת:

```
<return type> (*<name>(<parameters>)) = <initial value>;
```

- למשל הכרזה על מצביע עבור פונקציה המקבלת משתנה יחיד מטיפוס int ומחזירה int המאותחל ל-NULL תיראה כך:

```
int (*ptr)(int) = NULL;
```

מה קורה בלי הסוגרים?

- ניתן לאחסן במצביע לפונקציה את כתובתה של פונקציה בעלת חתימה זהה לו שהוגדרה במצביע

```
int square(int n);
...
ptr = square; // &square also works
```

- ניתן לקרוא לפונקציה דרך המצביע:

```
printf("%d", ptr(5)); // (*ptr)(5) also works
```

לא נהוג להשתמש ב-&-ו-\*  
עבור מצביעים לפונקציות

# דוגמה בסיסית

- ברשותנו שתי פונקציות הבאות:

```
bool isBigger(int a, int b) {
 return a > b;
}
```

```
bool isBiggerAbs(int a, int b) {
 int abs_a = a > 0 ? a : -a;
 int abs_b = b > 0 ? b : -b;
 return abs_a > abs_b;
}
```

- נכתוב את הפונקציה **max** המתקבלת מכך לפונקציה ומ>Returns את האיבר הגדול מביניהם בהתאם לкрיטריון שמועבר לה

```
int max(int a, int b, bool (*compare)(int,int)) {
 return compare(a,b) ? a : b;
}
```

- מה יהיו תוצאות כל אחת מההרצות הבאות של **max**?

```
max(-7, 5, isBigger);
max(-7, 5, isBiggerAbs);
```

# דוגמה - מיון

- כתבו פונקציה למיון מערך של מספרים שלמים המאפשרת מיון לפי קритריון משתנה

```
typedef bool (*CmpFunction)(int, int);
```

```
void sort(int* array, int n, CmpFunction compare)
assert(array != NULL && compare != NULL);
for (int i = 0; i < n; i++) {
 for (int j = i + 1; j < n; j++) {
 if (compare(array[i], array[j])) {
 int tmp = array[i];
 array[i] = array[j];
 array[j] = tmp;
 }
 }
}
```

כדי להימנע מהתחביר הלא  
נוח של מצביעים לפונקציות  
ניתן להשתמש ב-  
`typedef`

# דוגמה - מיון

## ■ דוגמה לשימוש:

```
int main(){
 int arr[] = { 1, -3, 9, -10, -5 };
 sort(arr, 5, isBigger); // -10 -5 -3 1 9
 sort(arr, 5, isBiggerAbs); // 1 -3 -5 9 -10
 return 0;
}
```

# **מצביים לפונקציות - סיכום**

- ניתן להגדיר ב-C מצבים אשר שומרים כתובת של פונקציה
- ניתן להריץ את הפונקציה שכותבה שמורה במצב
- השתמש במצבים לפונקציות כדי להעביר "הוראות" לחלקי קוד אחרים
- בעזרת מצבים לפונקציות ניתן לחסוך שכפול קוד

# עבודה עם קבצים

FILE\*

❖ פונקציות שימושיות

❖ דוגמה

# קבצים

- קבצים הם **משאב** מערכת בדומה לזכרון:
  - יש **לפתח** קובץ לפני השימוש בו
  - יש **לסגור** קובץ בסוף השימוש בו
  - אותן בעיות אשר צוות מניהול זיכרון לא נכון עלולות לגרום מניהול קבצים לא נכון
- כדאי להשתמש בקבצים מתוך התכנית שלנו נשתמש בטיפוס הנתונים FILE המוגדר ב-h.`stdio`
- כמו כל טיפוס מורכב השימוש בו נעשה בעזרת מצביעים, למשל שימוש בטיפוס **FILE\***

# פתיחת קובץ באמצעות fopen

- פתיחה קובץ נעשית ע"י הפונקציה **fopen** :
- FILE\* **fopen** (**const char\*** filename, **const char\*** mode);
- filename היא מחרוזת המכילה את שם הקובץ
- mode היא מחרוזת המתארת את מצב הפתיחה של הקובץ
  - "r" עבור פתיחה הקובץ לקרוא
  - "w" עבור פתיחה הקובץ לכתיבה
  - "a" עבור פתיחה הקובץ לשרשור (כתיבה בהמשכו)
- במקרה של כשלון מוחזר NULL

# סגירת קובץ באמצעות fclose

- סגירת הקובץ מתבצעת ע"י קריאה לפונקציה `fclose`:
- ```
int fclose (FILE* stream);
```
- בנגדול ל-`free` שליחת מצביע שהינו `NULL` תגרום להתרסקות התכנית
 - ככלל, לא ניתן לשולח `NULL` לפונקציות קלט/פלט המקבלת *FILE
- גם ניסיון לסגור קובץ פעמיים הוא שגיאה, ועשוי לגרום להתרסקות התכנית
- ערך ההחזרה הוא 0 בהצלחה ו-**EOF** במקרה של כשלון
 - קבוע שלילי המשמש לציין שגיאות בפונקציות קלט פלט, בדרך כלל כאלו שנובעת מהגעה לסוף הקובץ (End of file)

אין צורך לבדוק
הצלהת (`fclose`)
בקורס שלנו

פונקציות קלט/פלט עבור קבצים

- ניתן לכתוב ולקראן מקובץ בעזרת הפונקציות **fscanf** ו- **fprintf**:
`int fprintf (FILE* stream, const char* format, ...);`
`int fscanf (FILE* stream, const char* format, ...);`

- השימוש בהן זהה לשימוש ב- `printf` ו- `scanf` הרגילים מלבד הוספת הפרמטר `stream` שהוא הקובץ אליו יש לכתוב או ממנו יש לקרוא

- `stream` חייב להיות **פתוח** במצב המתאים כדי שהפעולה תצליח
- מצב המאפשר כתיבה עבורי `fprintf`
- – מצב המאפשר קריאה עבורי `fscanf`

- דוגמה - כתיבת המחרחת Hello world בfilename hello.txt לקובץ בשם Hello world
`FILE* stream = fopen("hello.txt", "w");`
`fprintf(stream, "Hello world!\n");`
`fclose(stream);`

מה חסר?

פונקציות קלט/פלט עבור קבצים

- stream מתקבל חוץ (זיכרון עבור ריק) ואת גודלו בbytes, קוראת שורה מ-stream וכותבת אותה אל החוץ

```
char* fgets (char* buffer, int size, FILE* stream);
```

- אם יש יותר מ-1-size תווים בשורה היא תחתר
- הפונקציה מחזיר את buffer אם הקריאה הצליחה, או NULL במקרה של שגיאת קלט.
- במקרה של הצלחה, הפונקציה מוסיפה تو'\'ו' בסוף המחרצת שנקרה.

- כותבת את המחרצת str לטור stream

```
int fputs (const char* str, FILE* stream);
```

- דוגמה - קראת שורה מקובץ:

```
char buffer[BUFFER_SIZE] = "";
fgets(buffer, BUFFER_SIZE, stream);
printf("Line is: %s", buffer);
```

תרגיל copy

- כתבו תוכנית המקבלת כפרמטרים שני שמות קבצים ו-
ומעתיקת את תוכנו של `<file1>` ל-`<file2>`

▪ דוגמה:

```
> cat file1.txt
This is text
in a file
> ./copy file1.txt file2.txt
> cat file2.txt
This is text
in a file
```

פתרון - copy

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

#define CHUNK_SIZE 256

void copy(FILE* input, FILE* output) {
    char buffer[CHUNK_SIZE];
    while (fgets(buffer, CHUNK_SIZE, input) != NULL) {
        fputs(buffer, output);
    }
}

void error(char* message, char* filename) {
    printf("%s %s\n", message, filename ? filename : "");
}
```

פתרון - copy

```
int main(int argc, char** argv) {
    if (argc != 3) {
        error("Usage: copy <file1> <file2>", NULL);
        return 0;
    }
    FILE* input = fopen(argv[1], "r");
    if (!input) {
        error("Error: cannot open", argv[1]);
        return 0;
    }
    FILE* output = fopen(argv[2], "w");
    if (!output) {
        fclose(input);
        error("Error: cannot open", argv[2]);
        return 0;
    }

    copy(input, output);

    fclose(input);
    fclose(output);
    return 0;
}
```

ערוצים סטנדרטיים

- הזכרנו בתרגול מס' 1 את שלושת ערוצי הקלט/פלט הסטנדרטיים: **הקלט הסטנדרטי, הפלט הסטנדרטי וعروץ השגיאות הסטנדרטי**.
- ערוצים אלו מיוצגים ב-C כמשתנים גלובליים מטיפוס *FILE:
 - ערוץ הפלט הסטנדרטי **stdout**
 - ערוץ הקלט הסטנדרטי **stdin**
 - ערוץ השגיאות הסטנדרטי **stderr**
- הטיפוס *FILE משמש כ**הפשטה (abstraction)** של ערוצי קלט/פלט
 - מאפשר עבודה אחידה על ערוצי קלט/פלט שונים ומקל על המשתמש בו

פונקציות קלט/פלט

- לרוב הפונקציות עבור קלט ופלט קיימות גרסאות נפרדות עבור שימוש בערכאים הסטנדרטיים:
 - למשל `printf` עבור `fprintf`
- הפונקציה `puts` כותבת מחרוזת ל-`stdout` ומוסיפה ירידת שורה בסופה
- `int puts (const char* str);`**
- הפונקציה `gets` קוראת שורה מהקלט הסטנדרטי. המחרוזת אינה מכילה את ירידת השורה בתוכה
 - מה חסר כאן? מה הופך את הפונקציה זו למסוכנת?

`char* gets (char* buffer);`

אסור לשתמש ב-`gets` ופונקציות דומות
המאפשרות חריגת מהזיכרון לקלט
זהו מקור לשגיאות זיכרון **וביעות אבטחה**

copy המשופרת

- נ衫ר את התכנית copy:
 - אם התכנית מקבלת שני שמות קבצים היא פועלת כמו קודם
 - אם התכנית מקבלת שם קובץ יחיד היא מדפיסה את תוכנו לפלט הסטנדרטי
 - אם התכנית אינה מקבלת פרמטרים היא מדפיסה מהקלט הסטנדרטי אל הפלט הסטנדרטי
 - כמו כן, הודעות השגיאה של התכנית יודפסו אל ערזן השגיאות הסטנדרטי

```
> ./copy file1.txt
```

```
This is text
```

```
in a file
```

```
> ./copy
```

```
Hello!
```

```
Hello!
```

המשופרת - פתרון copy

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

#define CHUNK_SIZE 256

void copy(FILE* input, FILE* output) {
    char buffer[CHUNK_SIZE];
    while (fgets(buffer, CHUNK_SIZE, input) != NULL) {
        fputs(buffer, output);
    }
}

void error(char* message, char* filename) {
    fprintf(stderr, "%s %s\n", message, filename ? filename : "");
    exit(1);
}
```

הפונקציה `copy` אינה משתמשת
מאחר והוא מתאימה גם
לערכזים הסטנדרטיים

הודעות שגיאה מומלץ
להדפיס לעזרן השגיאות

השימוש ב-`exit` הוא בדרך כלל
תכנות רע, המנוו משימוש בה

המשופרת - פתרון copy

```
FILE* initInputFile(int argc, char** argv) {
    if (argc < 2) {
        return stdin;
    }
    return fopen(argv[1], "r");
}
```

אם לא התקבל ארגומנט עם שם קובץ מהמשתמש, נחזיר את ערך הקלט הסטנדרטי

```
FILE* initOutputFile(int argc, char** argv) {
    if (argc < 3) {
        return stdout;
    }
    return fopen(argv[2], "w");
}
```

המשופרת - פתרון copy

```
int main(int argc, char** argv) {
    if (argc > 3) {
        error("Usage: copy <file1> <file2>", NULL);
    }
    FILE* input = initInputFile(argc, argv);
    if (!input) {
        error("Error: cannot open ", argv[1]);
    }
    FILE* output = initOutputFile(argc, argv);
    if (!output) {
        fclose(input);
        error("Error: cannot open ", argv[2]);
    }

    copy(input, output);
    fclose(input);
    fclose(output);
    return 0;
}
```

איזו בעיה עלולה
להיווצר כאן?

הכוונות קלט/פלט

- מה ההבדל בין שתי הפקודות הבאות? כיצד הוא מתרbeta בתכנית?

```
> ./copy data1.txt data2.txt
```

```
> ./copy < data1.txt > data2.txt
```

עבודה עם קבצים - סיכום

- קבצים הם משאב מערכת – יש להקפיד על ניהול נכון שלהם
- קבצים והערכזים הסטנדרטיים מיוצגים ע"י הטיפוס *FILE
- ניתן לכתוב ולקראם מקבצים בדומה לביצוע פעולות קלט ופלט רגילות
- הערכזים הסטנדרטיים מיוצגים ב-C כמשתנים הגלובליים stdin, stdout ו- stderr
- הכוונות קלט ופלט ניתנות לביצוע בתוך הקוד ומהוצאה לו
- ע"י שימוש ב-*FILE ניתן ליצור פונקציות קלט/פלט כך שיתאימו גם לעבודה עם הערכזים סטנדרטיים וגם עם קבצים

וידוא הנחות בזמן ריצה

- ❖ הערות בתוך הקוד
- ❖ שימוש במאקרו assert
- ❖ כיבוי המאקרו
- ❖ מתי משתמשים ב-assert

הערות בתוך הקוד

- מה הטעיה בקוד זהה?

```
int BMI(float height, float weight) {  
    if (height < 0 || weight <= 0)) {  
        return -1;  
    }  
    return BMI_calculator(height,weight)  
}  
  
int BMI_calculator(float height, float weight) {  
    // If we are here, height and weight are positive values.  
    ...  
}
```

הmacro assert

- המacro assert משמש לוידוא הנחות שנעשות בקוד:

```
assert(<expression>);
```

- המacro מוגדר בקובץ הממשק assert.h ועל מנת להשתמש בו יש לעשות **#include** בזמן היבוג הביטוי מוערך ונבדק
 - **אם הוא נכון** - לא קורה כלום והקוד ממשיר
 - **אם הוא לא נכון** - הוכנית נעצרת ומודפסת הודעה המפרטת את מקום ההנחה שפшла.

- השתמש ב-assert כדי להגן על הקוד מפני הכנסת באגים.

```
//If we are here, height and weight are positive values.—→assert(height>=0 && weight>0);
```

- שינויים עתידיים המפרים הנחות קיימות יגרמו ל**התראות מוקדמות**
- הנחות לא נכונות לגבי הקוד יימצאו כבר בפעם הראשונה שהן אינן מתקינות

```
> ./prog a b
```

```
prog: main.c:12: main: Assertion `argc==2' failed.
```

```
Abort
```

כיבוי המאקרו

- ניתןerc לכבות את המאקרו assert ע"י הגדרת הקבוע **NDEBUG**

#define NDEBUG

- אם DEBUG מוגדר המאקרו יוחלף בקוד שאינו עושה כלום
- כר ניתן לשחרר גרסה סופית של הקוד שאינה מוצעת ע"י הבדיקות ללא הסרתן
ידנית
- ניתן להגדיר את NDEBUG ישירות משורת הידור ע"י הוספת הדגל **-DNDEBUG**
- הדגל <name>- מוסיף בתחילת כל קובץ הגדרה של המאקרו בשם <name>

- שימושו לבב: **קוד שבתוֹר ה-assert לא יורץ כלל אם המאקרו כבוי**
- אסור לשימוש חישוביים הכרחיים לקוד בתוֹר assert.
- מה הבעה כאן? מה הפתרון?

```
assert(doSomethingImportant() != FAILED);
```

מתי משתמשים ב-assert

- ב-assert משתמשים לודאו **נכונות של הנחות** הנעשות בקוד
 - אם ההנחות שגויות יתכן וקיים באגים
 - נוח לבדוק עם assert את נכונות **הารגוונטיים, ערכי החזרה וAINERORIANTOT** של טיפוסי נתונים
- **לא משתמשים** ב-assert כדי לבדוק קלט מהמשתמש
- **לא משתמשים** ב-assert עבור בדיקות שרוצים שתבוצעו גם בתוכנית הסופית (release mode)

```
int getInput() {  
    int input;  
    printf("Enter a positive number:");  
    scanf("%d",&input);  
    assert(input > 0);  
    return input;  
}
```

מה הבעיה ב-assert כאן?
מה צריך לעשות במקום?

שימוש ב-assert - סיכום

- ניתן להשתמש במאקרו assert כדי לוודא קיום תנאים בתכנית
- מומלץ להשתמש ב-assert כדי להקל על דיבוג התכנית
- ניתן לכבות בקלות את התנהלות המאקרו בגרסאות סופיות בעזרת הגדרת NDEBUG
- אסור לשימוש חישובים הכרחיים בתוך assert
- השימוש ב-assert מתאים רק עבור מציאות באגים של המתכנת ואיןו מותאים עבור שגיאות אחרות

חלוקת למודולים

- ❖ תכנות מודולרי
- ❖ מודולים ב-C
- ❖ דוגמה: מודול תאריך

חלוקת למודולים

- אורך החיים של קוד יכול להיות עשרות שנים, לאור תקופה זו יש לתחזק את הקוד
 - תיקון באגים
 - הכנסת תוכנות חדשות
 - התאמת להתקנים חדשים
- הכנסת שינויים לתוכנית קיימת עלולה ליצור באגים חדשים
 - ככל שהתוכנה מסובכת יותר, הסיכוי לטעויות גדול
- הפתרון: **תכנות מודולארי**
 - חלוקת התוכנה למודולים לפי תפקידים
 - מאפשר שימוש חוזר במודולים עבור תוכנות אחרות
 - מסתיר פרטיימוש ומקטין את התלות בקוד

מודול

- מודול תוכנה מחולק לשני חלקים: הממשק והIMPLEMENTATION
 - מגדר את הפעולות שניתן לעשות בעזרת המודול
 - חלק זה חשוב למשתמש במודול
- **ממשק (interface):**
 - מספק את המימוש לפעולות שניתן לבצע דרך ממשק המודול
 - חלק זה אינו חשוב למשתמש
- למשל, עבור מכונית, הגה, דושת גז וברקס הם חלק מהממשק ואילו המנוע הוא חלק מהIMPLEMENTATION.

מודולים ב-C

- כדי לכתוב מודולים ב-C נוצר לכל מודול קובץ C וקובץ H
 - **קובץ ה-H** יכול את ממשק המודול:
 - הכרזות על פונקציות
 - הגדרות טיפוסים
 - הכרזות על קבועים שיש בהם עניין למשתמש (למשל קבוע שגיאה)
 - **קובץ ה-C** יכול את מימוש המודול:
 - מימושי פונקציות הממשק ופונקציות פנימיות נוספות
 - הגדרות טיפוסים לשימוש פנימי
 - הכרזות על קבועים שאין בהם עניין למשתמש
- נראה כעת כיצד יוצרים מודול מטיפוס הנתונים עבור תאריך מתרגול 2

ממשק המודול - date.h

```
#ifndef DATE_H_
#define DATE_H_
```

הגנה נגד include כפלי
מה קורה בלוודיה?

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

#define MIN_DAY 1
#define MAX_DAY 31
#define MONTH_NUM 12
#define DAYS_IN_YEAR 365
#define MONTH_STR_LEN 4
```

```
/**
 * A module for a date datatype
 */
typedef struct date_t {
    int day;
    char month[MONTH_STR_LEN];
    int year;
} Date;
```

ממשק המודול - date.h

```
/** Possible error codes */
typedef enum {
    DATE_SUCCESS, DATE_NULL_ARG, DATE_FAIL, DATE_INVALID
} DateResult;

/** writes the date to the stream fd */
DateResult datePrint(Date date, FILE* fd);

/** Reads a date from the stream fd */
DateResult dateRead(Date* date, FILE* fd);

/** Returns true if both dates are identical */
bool dateEquals(Date date1, Date date2);

/** Returns the number of days between the dates */
int dateDifference(Date date1, Date date2);

/** Checks if the date has valid values */
bool dateIsValid(Date date);

#endif /* DATE_H_ */
```

כדי לאפשר למשתמש שימוש נוח
במודול עליינו לספק לו קודי שגיאה
מפורטים, שכן גדר טיפוס מיוחד
למטרה זו ונשתמש בו

יש לספק תיעוד מפורט יותר
של ערכי החזרה והשגיאה.
תיעוד זה אינו מסופק כאן
נפאת חוסר המקום בשקוף

מימוש המודול - date.c

```
#include "date.h"
#include <string.h>

#define INVALID_MONTH 0

static const char* const months[] = { "JAN", "FEB", "MAR", "APR", "MAY", "JUN",
                                      "JUL", "AUG", "SEP", "OCT", "NOV", "DEC" };

static int monthToInt(char* month) {
    for (int i = 0; i < MAX_MONTH; i++) {
        if (strcmp(month, months[i]) == 0) {
            return i+1;
        }
    }
    return INVALID_MONTH;
}

static int dateToDays(Date date) {
    int month = monthToInt(date.month);
    return date.day + month*(MAX_DAY - MIN_DAY + 1) + DAYS_IN_YEAR * date.year;
}
```

אין צורך להזכיר מחדש מחדש על הפונקציות,
פשוט כוללים את הממשק בקובץ

פונקציות עזר פנימיות אינן
מעניינות את המשתמש ולכן
יש להזכיר עליהן בסיסטיות

מימוש המודול - date.c

```
DateResult datePrint(Date date, FILE* fd) {
    if (!fd) {
        return DATE_NULL_ARG;
    }
    fprintf(fd, "%d %s %d\n", date.day, date.month, date.year);
    return DATE_SUCCESS;
}

DateResult dateRead(Date* date, FILE* fd) {
    if (!date || !fd) {
        return DATE_NULL_ARG;
    }
    if (fscanf(fd, "%d %3s %d", &(date->day), date->month, &(date->year)) != 3) {
        return DATE_FAIL;
    }
    return dateIsValid(*date) ? DATE_SUCCESS : DATE_INVALID;
}
```

IMPLEMENTATION OF THE DATE MODULE - date.c

```
bool dateIsValid(Date date) {
    if (date.month[MONTH_STR_LEN-1] != '\0')
        return false;
    return date.day >= MIN_DAY && date.day <= MAX_DAY &&
           monthToInt(date.month) != INVALID_MONTH;
}

bool dateEquals(Date date1, Date date2) {
    return date1.day == date2.day &&
           strcmp(date1.month, date2.month) == 0 &&
           date1.year == date2.year;
}

int dateDifference(Date date1, Date date2) {
    int days1 = dateToDays(date1);
    int days2 = dateToDays(date2);
    return days1 - days2;
}
```

שימוש במודול התאריך

```
#include "date.h"

int main() {
    Date date1 = { 21, "NOV", 1970 };
    Date date2;
    DateResult result = dateRead(&date2, stdin);
    if (result == DATE_FAIL) {
        fprintf(stderr, "Bad date format\n");
        return 0;
    }
    else if (result == DATE_INVALID) {
        fprintf(stderr, "Invalid date\n");
        return 0;
    }
    datePrint(date1, stdout);
    datePrint(date2, stdout);
    if (!dateEquals(date1,date2)) {
        int diff = dateDifference(date1,date2);
        int absoluteDiff = diff < 0 ? -diff : diff;
        printf("The dates are %d days apart\n", absoluteDiff );
    }
    return 0;
}
```

כעת ניתן להשתמש במודול התאריך בתכניות שונות וגם לעדכן את מודול התאריך בקלות

חלוקת למודולים - סיכום

- נהוג לחלק את הקוד למודולים כדי לאפשר שימוש חזר ותחזוקה נוחה
- מודול מוגדר משני חלקים - מממשק ומימוש
- הממשק מכיל רק את מה שהמשתמש במודול זקוק לו
- ב-C מגדרים את ממשק המודול בקובץ `h` ואת המימוש בקובץ `c`
- יש להגן על קבצי `h` מפני `include` כפוף
- פונקציות פנימיות של המודול יש להגדיר כ-`static`

הידור של מספר קבצים

- ❖ שלבי הידור
- ❖ עיבוד מקדים
- ❖ הידור
- ❖ קישור
- ❖ פונקציות סטטיות

הידור של מספר קבצים

- לא נוח לשמר את כל הקוד בקובץ יחיד עבור תוכנה גדולה
- ניתן לחלק את הקוד למספר קבצים, לפחות כל קובץ בנפרד, ולקשר את כל הקבצים לקובץ הרצה יחיד בסוף התהיליך
- כדי להדר מספר קבצים ב-gcc ניתן פשוט לרשום את כל הקבצים כפרמטרים לשורת הפקודה של gcc:
 > `gcc a.c b.c mytest*.c`

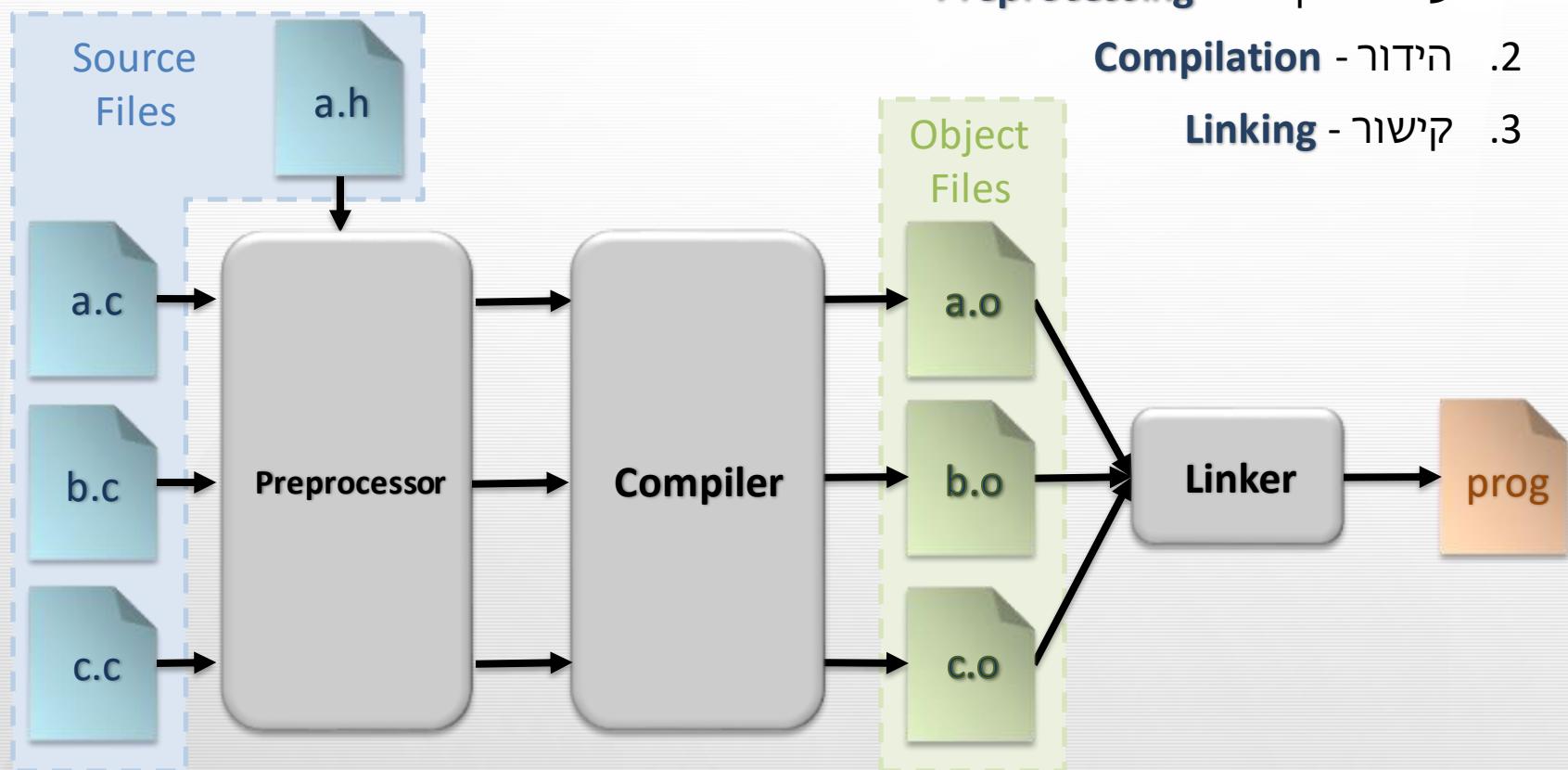
שלבי הקומpileציה

- את הידור הקוד ניתן לחלק לשלושה שלבים:

1. עיבוד מקדים - **Preprocessing**

2. הידור - **Compilation**

3. קישור - **Linking**



עיבוד מקדים – Preprocessing

- שלב העיבוד המקדים עובר על קובץ הקוד ומבצע את ההוראות עברו ה-preprocessor
 - הוראות עברו ה-preprocessor הן כל השורות המתחילה ב-#
 - פועלות העיבוד המקדים הן כולן פעולות גזירה והדבקה פשוטות
- **#define <macro> <value>**
 - הגדרת מacro חדש. לאחר ההגדרה בכל מקום שיפוריע <macro> בקוד תבוצע החלפה ל-<value>
 - ניתן להגדיר יותר מסתם קבועים בעזרת מacro אך חשוב לשים לב להשלכות
- **#include <filename>**
 - הוסיף את תוכן הקובץ לתוך הקוד במקום הנוכחי
 - הקוד שמתווסף עובר גם הוא עיבוד מקדים
 - אם שם הקובץ מופיע בתוך <> התייחסות היא לקובץ מן הספריה הסטנדרטית
 - אם שם הקובץ מופיע במרקאות " " התייחסות היא לקובץ שנמצא עם שאר הקוד שכתבונו
- **#ifdef <macro> ... #endif**
 - אם מוגדר מacro בשם <macro> המשך ברגיל, אחרת מחק את כל הקוד עד להופעת #endif

הידור - Compilation

- שלב ההידור מקבל קבצי קוד מקור שעברו את שלב העיבוד המקדים
 - שלב זה אינו מקבל קבצי `h`, קבצים אלו "הודבקו" לתוך קבצי ה-`c` הרלוונטיים
- לכל קובץ קוד נוצר קובץ בינארי אשר מכיל את הקוד המהודר מקובץ הקוד
 - קובץ זה קרוי **object file** וסיומתו היא `.o`. עבור `gcc`
- יתכן וchlק מהפונקציות שהוכחו לא מומשו ביחידת הקומpileציה הנוכחית,
במקרה זה קובץ ה-`object` יכול "חורים"
- קובץ זה תלוי מהדר ומערכת הפעלה - לכן קובץ אובייקט הנוצר עברו שרת ה-`stud` שונה מקובץ הנוצר עבור מחשב ביתי עם `Windows`

קישור - Linking

- שלב הקישור מקבל כקלט קבצי אובייקט ומקשר את כלם לקובץ הרצה היחיד
- בשלב זה לכל "חור" שהושאר בקובץ אובייקט מקשרת הפונקציה המתאימה
 - אם אכן קיים לה מימוש באחד מקבצי האובייקט
- יתכנו שגיאות בשלב זה הנקבעות שגיאות קישור:
 - לא נמצא מימוש עבור פונקציה מסוימת
- a.c:11: undefined reference to `my_function'
 - נמצא יותר מימוש אחד עבור פונקציה מסוימת
- a.c:10: multiple definition of `main'
- b.c:10: first defined here
- בكمפול של קובץ הרצה, אחד הקבצים (בדיקות) צריך להכיל הגדרה של פונקציית `main`.
 - בשלב הקישור פונקציה זו מסומנת בטור הפונקציה הראשונה שתיקרא כאשר התוכנית מורצת.

פונקציות סטטיות

- ניתן להזכיר על פונקציה כסטטית:

static <function declaration>

- במקרה זה הפונקציה אינה נראית מחוץ לקובץ הקוד שלה וה-linker לא ינסה לקשר אותה לקריאות לפונקציה מיוחדת קומpile'ציה אחרות.
- נשתמש בזה בשביל פונקציות עזר שאינן חלק מהממשק.
- פונקציות סטטיות אינן מוכחות בקובץ.h.
- דוגמה:

```
static int square(int n) {  
    return n*n;  
}
```

- במקרה זה אם תהיה הגדלה של פונקציית square וקריאה ל-square מקובץ קוד אחר, הפונקציה הסטטית לא תקשר לקריאה זו, ולא תהיה התנגשות בשלב הקישור.

הידור של מספר קבצים - סיכום

- ניתן לקמפל מספר קבצים בלבד לקובץ הרצה יחיד
- ההידור ניתן לחלוקת לשולשה שלבים: עיבוד-מקדים, הידור ו קישור
- בשלב העיבוד המקדים מוחלפים מאקרים ומבצעות פעולות `include`
- בשלב ההידור נוצר מכל קובץ `c` אובייקט
- בשלב הקישור מחוברים קבצי האובייקט לקובץ הרצה יחיד
- כדי להסתיר פונקציות מיחידות קומפילציה אחרות ניתן להגדירן כסטטיות
- פונקציות סטטיות אינן חלק מה ממשך ולא נמצאות בקובץ `h`.

תרגול מס' 4

- References ❖ מבוא ל-C++ ❖
- ❖ מחלקות הקצאת זיכרון דינמית ❖
- ❖ מבני נתונים והמחלקה Const ❖
- מחסנית Namespaces ❖

מבוא ל- C++

C++

```
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    cout << "Hello world" << endl;
    return 0;
}
```

- C++ היא שפת תכנות שנבנתה מעל C
- C++ מוסיפה **מגוון רב של תכונות מתקדמות**
 - עברו תכונות מונחה עצמים
 - עברו תכונות גנרי
 - עברו ניהול שגיאות (חריגות)
- השלד הבסיסי של הקוד בשפה זהה ל-C
- רוב הקוד של C ניתן לקומפול ישירות C++
- נהוג ב-C++ לבצע דברים אחרים תוך שימוש ב**יתרונות השפה**
 - לא כותבים ב-C++ **קוד בסגנון C!**
 - **לא נהוג להשתמש ב-C++ ועוד ב-C++ - יש תחליפים עדיפים**

C++

- כדי לkompile קוד C++ נשתמש בкомפיילר **g++**
 - אופן השימוש זהה ל-gcc (למעשה g++ הוא כמעט gcc בשינוי שם)
- קבצי ה-include # הסטנדרטים של C++ הם **לא סיומת**, למשל <iostream>
- עם זאת קבצי header הנקתבים על ידי המתכנתם בעלי סיומת **h** כמו ב-C
- קבצי ה-source ב-C++ הם בעלי סיומת **cpp, cc או C** (גדולה)
- דוגמא לפקודת kompol:

```
g++ -std=c++11 -Wall -Werror -pedantic-errors -DNDEBUG main.cpp -o HelloWorld
```

- DNDEBUG – עברור kompol של גרסה סופית.

הڪצאת ذيرون ديناميٰ

new ♦

delete ♦

הקצת זיכרון דינמית

- הקצאות זיכרון דינמיות ב- C++ מתבצעות בעזרת האופרטורים **new** ו-**delete**:
▪ ניתן להקצות משתנה חדשה (ולאתחלו) על ידי **new**:

```
int* ptr = new int; // No explicit initialization (random value)  
int* ptr = new int(7); // The new integer is initialized to 7
```

- ניתן להקצות מערך של משתנים על ידי **new[size_t]**:
int* array = **new int[n];**

- שחרור של עצם שהוקצה על ידי **new** מתבצע בעזרת **:delete**
delete ptr;

- שחרור של עצם שהוקצה על ידי **new[size_t]** מתבצע בעזרת **:delete[]**
delete[] array;

דוגמאות - delete ו new

```
int* ptr = new int(5);
int* array = new int[*ptr];
// No need to check for NULLs
```

C++

```
delete ptr;
delete[] array;
```

```
int* ptr = malloc(sizeof(int));
if (!ptr) {
    // handle errors ...
}
*ptr = 5;
int* array = malloc(sizeof(*array)* *ptr);
if (!array) {
    // handle errors ...
}

free(ptr);
free(array);
```

C

- אסור להתבלבל בין delete[] ו delete
- אסור לערבות בין free ו malloc לבין delete ו new
- למשל להקצות עם new ולשחרר עם free
- אין צורך לבדוק את ערך המצביע המוחזר מ-new
- new אינה מחזירה NULL
- טיפול בשגיאות ב-C++ מטבח בעזרת מנגןון החריגות (ילמד בהמשך)

הקצת זיכרון דינמית - סיכום

- ב-**C** משתמשים ב-`new` ו-`[size_t]new` כדי להקצת זיכרון חדש
- האופטור `new` מחזיר מצביע מהטיפוס שהוקצה (ולא מטיפוס `*void` כמו `(malloc)`)
- אין צורך לבדוק את הצלחת ההקציה כמו ב-**C**
- ב-**C++** משתמשים ב-`delete` ו-`[[]]delete` כדי לשחרר זיכרון
 - `new` משחררים עם `delete` ו-`[size_t]new` עם `[[]]delete`
- לא משתמשים ב-`malloc` ו-`free` (מלבד במקרה של חיבור קוד C ישן)

קבועים

❖ הגדרת משתנים קבועים

❖ מציבעים קבועים

קבועים - `const`

- בהכרזה על משתנה ניתן להוסיף לשם הטיפוס את המילה השמורה `const`
- **לא ניתן לשנות את ערכו** של משתנה אשר מוגדר כ-`const` (קבוע) לאחר אתחולו
 - לכן חובה לאותחל מעתנאים קבועים

```
const int n = 5;  
n = 7; // error: assignment to read-only variable 'n'  
const int m; // error: uninitialized const 'm'
```

- נכוונות השימוש במשתנים קבועים **נאכפת על ידי הקומפיאילר**

קבועים - `const`

- כאשר מוסיפים `const` להגדרת מצביע יתכו מספר אפשרויות:
 - **הכתובת** הנשמרת במצביע קבועה
 - **הערך** הנשמר במשתנה המוצבע קבוע
- ניתן למקם את המילה `const` במקומות שונים בהגדרת הטיפוס כדי לקבל כל אחת מהפתרונות הרצויות
 - אם ה-`const` מופיע **ראשון** הוא מתייחס לשם הטיפוס הבא אחריו
- `const int* cptr;`
 - בכל שאר המקרים `const` **מתייחס לשמאלו**
- `int const* cptr2;`
- `int* const ptr = NULL; // Must be initialized, why?`
 - ניתן לרשום **יותר מ-`const` אחד**
- `const int* const ptr = NULL;`

קבועים - **const**

- ניתן להמיר מצביעים קבועים רק כאשר המרה היא **מחמירה**, כלומר:
 - להמיר כתובת לא קבועה לכתובת קבועה
 - להמיר מצביע לערך לא קבוע, למצביע לערך קבוע

```
int i = 7;  
int* pi = &i;  
const int* cpi = pi; // this works
```

```
const int i = 7;  
const int* cpi = &i;  
int* pi = cpi; // compilation error!!
```

קבועים - דוגמאות

■ אילו מהשורות הבאות יגרמו לשגיאת קומPILEציה?

- 1) **int** i = 7;
- 2) **const int** ci = 17;
- 3) **const int*** pci = &ci;
- 4) *pci = 7;
- 5) pci = &i;
- 6) **int*** pi = &ci;
- 7) pci = pi;
- 8) pi = pci;
- 9) **int* const** cpi = &i;
- 10) *cpi = 17;
- 11) cpi = &i;
- 12) **int* const** cpi2 = &ci;
- 13) **const int* const** ccpi = &ci;
- 14) ccpi = &ci;

פרמטרים וערבי חזרה

- ניתן להכריז על **פרמטרים** של פונקציה כמצביים `const`:
`char* strcpy(char* destination, const char* source);`
 - מימוש הפונקציה מתחייב לא לשנות את ערך המוצבע על ידי הארגומנט
- ניתן להכריז על **ערך החזרה** של פונקציה כמצבי `const`:
`const char* studentGetName(Student s);`
 - משתמש הפונקציה לא יוכל לשנות את הערך המוצבע על ידי המצביע שmorphozar

קבועים - הערות נוספות

- משתנה אשר קבוע בהקשר מסוים אינו בהכרח קבוע בכל הקשר:

```
void h() {  
    int val; // val can be modified  
    g(&val);  
}
```

```
void g(const int* p) {  
    // can't modify *p here  
}
```

- ב- C++ מגדירים ערכיהם קבועים (לדוגמה, מספרי קסם) בעזרת משתנים קבועים (גלוובליים אם צריך) במקום בעזרת `#define`
 - אפשר לבדוק טיפוסים על ידי הקומpileר
 - אפשר הגדרת קבועים מטיפוסים מורכבים

אפשר גם להגדיר קבועים ב- C++ גם עם enum

```
const int MAX_SIZE = 100;  
const Complex ZERO = complexCreate(0,1);  
static const char* const MONTHS_NAMES[] ={ "JAN", "FEB", "MAR", "APR", "MAY", "JUN",  
                                         "JUL", "AUG", "SEP", "OCT", "NOV", "DEC" };
```

- חשוב להקפיד על שימוש נכון בקבועים (const correctness) ב- C++
 - הקפידו על const מהתחלת, הוספה מאוחרת של const יוצרת **כדור של שינוי**

const - סיכום

- ניתן להגדיר משתנים ב-`++C` קבועים כך שלא ניתן לשנות את ערכם
- ניתן להגדיר פרמטרים וערבי חזרה של פונקציות קבועים
- עבור מצביעים ניתן להגדיר את הערך המוצבע קבוע ו/או את ערך המצביע קבוע
- יש להשתמש ב-`const` כאשר ניתן כדי למנוע שינויים באגים
- הגדרת קבועים ב-`++C` מתרבצת בעזרת משתנים קבועים

Namespaces

❖ שימוש ב-namespaces ❖

Namespaces

- בתוכנית גדולה יש סבירות גבוהה ל-Name Clash – התנגשות בין הגדרות במקומות שונים.
- למשל, פונקציית `create()` בשני קבצים שונים.

```
typedef enum {  
    CONNECT_OK, CONNECT_NO_INTERNET, ...  
} Status;
```

```
Connection* connect(const char* ip_addr);
```

network.h

```
typedef enum {  
    PRINTER_ONLINE, PRINTER_NO_INK, ...  
} Status;
```

```
Printer* connect(const char* port);
```

printer.h

- מה יקרה אם נעשה `#include "printer.h"` וגם `#include "network.h"`...
 - שגיאת קומpileציה...

Namespaces

- ב-`C++` ניתן לחלק קוד למרחבי שמות – `.Namespaces`.
- כמו תיקיות במערכת קבצים היררכית.
- ניתן להגדיר **כל מזהה שנרצה** (טיפוסים, פונקציות, משתנים...) בתוך `.Namespace`.
- אם מזהה לא נמצא באף Namespace אז הוא ב-`"Global Scope"` (מתנהג כמו ב-`C`)

```
namespace fast_math {  
    double sqrt(double);  
    double cos(double);  
    double sin(double);  
}
```

- לאובייקט בתוך Namespace יש **שם מלא** מהצורה `<namespace>::<item>` –
למשל `fast_math::sqrt`, `fast_math::cos`, `fast_math::sin`.

Namespaces

- כדי להשתמש באובייקט מtower Namespace, נציין את שמו המלא ובכך נימנע Name Clashes.
- הספרייה הסטנדרטית של C++ נמצאת תחת ה-`std` Namespace.

```
#include <cmath> // math.h in C
#include "fast_math.h"

void example(double x) {
    cout << std::sqrt(x) << endl;
    cout << fast_math::sqrt(x) << endl;
}
```

Namespaces

- קוד בתוך Namespace יכול להשתמש ב-"שם הפרט" של אובייקטים מאותו Namespace.
- ניתן לגשת לאובייקטים מ-Namespace אחרים באמצעות ה-"שם המלא".

```
#include <cmath> // math.h in C
#include "fast_math.h"

namespace fast_math {
    double norm(double x, double y) {
        return sqrt(x*x + y*y); // calls fast_math::sqrt
    }
}
```

Namespaces

- ניתן לקנן Namespaces – כמו שמקננים תיקיות `<namespace1>::<namespace2>::<obj>`
- הגישה היא באמצעות תחביר של `<obj>`

```
namespace mtm {  
    namespace ex2 {  
        double grade(int student_id) { // mtm::ex2::grade  
            return 100;  
        }  
    }  
}
```

- ניתן להגדיר אובייקטים באותו Namespace מספר קבצים שונים (למשל, כל הקבצים בספרייה הסטנדרטית של C++ ביחד מוסיפים ל-`std`)

using

- ניתן להשתמש בפקודת **using** כדי להפוך שמות מ-namespace זמינים ללא שימוש באופרטור ::.

```
using namespace mtm::ex2;  
// ...  
grade(12346789) // no need to use full name
```

- ניתן גם לבצע **using** רק לשמות ספציפיים מתוך namespace

```
using mtm::ex2::grade;  
// ...  
grade(12346789) // same
```

using

- פקודה `using` רק מקצרת שמות שכבר מוגדרים, היא לא מגירה שמות חדשים

```
#include <cmath>
using namespace std;
// ...
sqrt(5); // allowed
strlen("Hello"); // strlen is not defined, no #include <cstring>
```

- לא נשתמש בפקודה `using` בקבצי `h`.
 - כר נימנע מ-"זיהום שמות" בכל קובץ המבצע `include` לקובץ `h`.

References

❖ שימוש ב-references ❖

❖ const references ❖

משתנים מטיפוס Reference

- משתנה שמצווגר כ- **reference** הוא למעשה שם נוסף למשתנה קיים
- עבור שם טיפוס X, **X&** הוא **רפרנס ל-X**

```
int n = 1;  
int& r = n; // r and n now refer to the same int  
r = 2; // n = 2
```

- העצם אליו מתיחס הרפרנס נקבע בהגדירה שלו, אך **חייב לאותחל רפרנס**
- **int& r; // error: initializer missing**

```
int& r; // a normal assignment between two ints
```

- בשונה ממצביעים, לאחר אותחול הרפרנס לא ניתן להפעיל פעולות "על הרפרנס"
 - או לשנות את העצם שהוא מתיחס אליו

```
int nn = 0;  
int& rr = nn;  
rr++; // nn is incremented by 1
```

משתנים מטיפוס Reference

- **אתחול של רפרנס:**

```
int& x = 1; // error: invalid initialization from temporary value  
int& y = a+b; // same thing
```

- עבור טיפוס X ניתן להגדיר רפרנס קבוע - **const X&**
 - ניתן לאותחל רפרנס קבוע גם על ידי ערכאים זמינים

```
const int& cx = 1;  
const int& cy = a+b;
```

- **פרמטרים (וערכי חזרה)** של פונקציה יכולים להיות רפרנס
 - הרפרנס יאותחל כך שיתיחס לעצם המועבר לפונקציה (מוחזר מהפונקציה)

```
void swap(int& a, int& b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

משתנים מטיפוס Reference

- הפקציה הבאה ממחירה רפרנס לאיבר המקורי במערך:

```
int& getMax(int array[], int size) {  
    int max_id = 0;  
    for (int i = 1; i < size; ++i) {  
        if (array[i] > array[max_id]) {  
            max_id = i;  
        }  
    }  
    return array[max_id];  
}
```

על מנת לטעד שבכוונתו לקבל מערך,
עדיף להשתמש בסינטקס עם סוגרים
רובעים, במקום `.int*`.

- **רפרנס שקול למשתנה** עצמו, שכן ניתן לרשום קוד מהצורה הבאה

```
int array[5] = { 1, 2, 3, 4, 5 };  
getMax(array,5) = 0 ; // array[4] = 0
```

אסור להחזיר רפרנס למשתנה מקומי

אמנם הקוד הזה נראה מחר,
אך בהמשך נראה שימושים
טבעיים עבור החזרת רפרנסים

- מה יקרה אם `getMax` תחזיר רפרנס ל-`max_id` למשל?

משתנים מטיפוס Reference

- קיימים דמיון רב בין רפרנסים למצביים, עם זאת קיימים גם הבדלים חשובים:
 - מצביע יכול להצביע ל-**NULL** בעוד רפרנס תמיד מתיחס לעצם חowi
 - עבור מצביעים יש צורך להשתמש ב-* ו-& כדי לבצע פעולות
 - מצביע (שאינו קבוע) יכול לשנות את הכתובת אליה הוא מצביע
 - עם מצביעים ניתן להשתמש בחשבון מצביעים ובאופרטור [] כדי להתייחס לאיברים שמופיעים לפני ואחרי הערך המוצבע
- עבור טיפוסים מורכבים, מומלץ **להעביר אותם לפונקציות כ-const & X**
 - במקום בהעתקה כשאין צורך לשנותם
 - עבור **משתנים פרימיטיביים** (int, double, bool, enum,...) אין יתרון בבחירה
 - בהעברה by reference, ולכן מומלץ **להעביר אותם by value** (יותר קצר וקריא)

משתנים מטיפוס Reference - סיכום

- עברו טיפוס X, הטיפוס & X מאפשר מתן שם חדש לעצם
- לאחר האתחול של רפרנס לא ניתן לשנות את העצם אליו הוא מתייחס & X const יכול להתייחס גם לערכיהם זמניים (& X לא יכול)
- ניתן להעביר פרמטרים ולהחזיר ערכיהם מפונקציות כרפרנסים אסור להחזיר רפרנס למשתנה מקומי
- מומלץ להעביר פרמטרים שאינם מטיפוסים פרימיטיביים כ-& X const

העמסת פונקציות

- ❖ העמסת פונקציות
- ❖ בחירת פונקציה מתאימה
- ❖ ערכי בירית מחדל

העמסת פונקציות - Function Overloading

- ב-**C++** פונקציה מזוהה על ידי **שםה** וגם על ידי **מספר וסוג הפרמטרים** שלה
 - בניגוד ל-**C** בה פונקציה מזוהה על ידי שם בלבד
- מבחינת **C++** הפונקציות **void print(double)** ו-**void print(int)** שונות
- ניתן ב-**C++** לכתוב מספר פונקציות בעלות אותו שם ו"להעמיס" על אותו שם **פונקציות שונות עבור פרמטרים מטיבוסים שונים**
- **הקומפיילר יבחר** את הפונקציה המתאימה לפי הארגומנטים בקריאה לה

```
void print(int);
void print(double);
void print(const char*);  
// ...  
print(5);           // print(int) called  
print(3.14);       // print(double) called  
print("Hello");    // print(const char*)
```

C++

```
void print_int(int);
void print_double(double);
void print_string(const char*);  
// ...  
print_int(5);
print_double(3.14);
print_string("Hello");
```

C

העמסת פונקציות - בחירת הפונקציה

- הקומפיילר מ Chapman פונקציה בעלת שם מתאים המקבלת את **מספר הפרמטרים המתאים**
- מבין כל הפונקציות המתאימות הקומפיילר מ Chapman את הפונקציה שהפרמטרים`void print(const char);`
`... print('a');`
`int print(int);`
`... print(false);`
`int print(void*);`
`... print(intArray);`שליה הם **מתאים ביותר** לפי הדרוג הבא
 - 1. התאמה **מדויקת** או המרה טריויאלית ($T \leftarrow \text{const}$, $\text{const} \leftarrow \text{מספר}$)
 - 2. התאמה בעזרת **המרת קידום** ($\text{float} \leftarrow \text{double}$, $\text{int} \leftarrow \text{char}$, $\text{int} \leftarrow \text{bool}$ וко')
 - 3. התאמה בעזרת **המרה לאמדויקת** ($\text{void*} \leftarrow \text{T*}$, $\text{int} \leftarrow \text{double}$, $\text{double} \leftarrow \text{int}$)
 - 4. התאמה בעזרת **המראות שהוגדרו על ידי המשתמש** (תרגום הבא)
- עבור יותר פרמטר אחד, הקומפיילר מ Chapman את הרתאמת הטובה ביותר עבור פרמטר אחד והתאמה **שווה או טובה יותר** לשאר הפרמטרים
- אם קיימות שתי התאמות באותו רמה אז הקריאה מכילה **דו-משמעות** והיא לא תתקמפל
- בחירת הפונקציה **אינה תלואה בערך ההחזרה**, לא ניתן לכתוב שתי פונקציות השונות רק בערך ההחזרה שלן

העמסת פונקציות - דוגמאות

- אילו מהפונקציות הבאות תיקרא בכל אחת מהקריאה?

```
void print(int);
void print(double);
void print(const char*);  
  
void print(char);  
  
void h(char c, int i, float f, double d, long l) {
    print(c);
    print(f);
    print(d);
    print(i);
    print(l);
    print('a');
    print(49);
    print(0);
    print("Hello");
}
```

במקרה של דו-משמעות יש להוסיף המרotas מפורשות כדי להבהיר לקומpileר באיזו פונקציה לבחור, למשל:

```
print(int(1));
print(double(1));
```

העמסות יכולות לסייע את הקוד, השתמשו בהן רק אם קורא הקוד יוכל להבין בקלות מה קורה מהסתכלות על הקריאה לפונקציה

ערבי בירית מחדל

- קיימות פונקציות אשר אחד הארגומנטים שלהן הוא ברוב המקרים אותו ערך

```
void print(int n, int base);
```

```
print(6, 10);
print(5, 10);
print(5, 8); // print in octal
print(17, 10);
print(14, 2); // binary
```

- ניתן להגדיר ערך בירית מחדל לפרמטרים עבור פונקציה:

```
void print(int n, int base = 10);
```

```
print(6, 10);
print(6); // same as above
```

ערבי בירית מחדל

- ניתן לתת ערך בירית מחדל רק לפרמטרים האחרונים של הפונקציה:

```
int f(int n, int m = 0, char* str = 0); // ok
```

```
int g(int n = 0, int m = 0, char* str); // error
```

```
int h(int n = 0, int m , char* str = NULL); // error
```

ערבי בירית מחדל

- את ערך בירית המחדל יש לכתוב **פעם אחת בלבד** בהכרזת הפונקציה

num_utils.h:

```
int sum_array(int arr[], int n, int start_val = 0, bool verbose = false);
```

num_utils.cpp:

```
int sum_array(int arr[], int n, int start_val, bool verbose) {
    int sum = start_val;
    for (int i = 0; i < n; ++i) {
        sum += arr[i];
    }
    if (verbose) {cout << "Total sum: " << sum << endl;
    }
}
```

```
sum_array(a, 5, -10, true);
sum_array(a, 5, -10);    // sum_array(a, 5, -10, false);
sum_array(a, 5);        // sum_array(a, 5, 0, false);
```

העמסת פונקציות - סיכום

- ב-++C פונקציה מזוינה לפי שמה והפרמטרים אותם היא מקבלת
- הקומפיילר אחראי לבחירת הפונקציה המתאימה מבין כל הפונקציות המוענסות על אותו שם
- אם אין פונקציה "מנצחת" מתאימה צריך לפתור את דו-המשמעותית
▪ ניתן להגדיר ערכי ברירת מחדל לפרמטרים האחרונים של פונקציה
- הקפידו להשתמש בהעמסת פונקציות רק כאשר היא קלה להבנה על ידי קורא הקוד

מחלקות - Classes

- ❖ משתנים ופונקציות סטטיות
- ❖ מethods
- ❖ מחלקות
- ❖ this
- ❖ private ו-public
- ❖ בנאים והורסים

טיפוסי נתונים – Data types

```
#include <iostream>
#include <cstring>

struct Date{
    int day;
    char* month;
    int year;
};

int main() {
    Date d1 = {21, "NOV", 1970};
    Date d2;
    std::cin >> d2.day >> d2.month >> d2.year;
    if (d1.day == d2.day && strcmp(d1.month,d2.month) == 0 &&
        d1.year == d2.year) {
        std::cout << "The dates are equal\n";
    }
    return 0;
}
```

בC++ אין צורך להשתמש במילה
השמורה struct כאשר מカリים
על משתנים מティפוס המבנה.

אלו בעיות יש
בקוד הבא?

טיפוסי נתונים - Data types

- תאריך הוא **יותר** מהרכבה של שני מספרים שלמים וארבעה תוים
 - לא כל צירוף של ערכים עבור המבנה Date הוא אכן **תאריך חוקי**
 - 2010 5 BLA - אין חודש מתאים ל-”BLA”
 - 31 SEP 1978 - ב-ספטמבר יש רק 30 ימים
 - 29 FEB 2010 - בפברואר 2010 יש רק 28 ימים
- מי שמשתמש במבנה התאריך צפוי להשתמש בו **בצורות מסוימות**
 - אתחול תאריך תקין
 - מציאת התאריך המוקדם יותר מבין שני תאריכים
 - מציאת מספר הימים בין שני תאריכים

טיפוסי נתונים - Data types

- כדי לוודא את נכונות השימוש בתאריכים ולמנוע את שכפול הקוד בשימוש בתאריכים علينا לכתוב **פונקציות** מתאימות לטיפול בתאריכים
- לצירוף של טיפוס והפעולות האפשריות עליו קוראים **טיפוס נתונים** - **Data type**
 - טיפוסי הנתונים המובנים בשפה נקראים **טיפוסי נתונים פרימיטיביים**
 - למשל `int`, `float` ומצביים (לכל אחד מהם פועלות שונות אפשריות)
 - יצרת טיפוסי נתונים מהוות את הבסיס לכתיבה תוכנה גדולה בצורה מסודרת ופешטה.

```
struct Date {  
    int day, year;  
    char* month;  
    int daysDifference(const Date& p);  
....  
};
```

טיפוסי נתונים ב-C++

```
struct Point {  
    double x, y;  
  
    double distance(const Point& p) {  
        double dx = this->x - p.x;  
        double dy = this->y - p.y;  
        return sqrt(dx*dx+dy*dy);  
    }  
  
};  
  
int main() {  
    Point p1 = { 3, 4 };  
    Point p2 = { 2, 8 };  
    double d = p1.distance(p2);  
    return 0;  
}
```

- יצירת טיפוסי נתונים הותמעה בשפת C++
- ב-C++ ניתן להגדיר את הפונקציות עבור טיפוס הנתונים **ישירות בתוך המבנה**
- פונקציות המוגדרות כחלק מהמבנה נקראות **METHODS (methods)** או **פונקציות חברות (member functions)**
- כדי להפעיל METHOD יש צורך **בעצמו** מהטיפוס המתאים להפעיל אותה עליו

METHODS - מетодות

```
struct Point {  
    double x, y;  
  
    double distance(const Point& p);  
};
```

Point.h:

```
double Point::distance(const Point& p) {  
    double dx = this->x - p.x;  
    double dy = this->y - p.y;  
    return sqrt(dx*dx+dy*dy);  
}
```

Point.cpp:

- מетодות ניתן לממש ישירות בתחום המבנה (כמו בשקף הקודם) או להכריז עליהן במבנה ולממשן מחוץ לו:

- **הגדנות טיפוסים** ב-**C++** יופיעו בדרך כלל בקובץ **h**

- אם מימוש הfonקציה נעשה **בתוך המבנה** אז הוא יופיע בקובץ **h** (בדרכם אליו יהיו פונקציות קצרות)

- אם מימוש הfonקציה **חיצוני** נשים אותה בקובץ **h-cc/cpp/C** המתאים

המצבי^{יע}

```
struct Point {  
    double x, y;  
  
    double distance(const Point& p) const;  
    void set(double x, double y);  
};
```

Point.h:

```
void Point::set(double x, double y) {  
    this->x = x;  
    this->y = y;  
}  
  
double Point::distance(const Point& p) const {  
    double dx = x - p.x;  
    double dy = y - p.y;  
    return sqrt(dx*dx+dy*dy);  
}
```

Point.cpp:

הדוע לא ניתן להגדיר
את set כ-const

- לכל מתודה של עצם מטיפוס X נשלח מצביע מטיפוס **X*** שבו **this**

- אין צורך להשתמש במצביע this כל עוד אין דו-משמעות

- ניתן להגדיר מתודה כך שתפעל גם על עצמים שהינם **const** על ידי **הוספת const** בסוף חתימת הפונקציה

- במתודה המוגדרת כ-const, המצביע **const X* const this** יהיה מטיפוס **this**

- אם מתודה מוגדרת כ-const, **כל השדות** של המצביע this (כלומר השדות של העצם עליו נקראת המתודה) יהיו קבועים

בקורת גישה - Access Control

```
struct Point {  
private:  
    double x, y;  
  
public:  
    double distance(const Point& p) const;  
    void set(double x, double y);  
};  
// ...  
int main() {  
    Point p;  
    p.x = 5; // error: 'int Point::x'  
             // is private  
    p.set(3, 4);  
    double d = p.distance(p);  
  
    return 0;  
}
```

- כדי להסתיר את מימוש הטיפוס מהמשתמש ניתן להגדיר חלקים פרטיים וחלקים פומביים
- פונקציות המוגדרות בתחום הטיפוס רשאית לגשת לחלקים הפרטיים
- קוד אשר כתוב מחוץ למבנה אינו יכול ל党的十 חלקים אלו
- לכל פונקציה, שדה או טיפוס בתחום struct מוגדרת בקורת גישה כלשהי
 - אם לא צוין אחרת, הם public
 - פונקציות עזר וכן השדות של הטיפוס יוגדרו בד"כ C-struct

מחלקות - Classes

- בדרכם כלל מגדירים טיפוסים ב-C++ עם המילה השמורה **class**
- ההבדל בין **class** ל-**struct** הוא בברירת המחדל עברו בקורת הגישה - **public** עברו בקורת הגישה - **private** ו-**struct** עברו בברירת המחדל עברו בקורת הגישה - **public**.
- נהוג להשתמש ב-**struct** עבור טיפוסים פשוטים שככל שדתויהם הינם **public**.
- טיפוס שבו השימוש מושתר ורק הממשק נתון למשתמש נקרא **Abstract Data Type** - **ADT**.
- **ADT** מוגדר רק על ידי **הפעולות** שהטיפוס תומך בהן, ולא על ידי השימוש. לעיתים אותו **ADT** ניתן למימוש בכמה דרכים, במקרה זה הקוד של המשתמש לא יהיה תלוי במימוש הספציפי שנבחר).

```
class Point {  
    double x, y;  
  
public:  
    double distance(const Point& p) const;  
    void set(double x, double y);  
};
```

```
struct Point {  
private:  
    double x, y;  
  
public:  
    double distance(const Point& p) const;  
    void set(double x, double y);  
};
```

בנאים - Constructors

```
class Point {  
    double x, y;  
  
public:  
    Point(double x, double y);  
    double distance(const Point& p) const;  
};
```

Point.h:

```
Point::Point(double x, double y) {  
    this->x = x;  
    this->y = y;  
}
```

Point.cpp:

```
int main() {  
    Point p1(3, 4);  
    const Point p2(2, 8);  
    cout << p1.distance(p2) << endl;  
    return 0;  
}
```

main.cpp:

- לכל מחלקה ניתן להגדיר סוג מיוחד של מתודות הנקראות **בנאים (Constructors)** או **C'tor** (בקיצור)

- מתודת בניאי מזוהה על ידי כר ששם הוא שם המחלקה, ואין לה ערך החזרה

- בנאים משמשים לאתחול של עצם חדש מהמחלקה

הורסאים - Destructors

```
class Array {  
    int* data;  
    int size;  
  
public:  
    Array(int size);  
    ~Array();  
    // More methods ...  
};
```

Array.h:

```
Array::Array(int size) {  
    data = new int[size];  
    this->size = size;  
}  
  
Array::~Array() {  
    delete[] data;  
}
```

Array.cpp:

```
int main() {  
    Array array(50);  
    // code ...  
    return 0;  
} // d'tor called
```

main.cpp:

- לכל מחלקה ניתן להגדיר סוג נוסף של מתודה
הקרויה הורס (D'tor Destructor)
- מתודת ההורס היא בעלת אותו שם כשם
המחלקה עם תחילית `~`, וללא ערך החזרה
- ההורס של המחלקה נקרא אוטומטית
בזמן שחרור עצם של המחלקה
- עוד על בנאים והורסים בתרגולים הבאים

פונקציות ושדות סטטיים

```
class Point {  
    double x, y;  
    static Point origin;  
public:  
    Point(double x, double y);  
    double distanceFromOrigin() const;  
    static void setOrigin(double x, double y);  
};
```

Point.h:

```
Point Point::origin(0,0);  
  
double Point::distanceFromOrigin() const {  
    double dx = x - origin.x;  
    double dy = y - origin.y;  
    return sqrt(dx*dx + dy*dy);  
}  
  
void Point::setOrigin(double x, double y) {  
    origin.x = x;  
    origin.y = y;  
}
```

Point.cpp:

- ניתן להגדיר **משתנים סטטיים** במחלקה, המשתנים אלו אינם שייכים לעצם ספציפי
- ניתן להגדיר **מתודות סטטיות**, מетодה סטטית **אינה מקבלת** **מצבייע this** ולא דרוש עצם כדי להפעילה
- משתנים ומетодות סטטיים מצויים לחוקי בקרת הגישה
 - אם למשל נגדיר משתנה סטטי כפרטי הוא יהיה נגיש רק מתוך המחלקה

מחלקות - סיכום

- ב-++ C ניתן להגדיר מethodות כחלק מהטיפוסים מethodות מקובלות פרמטר נסתר, `this`, אשר מאותחל להצביע לעצם עליו הפעלה המתודה כדי למנוע גישה מהמשתמש לימוש ניתן להגדיר חלקים מהמחלקה כפרטיים ואת המשך כפומבי.
- טיפוס שחווש רק את המשך למשתמש ולא את השימוש שלו נקרא ADT.
- ניתן להגדיר לכל מחלקה בנאים אשר ישמשו לאתחול משתנים חדשים מהטיפוס.
- ניתן להגדיר לכל מחלקה הורס אשר ישמש לשחרור משתנה ניתן (ונדרש) לסמן מethodות שאינן משנה את מצב האובייקט כ-`const`.
- ניתן להגדיר למחלקה משתנים ומethodות סטטיים אשר אינם שייכים לעצם ספציפי

מבני נתונים ומחלקות מחסנית

בעיה לדוגמה

- נרצה לקלוט עד 100 מספרים אי שליליים מהקלט ולהדפיס אותם בסדר הפוך
- בזמן הכניסת הקלט המשמש יכול להתרחט ולבטל את הכניסת המספר האחרון
 - לצורך כך הוא צריך להכנס 1.
- פעולה הביטול דומה לפעולה "undo" בעורכי טקסטים
- המשמש יכול לבצע "undo" כמה פעמים ולבטל כמה מספרים
- ברגע שהתקבלו 100 מספרים, התוכנית מדפיסה אותם בסדר הפוך ומסיימת את ריצתה.
- נפתר תחילה את הבעיה זו ישירות

פתרון ישיר

```
#include <iostream>
#include <cassert>

const int MAX_SIZE = 100;
const int UNDO_LAST_COMMAND = -1;

int main() {
    int input, size = 0, numbers[MAX_SIZE];
    while (size < MAX_SIZE && std::cin >> input) {
        if (input != UNDO_LAST_COMMAND) {
            assert(size >= 0 && size < MAX_SIZE);
            numbers[size++] = input;
        }
        else if (size > 0) {
            size--;
            std::cout << "undo" << std::endl;
        }
        else {
            std::cout << "Cannot undo" << std::endl;
        }
    }
    while (size > 0) {
        std::cout << numbers[--size];
        assert(size >= 0 && size < MAX_SIZE);
    }
    return 0;
}
```

ב++c נכלל את המאקרו
cassert assert

מדוע השימוש

בassert נכון כאן?

קוד קשה להבנה.

חסרונות הפתרון הישיר



- לא ניתן לעשות שימוש חזר בקוד עברו בעיות דומות
- **קל להכניס באגים**
 - ?--size או size--
 - ?++size או size++
 - ?size >= 0 או size > 0
 - ?size < 0 או size < 1
- **הפתרון אינו מתעד את עצמו, קשה להבחין ש:**
 - מוסיפים רק לסוף המערך
 - מורידים מספרים רק מסוף המערך
 - ההדפסה מתבצעת רק בסדר הפוך
- עברו בעיה גדולה יותר, כבר לא ניתן לשמר על הקוד פשוט כמו במקרה זה

מבנה נתונים

- **מבנה נתונים** הם טיפוסי נתונים מיוחדים שמטרתם לשמור אוסף של ערכים ולבסוף עליהם פועלות מסויימות

Array

Get(i)
Set(i)

Linked List

Head()
Next(node)
InsertAfter(node)
RemoveAfter(node)

דוגמאות:

- **מערך** - הממשק של מערך כולל קראת איברים לפי אינדקס והשמה לאיברים לפי אינדקס

- **רשימה מקוושרת** - הממשק של רשימה מקוושרת כולל קבלת האיבר הראשון, קבלת האיבר שבא אחרי איבר נתון, והכנסה/הוצאת של איבר אחרי איבר נתון

- נוח לכתוב מבני נתונים נוספים כטיפוס נתונים (מחלקה) ולהשתמש בהם עבור בעיות מתאימות

מחסנית - Stack

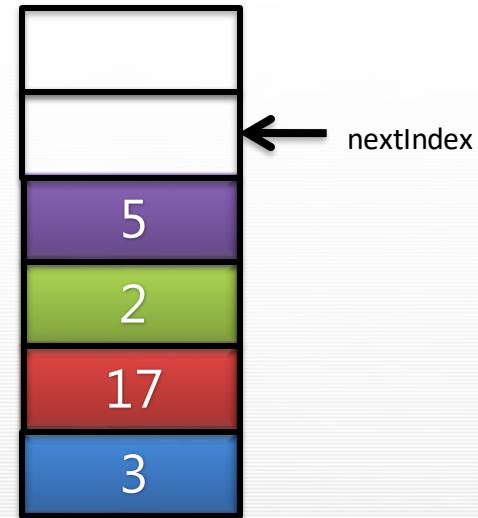


- מבנה הנתונים **מחסנית** מוגדר לפי הממשק הבא:
 - **push** - הוסף איבר בראש המחסנית
 - **pop** - הוצא את האיבר האחרון שהוכנס למחסנית (ambil להחזיר את ערכו)
 - **top** - החזר את ערכו של האיבר האחרון שהוכנס למחסנית (ambil להוציאו)
- מחסנית מאפשרת גישה רק לאיבר האחרון שהוכנס ורק אותו ניתן להוציא ברגע נתון (LIFO - Last In First Out)

ממשק מחלקת מחסנית

Stack.h:

```
class Stack {  
    int* data;  
    int maxSize;  
    int nextIndex;  
  
public:  
    Stack(int maxSize);  
    ~Stack();  
    int getSize() const;  
    void push(int n);  
    void pop();  
    int top();  
    bool isFull() const;  
    bool isEmpty() const;  
};
```



פתרון הבעה בעזרת מחסנית

```
#include <iostream>
#include "Stack.h"

const int MAX_SIZE = 100;
const int UNDO_LAST_COMMAND = -1;

int main() {
    Stack stack(MAX_SIZE);
    int input;
    while (!stack.isFull() && std::cin >> input) {
        if (input != UNDO_LAST_COMMAND) {
            stack.push(input);
        }
        else if (!stack.isEmpty()) {
            stack.pop();
            std::cout << "undo" << std::endl;
        } else {
            std::cout << "Cannot undo" << std::endl;
        }
    }
    while (!stack.isEmpty()){
        std::cout << stack.top();
        stack.pop();
    }
    return 0;
}
```

מימוש המחשבנית

Stack.cpp:

```
Stack::Stack(int maxSize) {
    data = new int[maxSize];
    this->maxSize = maxSize;
    nextIndex = 0;
}

Stack::~Stack() {
    delete[] data;
}

void Stack::push(int n) {
    if (this->isFull()) {
        error("Stack full");
    }
    data[nextIndex++] = n;
}

int Stack::getSize() const {
    return nextIndex;
}
```

```
void Stack::pop() {
    if (this->isEmpty()) {
        error("Stack empty");
    }
    nextIndex--;
}

int Stack::top() {
    if (this->isEmpty()) {
        error("Stack empty");
    }
    return data[nextIndex - 1];
}

bool Stack:: isFull() const{
    return this->getSize() >=
           maxSize;
}

bool Stack:: isEmpty() const{
    return nextIndex <= 0;
}
```

מבנה נתונים ומחסנית - סיכום

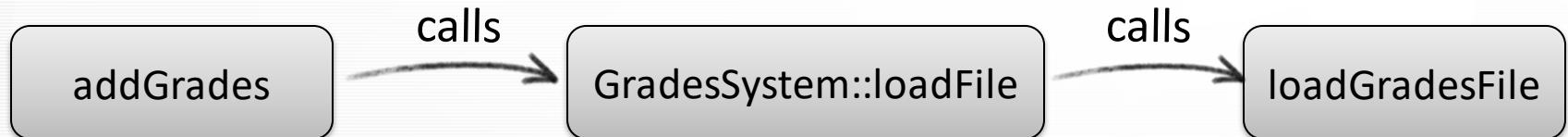
- ניתן להציג מבני נתונים כ-ADT, ספציפית במחלקות.
- ע"י פתרון הבעיה עם מבנה המחסנית מתאפשר פתרון עם סיכוי קטן יותר לבאגים.
- ניתן לבדוק ולדבג את המחלקה של המחסנית בנפרד מהקוד של התוכנית הראשית.
- הפתרון עם המחסנית קל לקרוא ונתען את עצמו.
- שימוש במבנה נתונים מונע שכפול קוד ומאפשר שימוש חזר בקוד.
- שימוש במבנה נתונים מקל על המתכנת בכתיבה קוד.

תרגול מס' 5

- ❖ טיפול בשגיאות באמצעות חריגות
- ❖ הגדרת מחלקות של חריגות
- ❖ חריגות ובנאים/הורסים
- ❖ שימוש נכון בחריגות

טיפול בשגיאות ב C++

- נניח שיש לנו מערכת לניהול ציונים שצריכה לקרוא ציונים חדשים מקובץ:



1. מקבלת שם קובץ מהמשתמש
2. טוענת את הקובץ ומוסיפה
את הציונים למערכת

1. טוענת את קובץ הציונים
2. מוסיפה את הציונים למערכת

1. פותחת את הקובץ הנוכחי
2. קוראת את הציונים
3. מחזירה את הציונים

הקובץ עשוי להכיל תוכן לא
חוקי, להיות בפורמט לא נכון,
או כלל לא להיות קיים

- איך על הפונקציה `loadGradeFile` להגיב במצב של שגיאה?

ניסיונן : Abort

- כאשר נתונים נתקלים בשגיאה, יוצאים מהתוכנית.
 - עזרת exit(), abort(), assert וכו'.
- הכי פשוט שאפשר.
- **חסרונות:** לא מקובל שתוכנית אמיתית תסתהים בלי לתת חיוי מסודר כלשהו למשתמש, או תנסה להטאוש מהשגיאה (לדוגמה, תשאל את המשתמש לשם קובץ חדש). תוכנית שמסיימת כר עלולה אף לאבד מידע או לעשות נזק!



1. מקבלת שם קובץ מהמשתמש
2. טוענת את הקובץ ומוסיפה
את הציונים למערכת

1. טוענת את קובץ הציונים
2. מוסיפה את הציונים למערכת

1. פותחת את הקובץ הנתון
2. קוראת את הציונים
3. מחזירה את הציונים

ניסיון 2: החזרת ערך לא חוקי

- כאשר קורית שגיאה, נחזיר **ערך מיוחד** שМОוחזר רק בעת שגיאה.

```
FILE* fopen();
```

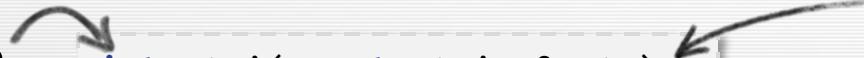


מחזר **NULL** בעת כישלון

לא תמיד אפשרי:

ממיר מחרחת של
ספרות ל-**int**

```
int stoi(const string& str);
```



כל **int** הוא ערך
החזרה אפשרי!

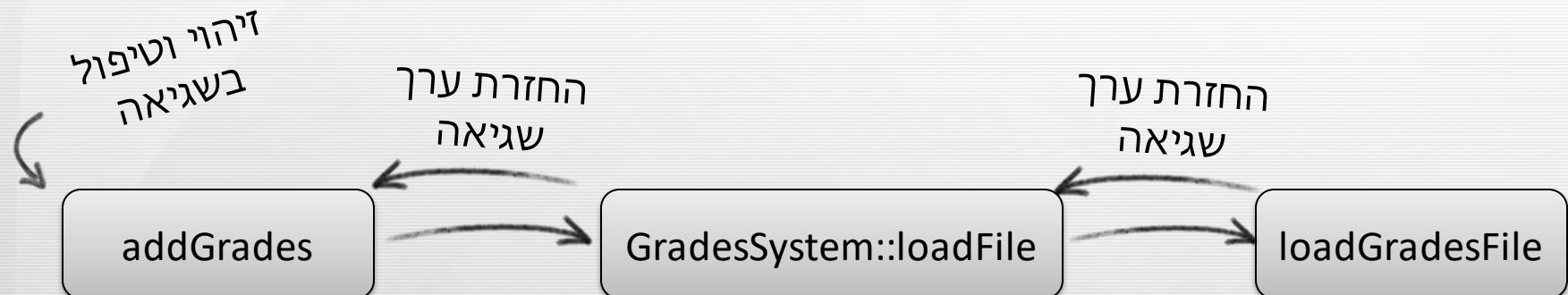
ניסיון 3: החזרת ערך שגיאה

- נשתמש בטיפוס מסויים (enum) בתור ערך חזרה.

```
Status sqrt(double x, double* result);
```

חסרונות:

- הfonקציה נדרשת להחזיר ערך החזרה **לא טבעי ולא נוח למשתמש**, רק כדי לטפל במקרה קצה. הדבר אינו מאפשר הרכבה של קריאות לפונקציות הללו ושימוש בערך ההחזרה שלهن בחישובי המשך.
- צורך לפעוף באופן ידני את השגיאה במעלה מחסנית הקריאה.
- עבור פונקציות ממודולים שונים, יש צורך לעיתים קרובות "לתרגם" ערכי שגיאה מטיפוס אחד לערכי שגיאה מטיפוס אחר (שנתאים למודול הנוכחי) – הרבה עבודה ידנית ועינה.
- נראה בעתיד: אי אפשר להודיע כרגע על שגיאות אופרטורים של C++



ניסיון 4: שימוש במשתנים גלובליים

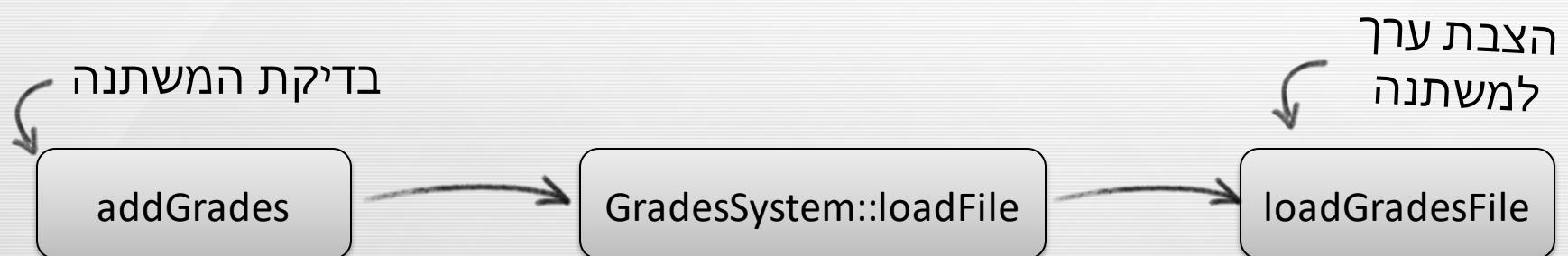
▪ שימוש **במשתנה גלובי**

- דוגמה 'errno' בספרייה הסטנדרטית של C

```
FILE* f = fopen("hello.txt", "r");
if (f == NULL) {
    printf("Error %d\n", errno);
}
```

▪ **חסרונות:**

- קל **להטעם מושגיות**.
- **מספר שגיאות** יכולות לקרות לפני שמספיקים לבדוק את המשתנה
- לעיתים צריך **זיכרון** לאפס את ערך המשתנה

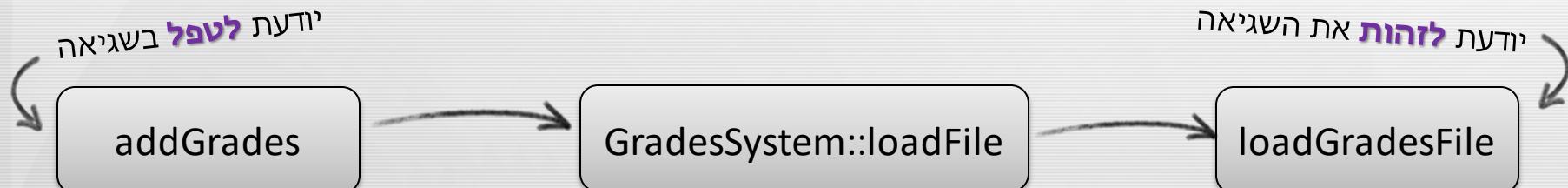


קשיים בטיפול בשגיאות

- **סיום פתאומי** של ריצת התוכנית זו לרוב לא אופציה.
- טיפול בשגיאות **מעמיס על הקוד** עם הרבה בדיקות, פוגע בקריאות הקוד, ופותח פתח לבאים.
 - הקוד שטוף בשגיאות יכול בקלות **להתנפח** ולתפוס יותר מקום מהקוד שמבצע את הלוגיקה הרצiosa של התוכנית
- **קל לשכח או להטעם** מבדיקה של שגיאות.
 - המתכנת צריך **באופן יום** לבדוק את ערכי ההחזרה, ערכי השגיאה או המשתנים הגלובאליים.
 - התוכנית עלולה למשיר **לרווח במצב של שגיאה** במידה והקוד לזייהו וטיפול בשגיאות הוחנה.

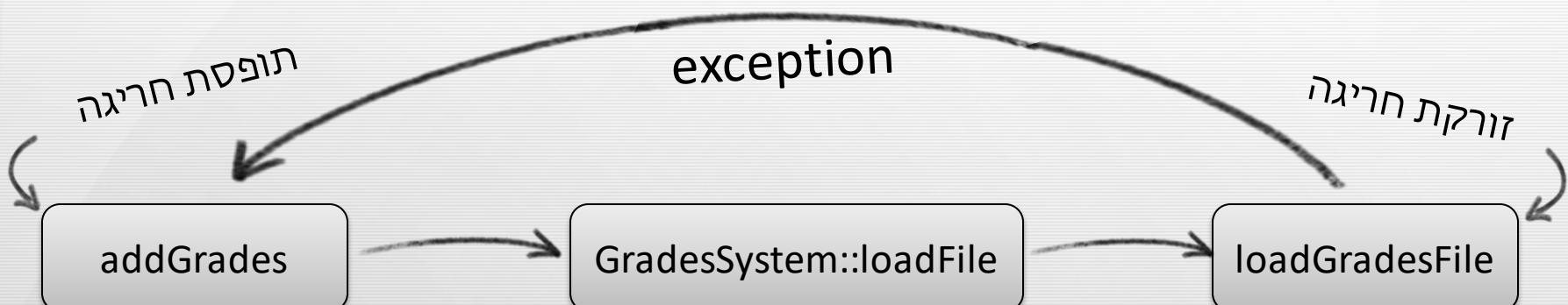
מבוא לטיפול בשגיאות

- לרוב יש 2 חלקים קוד שמעורבים בטיפול בשגיאה:
 - הfonקציה **שזהה** את השגיאה.
 - הfonקציה **שמטפת** בשגיאה.
- בדרכם כלל אלו פונקציות שונות.
- למה?**
 - שגיאות לרוב מוחות בפונקציות פנימיות** של התוכנית. לפונקציות אלה בד"כ אין את הקונטקס שבו הן נקראו - הן לא יודעות מה המצב הכללי של המערכת, ולרוב אין להן יכולת לתקשר עם המשתמש.
 - הפונקציות שודיעות את מצב התוכנית הן לרוב הפונקציות החיצונית יותר במערכת.** אלו בד"כ הפונקציות שודיעות כיצד יש לטפל בשגיאה ויש להן גם את היכולת לתקשר עם המשתמש
 - להודיע שקובץ לא קיים, לדוח שנגמר הזיכרון והפעולה שביקש אינה תבוצע, וכו'
 - אנו זקוקים בדרך עילית לקשר בין הפונקציה הפנימית שזהה את השגיאה, לחוץ החיצונית שיחדעת לטפל בשגיאה.**
 - ב-C זה נעשה על ידי שרשרת ידנית של return, ב-C++ יש מערכת שעשויה זאת אוטומטית.



טיפול בשגיאות ב C++

- ב C++ יש מנגנון חריגות (**exceptions**) שבו ניתן לטפל בשגיאות.
 - פונקציה שמצויה את השגיאה אבל לא יכולה לטפל בה **זורקת חריגה (throw)**.
 - פונקציה ש יודעת איך לטפל בשגיאה אבל היא לא יודעת מה השגיאה, יכולה **لتפוס את החריגה (catch)** ולטפל בה.
 - פונקציה שלא יודעת את השגיאה, וגם לא יודעת איך לטפל בה, יכולה **להתעלם ממנה**.



טיפול בשגיאות ב C++

- ניהול חריגות ב C++ נעשה על ידי 3 מילים שמורות.
- - משמשת את הפונקציה **stdexcept** את השגיאה בשל לזרוק חריגה. מקבלת כפרמטר את העצם אותו יש לזרוק כחריגה.
- - משמשת את הפונקציה **try** בשגיאה. מצינית תחילת בлок של קוד אשר יתכן ותירק ממנו חריגה שנרצה לתפוס.
- - משמשת גם כן את הפונקציה **catch** בשגיאה. מצינית את סוג החריגה שנרצה לתפוס ואת הקוד שיש לבצע כדי לטפל בחריגה.

```
int main() {  
    try {  
        printGradesFile("grades.txt");  
    } catch (string s) {  
        cerr << s << endl;  
    }  
    return 0;  
}
```

```
void printGradesFile(string filename) {  
    FILE* f = fopen(filename, "w");  
    if (!f) {  
        throw string("cannot open file");  
    }  
    int grade;  
    while (fscanf(f, "%d", &grade)) {  
        cout << grade << endl;  
    }  
}
```

זריקת חריגה

```
int factorial(int n) {  
    if (n < 0) {  
        throw -1;  
    }  
    if (n <= 1) {  
        return 1;  
    }  
    return n * factorial(n - 1);  
}
```

```
int choose(int n, int k) {  
    return factorial(n) /  
        (factorial(k) * factorial(n - k));  
}
```

```
int main() {  
    cout << choose(2, 4) << endl;  
    cout << choose(-1, 4) << endl;  
    return 0;  
}
```

- כasher נזרקת חריגה, הפונקציה הנוכחית **מפסיקה להבצע מידית**
 - הfonקציה הנוכחית איננה מחזיר ערך חזרה
 - במקום זאת, החריגה שנזרקה מועברת לפונקציה הקוראת את הפונקציה שקרה לה איננה תופסת את החריגה, **התהlijr ממשיר** והfonקציה הקוראת מסתiya גם היא מיד, והחריגה מפעפעת למעלה במחסנית הקראות.
- אם חריגה **לא מטופלת** כלל (זרקת מ-main) **התכנית קורשת**

תפישת חריגה

- פונקציה מצוינת שהיא יכולה לטפל בחריגה על ידי שימוש ב **try** ו- **catch**
- הפקודה **try** מגדירה בלוק של קוד שעלול לזרוק חריגות שהיינו רוצים לטפל בהן.
- הפקודה **catch** מגיעה מיד לאחר בלוק **try**, ומציין את טיפוס החריגות שאנו חווים לתפוס, וכן את הקוד שיש לבצע במידה ונתפסת חריגה כזו.

“אם קטע הקוד הבא זורק חריגה מטיפוס `int`, אני יודע איך לטפל בה!”

```
int main() {  
    int a, b;  
    cin >> a >> b;  
    try {  
        cout << choose(a, 4) << endl;  
        cout << choose(b, 4) << endl;  
    } catch (int n) {  
        cerr << "error no. " << n << endl;  
    }  
    return 0;  
}
```

תפישת חריגה

- מספר בלוקים של **try** יכולים להיות מוגדרים עבור בלוק אחד של **catch**
 - כל בלוק של catch מגדיר את הטיפוס של החריגה בה הוא יכול לטפל.
 - אם אף בלוק catch לא תופס את החריגה, החריגה ממשיכה לפעע במעלה מחסנית הקראות עד שהיא נתפסת (או שהתוכנית מסתיימת).
 - אם החריגה כן נתפסה, **מתבצע קטע הקוד בבלוק ה-try**, ולאחריו הפונקציה שתפיסה את החריגה ממשיכת **מחשורה שאחרי בлок ה-try האחרון**.

רק בлок ה-**catch הראשון**
שמתאים מתבצע

בלוק **catch** מיותר ...

אם לא נזרקה חריגה, או אם
נזרקה ונתפסה ע"י **catch**,
נמשיך את הריצה כאן

```
int main() {  
    int a;  
    cin >> a;  
    try {  
        cout << choose(a, 4);  
    } catch (int n) {  
        cerr << "error no. " << n;  
    } catch (int d) {  
        cerr << "caught " << d;  
    }  
    cout << a;  
    return 0;  
}
```

Stack Unwinding

- כאשר פונקציה זורקת חריגה, כל המשתנים הילוקאליים שלה **משוחררים באופן מסודר** ע"י קרייה להורס שלהם (בבדיקה כמו במקורה של `return`). רק לאחר מכן החריגה מועברת למעלה, לפונקציה הקוראת.
- זה מאפשר לנו לזרוק חריגות **בלי לדאוג לפינוי משאבים** באופן מפורש, בניגוד ל-C
- אם הפונקציה הקוראת לא תופסת את החריגה, היא נזקפת ממנה, והתהליך ממשיר תהליך היציאה מפונקציות עד לתפיסת החריגה נקרא **התרת המחסנית** (**stack unwinding**)
- החל מזירות החריגה ועד לתפיסתה, **הקוד היחיד שמתבצע** הוא של הורסים של המשתנים המקומיים המשוחררים **SetOfComplex::~SetOfComplex()**
- **נקרא אם**

נקרא אם **Complex::~Complex()** זורקת חריגה **readComplex()** נקרא אם

```
int main() {  
    try {  
        cout << readSet(5);  
    } catch (const char* error) {  
        cerr << error << endl;  
    }  
    return 0;  
}
```

```
SetOfComplex readSet(int n) {  
    SetOfComplex s;  
    for (int i=0; i<n; ++i) {  
        s.add(readComplex());  
    }  
    return s;  
}
```

```
Complex readComplex() {  
    Complex r;  
    cin >> r;  
    if (!cin) {  
        throw "invalid number";  
    }  
    return r;  
}
```

הגדרת חריגות

- מה ניתן לזרוק?
 - C++ מאפשרת לזרוק **אייה ערך שנרצה (מכל טיפוס)**
 - חריגות נתפסות **רק לפי הטעפוס שלו**
 - מה כדאי לזרוק?
 - נרצה לזרוק **טיפוס שונה לכל סוג של שגיאה**
- אפשר לתפוס int,
לא ספציפית 5 או
-1
- מחלקה ריקה
המייצגת שגיאה
מסויימת
- ```
class DivisionByZero {};
```
- ```
Complex& Complex::divideBy(const Complex& r) {
    if (r.real == 0 && r.imag == 0) {
        throw DivisionByZero();
    }
    // ...
}
```
- ```
Complex r1(1, 2), r2(0, 1);
try {
 r1.divideBy(r2);
 cout << r1;
} catch (DivisionByZero e) {
 cerr << "oops";
}
```
- 15
- Introduction to Systems Programming

# חריגות

נלמד בהמשך על מבני נתונים  
לטיפוסים כלליים

```
class Stack
{
 int* data;
 int size;
 int nextIndex;

public:
 Stack(int size = 100);
 Stack(const Stack& stack);
 ~Stack();
 Stack& operator=(const Stack& s);
 void pushint(int t);
 void pop();
 int& top();
 const int& top() const;
 int getSize() const;

 class Full {};
 class Empty {};
};

ניתן להגדיר מחלקות בתוך מחלקות
```

- נוסף חריגות למחסנית שלנו:

```
void Stack::push(int& t) {
 if (nextIndex >= size) {
 throw Full();
 }
 data[nextIndex++] = t;
}

int& Stack::top() {
 if (nextIndex <= 0) {
 throw Empty();
 }
 return data[nextIndex-1];
}
```

אין כל ערך הגיוני להחזיר מהפונקציה זו במקרה של שגיאה! חריגות הן פתרון מצוין שלא מסביר את המשך של הפונקציה

# חריגות

```
class Stack::Empty {
public:
 int index;
 Empty(int index) : index(index) {}
};

const T& Stack::top() const {
 if (nextIndex <= 0) {
 throw Empty(index);
 }
 return data[index];
}
```

בשביל להגדיר מחלוקת  
מקוונת כר יש להזכיר  
עליה בתוך Stack

מדוע יש להמנע?

טיפוסים של חריגות יכולים גם להיות  
מורכבים יותר

– ניתן לאחסן בהם יותר מידע על  
השגיאה

שימוש לב: הפרמטר של catch נתפס

– (כמו פרמטר של פונקציה) –  
יש להשתמש בrefrens כאשר תופסים  
חריגה מטיפוס מורכב, כדי להימנע  
מהפעלת בנאי העתקה

```
void f(String& s) {
 try {
 cout << s[50];
 } catch (const String::BadIndex& e) {
 cerr << "Bad Index: "
 << e.index;
 }
}
```

# חריגות - שימוש במחסנית

```
void f(Stack<int>& s) {
 try {
 s.pop();
 for (int i = 0; i < 10; ++i) {
 s.push(i);
 }
 } catch (Stack<int>::Empty& e) {
 cerr << "No numbers";
 } catch (Stack<int>::Full& e) {
 cerr << "Not enough room";
 } catch (...) {
 cerr << "Oops";
 throw;
 }
}
```

במציאות נדר מאד שצריך לבצע catch ... ואז לזרוק שוב (זה קורה אוטומטית אם לא תופסים את החריגה!) – הקוד כאן הוא רק לשם הדוגמה.

- ניתן להשתמש **במספר משפטי** עבור try יחיד
- ניתן להשתמש ב-... (שלוש נקודות) כדי לתפוס כל חריגה
  - במקרה זה לא ניתן לגשת לחריגה שנתפסה
- אם כמה מקרים מתאימים לתפיסת החריגה, יבחר הראשון מביניהם
  - לכן ... תמיד יופיע **אחרון** מtower בлок catch, ניתן לזרוק את החריגה הנוכחית שנתפסה לפני מעלה על ידי פקודת **throw** ללא פרמטרים

# חריגות בהקצאה דינמית

- כאשר `new` נכשל נזרקת החריגת `std::bad_alloc`

```
int main() {
 try {
 while (true) {
 new int[10000];
 }
 } catch (const std::bad_alloc& e) {
 cerr << e.what() << endl;
 }
 return 0;
}
```

- הקצתה זיכרון היא מקרה קלusi לחריגות: כמעט תמיד ההפונקציה שמקצתה את הזיכרון היא **פונקציה פנימית** שאינה יודעת לטפל בכישלון ההקצתה.
  - נפוץ מאוד במקרים אלו שנctrar **לפיעוף את החריגה כלפי מעלה**.
- לרוב, רק ההפונקציה שמקצתה את הזיכרון וההפונקציה שמתפלת בשגיאה יכללו קוד כלשהו שמתיחס לשגיאה, וכל ההפונקציות ש"בדרכן" לא יצטרכו להתייחס לכך.

# זהירות בשימוש בחיריגות

- לחיריגות יש חיסרון משמעותי - כל קריאה בקוד שלנו עלולה לגרום לזריקת חריגה (בצורה ישירה או עקיפה), ולכן קוד צריך להיות מוכן לזריקת חריגה בכל שלב.
- לדוגמה, מה הבעיה בקוד הבא?

```
static int* allocateAndCopy(const int* data, int size) {
 int* newData = new int[size];
 memcpy(newData, data, size*sizeof(int));
 return newData;
}

void Stack::copyFromAnotherStack(const Stack& s) {
 if (this == &s) { return; }
 delete[] data;
 data = allocateAndCopy(s.data, s.maxSize);
 maxSize = s.maxSize;
 nextIndex = s.nextIndex;
}
```

# קוד בטוח לחריגות

- כאשר מבצעים קוד שמעורב שחרור והקצאות - קודם נבצע את החלק המסוכן (שעלול לזרוק חריגות), ורק לאחר מכן נשחרר את המידע הישן:

```
static int* allocateAndCopy(const int* data, int size) {
 int* newData = new int[size];
 memcpy(newData, data, size*sizeof(int));
 return newData;
}

void Stack::copyFromAnotherStack(const Stack& s) {
 if (this == &s) { return; }
 int* tmp = allocateAndCopy(s.data, s.maxSize);
 delete[] data;
 data = tmp;
 maxSize = s.maxSize;
 nextIndex = s.nextIndex;
}
```

# זריקת חריגה לעומת החזרת ערך

- בתכנות נכון עם חריגות יש להפריד בין **התנוגות הרצויות** של הפונקציה (הчисוב או הפעולה שהיא אמורה לבצע), לבין **אירועים/מצבים לא רצויים** שמנועים מפעולה זו להצליח.
  - במקרה של **התנוגות תקינה**, נשתמש בערך החזרה רגיל
  - במידה **ולא ניתן להשלים את הפעולה**, ניתן לזרוק חריגה
- **אין לשימוש בחריגות כתחליף לערך חזרה!**
- מהי **נכון** לשימוש בחריגות?
  - כאשר הפונקציה נתקלת בבעיה שאינה מאפשרת לה לבצע את תפקידה.
  - לדוגמה: קלט לא תקין, תוכנה במצב שלא מאפשר את הפעולה, וכו'

# חריגות - שימוש נכון

- אם אפשר בבדיקה פשוטה לדעת שפעולה מסוימת תגרום לחריגה **ולהמנע ממנה**, לרוב כדאי לעשות זאת על פניו שימוש ב-try/catch-  
– קוד זהה לרוב יהיה קריא יותר, פשוט יותר ורציף יותר
- כדאי להשתמש בתפיסת חריגה רק כשהאין דרך קללה לדעת מראש אם פונקציה תזרוק חריגה או לא

```
void good(Stack& s) {
 while(s.getSize() > 0) {
 cout << s.top() << endl;
 s.pop();
 }
}
```

```
void bad(Stack& s) {
 while(true) {
 try {
 cout << s.top() << endl;
 s.pop();
 } catch (Stack::Empty& e) {
 break;
 }
 }
}
```

# חריגות ובנייה

- אם בניאי זורק חריגה, הוא מופסק מיד והאובייקט **aino** **נבנה**
  - במקרה זהה ההורס של האובייקט לא ייקרא!
- אם בוצעה הקצאה דינמית במבנה, יש לשחרר את הקצאה באופן ידני **לפני זריקת החריגה**.

```
class Map {
public:
 class City;
 class Road;

 Map();
 ~Map();
 // ...

private:
 City* cities;
 Road* roads;
 // ...
};
```

```
Map::Map() {
 cities = new City[INIT_CITY_VNUM];
 try {
 roads = new Road[INIT_ROAD_NUM];
 } catch (const bad_alloc&) {
 delete[] cities;
 throw;
 }
 // ...
}
```

צריך לשחרר את  
cities מפושנות.  
ההורס של  
Map לא יקרה במקרה  
של זריקת חריגה.

# חריגות והורסים

## בשם אופן אין **להזק חריגה מהורס**:

- כזכור, כאשר נזרקת חריגה, ההורסים של המשתנים המקומיים עדין נקראים לפני שהוא מועברת מעלה.
- בכל רגע נתון, יכולה להיות **תק חריגה אחת** ב"המתנה" להזק כלפי מעלה.
- אם הורס זורק חריגה תור כדין שכבר יש חריגה אחת שמשמעותה להזק מעלה – התוכנית לא תוכל המשיך והוא **תקROS**

```
class SomeClass {
 int n;
public:
 SomeClass(int n);
 ~SomeClass();
 // ...
};
```

```
SomeClass::~SomeClass() {
 if (n < 0) {
 throw NegativeValue(); // don't do this!
 }
}
```

```
void horribly_wrong() {
 SomeClass s(-1);
 throw VeryBadException();
}
```

1. ההורס של SomeClass נקרא  
במסגרת ה-  
stack unwinding

2. ההורס גורם לחריגה **שנייה** להזק

מידית והתוכנית קורסת  
**std::terminate()** נקרא

# חריגות - סיכום

- טיפול בשגיאות מתבצע ב-`C++` בעזרת `throw`, `try` ו-`catch`
- אפשר לזרוק כל טיפוס אבל נהוג לזרוק רק עצמים ממחלקות ייעודיות
- השתמש בחריגה במידה ובנאי נכשל ואינו יכול ליצור עצם חוקי
- כשלון של `new` מדוח על ידי זריקת חריגה `std::bad_alloc`
- המנוו מכתבת קוד אשר משתמש בחריגות כתחליף לערכי חזרה
- העדיף קוד פשוט שאינו מרבה ב-`try/catch` במידה ואפשר להמנע מכך
- לעומת זאת אין לזרוק חריגה מהורס

# תרגול מס' 6

- ❖ עבודה עם קבצים
- ❖ בנאים הורסים והשומות
- ❖ string - דוגמה ❖

# עובדת עם קבצים

המחלקות ifstream וofstream ♦♦♦

# קלט/פלט עם קבצים ב-C++

- כמו ב-C גם ב-C++ העבודה עם קבצים דומה לעבודה עם ערוצי הקלט/פלט הסטנדרטיים
- ב-C++ העבודה עם קבצים מתבצעת בעזרת מחלקות המוגדרות בקובץ **fstream**
  - המחלקה **ofstream** משמשת לכתיבה לקובץ
  - המחלקה **ifstream** משמשת לקרוא מקובץ
- לשתי המחלקות יש בנאי מקבל את **שם הקובץ לפתיחת**
- לשתי המחלקות יש הורס **המודא שהקובץ נסגר** כאשר העצם נהרס
- **הדפסה וקריאה** מתבצעות על ידי שימוש ב-<> ו->< כמו עבר אורוצים רגילים
- שאר הפעולות הדרשיות על ערוצי הפלט מבוצעות **כמתודות של המחלקות**
  - למשל המתודה **eof** מחזירה חיוי אם הגיעו לסוף הקובץ

# ממשק (חלקי) של ofstream & ifstream

```
class ofstream {
public:

 explicit ofstream(const char* filename);

 ofstream(const ofstream&) = delete;

 explicit operator bool() const;

 bool operator!() const
 ostream& operator<<(int val);
 ostream& operator<<(double val);
 ostream& put(char val);
 ostream& operator<<(char val);
 ostream& operator<<(char* val);

...
};
```

```
class ifstream {
public:

 explicit ifstream(const char* filename);

 ifstream(const ifstream&) = delete;

 explicit operator bool() const;

 bool operator!() const
 istream& operator>>(int val);
 istream& operator>>(double val);
 istream& get(char val);
 istream& operator>>(char val);
 istream& operator>>(char* val);

...
};
```

למה צריך למחוק את  
בנאי ההעתקה?

- יש עוד מגוון פונקציות ממשק – ניתן למצוא באתרים כמו [cppreference.com](http://cplusplus.com) ו- [cplusplus.com](http://cplusplus.com).

# קלט/פלט עם קבצים - דוגמה

```
#include <fstream>
using std::ifstream;
using std::ofstream;
using std::cerr;
using std::endl;

void copyFile(const char* sourceFileName, const char* targetFileName) {
 ifstream source(sourceFileName);
 if (!source) { אופרטור ! (סימן קריאה)
 cerr << "cannot open file " << sourceFileName << endl;
 return;
 }
 ofstream target(targetFileName);
 if (!target) {
 cerr << "cannot open file " << targetFileName << endl;
 return;
 }
 while (!source.eof()) {
 char c;
 source.get(c);
 target.put(c);
 }
}
```

למה לא צריך לסגור את הקובץ?

אפשר גם להשתמש ב  
operator<< ו operator>>  
שמקבלים תווים.

# שמירה וטעינה של Set (מה הרצאות)

```
#include <iostream>
#include <string>

using std::ifstream;
using std::ofstream;
using std::cerr;
using std::endl;
using std::string;

void saveSet(const Set& s, string targetFileName){
 ofstream target(targetFileName);
 if (!target) {
 ... // Cannot open output file
 }
 for (int value : s) {
 target << value << endl;
 }
}

Set loadSet(string sourceFileName){
 ifstream source(sourceFileName);
 if (!source) {
 ... // Cannot open input file
 }
 Set newSet;
 char line[256];
 while (source.getline(line, sizeof(line))) {
 newSet.add(std::stoi(line));
 }
 return newSet;
}
```

כנראה נרצה לזרוק כאן חריגה

מחזיר את ה-`ifstream` עצמו, ועבورو יש אופרטור `bool` שמאפשר להתייחס אליו כתנאי.  
האופרטור `bool` מחזיר `true` אם הקריאה الأخيرة מהקובץ הצלחה, או `false` אם לא.

# **בנייה, הורסים והשומות**

- ❖ **בנייה**
- ❖ **בנייה העתקה**
- ❖ **הורסים**
- ❖ **אופרטור ההשמה**

# C'tors & D'tors - בナイים והורסרים

- השימוש בפונקציות מפורשות לאתחול ושחרור עצמים (כמו ב-C) חשוף לטעויות:
  - המשתמש יכול **לשכוח אתחל/לשחרר** את העצם
  - המשתמש יכול **לאთחל/לשחרר** את העצם **פעמיים**
- לשם כך נוספה ב-++C האפשרות להגדיר פונקציות ייעודיות לאתחול ושחרור עצמים - **בナイים** (Constructors) ו**הורסרים** (Destructors)
- כל עצם שנוצר ב-++C מאותחל על ידי בניאי וכל עצם שמשוחרר מטופל על ידי הורס

# בנאים - Constructors

```
class X {
public:
 X();
 X(int n);
};

X globalX;

int main() {
 X x1;
 X x2(5);
 X* ptrx1 = new X;
 X* ptrx2 = new X();
 X* arrayx = new X[10];
 arrayx[0] = X();
 arrayx[1] = X(5);
 // ...
 return 0;
}
```

- בניאי מוגדר על ידי מתודה בשם בשם המחלקה

לבנייה אין ערך החזרה

- ניתן להגדיר מספר בניאים בהתאם לחוקי העמסת פונקציות

לבניאי שאינו קיבל פרמטרים קוראים גם **default c'tor**

אם לא מוגדר בניאי בצורה מפורשת למחלקה הקומפיילר ייצור בניאי חסר פרמטרים עצמו

- הבניאי הנוצר על ידי הקומפיילר קורא לבניאי חסר הפרמטרים של כל אחד מהשדות במחלקה

- הקומפיילר לא ייצור default c'tor במידה ומוגדר בניאי כלשהו במחלקה

כשמקצים מערך, כל אחד מאברי המערך מאותחל **default c'tor**

- לכן לא ניתן ליצור מערך מטיפוס שאין לו default c'tor

איזה בניאי נקרא בכל אחת מהשורות?

# הורסאים - Constructors

```
X globalX;

int main() {
{
 X x1;
}
X x2;
X* ptrx1 = new X;
X* ptrx2 = new X;
delete ptrx1;

X* arrayx = new X[10];
arrayx[0] = X();
delete[] arrayx;

return 0;
}
```

- ההורס של מחלקה X מוגדר כמתודה בשם **X~**
- הורס אינו מקבל פרמטרים ואין לו ערך חזרה
- ניתן להגדיר **הורס ייחיד** למחלקה
- כאשר עצם משוחרר נקרא ההורס שלו
  - הורס נקרא אוטומטית - לא כתבים דליית זיכרון מפורשת להורס
- אם לא מוגדר הורס הקומpileר יגדיר אחד בעצמו
- לאחר סיום ההורס** של עצם כלשהו ייקראו ההורסים של כל השדות שלו

מתי נקראים הורסים בקוד זה?

# זמן קריאה

```
class X {
 int n;
public:
 X(int n) : n(n)
 { cout << "X::X():" << n << endl; }
 ~X() { cout << "X::~X():" << n << endl; }
};

class Y {
 X x1, x2;
public:
 Y() : x1(1), x2(2)
 { cout << "Y::Y()" << endl; }
 ~Y() { cout << "Y::~Y()" << endl; }
};

int main() {
 Y y;
 return 0;
}
```

מה יודפס?

- **רשימת האתחול** של בניין  
מוצעת **לפני הכניסה לקוד**  
**הבניין** עצמו (לפני ה-{ })
  - סדר הקריאה לאתחול השדות  
הוא לפי סדר הגדרתם במחלקה
- **לאחר ביצוע הקוד** שבתווך ה-{ }  
בהורס **ייראו הורסים לכל**  
**שדותיו** של העצם הנhrs (בסדר  
ההפוך מבאתחול העצם)

# בנייה העתקה ואופרטור השמה

- לבנייה המקובל פרמטר מטיפוס העצם אותו הוא מאתחל קוראים **בנייה העתקה** (**Copy C'tor**) ויש לו מעמד מיוחד
  - חתימת בניאי העתקה היא:  
`x::x(const x& x)`
- שמה** היא הפעולה של שינוי ערכו של **משתנה קיים** כך שייהי זהה לערכו של משתנה אחר
  - ניתן להעMISS על אופרטור ההשמה שחתימתו היא:  
`x& x::operator=(const x& x)`
- אם לא נכתב אופרטור השמה/בנייה העתקה הקומפイルר ייצור אחד בעצמו
  - אופרטור ההשמה/בנייה העתקה שהקומפイルר יוצר קורא לאופרטורי ההשמה/בנייה העתקה של כל השדות
  - שימוש לב: בניאי העתקה נוצר על ידי הקומפイルר גם אם הוגדרו בניאים אחרים

# בנייה העתקה - Copy C'tor

- בניי העתקה הנוצר אוטומטית עבור Stack ייראה כך:

```
Stack::Stack(const Stack& s) :
 data(s.data),
 maxSize(s.maxSize),
 nextIndex(s.nextIndex) {
}
```

- מודיעו הוא לא מתאים לנו?

```
Stack::Stack(const Stack& s) :
 data(new int[s.maxSize]),
 maxSize(s.maxSize),
 nextIndex(s.nextIndex)
{
 for(int i = 0; i < nextIndex; i++) {
 data[i] = s.data[i];
 }
}
```

- בניי העתקה מתאים ייראה כך:

# אופרטור השמה

```
Stack& Stack::operator=(const Stack& s) {
 if (this == &s) {
 return *this;
 }
 delete[] data,
 data = new int[s.maxSize];
 maxSize = s.maxSize;
 nextIndex = s.nextIndex;
 for(int i=0; i < nextIndex; i++) {
 data[i] = s.data[i];
 }
 return *this;
}
```

## ▪ אופרטור השמה למחסנית

בדיקה השמה עצמאית  
מדוע בדיקה זו הכרחית?

מחיקת מידע ישן, בנגד לבנייה העתקה  
צריך לנוקוט את המידע הקודם

ביצוע הקצאות חדשות

העתקת המידע

החזרת \*this שרשור

מה קורה אם new נכשלת?

# פונקציות הנוצרות על ידי הקומפיילר

- הקומפיילר יוצר את הפונקציות הבאות בעצמו:
  - **בנאי חסר פרמטרים:** מתחילה את כל השדות של העצם בעזרת הבניי חסר הפרמטרים שלהם
    - אם נכתב בנאי **כleshho** במפורש הקומפיילר לא ייצור פונקציה זו
  - **בנאי העתקה:** מתחילה את כל שדות העצם בעזרת בנאי העתקה שלהם
    - הקומפיילר תמיד ייצור בנאי העתקה אם לא הוגדר אחד מפורשות (גם אם הוגדר בנאי אחר במחלקה!)
  - **אופרטור השמה:** קורא לאופרטור הרשמה של כל השדות של העצם
    - הקומפיילר תמיד יוצר אופרטור השמה אם לא מוגדר אחד מפורשות
  - **הורס:** קורא להורסים של כל השדות של העצם (גוף ההורס ריק)
- **אין צורך למשח מחדש** פונקציות שהקומפיילר יוצר בעצמו **ומתאימות** לנו
  - מימוש בלבד של פונקציה שהקומפיילר מייצר גם כרגע עצמו פותח פתח **לשגיאות** **וכן פוגע בקריאות הקוד**

# =default

- אם המימוש הדיפולטיבי שהקומפイルר מייצר עבור פונקציה מסוימת מתאים לנו, ניתן ב-C++11 **לציין זאת במשפט =default** עם
- משמש גם על מנת להורות לקומפイルר לייצר את ה-`ctor` במצב בו הגדרנו בנאי אחר במחלקה

```
class Event {
 Date m_eventDate;
 int m_numOfGuests;
public:
 Event(int num, Date d) : m_eventDate(d), m_numOfGuests(num){
 }
 Event(const Event &) = default;
 Event & operator=(const Event &) = default;
 ~Event() = default;
};
```

פונקציות שמיוצרות אוטומטית  
על ידי הקומפイルר, אבל מצוינות  
עם `=default` כדי ליתעד זאת

- בקורס זה **נדרש להשתמש ב-`=default`** עבור בנאים ואופרטור `=` שאנו רוצים את המימוש הדיפולטיבי שלהם (עבור ההורס הטריויאלי אין צורך בכור)

# Big three

- בין בניי העתקה, ההורס ואופרטור ההשמה קיימ קשר:  
**אם צריך למש אחיד מהם, צריך למש את שלושתם**
- איך יודעים מתי צריך למש את הפונקציות האלו?
  - אם מעורבים מצביעים בחלוקת בדרך כלל יש צורך בפונקציות האלו
  - אם יש שימוש ב-new ו-delete
  - אם מתבצעות הקצאות של משאים אחרים (למשל קבצים)

# בנאים, הורסים והשמות - סיכום

- בנאים משמשים לתחול עצמים
- לפני הכניסה לקוד הבניי, נקראים הבנאים של כל השדות של העצם
- אחרי ביצוע הקוד בהורס נקראים ההורסים של כל שדות העצם
- בגין העתקה משמש להעברת והחזרת ערכים ערכיהם `by value` ולאתחול העתקים
- אופרטור ההשמה (=) משמש להעתיקת ערכים בין שני עצמים קיימים
- בכתיבה אופרטור השמה חשוב לשים לב לשחרור המידע הקיים ולמניעת שימוש עצמיות
- הקומפיילר יכול לייצר בעצמו בגין חסר פרמטרים, בגין העתקה, אופרטור השמה והורס. אם המימוש של הקומפיילר מתאים, אין צורך למשח מחדש פונקציות האלו
  - ניתן (ומומלץ) לציין כל מקרה כזה באמצעות `=default`
  - אם פונקציה שמיצרת אוטומטית אינה רצiosa, יש לחסום אותה עם `=delete`
- אם צריך למשח מפorschות בגין העתקה, אופרטור השמה או הורס, אז צריך למשח את שלושתם

# דוגמה - String

- ❖ שימוש במבנה, הורסים והעמסת אופרטורים

# דוגמה - המחלקה String

- השימוש במחלקות, בנאים, הורסים והעמסת אופרטורים מאפשר לנו להגדיר מחלקה עבור String כך שייחסכו מאייתנו החסרונות של השימוש ב-`*char`
  - בזכות בנאים והורסים לא נדרש לנוהל את הזיכרון ידנית
  - בזכות העמסת אופרטורים נוכל לשמר על כל הנוחות של `*char`, למשל גישה כמערך
  - בזכות העמסת אופרטורים נוכל לאפשר פועלות בסיסיות בצורה נוחה - למשל שרשור

# דוגמה - קוד המשתמש ב-String

```
int main() {
 String s = "So long";
 String s2 = "and thanks for all the fish.";
 String s3 = s + " " + s2;
 s3[s3.size() - 1] = '!';
 cout << s3 << endl;

 return 0;
}
```

בשורה זו נקרא בנאי העתקה  
ולא אופרטור השמה.

כעת אין צורך בניהול זיכרון מפורש

# String.h

```
class String {
 int m_length;
 char* m_data;

 static char* allocateAndCopy(const char* data, int size);

public:
 String(const char* str = ""); // String s1; or String s1 = "aa";
 String(const String& str); // String s2(s1);
 ~String();
 int size() const;
 String& operator=(const String&); // s1 = s2,
```



מדוע אנחנו צריכים  
לממש בinati העתקה,  
הקורס, ואופרטור השמא?

# String.h

אפשר להשתמש ב-String  
עם פונקציות שמקבלות  
מחרוזת סטנדרטית

אופרטורים הכללים השמה כמו  
+= מאפשרים שרשור ולבן נשמר  
על התחנות הזו אצלנו

```
String& operator+=(const String& str); // s1 += s2;
const char& operator[](int index) const; // s[5] for const s
char& operator[](int index); // s[5] for non-const s
```

```
const char* c_str() const; // No
```

לא ניתן להגיד את אופרטור <> כמתודה  
של מחלקה שכותנו ( מדוע? )

```
friend ostream& operator<<(ostream&, const String&); // cout << s1;
friend istream& operator>>(istream& is, String& c);
```

```
friend bool operator==(const String&, const String&); // s1==s2
```

```
friend bool operator<(const String&, const String&); // s1<s2
```

```
};
```

הקוד שמופיע בגוף הפונקציה  
שהוכחה כחברה רשאי לגשת  
לחלקים פרטיים של המחלקה בה  
הוכח כ-friend

ניתן להגיד את האופרטורים  
כמתודות או כפונקציות חיצונית.  
אלו פונקציות חיצונית.

# – הרכות מחוץ למחלקה – String.h

```
bool operator!=(const String& str1, const String& str2);
bool operator<=(const String& str1, const String& str2);
bool operator>(const String& str1, const String& str2);
bool operator>=(const String& str1, const String& str2);
String operator+(const String& str1, const String& str2);
```

אופרטור + יוצר מחרחת חדשה, לנוכח התוצאה מוחזרת כעותק חדש

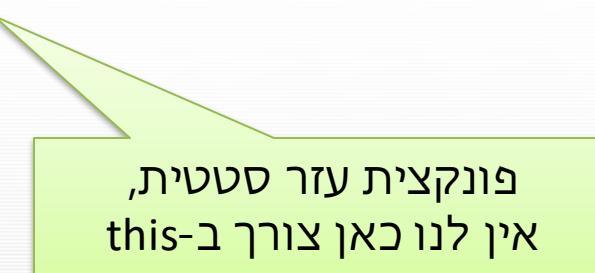
עבור אופרטורים אלו אין  
צורך ב-friend. מדוע?

אופרטורים סימטריים נהוג להכריז מחוץ למחלקה בשביל לתמוך המרות

**מודלץ להימנע שימוש ב-friend** - בדרך כלל זהו תסמין של **תיכון רע** של המערכת **ועודף תלויות** בין החלקים השונים בקוד. עם זאת קיימים מקרים בהם השימוש ב-friend נדרש.

# IMPLEMENTATION – פונקציות עזר String.cpp

```
char* String::allocateAndCopy(const char* str, int size) {
 return strcpy(new char[size + 1], str);
}
```



פונקציה עזר סטטית,  
אין לנו כאן צורך ב-this

בנאי המקבל פרמטר יחיד. בנאי זה משמש את הקומפיילר גם לביצוע המרות אוטומטיות

# מימוש string.cpp

```
String::String(const char* str) : m_length(strlen(str)),
 m_data(allocateAndCopy(str, strlen(str)))
{}
```

למה הפרמטר מועבר **?by reference**

```
String::String(const String& str) :
 m_length(str.size()),
 m_data(allocateAndCopy(str.m_data, str.m_length))
{}
```

הבנייה אחראי לאותחול השדות של המחלקה, אותחול השדות מתבצע **ברשימה האთחול** לפני הכניסה לגוף הפונקציה על ידי קראיה לבנים

```
String::~String() {
 delete[] m_data;
}

int String::size() const {
 return m_length;
}
```

מחזיר פוינטר לזכרון בבעלות המחלקה. יש להזכיר עם השימוש בפונקציה, ולא לשחרר את המחרוזת המוחזרת.

```
const char* String::c_str() const{
 return m_data;
}
```

# IMPLEMENTATION

# String.cpp

```
String& String::operator=(const String& str) {
 if (this == &str) {
 return *this;
 }
 char * tmp = allocateAndCopy(str.m_data, str.m_length);
 delete[] m_data;
 m_data = tmp;
 m_length = str.m_length;
 return *this;
}

String& String::operator+=(const String& str) {
 char* new_data = allocateAndCopy(m_data, this->m_length + str.m_length);
 strcat(new_data, str.m_data);
 delete[] m_data;
 m_length += str.m_length;
 m_data = new_data;
 return *this;
}
```

# IMPLEMENTATION

## String.cpp

```
const char& String::operator[](int index) const {
 if (index >= size() || index < 0) {
 throw OutOfBounds("Bad index");
 }
 return m_data[index];
}
```

פונקציות המחזירות & צריכות  
בדרך כלל שתי גרסאות - עברו  
עצמים רגילים ועברו קבועים

```
char& String::operator[](int index) {
 if (index >= size() || index < 0) {
 throw OutOfBounds("Bad index");
 }
 return m_data[index];
}
```

# IMPLEMENTATION

```
bool operator==(const String& str1, const String& str2) {
 return strcmp(str1.m_data, str2.m_data) == 0;
}

ostream& operator<<(ostream& os, const String& str) {
 return os << str.m_data;
}

istream& operator>>(istream& is, String& str) {
 return is >> str.m_data;
}

bool operator<(const String& str1, const String& str2) {
 return strcmp(str1.m_data, str2.m_data) < 0;
}
```

מחזירים את os כדי לאפשר שרשור

# IMPLEMENTATION

## String.cpp

```
bool operator!=(const String& str1, const String& str2) {
 return !(str1 == str2);
}

bool operator<=(const String& str1, const String& str2) {
 return !(str2 < str1);
}

bool operator>(const String& str1, const String& str2) {
 return str2 < str1;
}

bool operator>=(const String& str1, const String& str2) {
 return str2 <= str1;
}

String operator+(const String& str1, const String& str2) {
 return String(str1) += str2;
}
```

# std::string

- קובץ ה-**string** header מהספריה הסטנדרטית של C++ מכיל שימוש של המחלקה **std::string** אשר תומכת בכל הפעולות שמימשנו כאן ועוד
- הקפידו **להשתמש ב-std::string ולא ב-char\*** עבור מחרוזות ב-C++
- קיימת מתודה `str.c_str()`. (עבור תמיינה בקוד ופונקציות מ-C)

# תרגול מס' 7

- ❖ תבניות
- ❖ בחירת מבני נתונים

# תבניות

- ❖ תבניות של פונקציות
- ❖ תבניות של מחלקות
- ❖ מחסנית גנרטית

# תכנות גנרי

- נניח שברצוננו לכתוב פונקציה למציאת המקסימום במערך של עצמים
  - לכל טיפוס נctror לרשום מחדש פונקציה כמעט זהה

```
int max(const int* array, int size) { string max(const string* array, int size) {
 int result = array[0]; string result = array[0];
 for (int i = 1; i < size; ++i) {
 if (result < array[i]) {
 result = array[i];
 }
 }
 return result;
}
 }
 }
 return result;
}
```

- קל לנו לראות את **התבנית הכללית** של פונקציות המוציאות מקסימום במערך
  - רק **שם הטיפוס** משתנה (בדוגמאות שלנו int ו-string)
- בראצוננו **לכתוב קוד גנרי** שיתאים לכל טיפוס

# tabniot

- כפי שראינו בהרצאות, כדי להגדיר תבנית לפונקציה משתמשים במילה **השמורה:template**

```
template<class T>
T max(const T* array, int size) {
 T result = array[0];
 for (int i = 1; i < size; ++i) {
 if (result < array[i]) {
 result = array[i];
 }
 }
 return result;
}
```

בתוך קוד התבנית **T** מייצג שם של טיפוס וניתן לשימוש בו כמו בשם של טיפוס רגיל:  
להכריז על משתנים מסוג **T** ולבצע עליהם פעולות

- הקומpileר ישתמש בתבנית כדי ליצור קוד לפי הצורך
  - תהליך ייצור מופיע של הקוד המוגדר על ידי התבנית מתבצע בזמן הקומPILEציה וקרויה **instantiation**
- tabniot יש להגדיר תמיד **בקבץ** כך שהקוד המלא שלו יהיה זמין לקומpileר

# tabniot - shimush

- כדי להשתמש בפונקציה המוגדרת על ידי תבנית ניתן לקרוא לה לפי שמה  
ובתוספת הפרמטרים הדרושים ל התבנית:

```
int array[] = {4, -1, 2};
string array2[] = {"Hello", "cruel", "world"};
cout << max<int>(array, 3) << endl;
cout << max<string>(array2, 3) << endl;
```

- ניתן לקרוא לפונקציה ללא ציון הפרמטרים ל התבנית - במקרה זה **הקומפיילר** יסיק אותם בעצמו

- במצב זה הפונקציה תשתחף בתהlixir פתרון ההעמסה יחד עם פונקציות נוספות

```
int array[] = {4, -1, 2};
```

עם אותו שם

```
string array2[] = {"Hello", "cruel", "world"};
```

max<int> הוא array וכאן תיקרא max<int>(array)

```
cout << max(array, 3) << endl;
```

```
cout << max(array2, 3) << endl;
```

max<string> הוא array2 וכאן תיקרא max<string>(array2)

# tabniot vohamshat fonkziot

- בפתרון העמשה אשר מעורבות בו tabniot, הקומפיילר יתעדף את האפשרויות השונות בסדר הבא:
  1. בעדיפות הגבואה ביותר, הקומפיילר יחפש **fonkzia regila** (לא tabnit) שטיפוסי הפרמטרים שלה **matayim bdiok** לטיפוסי הארגומנטים שהועברו
  2. בעדיפות השנייה, הקומפיילר ישתמש **bfonkzia tabnit** שנייה להפעיל אותה על הארגומנטים **la hafulat hamrot**
  3. בעדיפות השלישית הקומפיילר ישתמש **bfonkzia regila** (לא tabnit) תוך הפעלת **hamrot otomiyot** על הארגומנטים
    - הקומפיילר **leolam la ibcu hamrot otomiyot** כדי להתאים ארגומנטים **lfonkzia tabnit**
    - ניתן להזכיר שימוש **bfonkzia tabnit** על ידי ציון הפרמטרים שלה במשמעות

# תבניות - העממת פונקציות

```
int max(int a , int b , int c) {
 if (a > b && a > c) return a ;
 if (b > c) return b ;
 return c;
}
// ...
int j = max (1,3,3); // ☺
char c = max ('w','w','w');// ☺
string s = max (s1,s2,s3); // ☹
int j = max (1,'a','a'); // ☺☺
```

```
template<class T>
T max (const T& a , const T& b ,
 const T& c) {
 if (a > b && a > c) return a ;
 if (b > c) return b ;
 return c;
}
// ...
int j = max(1,3,3); // ☺
char c = max ('w','w','w'); // ☺
string s = max (s1,s2,s3); // ☺☺
int j = max (1,'a','a'); // ☹
int j = max<int> (1,'a','a'); //☺
```

דוגמאות:

# tabniot - ייצור מופעים

בכדי שהשימוש בתבנית יצליח על הארגומנטים המועברים לתבנית לעמוד בדרישות (הלא מפורשות) על הטיפוס המופיעות בקוד

- אילו דרישות יש על הטיפוס T ב-`?max`?

```
template<class T>
T max(const T* array, int size) {
 T result = array[0];
 for (int i = 1; i < size; ++i) {
 if (result < array[i]) {
 result = array[i];
 }
 }
 return result;
}
```

```
Stack array[] = { Stack(50), Stack(60), Stack(70) };
cout << max(array, 3) << endl;
```

- 
- בניית העתקה
  - אופרטור <
  - אופרטור השמה
  - הורס

# tabniot - icirat mofeim

- האם הינו יכולים לספק פונקציה דומה שתדרוש פחות מהטיפוס T?

```
template<class T>
const T& max(const T* array, int size) {
 int max_id = 0;
 for (int i = 1; i < size; ++i) {
 if (array[max_id] < array[i]) {
 max_id = i;
 }
 }
 return array[max_id];
}
```

- הפונקציה זו מציבה רק את **הדרישה המינימלית האפשרית** – אופרטור <

- זאת בנויגוד לפונקציה בשקף הקודם, שדורשת 4 פעולות

- המימוש החדש מאפשר לנו להפעיל את max על **יוטר טיפוסים** (לא כל טיפוס ניתן להעתיק!). בהמשך נראה טיפוסים שאכן אין להם בנאי העתקה ואופרטור השמה.

- המימוש החדש גם יעיל יותר – אינו מבצע העתקות מיותרות

- לעיתים אפשר לספק למשתמש כמה גרסאות שונות של הפונקציה, שדורשות יותר או פחות מהטיפוס, ולפי זה פועלות קצת שונה, בסיבוכיות שונה, וכו'.

# מחסנית גנרייה ב-C++

```
template <class T>
class Stack {
 T* data;
 int maxSize;
 int nextIndex;

public:
 explicit Stack(int maxSize = 100);
 Stack(const Stack& s);
 ~Stack();
 Stack& operator=(const Stack&);

 void push(const T& t);
 void pop();
 T& top();
 const T& top() const;
 int getSize() const;
 class Full {};
 class Empty {};
 class InvalidSize {};
};
```

בתוך קוד המחלקה  
ניתן לרשום  
Stack<T>

- ניתן להגדיר תבנית עבור **מחלקה** ובכך למשתמש במחלקות גנריות שיתמכו במספר טיפוסים.
- בנגד לפונקציות, עבור מחלקות חייבות לרשום במבנה את הארגומנטים לתבנית כשמייצרים אובייקט מהמחלקה - **הקומפיאילר לא יסיק אותם** **בעצמם**
- כל שימוש בתבנית עם ארגומנטים אחרים יוצר **טיפוס חדש מה התבנית**

נספר את המחסנית שלנו כך שתהייה גנרית:

בנייה העתקה מ- Stack<T> ל- Stack<T>, כלומר **אותו סוג איבר**

הוסףנו חריגות

# מחסנית גנרייה ב-C++

```
template <class T>
Stack<T>::Stack(int size) :
 maxSize(size), nextIndex(0) {
 if (size <= 0){
 throw InvalidSize();
 }
 data = new T[size];
}
```

דרוש בinati חסר  
ארגומנטים

הקוד זהה נמצא  
בקובץ.h.

```
template <class T>
Stack<T>::Stack(const Stack<T>& s) :
 data(new T[s.maxSize]), maxSize(s.maxSize), nextIndex(s.nextIndex) {
try {
 for (int i = 0; i < nextIndex; ++i) {
 data[i] = s.data[i];
 }
} catch (...) {
 delete[] data;
 throw;
}
}
```

דרוש אופרטור  
השמה

מה יכול להשתבש  
כאן?

# מחסנית גנרייה ב-C++

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack<T>& s) {
 if (this == &s) {
 return *this;
 }
 T* tempData = new T[s.maxSize];
 try {
 for (int i = 0; i < s.nextIndex; ++i) {
 tempData[i] = s.data[i];
 }
 } catch (...) {
 delete[] tempData;
 throw;
 }
 delete[] data;
 maxSize = s.maxSize;
 nextIndex = s.nextIndex;
 data = tempData;
 return *this;
}
```

הקוד הזה נמצא  
בקובץ.h.

# מחסנית גנרייה ב-C++

```
template <class T>
Stack<T>::~Stack() {
 delete[] data;
}
```

הקוד זהה נמצא  
בקובץ.h.

```
template <class T>
void Stack<T>::push(const T& t) {
 if (nextIndex >= size) {
 throw Full();
 }
 data[nextIndex] = t;
 nextIndex++;
}
```

דרוש אופרטור השמאה

```
template <class T>
void Stack<T>::pop() {
 if (nextIndex <= 0) {
 throw Empty();
 }
 nextIndex--;
}
```

# מחסנית גנרייה ב-C++

```
template <class T>
T& Stack<T>::top() {
 if (nextIndex <= 0) {
 throw Empty();
 }
 return data[nextIndex - 1];
}

template <class T>
const T& Stack<T>::top() const {
 if (nextIndex <= 0) {
 throw Empty();
 }
 return data[nextIndex - 1];
}

template <class T>
int Stack<T>::getSize() const {
 return nextIndex;
}
```

כל הקוד הזה  
נמצא בקובץ **.h**.

# שימוש במחסנית הגנרטית

```
#include "Stack.h"
```

```
int main() {
 Stack<int> s;
 Stack<string> s2;
 s.push(20);
 s2.push("Hello");
 Stack<int> s3;
 s3 = s2; // error
 cout << s.top() << s2.top() << endl;
 return 0;
}
```

שגיאת קומפילציה, למה?

# אופרטור <> של מחסנית גנריית ב-C++

```
template <class T>
class Stack {
 T* data;
 int maxSize;
 int nextIndex;

public:
 int top() const; // הפונקציה הזו צריכה לקבל כריכה להיות friend
 int getSize() const; // מכיון שאין דרך להציג מבחרן (כגון r=(const T&) t) לערכים במחסנית, פרט בראש.
 void push(T); // מחייבים שpush יחזיר כריכה של Stack& (בדרך כלל כריכה של Stack<T>).
 void pop(); // מחייבים שpop יחזיר כריכה של Stack& (בדרך כלל כריכה של Stack<T>).
};

class Full {};
class Empty {};
class InvalidSize {};

template <class S>
friend ostream& operator<<(ostream& os, const Stack<S>& s);
```

כיוון שהאופרטור <> הוא חיצוני למחלקה, הצהרת ה-template של המחלקה עצמה לא תקפה. צריך להגדיר את הפונקציה כ-template כ-`template <class S> friend ostream& operator<<(ostream& os, const Stack<S>& s);` עם שם פרמטר אחר כי T תפוס).

```
template <class S>
ostream& operator<<(ostream& os, const Stack<S>& s) {
 for (int i = 0; i < nextIndex; ++i) {
 os << data[i] << endl;
 }
 return os;
}
```

כל הקוד הזה  
נמצא בקובץ **h**.

# פרמטרים של תבנית

```
template<class T, class S>
struct Pair {
 T first;
 S second;
 Pair(const T& t, const S& s)
 : first(t), second(s) {}
};
```

```
Pair<char, double> p('a', 15.0);
cout << "(" << p.first << "," << p.second << ")";
```

```
List<Pair<string, int>> phonebook;
phonebook.add(Pair<string, int>("Dani", 8343434));
phonebook.add(Pair<string, int>("Rami", 8252525));
```

- ניתן להגדיר תבניות המקבלות יותר מפרמטר אחד

הטיפוס זהה קיים בספרייה הסטנדרטית של C++ בשם `<utility>` בתוך `std::pair`

# פרמטרים של תבנית

- ניתן להגדיר תבניות גם עם פרמטרים בעלי טיפוס מסוים, למשל int:

```
template<class T, int N>
class Vector {
 T data[N];
public:
 Vector();
 Vector& operator+=(const Vector& v);
 //...
};
```

וקטור מתמטי ממימד N

N נקבע בזמן הקומpileציה, שכן אנו מגדירים  
למעשה מערך בגודל קבוע כshedah בחלוקת

החלוקת יכולה מוקצת על המחסנית ולכן N  
חייב להיות מספר קטן!

```
void f(Vector<int, 3>& v) {
 Vector<int, 5> v5;
 Vector<int, 3> v3;
 v3 += v; // o.k.
 v5 += v; // error
}
```

המספר הוא חלק מהטיפוס, שכן  $\langle 3, 5 \rangle$  הם טיפוסים שונים

ניסיונו לחבר וקטורים ממימדים  
שונים לא יתאפשר

# tabniot - shgaiot kompilyata

- כאשר הקומפילר עובר על תבנית הוא יכול לאתר בה רק שגיאות בסיסיות
- כאשר הקומפילר יוצר מופע של תבנית עבר טיפוס מסוים, רק אז יתגלו שגיאות התלויות בארגומנט הגרפי
- השגיאה המתקבלת מהקומפילר תהיה מורכבת ממספר שורות ותכלול את רשימת התבניות והארגומנטים שלhn

```
void f() {
 Stack<Pair<int, int>> ranges(10);
}
```

```
..\main.cpp: In constructor `Stack<T>::Stack(int) [with T = Pair<int, int>]':
..\main.cpp:241: instantiated from here
..\main.cpp:108: error: no matching function for call to `Pair<int, int>::Pair()'
..\main.cpp:233: note: candidates are: Pair<int, int>::Pair(const Pair<int, int>&)
..\main.cpp:237: note: Pair<T, S>::Pair(T, S) [with T = int, S = int]
```

- בגלל השגיאות המסווכות מומלץ לכתוב תחילת קוד רגיל ורק אחרי שהוא עובד להמיר אותו למבנה

# tabniot - सिकूम

- ניתן להגדיר תבניות לפונקציה
- ניתן להגדיר מחלקות גנריות בעזרת שימוש ב-template
- כדי שהשימוש בתבנית יתאפשר על הארגומנטים המועברים לתבנית להיות בעלי כל התכונות הדרשיות לפי התבנית.
  - הפרמטר של התבנית חייב למש רק את התכונות הדרשיות לשם קומפילציה של המתוודות שנעשה בהן שימוש בפועל. אם לא קוראים למתודה מסוימת אף פעם, אין צורך לטיפוס T לקיום הדרישות עבורה.
- ניתן להגדיר תבניות שהפרמטרים שלهن הם מספרים או טיפוסים אחרים (נדיר)
- ניתן להגדיר תבניות עם מספר פרמטרים
- מומלץ לכתוב תחילת קוד ללא תבניות ולהמירו אח"כ כדי למנוע התמודדות עם פلت מסובך מהקומפילר
- מומלץ ראותה לשימוש ב-T typedef int (או טיפוס אחר במקום int) ולהשתמש ב-T בכל הקוד של התבנית. לאחר שהקוד מתאפשר ועובד, ההמרה לתבנית תהיה פשוטה.

# **בחירה מבני נתונים**

- ❖ בחירת מבני נתונים
- ❖ מבני נתונים הנלמדים בקורס
- ❖ שאלה לדוגמא

# מבנה הנזtones הנלמדים בקורס

- בקורס זה אנו לומדים למש את **מבנה הנזtones הבסיסיים** הבאים:
  1. **List – רשימה מקוורת:** שומרת אוסף איברים עם סדר ביניהם ומאפשרת כפליות. (למדנו בחלק של C)
  2. **Set - קבוצה:** מאפשרת הכנסת איבר פעם אחת בלבד ואין שומרת סדר בין איברי הקבוצה. (נלמד בהרצאות)
  3. **Stack - מחסנית:** מאפשרת הכנסה, גישה והוצאה רק מסופה. שומרת על סדר ומאפשרת כפליות.
  4. **Queue – תור:** מאפשר הכנסה מסופו, גישה והוצאה רק מתחילה. שומר על סדר ומאפשר כפליות. (ילמד בתרגילי הבית)

# התאמת מבנה נתונים לבעה

- לכל בעיה חשוב להתאים את **מבנה הנתונים המתאים ביותר**
- בבחירה המבנה יש להתאים בין הצורך הספציפי ובין תוכנות מבני הנתונים:
  - האם המשק שלו מתאים?
  - האם מותרות כפליות של איברים?
  - האם יש משמעות לסדר האיברים?
  - סיבוכיות של הכנסה, גישה והוצאה? (יוסבר בקורס מבני נתונים).
- שימוש במבנה נתונים קיים יתרום ל:
  - **aicoot hakod** - בחירה טוביה יוצרת **קוד קצר יותר, פשוט יותר, מונעת שכפול קוד ומקשה על הכנסת באגים**
  - למשל בחירת set במקום list מונעת הכנסת איבר פעמיים, חוסכת התעסקות בסדר הרשימה ובדיקות לפני הכנסת איבר בשנית
  - **סיבוכיות** – כל מבנה נתונים מספק אוסף אחר של פעולות בסיסיות, ובחירה במבנה שפועלותיו מתאימות יותר לבעה יכולה **לשפר מאוד את יעילות התוכנית**.
  - למשל חיפוש ביןארי ניתן לבצע רק במערך ממויין, ולא ברשימה מקושרת ממויינת

# שאלה לדוגמה

- מבנה הנתונים תור עדיפויות (Priority Queue) מאפשר הכנסת איברים, והוצאה של האיבר המקסימלי. כלומר הפעולות הנדרשות מטור עדיפויות הן:
  1. יירה, שחרור, העתקה והשמה של תור עדיפויות. (כמו כל מחלוקת)
  2. הכנסת איבר, ניתן להכניס מספר עותקים של אותו איבר.
  3. החזרת האיבר המקסימלי (העדיף ביותר).
  4. הוצאה (וותק אחד של) האיבר המקסימלי (העדיף ביותר).
- א. באילו מבני הנתונים שנלמדו בקורס כדאי להשתמש למשתמש בתור עדיפויות? מדוע?
- ב. כתבו את קובץ הממשק עבור תור עדיפויות גנרי.
- ג. משטו את היראה, השחרור, העתקה והשמה של תור עדיפויות.

# סעיפים א' ו-ב'

```
#ifndef _PRIORITY_QUEUE_H
#define _PRIORITY_QUEUE_H
#include "List.h"

template <class Element>
class PriorityQueue{
 List<Element> m_data;
public:
 PriorityQueue();
 PriorityQueue(const PriorityQueue& s);
 ~PriorityQueue();
 PriorityQueue& operator=(const PriorityQueue&);

 void push(const Element& element);
 void popMax();
 const Element& Max() const;
};

#endif
```

נבחר להשתמש ב-List עבור מימוש תור העדיפויות:

- יתכנו העתקים של אותו איבר בערמה נוח יותר להכניס/להוציא איבד בודד מקום כלשהו ברשימה (לעומת מערך או מחסנית)
- סדר האיברים נקבע ע"י העמסת אופרטורי ההשוואה של הטיפוס על ידי המשתמש.

נמנש את תור העדיפויות כרך שהרשימה הפנימית ממויינת. ולכן, נספק רק גרסה const של פונקציית גישה, אשר לא מאפשרת שינוי של מידע פנימי.

# סעיף ג'

נשתמש בימוש הדיפולטי (מדוע?) עבור הבניי חסר הארגומנטים, ההורס, בנאי העתקה  
ואופרטור השמה.

```
PriorityQueue() = default;
PriorityQueue(const PriorityQueue& s) = default;
~PriorityQueue() = default;
PriorityQueue& operator=(const PriorityQueue&) = default;
```

# תרגול מס' 8

- ❖ מוצאים חכמים
- ❖ ספריית התכניות הסטנדרטית (STL)
- ❖ איטרטורים
- ❖ הספרייה algorithm

# מצביים חכמים

unique\_ptr ♦

shared\_ptr ♦

# מצביעים חכמים

- מצביעים אחרים לרוב המוחלט של שגיאות הזיכרון
- הצלחנו להיפטר מרוב השימושים הלא בטוחים במצביעים בעזרת שימוש בתכונות של **C++**
  - ניצול בנאים, הורסים והשماتות ל - **טיפול אוטומטי בזיכרון**
  - שימוש **באוספים של STL** לטיפול מסודר במבנה הנתונים
- עם זאת, יש מקרים שבהם עדין נשתמש בפונקציות, לדוגמה – במקרים המערבים ירושה (נלמד בהרצאות).
  - עבודה עם מצביעים מבטלת את כל היתרונות שהשנו ב- **-fC++!**
  - **צריך לנוהל זיכרון בצורה מפורשת** בכל מקום בתוכנית!

```
Stack<Circle*> circles;
circle.push(new Circle(3.5));
...
...
```

```
...
delete circles.top();
circles.pop();
```

# מצביים חכמים

מה הבעיה בקוד הבא?

```
int bad() {
 Circle* ptr1 = new Circle(5.0);
 Circle* ptr2 = new Circle(7.0);
 //...
}
```

אם הקריאה השנייה  
נכשלה יש דליפת זכרון!

```
int ok() {
 Circle* ptr1 = new Circle(5.0);
 try{
 Circle* ptr2 = new Circle(7.0);
 catch (...) {
 delete ptr1;
 throw
 }
 //...
}
```

פתרון אפשרי:

# unique\_ptr

- נוכל ליצור מחלקה, המתנהגת כמו צביע אך מוסיפה התנagoות נוספת

```
template<typename T>
class unique_ptr {
 T* data;
public:
 explicit unique_ptr(T* ptr = nullptr)
 : data(ptr) {}
 ~unique_ptr() { delete data; }
 T& operator*() const { return *data; }
 T* operator->() const { return data; }
};
```

– למשל, שחרור העצם המוצבע  
בזמן הריסת האובייקט

- ככלمر ניצור מצביע "חכם"  
шибצע שחרור אוטומטי

```
int good() {
 unique_ptr<Circle> ptr1(new Circle(5.0));
 unique_ptr<Circle> ptr2(new Circle(7.0));
 //...
}
```

```
int bad() {
 Circle* ptr1 = new Circle(5.0);
 Circle* ptr2 = new Circle(7.0);
 //...
}
```

# העתקה מצבי חכם

- מה יקרה אם ננסה לבצע השמה או העתקה למצבי?

```
unique_ptr<Circle> ptr1(new Circle(5.0));
unique_ptr<Circle> ptr2 = ptr1;
// uh-oh!
```

- העתקה כזו גורמת לשני מצביעים חכמים להצביע לאותו הזיכרון – כל אחד מהם ינסה לשחרר אותו כשהוא נהרס!
  - הרישת אותו זיכרון פגמיים גורמת להתנהגות לא מוגדרת!
- על מנת למנוע זאת, **ניתן לחסום העתקות** של unique\_ptr

```
template<typename T>
class unique_ptr {
public:
 // ...
 unique_ptr(const unique_ptr&) = delete;
 void operator=(const unique_ptr&) = delete;
};
```

# העתיקת מצבים חכמים

- אם יש אופציות אחרות לטיפול בהעתיקת מצבים חכמים, למשל כדי שנוכל להעביר או להחזיר אותם מפונקציות?
  - **שכפול האובייקט המוצבע**: לא! התנאות זו אינה דומה כלל לתנאות של מצבים אמיתיים. היא לא תעבור במצב שלא ממושך בנאי העתקה, או במצב של ירושה שילמד בהמשך.

```
unique_ptr(const unique_ptr& other) {
 data = other->data ? new T(*other->data)) : nullptr;
}
```

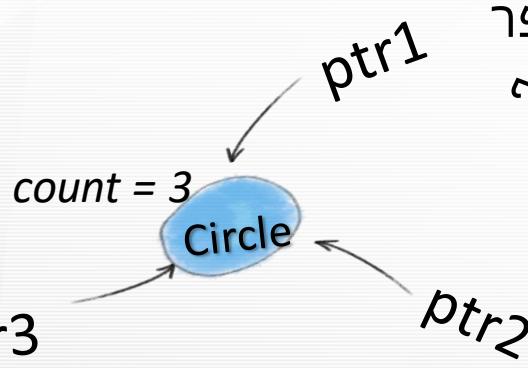


- בפועל קיימים פתרונות המתאים ל**העברה** (לא העתקה!) של unique\_ptr למקום אחר, אך זה מעבר למסגרת חומר הקורס...

# העתיקת מצביע חכם

- 

אלטרנטיבות לטיפול בהעתיקת מצביעים חכמים:



- **שיתוף בעלות**: שימוש מצביע חכם המאפשר למספר רב של מצביעים מטיפוס זה להצביע על עצם משותף אחד.
- מי אחראי על שחרור הזיכרון?
  - האخرון שמשחרר.
- המצביע החכם משתמש **במונח הצבעות** (**Reference Counting**) כדי לוודא שהעצם משוחרר רק כשהאין עוד מצביעים אליו

```
template<class T>
class shared_ptr {
private:
 T* data;
 int* counter;
 // ...
}
```

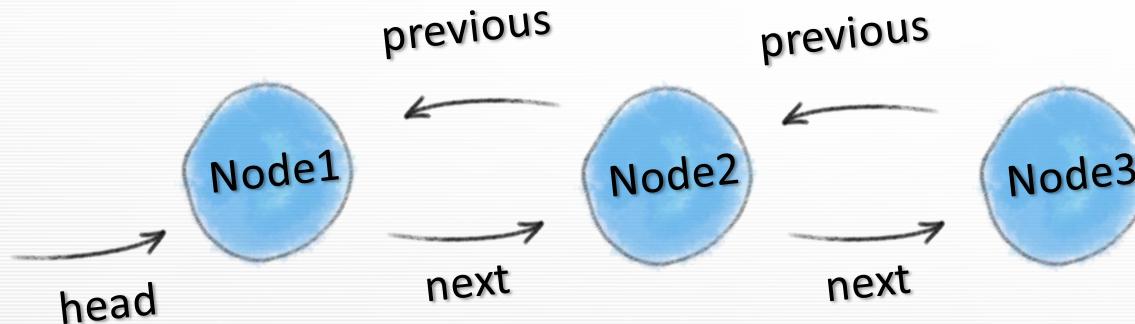
# שימוש חלקי לדוגמה – `shared_ptr`

```
shared_ptr(const shared_ptr& other) : data(nullptr), counter(nullptr) {
 if (other.data) {
 this->data = other.data;
 this->counter = other.counter;
 *counter++;
 }
}

~shared_ptr() {
 *counter--;
 if (*this->counter==0) {
 delete this->data;
 delete this->counter;
 }
}
```

# האם תמיד ניתן להשתמש ב- ?shared\_ptr

נניח שיש לנו רשימה מקוושרת דו כיוונית המורכבת מצמתים המחזיקים אחד לשני:



מה קורה אם אנחנו הורסים את המצביע `head`? האם הזיכרון ישוחרר?

- אם יש לנו הצבעה מעגלית, לא יוכל להשתמש ב-`!shared_ptr`

# מצביים חכמים - סיכום

- יתרונות:
  - כל הקוד לניהול הזיכרון **נכתב רק פעם אחת**
  - הקוד יהיה **פשוט ועמיד** יותר בפני שגיאות זיכרון
- std::unique\_ptr
  - מתאים **לבעלויות יחידה** על זכרון
  - לא מאפשר העתקה והשמה
- std::shared\_ptr
  - מתאים **لבעלויות משותפת** על זכרון
  - רק ההצעה האחורונה שנשארת מפנה את הזיכרון המוצבע
  - `shared_ptr` אינם מתאים לכל המצבים! השתמשו רק בסיטואציות מתאימות.

# ספרית התבניות הסטנדרטית (STL)

vector ♦

list ♦

iterator ♦

# ספרית התבניות הסטנדרטית

- קיימת בכל שימוש של `C++`.
- מכילה **אוסףים** (Containers) ו**אלגוריתמים**
- משתמש בתבניות (templates):
  - אוספי הנתונים גנריים
  - האלגוריתמים המספקים גנריים
  - מאפשרת הרחבה ע"י המשתמש
  - שומרת על יעילות הקוד
  - נועדה להכיל מחלקות ופונקציות שימושיות לפתרון של מגוון רחב של בעיות

# STL - סטנדרט

- ספירת התבניות הסטנדרטית מכילה מגוון רחב של מבני נתונים:
  - מבני נתונים שומרים נתונים לפי סדר הכניסה: vector, deque, array, list
  - מבני נתונים שומרים נתונים בסדר ממוין כלשהו: set, map, multiset
  - מבני נתונים שלא שומרים על סדר: unordered\_set, unordered\_map
  - ועוד רבים אחרים...
  - אנחנו נלמד רק על חלק קטן
  - [לקריאה נוספת:](http://cppreference.com) <http://cppreference.com>

# ווקטור

- הוסף (container) בשימוש **הכי נפוץ** הוא **std::vector**
  - #include <vector>
- ווקטור הוא **מערך דינامي**
  - איברים הם **רציפים בזיכרון**
  - גישה מהירה** בעזרת אינדקסים (**Fast Random Access**)
  - הכנסה והסרת **מהירה** של איברים מסוף הווקטור.
    - בגלל הקצאה של זיכרון מראש.
  - הכנסה והסרת מכל מקום אחר בווקטור עלול להיות **מאוד איטי**

```
void read_and_print() {
 vector<int> numbers;
 int input;
 while (cin >> input) {
 numbers.push_back(input);
 }
 for (int i = 0; i < numbers.size(); ++i) {
 cout << numbers[i] << " ";
 }
}
```

הוספת איבר לסוף  
הווקטור



# METHODS OF vector

vector.h

```
template<typename T>
class vector {

 vector();
 vector(const vector& c);
 vector(size_t num, const T& val = T());
 vector& operator=(const vector& v);
 ~vector();

 T& operator[](size_t index);
 const T& operator[](size_t index) const;

 void push_back(const T& val);
 void pop_back();

 void resize(size_t size, T val = T());
 size_t size() const;
};
```

קובץ ממשק חלק של vector.

עוד מידע על הממשק של וקטור או אוסףים אחרים נמצא **באינטרנט**.

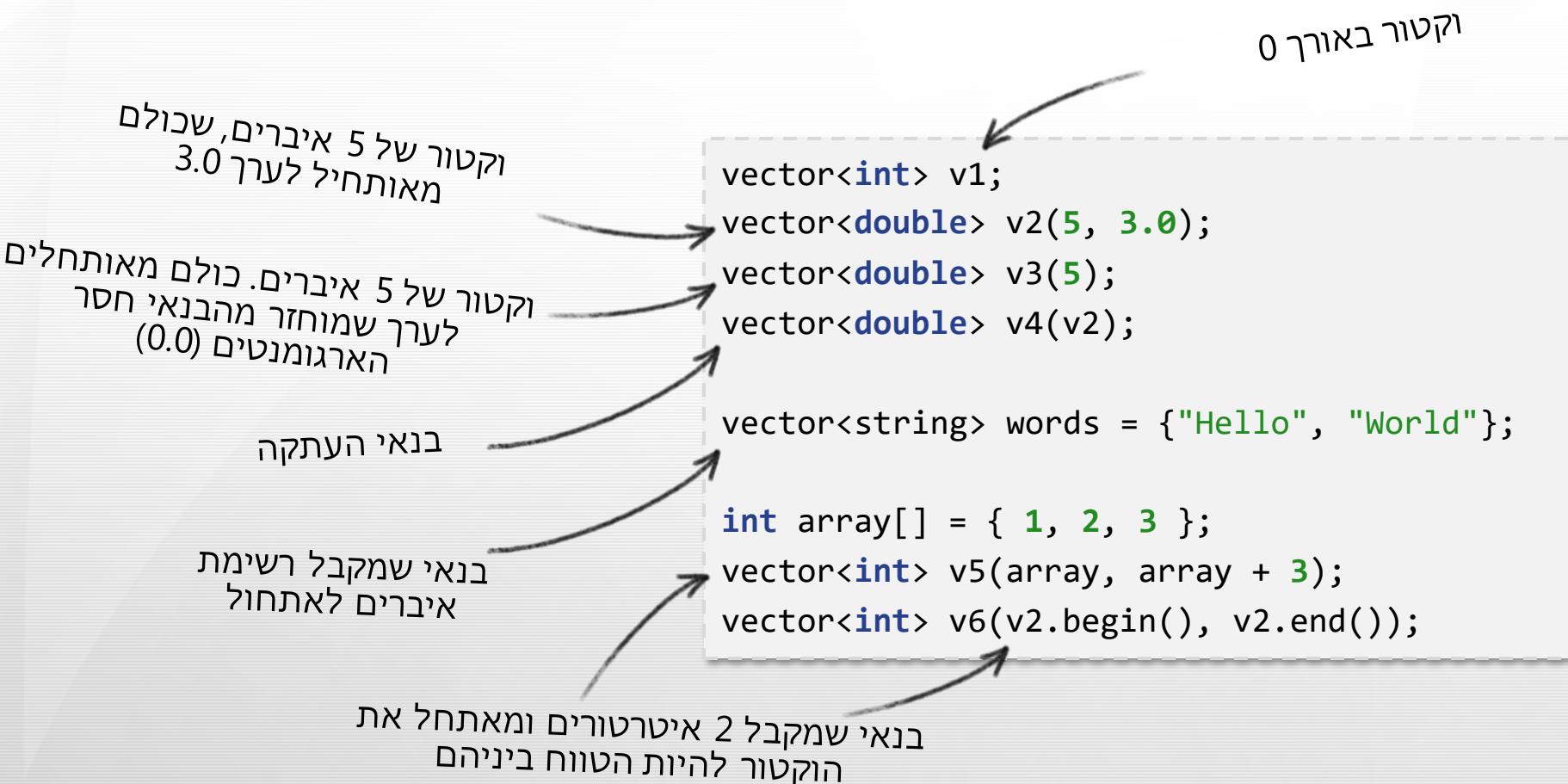
ל-vector יש מетодות ואופרטורים רבים  
נוספים

- ניתן פשוט לחפש באינטרנט:
  - למשל

<http://www.cppreference.com/>

# אתחול וקטורים

- `std::vector` יש מגוון רחב של בناים. הנה דוגמה לחלקם:



# גישה לוקטור

- ניתן לגשה לאיברים בוקטור כמו במערך של C
- **ניתן לגשת בעזרת operator [ ]**. אם האינדקס אינו תקין, אז **התוצאה אינה מוגדרת** – ערכו זבל, שגיאת זיכרון..
- קיימת מתודה בשם **(at)** המעניקה גישה בטוחה יותר אך איטית יותר. אם האינדקס אינו תקין **נזרקת חריגה**.

```
vector<string> words = { "Hello", "World" };

for (int i = 0; i < words.size(); ++i) {
 cout << words[i] << endl;
}

try {
 cout << words.at(2) << endl;
} catch (const std::out_of_range& e) {
 cerr << e.what() << endl;
}
```

נעדי להשתמש ב**(at)** כאשר ניגש לוקטור שלא בוחר לולאת **for**. יש להשתמש ב**[ ]** כאשר היעילות חשובה.

# איטרטור של וקטור

▪ תומך באיטרציה.  
std::vector

▪ מחייב איטרטור לתחילה הווקטור, ואיטרטור במקום אחד אחריו הסוף בהתאם.

```
vector<int> v = { 2, 3, 4, 1, 2, 4 }
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
 cout << *it << " ";
}

```

2 3 4 1 2 4

▪ מספקות ממשק לאיטרציה בסדר הפוך.

```
vector<int> v = { 2, 3, 4, 1, 2, 4 }
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend();
++it) {
 cout << *it << " ";
}

```

4 2 1 4 3 2

# לולאת foreach

- מבנה הלולאה הבא מאוד נפוץ ב-C++:

```
for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
 int num = *it;
 // ... do something with num
}
```

▪ C++11 הוסיף לנו את הסינטקסט הבא עבור לולאה שקולה:

```
for (int num : vec) {
 // ... do something with num
}
```

- הסינטקס עובד עבורי:
  - כל האוספים הסטנדרטיים ב-C++.
  - כל מחלקת המגדירה פונקציות `(begin() -() end)` המחזירות אובייקט איטרציה, כולל אובייקט שמוגדר בשולשת הפעולות הבסיסיות: `*`, `+`, `=!` (קידום, קראיה והשווואה).

# הוספה והסרה

- ניתן להסיר ולהוסיף איברים **ביעילות** מסוף הוקטור.

{ "Hello", "World", "!!!", }

```
vector<string> words = { "Hello", "World" };
```

{ "Hello", "World" }

```
words.push_back("!!!");
```

שינוי הגודל ל-5.

```
words.pop_back();
```

```
words.resize(5);
```

- ניתן להסיר ולהוסיף איברים מכל מקום בוקטור, אבל **לא ביעילות**.

מכניס איבר לפני האיטרטור

(it points to "World")

{ "Hello", "Lovely", "World" }

```
vector<string> words = { "Hello", "World" };
```

```
vector<string>::iterator it = words.begin();
++it;
```

```
words.insert(it, "Lovely");
```

```
words.erase(words.begin());
```

מוחק את "Hello"  
כדי לשם האיטרטור מצביע

# וקטור - הערות



- האובייקטים המאוחסנים בתווך וקטור חייבים להגדר: **בנאי העתקה, ואופרטור השמה.**
  - אין צורך בבנאי חסר ארגומנטים. ↪ בשונה ממערכות רגילים

# std::list

- אוסף שמשמש רשימה מקוشرת דו-כיוונית
  - מאפשרת הוספת איברים מהירה לכל מקום בראשימה
  - מאפשרת גישה סדרתית קדימה ואחוריה בראשימה
  - אינה מאפשרת גישה מהירה למקומות שרירוטי באוסף
- חלקים גדולים מהמנשך של `list` ו-`vector` משותפים
  - השימוש במנשך דומה, מאפשר למשתמש להחליף בקלות את סוג האוסף שהוא משתמש בו, גם בשלב מאוחר בתכונות.
- בפועל, `vector` הוא פתרון פשוט ויעיל ברוב המקרים, ומומלץ לשימוש בו.

# METHODS CHOSEN FOR list

```
template<typename T> list.h
class list {
 list();
 list(const list& c);
 list(size_t num, const T& val = T());
 list& operator=(const list& c);
 ~list();

 void push_back(const T& val);
 void pop_back();
 void push_front(const T& val);
 void pop_front();

 size_t size() const;
};
```

- נוספו מетодות להסרת והוספת איברים גם בראש הרשימה
- מетодות לגישה מהירה לאיבר באמצעות הרשימה נעלמו
- כיצד נוכל לגשת לאברי הרשימה בצורה נוחה...?

# std::deque

- מבוטא **deck** – קיצור של double ended queue
- כמו וקטור אבל מאפשר הכנסה והוצאה מהירה משני הקצוות
  - תור דו כיוני: כי אפשר גם להוציא וגם להכניס משני קצאות התוור
- מאוד דומה לממשק של vector
  - משתמשים בMETHODS push\_front ו pop\_front

```
deque<string> words = { "Hello", "World" };

words.pop_front();
words.push_front("Hi");
```

# std::set

- במבנה נתונים "אโซצייאטיביים", סדר הכנסה לא מופיע על סדר האיטרציה על האיברים.
- דוגמה למבנה נתונים אโซצייאטיבי: בניית הנתונים **std::set** השומר על איבריו בסדר ממוקן באמצעות `<operator<`.
- בניית הנתונים **std::set** מימוש קבוצה: אין בו כפליות.

```
set<int> my_set;
my_set.insert(5);
my_set.insert(3);
my_set.insert(5);
cout << my_set.size(); // prints 2
```

# std::map

- עוד מבנה נתונים אסוציאטיבי: מבנה הנתונים **std::map**
- מבנה הנתונים **std::map** שומר זוגות של  $\text{key} \mapsto \text{value}$  כאשר יש לכל היותר זוג אחד עם כל מפתח.

```
map<string, int> grades;
grades["Baraa"] = 100;
grades["Ron"] = 95;
grades["Daniel"] = 92;
std::cout << grades["Baraa"];
```

# std::map

- ניתן לבצע איטרציה על הזוגות השמורים ב-map:

```
for(const pair<string, int>& grade : grades) {
 cout << grade.first << " got grade " << grade.second << endl;
}
```

- ניתן להשתמש ב-[operator []] כדי לגשת לזוג עם מפתח נתון, כדי לשנות ערך השיר למפתח או כדי להכניס זוגות חדשים:

```
map<int, int> m;
m[1] = 2; // adding pair - key: 1, value: 2
m[1] = 3; // changes existing key 1 to contain the value 3
cout << m[1]; // prints 3
cout << m[42]; // prints value provided by int default c'tor...
```

# std::map

- ניתן למצוא אם מפתח נתון נמצא ב-map באמצעות המתודה `:find`

```
map<int, int> m;
map<int, int>::iterator it;
// some code...
it = m.find(42);
if (it != m.end()) {
 cout << it->second << endl;
}
```

– במידה והמפתח קיים ב-map – יוחזר איטרטור ה"מצבייע" לצמד עם המפתח  
הمبرוקש וה-value המתאים

– במידה והמפתח לא קיים – יוחזר איטרטור ל"סוף" ה-map, המציג כשלון או  
הצבעה לא חוקית

# דוגמה - std::map

- תרגיל: קיבל קלט מהמשתמש ונספר את המילה הכי נפוצה שופיעיה בקלט.

```
void most_frequent_word() {
 string word;
 map<string, int> count;
 int max_count = 0;
 string max_word = 0;
 while(cin >> word) {
 count[word]++;
 if (count[word] > max_count) {
 max_count = count[word];
 max_word = word;
 }
 }
 cout << max_word << endl;
}
```

# איטרטורים ו-STL

- כל האוסףים ב-STL מספקים iterator בעזרתו ניתן לעבור על איבריהם. איטרטוריםמאפשרים לנו לכתוב קוד אחד שיכל לפעול עם מספר אוספים.
- המתודה `(begin()`מחזירה iterator לתחילת האוסף (כמו בוקטור).
- המתודה `(end())`מחזירה iterator לאיבר "DMA" שנמצא אחד אחרי האיבר האחרון (כמו בוקטור).
- למשתמש אין צורך לדעת כיצד ממומש האיטרטור, הוא רק צריך לדעת באילו פעולות הוא תומך
  - כל האיטרטורים תומכים לפחות בקידום `(++)`, גישה `(>,*)` והשוואה `(!=,==)`.

# const\_iterator

- בנוסף, כל אוסף מגדיר גם טיפוס **const\_iterator**
  - const\_iterator מאפשר לעבור על אברי האוסף ובשונה מ iterator יכול רק לקרוא אותו, אך לא לשנות.
  - אוסף שמוגדר כ-const מחזיר const\_iterator

```
const vector<int> v(10, 3);
vector<int>::const_iterator i = v.begin();
*i = 7; // error! *i is a const int&
```

- אוסף שאינו const מחזיר איטרטור רגיל (לקראיה וכתיבה)
- מה אם האוסף שלנו אינו const, אך אנו רוצים לעבור עליו עם איטרטור ולקראת ממו, **ושהköםפילר יודא** שאיננו משנה את האוסף דרך האיטרטור?
  - ב-STL מוגדרת המרת המרה מ-iterator ל-const\_iterator (אך לא הפוך):

```
vector<int> v(5,7);
vector<int>::const_iterator i = v.begin();
*i = 7; // error, i is a const_iterator
```

# סוגי איטרטורים

- אוספים שונים מוחזרים איטרטורים התומכים בפעולות שונות, למשל:
  - האיטרטור של `list` הוא מסוג **bidirectional iterator**, המאפשר קידום האיטרטור עם אופרטור`++` והחזרתו לאחר עם אופרטור`--`.
  - האיטרטור של `vector` הוא מסוג **random access iterator**, ומאפשר בנוסף לפעולות הנ"ל גם קידום ונסיגה עם מספר שלם כלשהו, וגם גישה עם האופרטור `[]`.
  - התייעוד של STL מציין לכל אוסף את סוג האיטרטור שהוא מספק.

```
for(vector<double>::iterator it = myVector.end()-1; it != myVector.begin()-1; --it)
{
 cout << *it << endl;
}
```

# עוד שימושים של איטרטורים

- מетодות רבות של אוספים משתמשות באיטרטורים כדי לציין **מיקומים** באוסף
  - למשל, הוספת איבר באמצעות רשימה ממוינת:

```
list<double>::iterator it = myList.begin();
while (it != myList.end() && *it < num) {
 ++it;
}
myList.insert(it, num); // insert num before it
```

- מחיקת האיבר החמישי בוקטור:  

```
myVector.erase(v.begin() + 4); // note: not efficient
```
- שינויים באוסף עלולים לגרום לביטול התוקף של איטרטור (**invalidation**)
  - גישה לאיטרטור שאינו בתוקף **אינה מוגדרת**
  - למשל `vector::insert` עלולה לגרום לכל האיטרטורים הקיימים לצאת מתוקף
  - מומלץ להניח את **הנחה המחייבת** שככל שינוי של אוסף מוציא מתוקף את כל האיטרטורים שלו
  - אם הנחה זו מחייבת מדי ניתן למצוא מידע מדויק בתיעוד האוסף

# הוספת תמייה באיטרטורים

```
class String {
 int length;
 char* value;
public:
 //...
 typedef char* iterator;
 typedef const char* const_iterator;

 iterator begin() {
 return value;
 }
 const_iterator begin() const {
 return value;
 }
 iterator end() {
 return value + length;
 }
 const_iterator end() const {
 return value + length;
 }
 //...
};
```

- נזכר במחלקה `String` שמייננו, כיצד נוכל להתאים את המחלקה לשימוש באיטרטורים?
- במקרה של `String`, נוכן פשוט להשתמש במצביע ל-`char`
- שימושם לב להגדרות ה-`typedef` שהוא מוסיף – המשתמש אינו נדרש לדעת מהו הטיפוס האמתי שעומד מאחורי טיפוסי האיטרטורים
- לא תמיד ניתן להשתמש בטיפוס קיים כטיפוס האיטטור, לעיתים נוצר להגדיר מחלוקת חדשה לשם כך

# STL - סיכום

- STL מגדירה מספר אוסףים נוחים ויעילים לשימוש
- שימוש באוספים אלו חוסך **ניהול זיכרון ידני עם new ו-delete**
- **האוסף std::vector** מתאים לרוב המקרים – השתמשו בו
  - כולל במצבים בהם לא יודעים מראש את מספר האיברים הסופי
  - לאורק זמן, ההבדלים בין האוספים יהיו יותר ברורים ותוכלו לבחור את האוסף המתאים ביותר
- הגישה לאוספים מבוצעת דרך **איטרטורים**, כך שניתן לגשת באופן דומה לכל האוספים של STL
- השימוש של חלק מהאוספים לימד בקורס מבני נתונים 1.

# algorithm

# הספריה algorithm

- איטרטורים מאפשרים לנו להשתמש במשק אחיד למספר אוסףים:
  - אוסף STL
  - חלקים מאוסף: תת-קבוצה, תת-רשימה, תת-וקטור...
  - std::string
  - מערכים רגילים
- STL מכילה מימושים מובנים למספר **אלגוריתמים** הפעלים על איטרטורים.
- כדי להשתמש בספריה, נבצע `#include <algorithm>`

```
/* prototype:
template <class Iterator, class T>
int std::count(Iterator first, Iterator last, const T& value);
*/
vector<int> numbers = {1, 5, 2, 5, 7}
cout << count(numbers.begin(), numbers.end(), 5); // prints 2
```

# דרישות נוספות

- חלק מהאלגוריתמים דורשים תמינה בפעולות מסוימות מצד האיטרטורים.

```
/* prototype:
template <class Iterator>
void std::sort(Iterator first, Iterator last);
Container must have operator[]
*/
vector<int> numbers_vec = {1, 5, 2, 5, 7}
sort(numbers_vec.begin(), numbers_vec.end()); // works
list<int> numbers_list = {1, 5, 2, 5, 7}
sort(numbers_list.begin(), numbers_list.end()); // does not compile
```

# יעילות

- האלגוריתמים משתמשים תמיד במשמעות היעיל ביותר לפי הפעולות שהאיטטור תומך בהן.

```
/* prototype:
template <class Iterator>
bool std::binary_search(Iterator first, Iterator last, const T& value);
*/
vector<int> numbers_vec = {1, 5, 2, 5, 7}
binary_search(numbers_vec.begin(), numbers_vec.end(), 6); // O(logn)
list<int> numbers_list = {1, 5, 2, 5, 7}
binary_search(numbers_list.begin(), numbers_list.end(), 6); // O(n)
```

# גמישות

- חלק מהאלגוריתמים ניתנים להטאה באמצעות הפרמטרים שלהם.
  - למשל, נמצא את כל האיברים המקיימים פרדיקט כלשהו:

```
/* prototype:
template <class Iterator, class Condition>
bool std::any_of(Iterator first, Iterator last, Condition value);
*/
class SomeCondition { bool operator()(int elem) { /* ... */ } }
vector<int> numbers = {1, 5, 2, 5, 7}
bool res = any_of(numbers.begin(), numbers.end(), SomeCondition());
```

# algorithm - סיכום

- STL מגדירה מספר כלים נוחים לתוכנות בשפת C++
- השתמשו באוספים אלו במקומם **להמציא מחדש את הgalgal**
- אל תצפו לזכור את כל ה-STL בעל פה – הוא גדול מדי.
  - חפשו באינטרנט משאבים נוספים
  - בהרבה מקרים מה שאתם רוצים למשם כבר קיימים ☺
- יש עוד הרבה – אבל מחוץ לחומר הקורס שלנו.

# תרגול מס' 9

❖ שאלות לתרגול - C++

# שאלה 1

```
#include <iostream>
using std::cout;
using std::endl;

template<class T>
class A {
public:
 A() { cout << "A::A()" << endl; }
 A(const A& a) : i(a.i) { cout << "A::A(A&)" << endl; }
private:
 T i;
};

template<class T>
class B {
public:
 B(A<T> aa) : a(aa) { cout << "B::B(A)" << endl; }
 B(const B& b) : a(b.a) { cout << "B::B(B&)" << endl; }
 A<T> a;
};

class C: public B<int> {
public:
 C(A<int> aa) : B<int>(aa), a(aa) {
 cout << "C::C(A aa)" << endl; }
 ~C() { cout << "C::~C()" << endl; }
 A<int> a;
};
```

■ מה מדפסה התכנית הבאה?

```
int main() {
 cout << "--1--" << endl;
 A<int> a;
 cout << "--2--" << endl;
 A<double> a1;
 cout << "--3--" << endl;
 B<int> b(a);
 cout << "--4--" << endl;
 B<int> b1(b);
 cout << "--5--" << endl;
 C c(a);
 cout << "--6--" << endl;
 B<int>& b2 = c;
 cout << "--7--" << endl;
 return 0;
}
```

# שאלה 1 - פתרון

--1--

A::A()

--2--

A::A()

--3--

A::A(A&)

A::A(A&)

B::B(A)

--4--

A::A(A&)

B::B(B&)

--5--

A::A(A&)

A::A(A&)

A::A(A&)

B::B(A)

A::A(A&)

C::C(A aa)

--6--

--7--

C::~C()

יודפס:

# שאלה 2

- בתרגול 9 ראיינו את אוסף המחלקות עבור צורות, להלן תזכורת של הקוד עבור המחלקות

```
class Circle : public Shape {
 int radius;
public:
 Circle(int x, int y, int radius) :
 Shape(x,y), radius(radius) {}
 double area() const override {
 return radius*radius*PI;
 }
};
```

```
class Square : public Shape {
 int edge;
public:
 Square(int x, int y, int edge) :
 Shape(x,y), edge(edge) {}
 double area() const override{
 return edge*edge;
 }
};
```

```
class Shape {
 int center_x, center_y;
public:
 Shape(int x, int y) :
 center_x(x), center_y(y) {}
 virtual ~Shape() {}
 virtual double area() const = 0;
};
```

הalon:

▪ ברצוננו לאפשר הדפסת של צורות ע"י  
אופרטור ההדפסה <> כרך שעבור עיגול  
שרדיוסו 3 יודפס Circle: radius=3  
ואילו עבור ריבוע שאורך הצלע שלו היא 2  
יודפס Square: side length=2  
▪ הוסיפו תמיינה באופרטור ההדפסה למחלקות  
הנ"ל. תארו במדוייק את שינויי הקוד שלכם  
והיכן הם צריכים להתבצע.

# שאלה 2 – פתרון מספר 1

```
class Shape {
 //...
protected:
 friend ostream& operator<<(ostream& os,
 const Shape& s);

 virtual void print(ostream& os) const = 0;
};

class Circle: public Shape {
 //...
protected:
 void print(ostream& os) const override{
 os << "Circle: radius=" << radius;
 }
};
```

```
class Square: public Shape {
 //...
protected:
 void print(ostream& os) const override{
 os << "Square: side length=" << edge;
 }
};

ostream& operator<<(ostream& os, const Shape& s) {
 s.print(os);
 return os;
}
```

# שאלה 2 – פתרון מספר 2

```
class Shape {
 //...
protected:
 friend ostream& operator<<(ostream& os,
 const Shape& s);

 virtual string getRepresentation() const = 0;
};

class Circle: public Shape {
 //...
protected:
 string getRepresentation() const override {
 ostringstream os;
 os << "Circle: radius=" << radius;
 return os.str();
 }
};
```

```
class Square: public Shape {
 //...
protected:
 string getRepresentation() const override {
 ostringstream os;
 os << "Square: side length=" << edge;
 return os.str();
 }
};

ostream& operator<<(ostream& os, const Shape& s) {
 return os << s.getRepresentation();
}
```

# שאלה ממבחן – אביב 2020 מועד ב'

שאלה זו עוסקת ב-**function objects** (כלומר, מחלקות עם אופרטור סוגרים).  
שימוש לב:

1. בכל השאלה, ניתן למשתמש פונקציות של מחלקה ישירות בתוך הגדרת המחלקה.

**סעיף א (11 נקודות):**

נתון קטע הקוד הבא:

```
int main() {
 int k;
 cin >> k;
 try {
 Divides div(k);
 int x;
 cin >> x;
 cout << (div(x) ? "k divides x" : "k does not divide x");
 }
 catch (const std::exception& e) {
 cout << e.what();
 }
 return 0;
}
```

# שאלה ממבחן – אביב 2020 מועד ב'

1. סעיף א (11 נקודות):

נתון קטע הקוד הבא:

```
int main() {
 int k;
 cin >> k;
 try {
 Divides div(k);
 int x;
 cin >> x;
 cout << (div(x) ? "k divides x" : "k does not divide x");
 }
 catch (const std::exception& e) {
 cout << e.what();
 }
 return 0;
}
```

משו את המחלקה Divides כך שקטע הקוד לעיל ידפיס:

- “x” אם x מחלק ב-k ללא שארית.
- “k divides x” אם x אינו מחלק ב k ללא שארית.
- “k does not divide x” אם x מחלק ב 0!
- “Cannot divide by 0!”

# שאלה מבחן – אביב 2020 מועד ב'

## סעיף א - פתרון

```
int main() {
 int k;
 cin >> k;
 try {
 Divides div(k);
 int x;
 cin >> x;
 cout << (div(x) ? "k divides x" : "k does not divide x");
 }
 catch (std::exception& e) {
 cout << e.what();
 }
 return 0;
}
```

ממשו את המחלקה Divides כר שקטע הקוד הבא ידפיס:

- “ $x$  אם  $x$  מתחלק ב- $k$  ללא שארית.
- “ $k$  does not divide  $x$ ” אם  $x$  אינו מתחלק ב- $k$  ללא שארית.
- $.k==0$  “Cannot divide by 0!”

```
#include <stdexcept>
class Divides {
 int m_k;
public:
 Divides(int k) : m_k(k)
 {
 if (k==0)
 throw std::runtime_error("Cannot divide by 0!");
 }
 bool operator()(int x) const{
 return x%m_k==0;
 }
};
```

ניתן גם להגדיר מחלקה נפרדת לחירגה (למשל DivideByZero). יש לוודא כמהון שהיא יורשת מ- `std::exception` ומימושה את `what` כנדרש.

# שאלה ממבחן – אביב 2020 מועד ב'

## סעיף ב (11 נקודות):

בסעיף זה נגדיר **מחלקת בסיס** אשר ממנו יהיה ניתן לרשות מחלקות תנאי נוספות, בדומה למחלקה Divides. המחלקה שנגדיר תהיה גנרטית, בשם **Cond<T>** (קיצור של *Condition*), כאשר T הוא הטיפוס **עליו התנאי פועל** (למשל, במקרה של Divides שפועלת על משתנים מטיפוס int, הטיפוס T יהיה int)

a. כתבו את הגדרת מחלקת הבסיס Cond מນשו אותה כך שבאופן דיפולטיבי התנאי שלו תמיד מתקיים.

ב. הסבירו כיצד יש **לשנות את הקוד שלכם מסעיף א'**, כך שהמחלקה Divides תירש מ- Cond. ציינו רק את השינויים הדרושים בתשובתכם לסעיף א' (אין צורך לכתוב את ההגדרה המלאה).

# שאלה מבחן – אביב 2020 מועד ב'

## סעיף ב - פתרון

```
class Divides {
 int m_k;
public:
 Divides(int k) : m_k(k)
 {
 if (k==0)
 throw std::runtime_error("Cannot divide by 0!");
 }
 bool operator()(int x) const{
 return x%m_k==0;
 }
};
```

סעיף א

א. כתבו את הגדרת מחלקה הבסיס Cond ממשו אותה כך שבאופן דיפולטיווי התנאי שלה תמיד מתקיים.

זכור בפתרון שלנו לסעיף א':

פתרון לסעיף ב חלק א:

```
template <class T>
class Cond {
public:
 virtual bool operator()(const T& x) const{
 return true;
 }
 virtual ~Cond {}
};
```

# שאלה מבחן - אביב 2020 מועד ב'

## סעיף ב - פתרון

```
class Divides {
 int m_k;
public:
 Divides(int k) : m_k(k)
 {
 if (k==0)
 throw std::runtime_error("Cannot divide by 0!");
 }
 bool operator()(int x) const{
 return x%m_k==0;
 }
};
```

סעיף א

ב. הסבירו כיצד יש לשנות את הקוד שלבם מסעיף א' כך שהמחלקה Divides תירש מ- Cond. ציינו רק את השינויים הדרושים בתשובהתכם לסעיף א' אין צורך לכתוב את ההגדרה המלאה).

```
template <class T>
class Cond {
public:
 virtual bool operator()(const T& x)
const{
 return true;
 }
 virtual ~Cond {}
};
```

סעיף ב, תת סעיף א

פתרון:

השינויים הנדרשים במחלקה Divides הם:

1. ירושה מ- Cond<int>

```
class Divides : public Cond<int>
:override
bool operator()(const int& x) const
override{
```

# שאלה מבחן – אביב 2020 מועד ב'

## סעיף ג (21 נקודות):

בסעיף זה נמשח אופרטור שיאפשר לנו לחבר כמה תנאים יחד (למשל,  $x$  מחלק גם ב-3 וגם ב-5). אופן השימוש באופרטור יהיה כזה:

```
Divides div3(3), div5(5);
CondAnd<int> cond = div3 && div5;
// cond(15)==true, cond(13)==false
```

שימוש לבשנית לבצע פעולה  $\&$  רק בין שני תנאים **שפועלים על אותו טיפוס** (למשל, `int` בדוגמה לעיל). כמו כן, על המחלקה `CondAnd<int>` לרשות אף היא מ-`Cond`.

משמעותו את המחלקה הגדנית `CondAnd` ואת האופרטור  $\&\&$  כך שהקוד לעיל יתאפשר ויעבד נכון.

# שאלה מבחן – אביב 2020 מועד ב'

## סעיף ג - פתרון

משו את המחלקה הכלכלית CondAnd ואת האופרטור `&&` כך שהקוד הבא יתאפשר ויעבד כמפורט.

```
Divides div3(3), div5(5);
CondAnd<int> cond = div3 && div5;
// cond(15)==true, cond(13)==false
```

פתרון:

זכור בהגדה של המחלקה Cond

```
template <class T>
class CondAnd : public Cond{
 const Cond<T>& m_cond1;
 const Cond<T>& m_cond2;
public:
 CondAnd(const Cond<T>& c1, const Cond<T>& c2)
 m_cond1(c1), m_cond2 (c2) {}
 bool operator()(const T& x) const override{
 return m_cond1(x) && m_cond2(x);
 }
};
```

```
template <class T>
CondAnd<T> operator&&(const Cond<T>& c1, const Cond<T>& c2){
 return CondAnd<T>(c1,c2);
}
```

```
template <class T>
class Cond {
public:
 virtual bool operator()(const T& x)
const{
 return true;
}
 virtual ~Cond {}
```

# שאלות C++ בבחן

בחן יהיו 2 שאלות גדולות על C++ - עם משקל של 30-35% כל אחת.

ה שאלה ראשונה תהיה כללית על C++ ותcosa את כלל החומר שנלמד בהרצאות ותרגולים. השאלה תתמקד בבדיקה הידע על התכונות שונות של C++ ושימוש במבני הנתונים שנלמדו.

ה שאלה השנייה תתמקד בתכנון נכון ב C++ עם שימוש בירושה ופולימורפיזם, ותכלול כתיבת הוכנת TUM. תכנון של מערכות גדולות יודגמו בתרגיל בית 4 ובהרצאות.

# תרגול מס' 10

- ❖ **python** - הכרת השפה וסביבת העבודה
  - ❖ **משתנים** - טיפוסי משתנים, שימושות ופעולות ארכיטמטיות
  - ❖ **מחרחות**
  - ❖ **מבנה בקרה** - if, while-loops



# שפת התכנות פיתון

פיתון היא שפת תכנות שזוכה לפופולריות רבה.

קיימת קהילת מתכננים גדולת משתמש בה ומגוון רב של חבילות הזמןנות לכל שימוש אפשרי.

קלת השימוש עבור פיתוח ובחינה של קוד חדש.

עבודה עם `interpreter` (הרחבה בשוקופית הבאה).

הסינטקס של השפה מאד פשוט, ואף דומה לשפה האנגלית.

היא תומכת במגוון רחב של מערכות הפעלה וקוד הנכתב בה יכול, במקרים רבים, לעבוד על מערכות הפעלה שונות ללא צורך להתאמה.

היא חינמית ו-Open Source (קוד פתוח).

Python

```
1 print("hello world!")
2
3
```

C

```
1 #include <stdio.h>
2 int main(){
3 printf("hello world!\n");
4 return 0;
5 }
6
```

אין צורך לנהל זיכרון בצורה ידנית. הקצאות וחרור זיכרון מתבצעות אוטומטית.

# Interpreter vs. Compiler

- compiler מתרגם את כל התוכנית לקוד מכונה יעיל (קובץ הרצה), שנitinן להפעיל ולהריץ ישירות על ידי המעבד.
  - Interpreter מבצע שורה אחר שורה בתוכנית, בצורה אינטראקטיבית.
  - זמן הריצה של תכנית שהודרכה עם compiler עם compiler כל מהיר יותר מאשר כבר למדנו על שפה אחת שעבדה תכנית העובדת עם interpreter.
- bash - interpreter עם





# הרכבת קוד פיתון

- תחילה, יש להתקין פיתון על המחשב (המחיר של interpreter (...interpreter
- בקורס נעבד עם python 3.6 (כל גרסה x. 3 תחתים)
- בשרת ה-CSL3 יש להשתמש בפקודה **python3.6** (כבר מותקן)
- ניתן להריץ תוכנית בפייתון בכל אחת מה דרכים הבאות:
  1. הפעלה אינטראקטיבית של python (אפשר בשורת הפקודה) והזנה של פקודות שורה-שורה
    - יציאה ע"י Ctrl+d (EOF) או הזנת הפקודה exit()

```
[mtmchk@cs13 ~]$ python3.6
Python 3.6.3 (default, Jan 27 2019, 09:41:42)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> █
```

# הרכבת קוד פיתון

2. הרצה של תסריט: יצרת קובץ טקסט עם קוד פיתון ("תסריט") והרכתו ע"י הרצה הפקודה

```
python hello.py
```

```
[mtmchk@csl3 ~]$ python hello.py
Hello, World!
[mtmchk@csl3 ~]$
```

**hello.py:**  
print("Hello, World!")

3. הרצת תסריט כ-"קובץ הרצה" (ב-unix): הפניה לפיתון בראש התסריט ע"י שורה שמתחליה ב-#! (shebang) והרצה כפקודה תוך מתן הרשות ריצה ע"י chmod – תוך מתן הרשות ע"י chmod – עובד בצורה דומה לכל interpreter

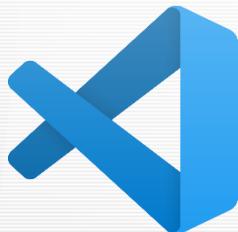
```
[mtmchk@csl3 ~]$ cat hello.py
#!/usr/local/bin/python3.6

print("Hello, World!")

[mtmchk@csl3 ~]$ chmod u+x hello.py
[mtmchk@csl3 ~]$./hello.py
Hello, World!
[mtmchk@csl3 ~]$
```

# עובדת עם פיתון בסביבת עבודה (IDE)

- ניתן להתקין תוסף לVSCode המאפשר עבודה עם פיתון.
  - יתרון של VSCode בעבודה על פרויקטים מרובי שפות – מאפשר עבודה עם כמה שפות בסביבת עבודה אחת.
- ישנן סביבות פיתוח פופולריות נוספות שנitin להשתמש בהן, למשל PyCharm
  - (חינמית לסטודנטים בטכניון)



# משתנים

- ❖ משתנים ב-`python3`
- ❖ טיפוסי משתנים
- ❖ השמות ו שינוי ערכי משתנים
- ❖ פעולות אРИתמטיות

# משתנים מספריים



- ניתן ליצור משתנה ב-`python` על ידי השמה ישירה:

```
> <varname> = <value>
> x = 7
```

- **אין צורך להזכיר** על משתנים ב-`python`, לאחר ביצוע השמה למשתנה הוא מוגדר אוטומטית.
- למשתנים אין טיפוס קבוע – רק לערכים.
- ניתן **להחליף** ערך של המשתנה על ידי השמה נוספת.

```
> x = 7 # Create a new variable with the value 7
> x = 5 # Change the value of x to 5
```

הערות בפייתון  
מתחילת בתו #  
ונמשכות עד סוף  
השורה

# טיפוס המשתנים בפייתון

- 
- 

טיפוס המשתנה נקבע על פי הערך שנתנו לו בהשמה.

ניתן לקבל את הטיפוס של המשתנה באמצעות הפקודה `(type)`

```
> x = 7
```

```
> type(x)
```

```
<class 'int'>
```

x הוא מטיפוס int כי  
יצרנו את x עם הערך 7

```
> x = 3.14
```

```
> type(x)
```

```
<class 'float'>
```

עכשו x הוא מטיפוס float  
כי שמננו בו מספר ממשי –  
3.14

▪ הפקודה `(type)` יכולה לפעול גם על ערכים ולא רק על משתנים.

– לדוגמה:

```
> type(56.5555)
```

```
<class 'float'>
```

# משתנים והשימוש

- המתרגם ( interpreter ) פועל באופן הבא:

1. מחשב את תוצאה **הביטוי**

2. מבצע השמה של תוצאה הביטוי אל תוך **המשנה**

חלק שמאל של ההשמה  
חייב להיות משתנה.

משנה

ביטוי (expression)

חלקימין יכול להיות כל  
ביטוי הנitin לחישוב.

> n = 10

> y = (35 + 99) \* 6

> 56 = y \* 7

שגיאה ! צד שמאל הוא  
לא משתנה

- שם המשתנה – רצף של תווים וספרות, חייב להתחיל בתרו.

# משתנים והשומות - דוגמאות

```
> n = 2 * 5
```

```
> print(n)
```

```
10
```

הפקודה `print` מדפסה למסך את ערך הביטוי שבסוגרים.

- ניתן להשתמש בערכו של המשתנה כחלק מחשבון של ביטוי:

```
> m = n * 2 + 1
```

`m = 21`

- ניתן לאתחל מספר משתנים באותה שורה עם ערכים שונים:

```
> a, b = 1, 2
```

`a = 1  
b = 2`

# משתנים והשומות - דוגמאות

- ניסיון לגשת לשם של משתנה אשר לא קיים יגרום **לשגיאת זמן ריצה** אשר תגרום ליציאה מהתוכנית (במצב של הרצת הקוד כתרريط), או להפסקת ריצת הקוד וחרזה לשורת הפקודה (במצב של הפעלה אינטראקטיבית של פיתון):

```
> n = 1.7456
```

```
> n = t + 5
```

```
> NameError: name 't' is not defined
```

# אופרטורים ארכיטמטיים

- ניתן לבצע פעולות ארכיטמטיות פשוטות בין משתנים מסוימים על ידי הפעלת האופרטור המתאים.
- האופרטורים המתמטיים המוגדרים בשפה מוצגים בטבלה:

| אופרטור | שימוש      | תיאור                                |
|---------|------------|--------------------------------------|
| +       | $y + x$    | חיבור בין שני משתנים                 |
| -       | $y - x$    | חיסור בין שני משתנים                 |
| *       | $y * x$    | הכפלת הערך של $x$ בערך של $y$        |
| **      | $y^{**} x$ | העלאת $x$ בחזקת $y$                  |
| /       | $y / x$    | מחלק את $x$ ב $y$ במדוקן             |
| %       | $y \% x$   | מודולו: השארית של חלוקת $x$ ב $-y$   |
| //      | $y // x$   | חלוקת בלמים (הפעולה המשלימה למודולו) |

# פולות ארכיטמטיות - דוגמאות

```
> n = 2 * 5 # n = 10 > y = m % n
> m = n ** 2 # n squared > print(y)
> print(m) 2
100
```

```
> print(m / n)
```

```
10
```

```
> m = 5
```

```
> n = 3
```

```
> x = m / n
```

```
> print(x)
```

```
1.6666667
```

למרות שהילקנו שני  
משתנים מטיפוס `int`,  
תוצאת החלוקת יצא  
מספר לא שלם מטיפוס  
`float`

חלוקת שלמים יש  
להשתמש באופרטור `//`

```
> x = m // n
```

```
> print(x)
```

```
1
```



# strings

- משתנים מטיפוס string (מחרחת) משמשים לשימרת טקסט.
- מחרחת היא אוסף סדר של תווים.
- בפייטון, מחרחות קבועות ניתנות לתייאור על ידי גרשיים כפולים או גרש בודד. אין הבדל, בחרו את הדרך הנוחה לכם.

```
> print("Hello World")
Hello World
```

```
> s = "Hello"
> print(s)
Hello
> type(s)
<class 'str'>
```

```
> s = 'Hello'
> print(s)
Hello
```

|    |    |    |    |    |
|----|----|----|----|----|
| H  | e  | I  | I  | o  |
| 0  | 1  | 2  | 3  | 4  |
| -5 | -4 | -3 | -2 | -1 |

נתן לגשת לתו  
ספציפי במחרחת  
באמצעות האינדקס  
המתאים

```
> print(s[0])
H
> print(s[4])
o
> print(s[-1])
o
> print(s[-3])
l
```

אינדקס -1  
תמיד ייתן את  
התו האחרון  
במחרחת

# עוד דוגמאות - strings

```
> print('Hello "World"! ')
Hello "World"!
```

```
> print("Hello \"World\"!")
Hello "World"!
```

```
> s = 'Hello'
> type(s[0])
<class 'str'>
```

```
> s1, s2 = 'Hello', "World"
> print(s1 + ' ' + s2)
Hello World
```

- אם המחרצת מכילה סוג מסוים של גרשים בתוכה, ניתן להשתמש בסוג השני כדי לתחום אותה:
- או להשתמש ב-escaping characters : \
- בפייתון אין טיפוס 'תו' ותו בודד מיוצג כמחרצת באורך 1.
- ניתן לשרשן מחרחות באמצעות האופרטור '+'.

# המשר - Strings

- מחרחות בפייתון אינן ניתנות לשינוי – **Immutable**.
- לא ניתן לשנות את התווים של מחרחות קיימת, רק **ליצור חדשה**.
- דוגמה:

```
> s = "Dog"
> s[0] = "d"
```



**TypeError: 'str' object does not support item assignment**

- כן ניתן לגרום למשתנה המתיחס למחרחת להתייחס למחרחת חדשה:

```
> s = "Dog"
> s = "Cat"
```



- לא ניתן לשנות את התוכן של ערכים שהם **Immutable** לאחר יצרתם.
- **פעולות על משתנים Immutable** מוחזירות **ערך חדש** ואינן מעדכנת את הערך **הנוכחי**.

# String Slicing

- ניתן לקבל תת מחרצת מתוך מחרצת באמצעות slicing :

```
> s = "Some String"
> s[start:end]
```

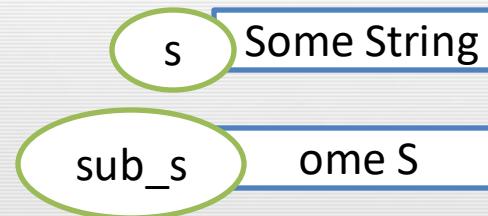
|   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|----|
| s | o | m | e |   | s | t | r | i | n | g  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

הפעולה תחזיר מחרצת  **חדשה** המבוססת על התווים של המחרצת s המקורי  
באופן הבא:

- מהתו המתחילה באינדקס start במחרצת s (כולל)
- עד התו באינדקס end במחרצת s (**לא כולל**)

דוגמא:

```
> sub_s = s[1:6]
> print(sub_s)
ome S
```



# String Slicing

```
> s = "Some String"
> s[start:end:step_size]
```

- ניתן גם לציין `step_size` שיציין גודל קפיצה. בברירת המחדל – הקפיצות בגודל 1.

```
> s[5:7]
St
> s[5:7:]
St
```

- אם לא מצוין `end` – אז תת-המחרצת הנוצרת תגיע עד התו האחרון של המחרצת המקורית:

```
> s[5:]
String
> s[5::2]
Srn
```

- אם לא מצוין `start` – אז תת-המחרצת הנוצרת מתחילה מהתו הראשון של המחרצת המקורית:

```
> s[:3]
Som
```

# פעולות נוספות על מחרוזות

- הפקציה **len** מחזירה את אורך המחרצת

```
> mystr = "Hello"
> len(mystr)
5
```

- המетодות **endswith** ו- **startswith** בודקות האם המחרצת מתחילה או מסתיימת  
במחרצת נתונה

```
> mystr = "Hello"
> mystr.startswith("He")
True
```

- המethode **strip** מוחקת מהמחרצת רווחים בהתחלה ובסוף

```
> mystr = " Hello "
> mystr.strip()
Hello
```

# פעולות נוספות על מחרוזות

- המethodות **islower**, **isupper** מחזירות האם מחרצת היא כולה אותיות גדולות או קטנות

```
> mystr = "hello"
> mystr.islower()
True
```

- המethodות **lower** ו-**upper** מנירות מחרצת להיות כולה באותיות גדולות או קטנות

```
> mystr = "Hello"
> mystr.upper()
HELLO
```

- המethodות **isdigit**, **isalpha** בודקות האם מחרצת היא כולה אותיות או כולה ספרות

```
> mystr = "021334"
> mystr.isdigit()
True
```

# פעולות נוספות על מחרוזות

- המתודה **find** מחזירה את האינדקס הראשון במחרוזת שהחל ממנו מופיעה המחרצת הנтונה

```
> mystr = "hello"
> mystr.find("lo")
3
```

- המתודה **replace** מקבלת שתי מחרוזות ומחליפה כל מופיע של המחרצת הראשונה במחרצת השנייה

```
> mystr = "Hello"
> mystr.replace("el", "ro")
Hrolo
```

- המתודה **isspace** בודקת אם מחרצת כולה תווי רווח (רווח, טאב, ירידת שורה...)

```
> mystr = " "
> mystr.isspace()
True
```

# הפקודה print

- הפקודה `print` מאפשרת מספר שימושים מתקדמים נוחים.
- למשל, ניתן להעביר לה מספר שירותים של ארגומנטים מופרדים בפסיק, והיא תדפיס אותם עם רווחים ביניהם:

```
> print(3, 5, 6)
3 5 6
```

- ניתן גם לבחור מפריד אחר באמצעות התחביר הבא (שיילמד לעומק בהמשך):

```
> print(3, 5, 6, sep=";")
3;5;6
```

- `print` יכולה לקבל מגוון רחב של טיפוסים:

```
> print(3, 5.4, "Cat")
3 5.1 Cat
```

# f-Strings

- בפייתון ניתן לבצע formatting של מחרוזת, כולל השטלת ערכים בתוך מחרוזת, באמצעות התחביר הבא:

```
> x = 5
> s = f"My number is {x} and twice my number is {2*x}"
> print(s)
My number is 5 and twice my number is 10
```

- ניתן להציג גם את הפורמט בו יודפסו הערכים בסוגרים:

```
> pi = 3.14159265358979323846264338327950288419716939937510582097494...
> s = f"Pi equals {pi:.2f}, or more accurately {pi:.6f}"
> print(s)
'Pi equals 3.14, or more accurately 3.141593'
```

# רשימות - lists

- ❖ אתחול רשימה
- ❖ גישה לאים
- ❖ הוספת איברים ושרשור רשימות
- ❖ מחיקת איברים
- ❖ פעולות נוספות על רשימות

# lists

- **list** היא מבנה נתונים מסווג רשיימה אשר מסוגל להכיל איברים לפי סדר, מכל טיפוס
- בשביל ליצור רשיימה עם איברים  **уникальнים**, יש להקיפם בסוגרים מרובעים [ ]
- **list** יותר דומה ל-**vector** מ-**C++** מאשר לרשימה מקוشرת

```
> new_list = [1, 2, 3, 4, 5]
> print(new_list)
[1,2,3,4,5]
```

```
> empty_list = []
> print(empty_list)
[]
```

```
> zeros = [0]*5
> print(zeros)
[0, 0, 0, 0, 0]
```

```
> mixed_types = [7, 'dog', [7], 3.14]
> print(mixed_types)
[7, 'dog', [7], 3.14]
```

# indexing

- גישה לאיברים וביצוע slicing מתבצע באותו אופן כמו עם מחרוזות.

```
my_list = ["Baraa", "Daniel", "Ron", "Regev"]
```

```
> my_list[0]
```

"**Baraa**"

```
> my_list[3]
```

"**Regev**"

```
> my_list[4]
```

**Traceback (most recent call last):**  
File "<stdin>", line 1, in <module>  
**IndexError: list index out of range**

```
> my_list[-1]
```

"**Regev**"

# Slicing

0 1 2 3 4 5 6 7 8 9

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
> my_list[1:5] # Slicing
['b', 'c', 'd', 'e']
```

# Slicing does NOT change the original list

```
> my_list
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

# Slicing using an arithmetic progression

```
>>> my_list[0:9:2]
['a', 'c', 'e', 'g', 'i']
 start stop step
```

# We can omit any of these

```
>>> my_list[0::2]
['a', 'c', 'e', 'g', 'i']
```

```
>>> my_list[1::2]
['b', 'd', 'f', 'h', 'j']
```

# הוספת איברים

- ניתן להוסיף איברים לרשימה במספר דרכים
- באמצעות המתודה **append(value)** ניתן להוסיף איבר לסוף הרשימה
- באמצעות המתודה **insert(pos, value)** ניתן להוסיף איבר למיקום ספציפי

```
new_list = [1, 2, 3, 4, 5]
print(new_list)
```

```
> [1,2,3,4,5]
```

```
new_list.append(8)
print(new_list)
```

```
> [1,2,3,4,5,8]
```

```
new_list = [1, 2, 3, 4, 5]
print(new_list)
```

```
> [1,2,3,4,5]
```

```
new_list.insert(1, 'one')
print(new_list)
```

```
> [1,'one',2,3,4,5]
```

```
new_list[1] = 10
```

```
> [1,10,2,3,4,5]
```

# הוספת איברים

- באמצעות הפעולה `extend(other_list)` ניתן לשרר רשימה אחרת לסוף הרשימה.
- ניתן לשרר רשימות גם באמצעות אופרטור `+`

```
new_list = [1, 2, 3, 4, 5]
new_list.extend([23, 22])
[1,2,3,4,5,23,22]
```

```
new_list.append([55, 56])
[1,2,3,4,5,23,22, [55, 56]]
```

```
new_list = [1,2,3] + [4,5]
new_list += [23, 22]
[1,2,3,4,5,23,22]
```

# שינוי ערכים קיימים

- שינוי ערכים בראשימה מתרבע באמצעות גישה לאיברים עם האינדקס המתאים וכתיבה מחדש לתוכם (בניגוד למחרוזות, רשימות הן `mutable`):

```
new_list = [1, 2, 3, 4, 5]

new_list[3] = 77

[1, 2, 3, 77, 5]
```

# הסרת איברים מרשימה

- ניתן להסיר איברים מרשימה במספר דרכים
- באמצעות המethode **remove(value)** ניתן למחוק המופיע הראשון של האיבר **value** ברשימה.
- באמצעות הפקודה **[del] my\_list[index]** ניתן למחוק מהרשימה **my\_list** את האיבר במקום ה-**index**.
- המתודה **(pop(index))** מוציאה את האיבר במקום ה-**index** ומחזירה אותו.

```
new_list = [1, 2, 3, 4, 2, 5]
new_list.remove(2)

[1, 3, 4, 2, 5]

del new_list[2]

[1, 3, 2, 5]
x = new_list.pop(1)

[1, 2, 5]
```

```
print(3)
2

list_ch = ['a', 't', 'l', 'm']
s = list_ch.pop()

['a', 't', 'l']

print(s)
'm'
```

# העתקה של רשימות - copy

- שימוש לב: ערכים שאינם מטיפוסי בסיס (טיפוסים מספריים) מאוחסנים ומועברים reference by בפייתון (כלומר, הם מתנהגים כמצביעים). בפרט, שינוי של הערך דרך משתנה אחד יגרום לו לשינויו גם בכל המשתנים האחרים שמצביעים אותו הערך.
- לדוגמה, מה יהיה הפלט של קטע הקוד הבא?

```
> a = [1, 2, 3]
> b = a
> b[1] = 5
> print(a)
[1, 5, 3]
```

לחלופין, ניתן ליצור עותק בצורה מפורשת:

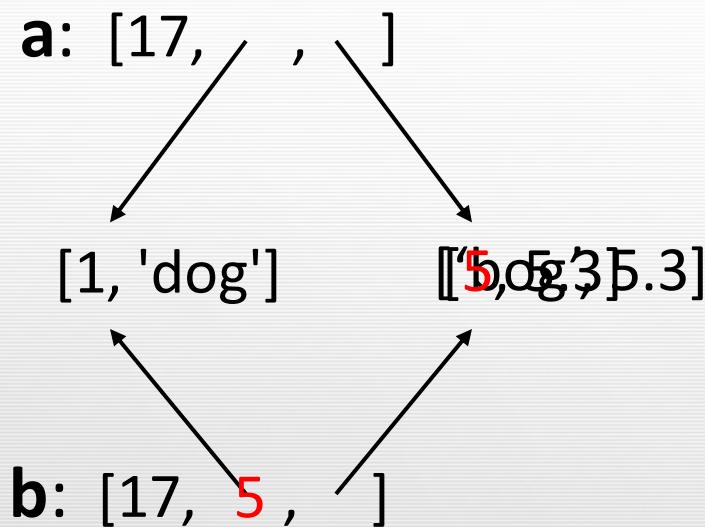
```
> b = a.copy()
> b[1] = 5
> print(a, b)
[1, 2, 3] [1, 5, 3]
```

# העתקה של רשימות - deepcopy

- מה קורה בرمת כינון יותר عمוקה (רשימה של רשימות)?

```
> a = [17, [1, 'dog'], ['bog', 5.3]]
> b = a.copy()
> b[1] = 5
> b[2][0] = 5
```

```
> print(b)
[17, 5, [5, 5.3]]
> print(a)
[17, [1, 'dog'], [5, 5.3]]
```



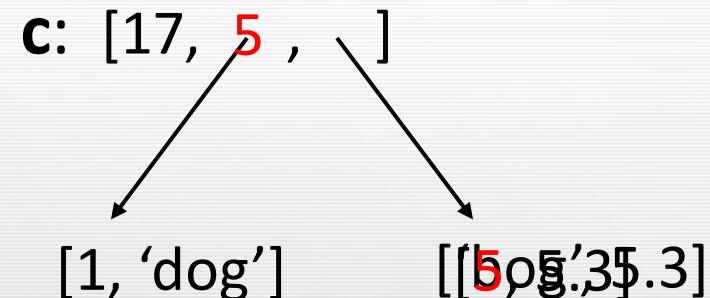
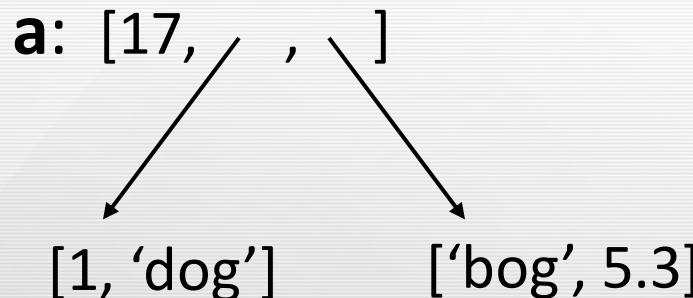
# העתקה של רשימות - deepcopy

- להעתקה ברמת כינון יותר عمוקה (רשימה של רשימות) יש להשתמש ב-deepcopy:

```
> from copy import deepcopy
> c = deepcopy(a)
> c[1] = 5
> c[2][0] = 5
```

```
> print(c)
[17, 5, [5, 5.3]]
> print(a)
[[1, 'dog'], ['bog', 5.3]]
```

מתווך קובץ הספרייה הסטנדרטית  
deepcopy ייבא את הפונק' copy  
שמוגדרת בו



# שני סוגי השוואות

- בפייתון מוגדרים שני סוגי אופרטורים להשוואה:

1. אופרטור `is` שבודק זהות עצמים, האם מדובר באותו מקום בזיכרון  ממש (identity)
2. אופרטור `==` שבודק שיוויון, בהתאם לティפוס (equality)

```
> a = [1, 2, 3]
```

```
> b = a
```

```
> c = a.copy()
```

```
> a is b
```

```
True
```

```
> print(a is c, a == c)
```

```
False True
```

```
> a = 3
> b = 3
> c = a
> print(a is b, a is c)
??
```

תליי מימוש / אופטימיזציות  
(תקף גם למחזרות)

```
> a = 3
> b = a
> a = 4
> print(a is b, b)
False 3
```

מובטח!

- ומה לגבי טיפוסים בסיסיים?

# פעולות נוספות על רשימה

- הפקציה `len` ממחירה את אורך הרשימה

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> len(my_list)
7
```

- הפקציות `max` ו-`min` מוחירות את האיבר המקסימלי והמינימלי ברשימה

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> max(my_list)
3
```

- המתודה `list.count(value)` ממחירה את מספר המופיעים של ערך מסוים ברשימה

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> my_list.count(1)
3
```

# פעולות נוספות על רשימה

- הmethod `(list.index(value))` מחזירה את האינדקס **הנמור ביותר** בו איבר נמצא ברשימה, או זורקת שגיאה אם האיבר לא נמצא ברשימה.

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> my_list.index(1)
0
```

- הmethod `(list.sort())` ממיינת את הרשימה בסדר עולה (משנה את הרשימה).

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> my_list.sort()
> print(my_list)
[-1, 0, 1, 1, 1, 2, 3]
```

- ה פעולה `sorted(list)` מחזירה עותק ממויין של הרשימה.

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> sorted(my_list)
[-1, 0, 1, 1, 1, 2, 3]
```

# פעולות נוספות על רשימה

- המетодה `list.reverse()` ההפוך את סדר האיברים ברשימה (משנה את הרשימה).

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> my_list.reverse()
[1, 1, -1, 0, 2, 3, 1]
```

- בדינה ל-`sort`, `sorted` קיימת הפעולה `reversed` אשר מחזיר עותק הפוך של הרשימה.
- בדיקה האם איבר קיים ברשימה באמצעות אופרטור `in`:

```
> my_list = [1, 3, 2, 0, -1, 1, 1]
> 4 in my_list
False
```

- האופרטור `in` עובד גם לסוגי רצפים ומבנה נתונים אחרים, למשל מחרוזות.

# tuples - א-יות

- tuple היא רשימה שהיא `immutable`. כלומר, לא ניתן לשנותה לאחר שנוצרה (אבל אפשר להעתיק)
- בשביל ליצור tuple עם איברים ספציפיים, יש להקיפם בסוגרים עגולים ()

```
> new_tuple = (1, 2, 3, 4, 5)
> print(new_tuple)
(1,2,3,4,5)
```

```
> new_tuple[2] = 0
Type Error: 'tuple' object does not
support item assignment
```

```
> empty_tuple = ()
> print(empty_tuple)
()
```

```
> zeros = (0,)*5
> print(zeros)
(0, 0, 0, 0, 0)
```

```
> mixed = (7, 'dog', [7], 3.14)
> mytuple = (5, 3) + mixed
> print(mytuple)
(5,3,7,'dog',[7],3.14)
```

מקרה מיוחד: עבור tuple עם איבר אחד מצינים עם פסיק שהסוגרים תוחמים tuple והם אינם סוגרים של חישוב מתמטי

פעולה חיבור בין שני tuples ממחזירה tuple חדש עם שרשרת התכנים שלהם

# Tuple מול List

- לאחר ש-Tuple מיצג רצף שהוא `Immutable`, במקרים בהם נדרש לשנות את הרצף לאחר ייצורו, נעדיף להשתמש ב-`List`.
- מתי בכלל זאת משתמש ב-Tuple?
  - אם נרצה לוודא שה תוכן של הרצף נשאר קבוע ואיןו משתנה בטעות.
  - לעיתים קרובות נרצה להעביר כמה ערכים יחד למקום מסוים ללא כוונה לשנותם או להוסיף/להוריד מהרצף.
  - במקרה זהה, שימוש ב-Tuple אוכף תכוונה זו ומקל על הקריאה של הקוד – מדגיש שהרצף הוא `Immutable`.
  - לאחר שה תוכן של Tuple הוא קבוע, הוא מעט יותר יעיל בזכרון ובזמן ריצה.
  - כפי שנלמד בהמשך, ניתן לאחסן Tuple במבנה הנתונים `Set` המממש קבוצה בפייתון (כמו גם במבנה נתונים אחרים), בעוד שלא ניתן לאחסן `List` ב-`Set`.זה מכיוון שאם נכנס שניים שונים ואז נשנה אחד מהם להיות שווה לאחר, אנחנו נשבר את הקבוצה (שאמורה להכיל רק עותק אחד מכל ערך)

# רצפים - Sequences

- מחרוזות, רשימות ו-ח-יות אלו טיפוסי רצפים מובנים בפייתון
- על כולם מוגדרותפעולות `len`, `slicing` ו-`index` גם שרשור, `xrange` ו-`count`
- מה ההבדלים ביניהם?
- בהמשך נראה כיצד סינטקס לולאות `for` מבוסס על רצפים

# אופרטור הפיזור \*

- קיימ בפייתון האופרטור \* (כוכבית) הפועל על טיפוסים המייצגים רצף.
- האופרטור הופך את איברי הרצף לסדרת ערכים עם פסיקים ביניהם, שנייתן להשתמש בה בכל הקשור שבו פייתון מצפה לקבל סדרת ערכים עם פסיקים ביניהם. למשל:

```
> mixed = (5, 'dog', [7], 3.14)
> print(mixed)
(5, 'dog', [7], 3.14)

> mytuple = (3, mixed)
> print(mytuple)
(3, (5, 'dog', [7], 3.14))

> mytuple = (3, *mixed) # like (3, 5, 'dog', [7], 3.14)
> print(mytuple)
(3, 5, 'dog', [7], 3.14)

> mytuple = (3, *mixed) # like print(3, 5, 'dog', [7], 3.14)
> print(*mytuple)
3 5 dog [7] 3.14
```

# Unpacking

- כאשר ישנו רצף עם *n* איברים באגף ימין של השמה, פיתון מאפשר לבצע השמה של הרצף ל-*n* משתנים שכל אחד מהם יכול לקבל ערך אחד מהרצף. פעולה זו נקראת *unpacking* – פירוק של הרצף:

```
> my_tuple = (1, 2)
> a, b = my_tuple
> print(a, b)
1 2
```

- אם אנחנו מועוניינים לדלג על ערך, הקונבנצייה היא לשים אותו ב-*\_*, כה:

```
> my_tuple = (1, 2, 3)
> a, _, b = my_tuple
> print(a, b)
1 3
```

# מבני בקרה

if statements ❖

if-else ❖

while loop ❖

# משפטים if ב-**python**

```
if <condition>:
<TAB>statement1
<TAB>statement2
...
...
```

אפשר גם 4 רווחים (מקובל),  
חייב להיות **אחד** בקובץ הקוד

- בפייתון משפטי תנאים מתבצעים באופן הבא:
- ה-**condition** הוא כל ביטוי אשר מחזיר ערך בוליאני – לוגי מטיפוס **bool**
- ההזחה מגדירה את הסקופ של בלוק ה-**if**
- בנגד לשפת C, בפייתון אין סגירה ופתיחה מפורשימים של בלוקים (אין סגירה ופתיחה עם סוגרים מסולסלים)

**Indentation!**

# משפטי if ב-python

- דוגמאות למשפטי if:

```
if 32 % 8 == 0: # the remainder of 32 divided by 8
 print ("32 is divisible by 8")
 print ("32 is also divisible by 2 and 4")
```

```
32 is divisible by 8
32 is also divisible by 2 and 4
```

```
x = 9
y = 7
if x > y:
 print("x is greater than y")
 x -= 1
print(x)
```

```
x is greater than y
8
```

# if-else

- ניתן לכתוב ביטויי if-else בפייתון:

```
if x > y:
 print("x is greater than y")
else:
 print("y is greater than or equal to x")
```

- במידה ויש יותר מ-2 אפשרויות ניתן לרשום ביטויי if-elif-else

```
if x > y:
 print("x is greater than y")
elif x == y:
 print("x is equal to y")
else:
 print("y is greater than x")
```

- מותר גם יותר מ-elif אחד.
- ה-else הסופי הוא אופציוני.
- קיים גם אופרטור טרנארי:

```
print("x is greater than y") if x > y else print("y is greater than or equal to x")
```

# אופרטורים לוגיים וקבועי Boolean

- באמצעות האופרטורים **not**, **and**, **or** ניתן לשלב ביטויי תנאי מורכבים:

```
x = 3
y = -3
if x >= 2 and y <= -2:
 print("True")
if x > 3 or y < -3:
 print("True")
if not x == y: # also: x != y
 print("True")

if x > y or not (x >= 3 and y <= -3):
 print("True")
```

```
x = True
y = False
x and y
> False

type(x)
<class 'bool'>
```

- סדר הקידימות של האופרטורים:

not .1  
and .2  
or .3

- יצוג משתני Boolean בפייתון:
  - "אמת" מיוצג על ידי `True`
  - "שקר" מיוצג על ידי `False`
  - שימוש לבאות הראשונה הגדולה

# לולאות while

- בפייתון משפטי **while** מתבצעים באופן הבא:

```
while <expression>:
 statement1
 statement2
 ...
```

- ה-<expression> הוא כל ביטוי אשר מחזיר ערך Boolean
- ההזאה מגדירה את הסעיף של בלוק ה-<while>
- ניתן להשתמש בפקודות **break** ו- **continue** בתחום לולאות while
- כמו בשפת C

- דוגמה:

```
count = 0
while count < 3:
 print("count:", count)
 count += 1
print("Done")
```

```
count: 0
count: 1
count: 2
Done
```

# תרגול מס' 12

- ❖ הרצת קבצי פיתון
- ❖ לולאות for
- ❖ ערכי ברירת מחדל
- ❖ מיליוןים
- ❖ ופרמטרים שמיים
- ❖ פונקציות

# לולאות for

# לולאת **for**

- ניתן להשתמש בלולאת **for** בפייתון בכמה דרכים.
- בשביל לעבור על איברי רשימה או כל מבנה נתונים אחר (נקרא טיפוס **iterable**)  
משתמשים בתבנית הבאה:

```
for element in iterable:
 <TAB>statement1
 <TAB>statement2
```

- ההזחה (indentation) מגדירה את ה-scope של הלולאה:

```
> elements = [3, 'a', 5.5]
> for element in elements:
> print(element)
> print("Done")
3
a
5.5
Done
```

# Range

- `range` היא פונקציה מובנית בפייתון (ומאוד שימושית).
- היא ממחישה אובייקט `iterable`, המיצג את הטווח המבוקש של איברים מטיפוס `int`.

```
list(range(6,10))
> [6,7,8,9]
```

ניתן להמיר את תוצאת  
רשימה `range`

```
list(range(5)) # same as range(0,5)
> [0,1,2,3,4]
```

start=0  
ומקבול להמשתו.

```
list(range(4,2))
> []
```

```
list(range(10, 0, -2))
range(start, stop, step)
> [10,8,6,4,2]
```

ערך ברירת המחדל עבור פרמטר  
step הוא 1

```
type(range(10))
> <class 'range'>
```

```
list(range(10))
> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# לולאת **for** עם **range**

- בשביל לייצר לולאה פשוטה עם אינדקס *i*, משתמשים בערך ההחזרה של `:range`

```
elements = [3, 'a', 5.5]
for i in range(3):
 print("iteration:", i)
 print("element:", elements[i])
print("Done")
> iteration: 0
> element: 3
> iteration: 1
> element: a
> iteration: 2
> element: 5.5
> Done
```

- הערה: פקודות `break`, `continue` עובדות כמו זהה (גם בלולאות `while`)
  - אם אין צורך במשתנה האינדקס, ניתן להמיר את שמו בקו תחתון `_`

# לולאת for עם enumerate

- באמצעות הפונקציה המובנית `enumerate` ניתן לבצע לולאה על אובייקט `iterable` כלשהו, ובכל איטרציה לקבל גם את האלמנט וגם את האינדקס הנוכחי (מספר האיטרציה):

```
elements = [3, 'a', 5.5]
for i, e in enumerate(elements):
 print("iteration:", i)
 print("element:", e)
print("Done")
> iteration: 0
> element: 3
> iteration: 1
> element: a
> iteration: 2
> element: 5.5
> Done
```

המייצגים זוגות אינדקס + איבר  
tuple של iterable ממחזירה enumerate

# פקודת zip

- למעבר על מספר רשימות (או אובייקטים iterable) בו-זמנית נשתמש בפקודה zip:

גם zip מחזירה אובייקט iterable

```
> l1, l2 = ['C', 'for'], ['is', 'Cookie']
> for e1, e2 in zip(l1, l2):
> print(e1, e2)
C is
for Cookie
```

- למעשה, הפקודה מייצרת רשימה של tuples, ומבצע מעבר עליו, בשני משתנים:

```
> l1, l2 = ['C', 'for'], ['is', 'Cookie']
> print(zip(l1, l2))
[('C', 'is'), ('for', 'Cookie')]
```

- שאלת: ממשו את הפקודה enumerate באמצעות range ו zip ו enumerate

```
> list(zip(range(len(l1)), l1))
[(0, 'C'), (1, 'for')]
```

# לולאת for

- מעבר על איברי מחרוזת:

```
> str_mtmm = 'mtm'
> for letter in str_mtmm:
> print(letter)
m
t
m
```

- ספרת מספר המופיעים של האות s במחרוזת:

```
> seq = 'supercalifragilisticexpialidocious'
> count = 0
> for letter in seq:
> if letter == 's':
> count += 1 # count = count + 1
> print(count)
3
```

השוואת מחרוזות

seq.count('s')

# List Comprehensions

- מנגןן ליצור רשימה חדשה על ידי הפעלת חישוב על כל איבר ברשימה קיימת
  - מנגןן זה מורכב מחלקים הבאים:
    - קלט: רשימה (או אופן כללי כל אובייקט iterable)
    - שם משתנה: המשתנה שיקבל את האיברים זה אחר זה
    - ביטוי תנאי (אופציונלי): קובע אילו איברים מהרשימה הנוכחית ייכללו
    - ביטוי פלט: החישוב שיופעל על כל איבר מרשימה הקלט
  - דוגמה: יצירת רשימה חדשה ע"י העלאה בריבוע של כל מספר שלם ברשימה נתונה
- הערה: ניתן להמיר את הסוגרים המרובעים [] בעגולים () לקבלת ח-יה.

```
[elem**2 for elem in iterable if type(elem) is int]
```

ביטוי תנאי (אופציונלי)

רשימת הקלט

ביטוי פלט

# List Comprehensions

דרך יותר נכונה:  
if isinstance(elem, int)

```
a_list = [1, 4, 9, 'a', 'b', 0, 4]
squared_ints = [elem**2 for elem in a_list if type(elem) is int]
> [1, 16, 81, 0, 16]
```

- מתרצע תהליך איטרטיבי בו בכל איטרציה המשתנה `elem` מקבל ערך אחר מתוך הרשימה `a_list`
- ביטוי ההשוואה בודק האם `elem` הוא מטיפוס `int`
- אם האיבר הוא מטיפוס `int` אז הוא מועלה בריבוע ונכנס לרשימת הפלט.
- ניתן להשミニ את ביטוי התנאי:

```
x = 1
squared_plus_x = [elem + x for elem in squared_ints]
> [2, 17, 82, 1, 17]
```

# לולאות for – סיכום

- לולאת for בפייתון משמשת לאיטרציה על כל רצף iterable
- הfonק' המובנית range מייצרת סדרה של מספרים שלמים
- ניתן להשתמש ב-for בתוך comprehension לקבלת רשימה על בסיס אובייקט iterable קיימ .tuple comprehension – קיימ גם

# מילוניים

# dict

- dict הוא מבנה נתונים מסוג מילון אשר ממפה בין מפתחות (keys) לבין ערכים (values).
- סדר שמירת הזוגות הסדריים במילון (מפתח, ערך) אינו מוגדר.
- בשביל לאותל מילון עם תוכן מסוים, יש להשתמש בסוגרים מסולסלים {}.

```
> new_dict = {
> 'a': 1, 'b': 2, 'c': [3]
> }
> print(new_dict)
{'a': 1, 'b': 2, 'c': [3]}
```

```
> empty_dict = {}
> print(empty_dict)
{}
```

מותר לשבר שורה בתוך סוגרים.

```
> d = {[3]: 4}
TypeError: unhashable type: 'list'
```

תנאי הכרחי: מפתחות חייבים להיות בלתי-  
 לשנות (immutable)

```
> d = {(3): 4}
```

Tuple הוא טיפוס immutable ולכן  
הקוד זהה תקין. לא תזדקק חריגה.

# פעולות על מילון

```
> d = {'one': 1, 'two': 2, 'three': 3}
```

- גישה לאיבר:

```
> d['two']
2
```

- השמה לאיבר:

```
> d['two'] = [2]
> print(d)
{'one': 1, 'two': [2], 'three': 3}
```

- הוספה איבר:

```
> d['four'] = 'quattro'
{'one': 1, 'four': 'quattro', 'two': [2], 'three': 3}
```

- הסרת איבר:

```
> del d['two']
{'one': 1, 'four': 'quattro', 'three': 3}
```

- בדיקה האם מפתח מסוים:

```
> 'four' in d
True
```

# על מילון for

- הפקציות `keys()`, `values()`, `items()` של מפתחות, ערכים, וחוגות (מפתח, ערך) בהתאם.
- ניתן להשתמש ב-`for`:

```
> d = {'C': 'cookie', 'B': 'Big bite', 'A': 'Apple'}
```

```
> for k in d.keys(): # also "for k in d:"
> print(k)
C
B
A
```

```
> for v in d.values():
> print(v)
cookie
Big bite
Apple
```

```
> for k, v in d.items():
> print(k + ' is for ' + v)
C is for cookie
B is for Big bite
A is for Apple
```

כל איבר ב-`d.items()` הוא זוג (`tuple`) של מפתח וערך

- איטרציה על המילון עצמו שקופה לאיתרציה על `d.keys()`.
- סדר הדפסה / החזרה של האיברים עלול להשתנות לאחר כל שינוי במילון!

# dictionary comprehensions

- מנגנון ליצור מילון על ידי הפעלת חישוב על כל איבר בסדרת איברים קיימת.
- דוגמה מאוד ל-`list comprehensions`.
- מנגנון זה מורכב מהחלקים הבאים:
  - קלט: איברי מילון/רשימה קיימת (או אופן כללי אובייקט iterable)
  - שם משתנה: המשתנה (או המשתנים) שיקבלו את האיברים בזיה אחר זה
  - ביטוי תנאי (אופציונלי): קובע אילו איברים מהקלט הנוכחי ייכללו
  - ביטוי פלט מהצורה `key : value`: שמתאר את המפתח והערך שלו (יכולים להיות תוכאת חישוב)
- דוגמה: יצירת מילון חדש ע"י העלאה בריבוע של כל מספר שלם ברשימה נתונה:

```
{elem: elem**2 for elem in a_list if type(elem) is int}
```

ביטוי פלט

שם משתנה

רשימת  
קלט

ביטוי תנאי (אופציונלי)

# dictionary comprehensions

- ניתן להשתמש ב-dictionary comprehension גם עם מילון קלט, למשל:

```
a_dict = {'a': 'Alfa', 'b': 'Bravo', 'c': 'Charlie', 'd': 'Delta'}
```

```
result = {letter: code.lower() for (letter, code) in a_dict.items()}
```

```
> {'a': 'alfa', 'b': 'bravo', 'c': 'charlie', 'd': 'delta'}
```

הסוגרים אופציונליים

- מתרבע תהליך איטרטיבי בו בכל איטרציה, משתני הלולאה key, val מקבלים ערכים אחרים מתוך המילון (זוג של מפתח + ערך תואמים)
- מילון הפלט מתקיים על ידי המפתח המקורי והערך val.lower() ששממופה אליו
- דוגמה נוספת:

```
result = {'-' + letter + '-': code for (letter, code) in a_dict.items()
 if len(code) >= 5}
```

```
> {'-b-': 'Bravo', '-c-': 'Charlie', '-d-': 'Delta'}
```

# פונקציות

# פונקציות בפייתון

- בפייתון מגדירים פונקציה באמצעות התבנית הבאה:

```
def function_name(arg1, arg2, ...):
 statements
```

- לא מגדירים את טיפוס ההחזרה של הפונקציה.
- לא מגדירים את טיפוסו של כל ארגומנט.
- לא רושמים הכרחה של הפונקציה, אלא רק הגדרה ומיושן.
- ניתן להשתמש בפונקציה החל מהשורה שמתוחת למימושה.
- הפקודה `pass` מציננת פקודה ריקה (שימושי לניפוי שגיאות)
- ערך החזרה של פונקציה שלא מכילה פקודת `return` הוא הקבוע `None`
- בניגוד ל-`++C`, בפייתון אין העמסת פונקציות.

# דוגמאות

- פונקציה למציאת סכום שני איברים:

```
def sumTwo(a, b):
 return a + b

sumTwo(1,5)
> 6
sumTwo('Hello_ ', 'World')
> 'Hello_World'
```

הfonקציה sumTwo עובדת  
לכל טיפוס שМОדרת  
עבורה פועלת +

- פונקציה למציאת הערך הממוצע ברשימה:

```
def avg(elements):
 sum = 0
 for elem in elements:
 sum += elem
 return sum / len(elements)
```

# העברה פונקציה כפרמטר

- למעשה, פונקציה בפייתון היא "עצמה", שמודרת עבורה הפעולה סוגריים.

- לדוגמה, עבור הפונקציה `(b,a)sumTwo` מהשאוף הקודם:

```
> type(sumTwo)
<class 'function'>
```

- תרגיל: כתבו פונקציה המקבלת פונקציה ורץ. הפונקציה תחזיר רשימה חדשה, שאיבריה הם איברי רשימה הקלט לאחר הפעלת הפונקציה הנottonה עליהם.

פתרון:

```
def map(seq, f):
 return [f(el) for el in seq]
```

```
> def square(x):
 return x**2
> map(square, [1, 2, 3])
[1, 4, 9]
```

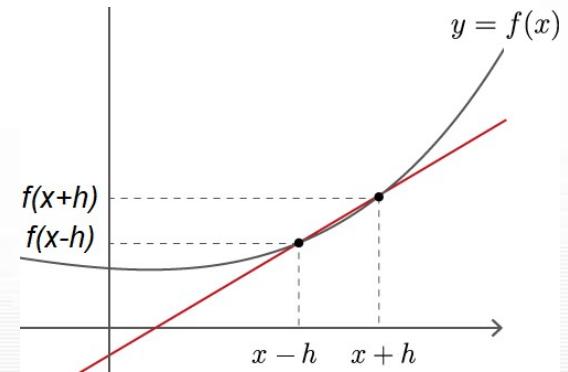
דוגמת שימוש:

- הערה: פונקציה `map` צו אcn קיימת ב-`python`.

# דוגמה: חישוב נגזרת (נומרית)

- נכטו פונק' שמקבלת פונק' מתמטית (במשתנה אחד), נקודה  $x$  ואפסילון קטן, ומחזירה נגזרת מקרבת של הפונקציה בנקודת.

```
def derivative(f, x, epsilon):
 assert epsilon > 0
 dy = f(x + epsilon) - f(x - epsilon)
 dx = 2 * epsilon
 return dy / dx
```



```
from math import sin, cos, pi
```

```
derivative(sin, pi/4, 0.001)
0.707106663335455
```

```
cos(pi/4)
0.7071067811865475
```

- נבדוק:

# ייבוא והרצת קבצי פיתון, import

# ייבוא והרצה קבצי פיתוח

## myfile1.py:

```
def getlanguageName():
 return 'Python'
print('Hello ' + getlanguageName() + '!')
```

```
$ python3 myfile1.py
Hello Python!
```

- תזכורת

## myfile2.py:

```
import myfile1
print(myfile1.getlanguageName() + ' is
cool!')
```

```
$ python3 myfile2.py
Hello Python!
Python is cool!
```

## myfile3.py:

```
from myfile1 import getlanguageName
print(getlanguageName() + ' is cool!')
```

```
$ python3 myfile3.py
Hello Python!
Python is cool!
```

- כל הגדרה שמופיעה בקובץ נשמרת תחת namespace בשם הקובץ
- אפשר לייבא רק חלקים ספציפיים ממודול מסוים, לדוגמה:

# הרצה בתסריט

- יש שתי דרכים עיקריות להשתמש בקובץ פיתון: אפשר להריץ אותו ישירות (**בתור תסריט**) ואפשר לבצע לו `import`.
- יתכן שנרצה להשתמש בקובץ הקובץ לשתי המטרות: למשל, לכלול באותו הקובץ פונקציות שאנו משתמשים בהם, וטסטים לאותן הפונקציות.
  - כדי להריץ את הטסטים, נריץ את הקובץ בתסריט.
  - כשהנעsha לקובץ `import`, ניבא את הפונקציות ללא הרצת הטסטים.
- אם נרצה להריץ פקודות מסוימות רק כאשר אנחנו מרים את הקובץ ישירות (לא דרך `import`), נטוף את הקוד בבדיקה הבאה:

```
if __name__ == "__main__":
 <code>
```

# ייבוא והרצה קבצי פיתוח

- ניתן להוסיף פקודות שיתבצעו כשהקובץ מורץ ישירות משורה הפקודה, אבל לא כשהוא נכלל כ-`:import`:

```
def getlanguageName():
 return 'Python'

if __name__ == "__main__":
 print('Hello ' + getlanguageName()+' !')
```

- הפקודות תחת התנאי יבוצעו בהפעלה ישירה של הקובץ, אבל לא בזמן `import`

## myfile2.py:

```
import myfile1
print(myfile1.getlanguageName()
+ ' is cool!')
```

```
> python3 myfile1.py
Hello Python!

> python3 myfile2.py
Python is cool!
```

# הרצה קבצי פיתון - סיכום

- בהרצה ובייבוא (import) של קובץ פיתון שורות הקובץ מתרבעות אחת אחרי  
השנייה
- ניתן להשתמש בתנאי "`if __name__ == "main__"` לעיטוף פקודות שאינן  
לביצוע בעת import

# Named & Unnamed Arguments

# פרמטרים ברירת-מחדר

- כמו ב-`C++`, ניתן להגדיר בפיזטון פרמטרים ברירת מחדר ( `default` ) `arguments` (פונקציות). הת לחבר וההתנהגות דומות לאלה שב-`C++`.
- אפשר להגדיר את ערכי `default` רק לפרמטרים האחרונים של הפונקציה, כמו ב-`C++`.

```
def student(name, grade = 100):
 print('The student', name, 'got a grade of', grade)

student('Alice') # The student Alice got a grade of 100
student('Bob', 90) # The student Bob got a grade of 90.
```

# פרמטרים עם שם

- בפייתון, ניתן להעביר ארגומנטים לפונקציה בשתי דרכים:
  - בדרך המיקומית (positional), כאשר מציבים את הפרמטרים לפי הסדר בחתימת הפונקציה.
  - בדרך השמית (named), אשר כל פרמטר ניתן בתוספת השם שלו.
  - הדרך השנית שימושית במיוחד במקרים בהם יש פרמטרים רבים ורבים מהם אופציונליים.

```
def student(name, grade = 100):
 print('The student', name, 'got a grade of', grade)

student('Alice') # Positional
student('Bob', 90) # Positional
student(name = 'Alice') # Named
student(name = 'Bob', grade = 90) # Named
student(grade = 90 , name = 'Bob')
```

כasher הפרמטרים  
מצוויים לפי שמותם, אין  
צורך לשמור על הסדר  
המקורית שלהם לפי  
חתימת הפונקציה

# פרמטרים עם שם

- יתרונות של השיטה השמית להעברת פרמטרים:
- שמות הפרמטרים מופיעים ליד הערכים המתאימים, מה שתורם לקריאות ומונע שגיאות.
- למשל, האם פונקציה המעתיקה בין שני קבצים מקבלת קודם את המקור או את היעד?).
- הפרמטרים לא חייבים להופיע לפי הסדר שלהם בחתימת הפונקציה.
- ניתן לשלב בין שתי השיטות אבל חייבים לשים קודם כל הפרמטרים המיקומיים על פי סדרם בחתימת הפונקציה, ורק לאחר מכן את הפרמטרים השמיים.:

```
def student(name, grade = 100):
 print('The student', name, 'got a grade of', grade)

student('Bob', grade = 90) # Named
```

# פרמטרים שימושיים

- מה הקריאה הבאה עושה?

```
def applyFactor(grades, additive_factor, multiply_factor,
use_factor_shoresh = False):
 if use_factor_shoresh :
 grades = [sqrt(grade)*10 for grade in grades]
 grades = [grade* multiply_factor + additive_factor
 for grade in grades]
 return [round(grade) for grade in grades]
```

```
applyFactor(grades,5,1.1, True)
```

יותר ברור עכשו?

```
applyFactor(grades, additive_factor = 5, multiply_factor = 1.1,
use_factor_shoresh = True)
```

# פונקציות עם מספר פרמטרים משתנה

- ניתן להגדיר פונקציה שמקבלת מספר לא ידוע מראש של פרמטרים מיקומיים (ללא שם). אופציונלית, ניתן גם לקבל לאחריהם פרמטרים שימושיים.
- כל הפרמטרים המיקומיים יתקבלו בפונקציה כרשימה (list) אחת, שאורכה כמספר הפרמטרים המיקומיים שהועברו.

```
def print_numbers(*numbers, separator=', '):
 for num in numbers[:-1]:
 print(num, end = '')
 print(separator, end = '')
 print(numbers[-1])
```

הפרמטר השמי `end` מגדיר את התו/מחרוזת שיעודפס אחרי תוכן `print`. (ברירת המחדל היא ירידת שורה)

```
print_numbers(1, 2, 3, 4)
1, 2, 3, 4
print_numbers(1, 2, 3, 4, 5, separator = ' ')
1 2 3 4 5
```

# פונקציות עם מספר פרמטרים משתנה

נתן להגדיר פונקציה המתקבלת מספר לא ידוע מראש של פרמטרים שמיים, שיתקבלו בתור מילון (dict) שהמפתחות שלו הם שמות הפרמטרים, והערכים שלו הם ערכי הפרמטרים שהועברו:

```
def print_grades(**students):
 for student, grade in students.items():
 print(student, grade)
```

```
print_grades(alex=100, bob=90, alice=95)
alex 100
bob 90
alice 95
```

# פרמטרים שמיים - סיכום

- כמו ב-`++C`, בפייתון יש אפשרות להגדיר ערכי ברירת-מחדר לפרמטרים לפונקציות ולMETHODS.
- בפייתון ניתן לקרוא לפונקציות באמצעות שמות הפרמטרים ולא רק באמצעות הסדר שלהם.
- סגנון זה נקרא `Named` והיתרונות שלו כוללים קראות משופרת וחוסר תלות בסדר הפרמטרים.
- ניתן להגדיר פונקציות עם מספר פרמטרים משתנה. ניתן לבחור האם לקבל אותם לפי `Positional` (ואז תתקבל רשימה) או לפי `Named` (ואז יתקבל `dict`).
- אפשר לשלב בין האפשרויות.

# תרגול מס' 12 - המשך

- ❖ ספירות נפוצות בפייתון
- ❖ קלט/פלט עם קבצים
- ❖ חריגות
- ❖ מחלקות

# עובדת עם קבצים

# עבודה עם קבצים

- בפייתון ניתן לכתוב לקבצים ולקראם מהם בצורה פשוטה.
- עובד עם קבצי טקסט בלבד ונניח כי קיימת הרשות קריאה וכתיבה לקבצים.
- הfonkcija `(open)` מ带回 לה אובייקט אשר באמצעותו קוראים וכותבים אל תוך הקובץ:

`open(filename, mode)`

**פרמטרים:**

– מסלול (path) אל קובץ כלשהו במערכת הקבצים **filename**

– באיזה מצב לפתח את הקובץ. אחד מ-3 אפשרויות: **mode**

- `'w'` – פותח את הקובץ במצב כתיבה (מוחק את תוכן הקובץ אם הוא כבר היה קיים, אחרת יוצר קובץ חדש בשם זה)
- `'r'` – פותח במצב קריאה (הקובץ חייב להיות קיים אחרת נקבל שגיאה)
- `'a'` – append פותח את הקובץ להוספה (כתיבה בסוף שלו)

# קריאה

file.txt

1. Sunday
2. Monday
3. Tuesday

- קיימות מספר דרכים לקרוא מתוך קובץ:

- הפקציה `read` ממחישה את תוכן הקובץ (כולם) אל תוך מחרוזת אחת

```
f = open('file.txt', 'r')
f.read()
```

'1. Sunday\n2. Monday\n3. Tuesday'

- הפקציה `readlines` ממחישה רשימה אשר כל איבר בה הוא מחרוזת המכילה תוכן של שורה.

```
f = open('file.txt', 'r')
f.readlines()
```

['1. Sunday\n', '2. Monday\n', '3. Tuesday']

# קריאה

file.txt

1. Sunday
2. Monday
3. Tuesday

- הפקציה readline תחזיר שורה בודדת מתוך הקובץ.  
בכל קרייה השורה שמוחזרת היא השורה הבאה בקובץ:

```
f = open('file.txt', 'r')
f.readline()
```

'1. Sunday'

```
f.readline()
```

הזרת השורה הבאה

'2. Monday'

- הדריך הנוחה לקרוא מקובץ היא על ידי לולאה, בקריאה של שורה-שורה מתוך הקובץ:

```
f = open("file.txt", 'r')

for line in f:
 print(line)
```

# כתיבה לקובץ

- כתיבה מתבצעת באמצעות הפונקציה **write**:
- הפונקציה מקבלת כפרמטר מחרוזת וכותבת אותה לתוך הקובץ.  
במידה ורוצים לרדת שורה בקובץ יש להוסיף למחוזת את **\n** ירידת שורה:

```
f = open("new_file.txt", 'w')
s1 = "Hello World\n"
s2 = "End of file"

f.write(s1)
f.write(s2)
```

new file.txt  
Hello world  
End of file

# סגירת קובץ

- חובה **lsegor כל קובץ שנפתח באמצעות open על ידי הפונקציה () close**
- בסגירת קובץ הקישור בין הקובץ בזיכרון המחשב לבין המשטנה בתוכנה מסתירים בצורה מפורשת.
- סגירת קובץ מאפשרת לתוכניות אחרות לגשת אליו.
- שמירת המידע שנכתב לקובץ הינה וודאית רק לאחר סגירת הקובץ.

```
f.close()
```

# סגירה אוטומטית של קובץ

- הדרך הממלצת והמקובלת לפתיחת וסגירה של קבצים בפייתון.
- שימוש במנגןון `with` לפתיחת קובץ וסגירתו בצורה אוטומטית:

```
file_name = 'new_file.txt'

with open(file_name, 'r') as f:
 lines = f.readlines()

print(lines[0])

'1. Sunday\n'
```

בນקודה זו הקובץ נסגר  
אוטומטית (סיום ה-scope  
של `with`)

ניתן לבחור בין המצביעים:  
'w', 'r', 'a'

`file.txt`

1. Sunday
2. Monday
3. Tuesday

- הקובץ נסגר אוטומטית בסוף ה-scope של `with`
- אין צורך לדאוג ידנית לסגירת הקובץ

# כתיבה של מילון לקובץ

- פורמט JSON (JavaScript Object Notation) הוא דרך מקובלת לייצוג מידע באמצעות קובץ טקסט פשוט, ונתמן על ידי שפות תכנות רבות.
- הספרייה הסטנדרטית של פיתון מכילה פונקציות לקרוא וכותבה של מיליוןים לפורמט JSON בצורה פשוטה ומהירה
- המפתחות חיבים להיות מחורזות

```
> import json
> my_dict = { 'key_1': 17,
 > 'key_2': ['a', 'list'] }
>
> # Write my_dict to json
> with open('myfile.json', 'w') as f:
 > json.dump(my_dict, f, indent=4)
>
> # Read myfile.json
> with open('myfile.json', 'r') as f:
 > loaded_dict = json.load(f)
>
> print(loaded_dict)
{'key_1': 17, 'key_2': ['a', 'list']}
```

myfile.json

```
{
 "key_1": 17,
 "key_2": [
 "a",
 "list"
]
}
```

פרמטר גודל זהה. גורם לפלאט מעצב וקריא

- קיימות גם פונק' `json.loads()` ו- `json.dumps()` לקריאה/כתיבה ישירות ממחרוזת.

# חריגות

# חריגות

- כמו ב-`C`, גם ב-`Python` קיים מנגנון של חריגות.
- כדי לזרוק חריגה ב-`Python`, נשתמש במילה השמורה `raise` (ולא `throw` כמו ב-`C++`).
- ניתן לזרוק ב-`Python` עצמים מטיפוס `Exception`, או מטיפוס הירוש מטיפוס זה.
- טיפוסים נפוצים של חריגות ב-`Python`:  
`ValueError`, `TypeError`, `RuntimeError`

```
def print_pos_number(num):
 if num < 0:
 raise ValueError(num)
 print(num)
```

# חריגות מובנות

- בפייתון יש מספר חריגות מובנות, שהשפה תזרוק במקרים מסוימים (רשימה חלקית):

| תירוק כאשר:                                                                                    | החריגה:                          |
|------------------------------------------------------------------------------------------------|----------------------------------|
| מנסים לגשת ל- <code>index</code> ברשימה או ל- <code>key</code> ב- <code>dict</code> שלא קיימים | <code>IndexError/KeyError</code> |
| מנסים לבצע על עצם פעולה שלא מתאימה לטיפוס שלו (נזרק בזמן ריצה)                                 | <code>TypeError</code>           |
| פונקציה מקבלת כפרמטר ערך מהטיפוס הנכון אבל ערך שלא מתאים לפונקציה (למשל ערך שלילי עבור שורש)   | <code>ValueError</code>          |
| מחלקיים באפס                                                                                   | <code>ZeroDivisionError</code>   |
| שגיאה שקשורה לטיפול בקבצים                                                                     | <code>IOError</code>             |

# טיפול בחיריגות

- בדומה ל-C++, גם בפייתון יש מנגנון לטיפול בחיריגות.
- בפייתון אנחנו משתמשים ב-.try ... except

```
def print_pos_number(num):
 if num <= 0:
 raise ValueError(num)
 print(num)

try:
 print_pos_number(int(input('Enter a positive number: ')))
except ValueError:
 print('This was not a positive number...')
```

print\_positive\_number.py

הfonקציה input מדפיס את  
הprompt המועבר לה כארוגמנט,  
ומחזיר לה קלט המוכנס על ידי  
המשתמש לאחר הדפסה הפורמatted

```
>python3 print_positive_number.py
Enter a positive number: -1
This was not a positive number...
```

# טיפול בחיריגות

- מותר شيיו כמה בלוקים של `.except`.
- ניתן לתפוס כמה חריגות באמצעות `:except`

```
def print_pos_number(num):
 if num <= 0:
 raise ValueError(num)
 print(num)

try:
 print_pos_number(int(input('Enter a positive number: ')))
except (ValueError, TypeError):
 print('This was not a positive number...')
except KeyboardInterrupt:
 print('exiting...')
```

חריגה הנזרקת כאשר המשתמש לוחץ על `ctrl-C`

```
Enter a positive number: aaaa
This was not a positive number...
```

# טיפול בחיריגות

- ניתן גם לקבל את ערך החריגה ב-`except` כדי שנוכל להשתמש בו באמצעות המילה השמורה `as`:

```
def print_pos_number(num):
 if num <= 0:
 raise ValueError(num)
 print(num)

try:
 print_pos_number(int(input('Enter a positive number: ')))
except ValueError as err:
 print('This was not a positive number... exception:', err)
```

```
Enter a positive number: -1
This was not a positive number... exception: -1
```

# טיפול בחיריגות

- ניתן להגדיר ל-`try` בלוק `else`, שירוץ אם לא נזקקה חיריגה.

```
def print_pos_number(num):
 if num < 0:
 raise ValueError(num)
 print(num)

try:
 print_pos_number(int(input('Enter a positive number: ')))
except ValueError as err:
 print('This was not a positive number... exception:', err)
else:
 print('That was a positive number, good job! ')
```

**Enter a positive number: 5**

**5**

**That was a positive number, good job!**

# טיפול בחיריגות

- ניתן להגדיר ל-`try` בלוק `finally`, שירוץ **תמיד**, בין אם נדרש חיריגה או לא, בין אם נתקשה חיריגה או לא.

```
def print_pos_number(num):
 if num < 0:
 raise ValueError(num)
 print(num)

try:
 print_pos_number(int(input('Enter a positive number: ')))
except ValueError as err:
 print('This was not a positive number... exception:', err)
finally:
 print('All in all, a learning experience')
```

Enter a positive number: 5

5

All in all, a learning experience

# Stack unwinding

- כמו ב C++ גם ב פיריטון החריגה מפועעתת כלפי מעלה במחסנית.
- אם חריגה לא נתפסת בפונקציה/תסריט ראשי אז התוכנית מסיימת את ריצתה ומודפס traceback של מחסנית הקראות.

```
def print_pos_number(num):
 if num < 0:
 raise ValueError(num)
 print(num)

print_pos_number(int(input('Enter a positive number: ')))
```

test.py

```
Enter a positive number: -1
Traceback (most recent call last):
 File "test.py", line 5, in <module>
 print_pos_number(int(input('Enter a positive number: ')))
 File "test.py", line 3, in print_pos_number
 raise ValueError(num)
ValueError: -1
```

# assert

- כמו שראינו ב++c/c גם בפייתון קיים מנגנון של וידוא הנחות בזמן ריצה בעזרת המילה השמורה – **.assert**.
- כאשר התנאי ב**assert** לא מתקיים, נזרקת חריגה בשם **AssertionError**.

```
x = "hello"
#if condition returns False, AssertionError is raised:
assert x == "goodbye"
```

```
Traceback (most recent call last):
 File "<string>", line 3, in <module>
AssertionError
```

- ניתן גם להוסיף הודעה שגיאה. לדוגמה:

```
x = "hello"
assert x == "goodbye", "x should be goodbye'"
```

```
Traceback (most recent call last):
 File "<string>", line 2, in <module>
AssertionError: x should be 'goodbye'
```

# חריגות - סיכום

- כמו ב-`C++`, בפייתון יש מנגנון של חריגות לטיפול בשגיאות.
- בפייתון המנגנון משוכל במעט על פנוי `C`.
- בлок `except` אחרי `try` משמש כדי להריץ קוד שירוץ אם נזרקה חריגה המצוינת ב-`except`.
- ניתן לתפוס כמה חריגות באותו `except`.
- בлок `else` משמש כדי להריץ קוד שירוץ אם לא נזרקה חריגה.
- בлок `finally` משמש כדי להריץ קוד שירוץ בכל מקרה, בין אם נזרקה חריגה ובין אם לא.
- בлок `finally` משמש, למשל, כדי לסגור קבצים.
- בפייתון מתבצע `unwinding stack` כמו ב-`C`.
- אפשר להדפיס את מצב המחסנית בעזרת הפקצתה `print_stack` בספירה `.traceback`.
- המילה השמורה `assert` משמשת כדי לוודא הנחות בזמן ריצה.
- עקרונות התכנות הנכון עבור חריגות ב-`C++` תקפים גם בפייתון.

# מחלקות

# הגדרת מחלקה

- כמו ב-`C++`, גם ב-`פייתון` קיימות מחלקות ומנגנון של ירושה.
- ב-`פייתון` מגדירים מחלקה עם המילה השמורה `class` עם בלוק.
- בדומה ל-`C++`, ב-`פייתון` מethodות של המחלקה מוגדרות בתוך המחלקה.
- כדי לציין שאלו מethodות של המחלקה, משתמשים ב-"`פרמטר`" הנוסף `self` הממלא תפקיד דומה לזה של `this` ב-`C++`.
- למחלקות ב-`פייתון` יש בנאי בירית מיוחד אם לא הוגדר להן בנאי אחר.

```
class Dog:
 def bark(self):
 print('Woof')

dog = Dog()
dog.bark() # Woof
```

# שדות

- ניתן לגשת לשדות במחלקה באמצעות התחביר `<object>.field`. בפייתון, לא צריך להציג במפורש על שדות.
- לא קיים מנגנון של `access control` – כל השדות וכל המethodות הם בחזקת `public`.
- מקובל לאותחל את כל השדות שנשתמש בהם בבנייה.
- בפייתון, בניאי הוא מתודה המוגדרת במחלקה עם השם `__init__`.

```
class Dog:
 def __init__(self, name):
 self.name = name

 def bark(self):
 print('Woof, says', self.name)

dog = Dog('Rexy')
print(dog.name) # Rexy
dog.bark() # Woof, says Rexy
```

# הערות

- אין צורך בMETHOD\_OF\_DESTRUCTOR (destructor) לאור שחרור הזיכרון האוטומטי בשפה.
- לא ניתן לגשת לשדות מטור המחלקה ללא **self**. למשל, הקוד הבא אינו חוקי:

```
class Dog:
 def __init__(self, name):
 self.name = name

 def bark(self):
 print('Woof, says', name) # must use self.name
```

# ירושה

גם בפייתון קיימן מנגנון של ירושה ופולימורפיזם:

```
class Dog:
 def __init__(self, name):
 self.name = name

 def bark(self):
 print('Woof, says', self.name) # must use self.name

class Chiwawa(Dog):
 def bark(self):
 print('Weef, says', self.name) # must use self.name

dogs = [Dog("Tino"), Chiwawa("Toy")]
[dog.bark() for dog in dogs]
```

```
woof, says Tino
weef, says Toy
```

# מחלקות - סיכום

- כמו ב-`++C`, בפייתון יש מחלקות.
- כדי לקרוא לבנאי, משתמשים בשם המחלקה כאילו הייתה פונקציה.
- לא קיימים הורסים בפייתון כי ניהול הזיכרון בשפה אוטומטי.
- בנאי הוא מתודה במחלקה בשם `__init__`.
- לא צריך להציג על שדות במפורש - רק לאותם בבנאי.
- המקבילה בפייתון של `this` היא `self`. חייבים תמיד להשתמש ב-`self` כדי להתייחס לשדות במחלקה.

# ספריות נפוצות בפייתו

# ספריות בפייתו

- בפייתון ישן ספריות רבות לכל תחומי העיסוק: למידת מכונה, ניתוח מידע, אלגברה לינארית, עבודה עם מערכת הפעלה, תקשורת, קבלת ארגומנטים משורת הפקודה (כמו ב-C) ועוד.
- ניתן להתקין ספריות חדשות עם מנהל הספריות של פייתון ששמו **cipk**.
- כתעת נציג מספר ספריות מובנות בסיסיות בשפה.

# הספריה sys

- הספריה sys היא מובנית בפייתון ומאפשרת גישה למשתנים ופונקציות שלתחרירים לאינטראפטור.

```
import sys
```

- דוגמה לחלק מהפונקציונליות בספריה sys מספקת לנו:

```
import sys

print('Error', file = sys.stderr) #prints to stderr

for arg in sys.argv: #prints the command line argument of a script
 print(arg)
```

# הספריה os

- הספריה os בפייתון מאפשרת גישה למתודות רבות של מערכת הפעלה.

```
import os
```

- היא בלתי תלולה במערכת הפעלה - היא תעבור ללא תלות בהאם היא רצה על Windows, Linux או כל מערכת הפעלה אחרת.

# הספרייה os

- מספר פקודות חשובות לעבודה עם קבצים בספריה:
- `(path)os.listdir()` – מדפיסה את רשימת הקבצים בתיקייה הנתונה.
- `os.sep` – התו המפריד בין ספריות במערכת הפעלה הנוכחית.

```
> path = 'home{}mtm{}top_secret_dir'.format(os.sep,os.sep)
path = 'home/mtm/top_secret_dir' in linux.
> os.listdir(path)
['tutorial13_python3.pptx', 'matam_exam_final.pdf',
'secret_nuclear_codes.txt']
```

# הספרייה os

- `(*)os.path.join(*path)` – מקבלת אלמנטים של כתובות ומחברת אותם לכדי כתובות אחת בדרך המקובלת במערכת הפעלה הנוכחית.

```
> path = os.path.join('home', 'mtm', 'top_secret_dir')
'home/mtm/top_secret_dir' in linux.
```

- `os.path.dirname(path)` – מקבלת אלמנט של כתובות ומחזירה את הכתובת של תיקיית האב.

```
> dir_path = os.path.dirname(path)
dir_path = 'home/mtm'.
```

- `os.path.basename(path)` – מקבלת אלמנט של כתובות ומחזירה את שם הקובץ/תיקייה אליו מובילת הכתובת.

```
> dir_path = os.path.basename(path)
dir_path = 'top_secret_dir'.
```

# הספרייה os

- `os.path.split(path)` – מקבלת אלמנט של כתובות ומחזירה tuple עם 2 איברים – תחילת הכתובת וסופה.

```
> os.path.split(path)
('home/mtm', 'top_secret_dir')
```

- `os.path.splitext(path)` – מקבלת אלמנט של כתובות ומחזירה tuple – הכתובת בלי הסיומת, והסיומת.

```
> full_path = os.path.join(path, 'secret_nuclear_codes.txt')
> os.path.splitext(full_path)
('home/mtm/top_secret_dir/secret_nuclear_codes', '.txt')
```

- יוצרת תיקייה עם הכתובת המתקבלת.

```
> directory_name = 'cats_images'
> directory_path = os.path.join(path, directory_name)
> os.makedirs(directory_path)
> os.listdir(path)
['tutorial13_python3.pptx', 'matam_exam_final.pdf',
'secret_nuclear_codes.txt', 'cats_images']
```

# דוגמת סיכון

- נכ调皮 תוכנית המדפיסת את תוכני כל הקבצים בתיקייה המועברת כפרמטר לסקריפט.

```
import os
import sys

if len(sys.argv) != 2:
 print('usage: script.py <path_to_dir>')

path = sys.argv[1]
for file in os.listdir(path):
 file_path = os.path.join(path, file)
 if os.path.isfile(file_path):
 with open(file_path, 'r') as f:
 print(f.read())
```

הfonקציה `os.path.isfile` ממחירה True אם הכתובת היא של קובץ False אם היא תיקייה. קיימת פונקציה נוספת שבודקת את המקרה ההפוך – `os.path.isdir`.

```
> python3 script.py /home/mtm/top_secret_dir
Nuclear codes: 4, 8, 15, 25, 47, 42
Final exam: File too large
...
```

# ספריות מוכנות נוספות

- `argparse` - ניהול מתקדם של פרמטרים משורת הפקודה, כולל דגלים, פרמטרים אופציונליים ועוד.
- `shutil` - ספרייה משלימה לזה שמספקת פונקציונליות כמו העתקת ומחיקת קבצים וספריות.
- `math` - אוסף פונקציות מתמטיות נפוצות.
- `abc` - ספרייה להגדרת מחלקות אבסטרקטיות.
- `itertools` - מגוון פונקציות לעבודה עם איטרטורים, כמו שרשור איטרטורים, חזרה על איטרציה מספר פעמים.
- `random` - אוסף פונקציות לייצור מספרים אקראיים לפי מגוון התפלגות.
- `time` - קבלת הזמן הנוכחי, מדידת זמנים, השהיית ריצת התוכנית לזמן קצר ועוד.
- `Pickle` - שמירה וטעינה אובייקטים של פיתון לקבצים.
- ועוד ספריות רבות.
- `async` - עבודה אסינכרונית.

# ספריות חיצונית נפוצות

- Numpy - ספרייה מתמטית מקיפה לטיפול במטריצות, וקטורים, פונקציות סטטיסטיות וחישובים מתמטיים נוספים.
- Scipy - ספריית המרחיבה את Numpy עם מגוון חישובים נומריים ואלגוריתמים מתמטיים (אינגרציה, אינטראפולציה, אופטימיזציה ועוד).
- matplotlib - ספרייה לויזואלייזציה אינטראקטיבית של גרפים 2D ותחת ממדים.
- openCV - ספרייה לעובדה עם תמונות וסרטים.
- Requests - ניהול תקשורת עם אתרים ושרותי אינטרנט
- (http) - ייצור חיבורים, העברת מידע, תקשורת מאובטחת ועוד.
- TensorFlow/PyTorch - ספרייה עבור מודלי למידה.
- ועוד הרבה..



# ספריות בפייתו - סיכום

- בפייתו יש מגוון רחב של ספריות זמיןות לשימושים.
- ניתן להתקין ספריות חדשות באמצעות מנהל החבילות **diof** הזמין בפייתו.
- מומלץ תמיד להשתמש בקוד מה빌ה במידת האפשר, במקום לכתוב את הקוד בעצמו.
- הספריה **os** מכילה כלים שימושיים לתקשורת עם מערכת הפעלה. כלים אלו שימושיים במיוחד בעבודה עם קבצים.

# תרגול מס' 13

- ❖ מבנה וחומר המבחן
- ❖ שאלת הכוללת תכנן ב++C

# מבנה וחומר המבחן

# מבנה המבחן

- ב מבחן יהיו 3 שאלות:
  - שאלת C++ "קלאסית".
- שאלה כללית על C++ הבודקת ידע ושימוש נכון בתכונות שונות של C++ שנלמדו. בדרך כלל בשאלת זו נדרש למשר מחלוקת או מבנה נתונים, לפחות עם מבנים.
- הדגש בשאלת זו – שימוש נכון בכלים של מדרנו בקורס (מבנה נתונים, העממת פונקציות ואופרטורים, מצביים חכניים, STL)
- שאלת תכנון (ב-C++).
- שאלה זו תתמקד בתכנון נכון של מערכת. תקבלו סיפור רקע כלשהו ותשאלו מחלוקת לתכנון ולמשר מערכת העונה לצרכים של הסיפור.
- הדגש בשאלת זו – תכנון נכון, בפרט בר הרחבה ותחזוקה (Extensible and Extensible), הבנה ושימוש נכון בתכונות תכנון (Maintainable), שאלת פיתון.
- בשאלת זו תידרשו למשר תכנית בפייתון, תוך כדי שימוש במנגנוןים השונים בשפה של מדרנו.

# איך לומדים באופן כללי?

- לפני שמתחלים לפתרו מבחנים חשוב מאד לעבור על החומר.
  - לחזור על השקפים של הרצאות והתרגולים.
  - מומלץ ללמידה בצורה רוחנית – לעבור על חומר מסוים ובסופה לפתרו שאלות.
- פתרת מבחנים וסימולציות.
  - כדאי לSAMPLE פתרת של מבחנים כדי לתרגל כתיבת קוד על דף ולהתרגל לחומר הפתוח.
  - במחן יש חומר פתוח. כדאי לסדר את החומר בצורה נוחה, כדי לא לbezבז זמן במחן על חיפוש.
  - אסור לכתוב בעיפרון – רק בעט. (לא בשליטהנו – הסורקים בטכניון לא מסתדרים עם עופרת)
  - תבאו לשעות קבלה לפני המבחן לשאול שאלות.
- כל אחד מוצא את הדרך שלו ללמידה. כל העצות כאן הן בגדר הצעה.

# AIR ללמידה לכל שאלה?

- שאלת C++ קלאסית:
  - חזרו על הדוגמאות לשימוש בתכונות השונות של C++ מהתרגולים ועל שאלות החזירה מתרגול 9.
  - פתרו שאלות ממבחןים קודמים.
- שאלת תכנון:
  - חזרו על על הרצאות העוסקות בתכנון, הבינו את התפקידים השונים של תבניות התכנון.
  - פתרו שאלות ממבחןים קודמים.
- שאלת פיתון:
  - פתרו שאלות ממבחןים קודמים.
  - ניתן להשתמש לתרגול בחומרים מהאינטרנט ומהקורס מבוא לפיתון.

# שאלת הכללה תכנון ב+C לסייע

# שאלות תכנון

- מה הן שאלות הכללת תכנון? בשאלות תכנון יהיה לנו סיפור כלשהו המתאר דרישות ממערכת מסוימת.
- נתבקש לתאר את התכנון באמצעות UML או/ו לספק ממשק למחלקות/מערכת או/ו מימוש לחלק מהמחלקות הנכללות בתכנון.
- לעיתים הסיפור יכול כולל מקרים שבהם יתאים להשתמש בחומר design pattern מוכר, כמו אלה שנלמדו בהרצאות.
- תזכורת: בהרצאות נלמד design patterns הבאים:
  - Behavioral: Strategy, Command.
  - Structural: Composite, Adapter, Decorator.
  - Creational: Abstract Factory, Singleton.

# שאלה סיכון

בשאלה זו נרצה לתכנן ולממש מערכת של בנק המנהלת את חשבונות לקוחות הבנק. כל חשבון מאופיין במספר חשבון, שם של בעלי החשבון, יתרה והיסטורית פעולות (הפקדה, משיכה, העברה מחשבון לחשבון).

```
class BankSystem {
 std::map<int, std::unique_ptr<Account>> m_accounts;
public:
 ...//TODO
};
```

```
int main(){
 BankSystem bank();
 int accountNumAlice = bank.addAccount("Alice", 1100);
 int accountNumBob = bank.addAccount("Bob");
 // Alice's balance = 1100, Bob's balance = 0.
 bank.deposit(accountNumAlice, 100);
 bank.withdraw(accountNumBob, 100);
 // Alice's balance = 1000, Bob's balance = 100.
 bank.undoLastAction(accountNumAlice);
 // Alice's balance = 1100, Bob's balance = 100.
 bank.transfer(accountNumAlice, accountNumBob, 100);
 // Alice's balance = 1000, Bob's balance = 200.
 bank.undoLastAction(accountNumAlice);
 // Alice's balance = 1100, Bob's balance = 100.
 double balanceAlice = bank.getBalance(accountNumAlice);
 // balanceAlice = 1100.
 return 0;
};
```

השלימו את ממשק המחלקה  
המייצגת מערכת  
של בנק.

לפי פונקציית ה`main` הבאה:

# סעיף א - פתרון

מפתח מחסור במקום  
לא הוספנו בנאי,  
והורס. נשתמש  
במימוש הדיפולטי.  
בנאי העתקה  
– אופרטור השמה –  
אם נרצה למעשה  
ntsentr להגדיר  
מתודת clone  
לחשבונות

```
int main(){
 BankSystem bank();
 int accountNumAlice = bank.addAccount("Alice", 1100);
 int accountNumBob = bank.addAccount("Bob");
 // Alice's balance = 1100, Bob's balance = 0.
 bank.deposit(accountNumAlice, 100);
 bank.withdraw(accountNumBob, 100);
 // Alice's balance = 1000, Bob's balance = 100.
 bank.undoLastAction(accountNumAlice);
 // Alice's balance = 1100, Bob's balance = 100.
 bank.transfer(accountNumAlice, accountNumBob, 100);
 // Alice's balance = 1000, Bob's balance = 200.
 bank.undoLastAction(accountNumAlice);
 // Alice's balance = 1100, Bob's balance = 100.
 double balanceAlice = bank.getBalance(accountNumAlice);
 // balanceAlice = 1100.
 return 0;
}

class BankSystem {
 std::map<int, std::unique_ptr<Account>> accounts;

public:
 int addAccount(const std::string& ownerName, int initialBalance = 0);
 void deposit(int accountNum, double amount);
 void withdraw(int accountNum, double amount);
 void transfer(int transmitterAccountNum, int receiverAccountNum, double amount);
 void undoLastAction(int accountNum);
 double getBalance(int accountNum) const;
};
```

השלימו את  
שימוש המחלקה  
BankSystem  
המייצגת מערכת  
של בנק.  
לפי פונקציית  
הchnin הבאה:

# סעיף ב'

חובו על תכנן למערכת, רשמו UML לתכנן שלכם ורשמו אם השתמשם ב design patterns כלשהם.

מזכורת – UML:



# סעיף ב'

תארו תוכן המתאים לדרישות מסעיף א' באמצעות UML וצייןו אם השתמשתם בו-  
design patterns כלשהם.

בד"כ, שמות עצם הם  
אינדיקציה למחלקות

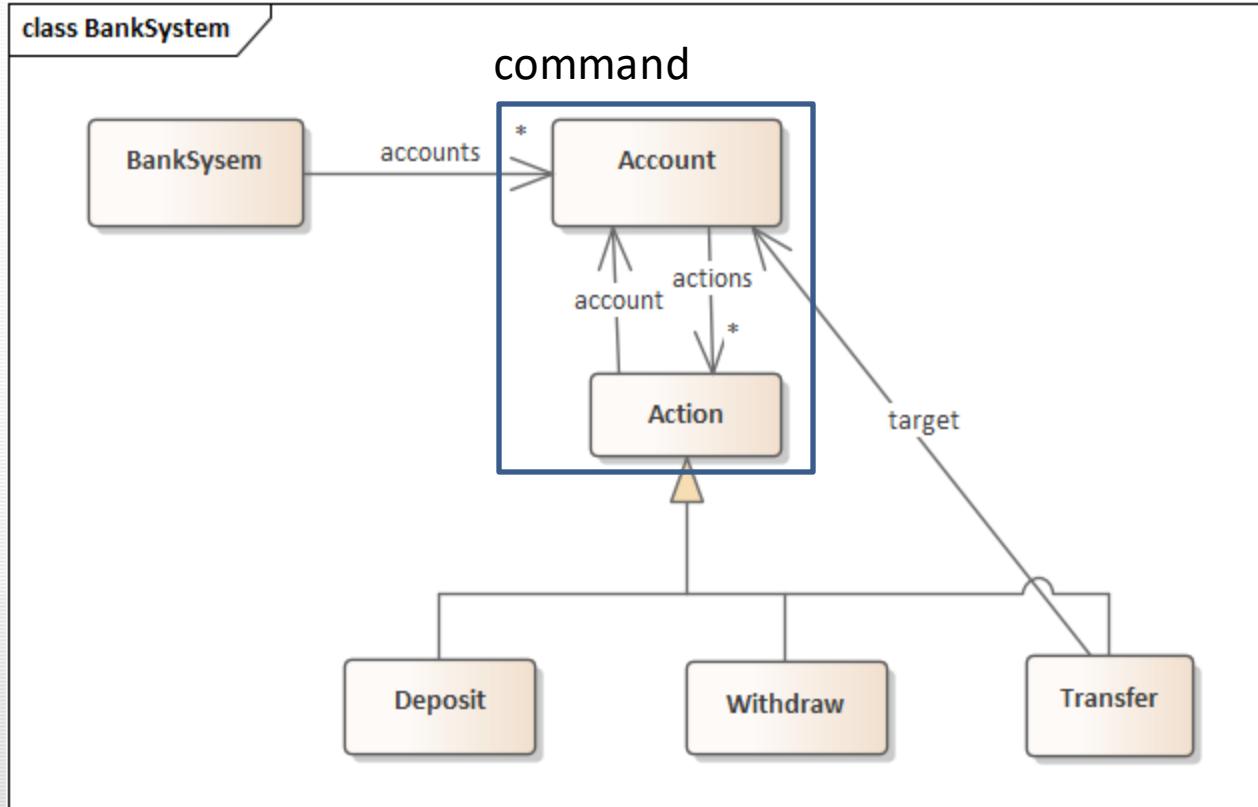
זכור בסיפור השאלה ובממשק המערכת:

מערכת של בנק המנהלת את **חוויות לקוחות הבנק**. כל חשבון מאופיין במספר,  
שם של בעלי החשבון, יתרה והיסטוריה של פעולות (הפקדה, משיכה, העברה  
מחשבון לחשבון).

```
class BankSystem {
 std::map<int, std::unique_ptr<Account>> accounts;

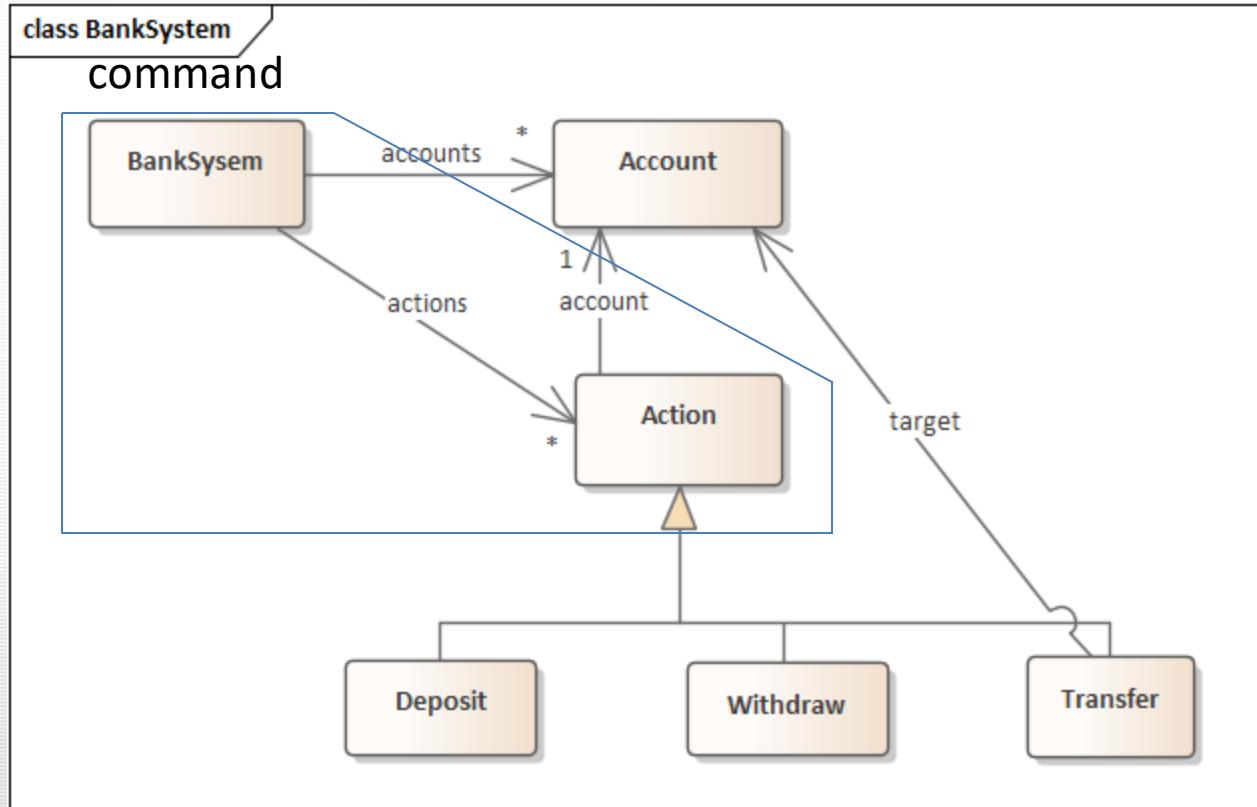
public:
 int addAccount(const std::string& ownerName, int initialBalance = 0);
 void deposit(int accountNum, double amount);
 void withdraw(int accountNum, double amount);
 void transfer(int transmitterAccountNum, int receiverAccountNum, double amount);
 void undoLastAction(int accountNum);
 double getBalance(int accountNum) const;
};
```

# פתרונות – סעיף ב'



האם השתמשנו בdesign patterns מוכרים?  
השתמשנו בdesign pattern התנהוגתי – **command**. הפעולה מיוצגת על ידי מחלקת  
יעודית אשר מבצעת ומחזיקה את כל המידע כדי לבצע את הפעולה.

# פתרון - סעיף ב' – אופציה נוספת



# סעיף ג' – הפונקציה transfer

ממשו את הפונקציה transfer של BankSystem וכל מחלוקת שהשתמשה בה בדרך.

```
void BankSystem::transfer(int transmitterAccountNum, int receiverAccountNum, double amount){
 std::unique_ptr<Transfer> transferAction(new
 Transfer(accounts[transmitterAccountNum].get(),
 accounts[receiverAccountNum].get(),
 amount));
 transferAction->apply();
 accounts[transmitterAccountNum]->saveAction(std::move(transferAction));
}
```

.Account-ו Action ,Transfer: נספק מימוש למחלקות

# תזכורת - בעלות על זיכרון

- נשתמש ב-**ptr\_unique** כאשר קיימת **בעלות יחידה** על הזיכרון הדינامي.
- **הבעלים** של הזיכרון הוא מי שאחראי **לפנות** אותו, כאשר משתמשים בו smart pointers זה מתבצע אוטומטית.
- כאשר אין דרישת בעלות אלא רק גישה ניתן להסתפק בפונקטור פשוט **שאין** לפנותו.
- עבור המחלקות **Action**-**Transfer** נדרש רק גישה לאובייקט מסווג **Account**.
- ב-**ptr\_shared\_ptr** נשתמש במצב של בעלות מרובה, כאשר לא ידוע מי מהבעלים אחראי על פינוי האובייקט.

# סעיף ג' המחלקות Action-1 Transfer

```
class Action {
protected:
 Account* account;
public:
 Action(Account* account) : account(account){}
 virtual void apply() = 0;
 virtual void undo() = 0;
};

class Transfer : public Action {
 Account* target;
 double amount;
public:
 Transfer(Account* transmitter,
 Account* target, double amount) :
 Action(transmitter), target(target), amount(amount){}
 void apply() override {
 account->reduceBalance(amount);
 target->addToBalance(amount);
 }
 void undo() override {
 account->addToBalance(amount);
 target->reduceBalance(amount);
 }
};
```

מפהת מחסום  
במקום לא  
הוספנו בנאי,  
בנאי העתקה,  
אופרטור  
השמה והורס.  
נשתמש  
במימוש  
הדיפולטי

# סעיף ג' המחלקה Account

```
class Account {
protected:
 std::string ownerName;
 double balance;
 std::vector<std::unique_ptr<Action>> actions;
public:
 Account(const std::string& ownerName, double initialBalance) :
 ownerName(ownerName), balance(initialBalance){}
 void saveAction(std::unique_ptr<Action> action){
 actions.push_back(std::move(action));
 }
 void addToBalance(double amount){
 balance+=amount;
 }
 void reduceBalance(double amount){
 balance-=amount;
 }
};
```

בעלויות ייחודיות

אפשר היה להשתמש גם בstack.

מה קורה אם מקבלים amount שלילי? מה  
קורה אם היתרה יודת מתחת לאפס? מה לגבי  
מסגרת אשראי?  
שאלות טובות – נתעלם.

## סעיף ד

בבנקים מסויימים ניתן לפתח חשבון רק לאחר אישור ממערכת ממשלתית.

- מסופקת לכם הפקציה:

`bool govAPI::getIdentityApproval(std::string accountOwnerName)`

הפקציה מחזירה אמת אם יש אישור, וסקר אחרת.

- במקרה של כישלון לפתח חשבון יש לזרוק חריגה `UnapprovedAccount` ממשו מחלקה הנקרהת `SecuredBankSystem` אך תומכת בדרישה זו.

פתרונות:

```
class SecuredBankSystem : public BankSystem {
public:
 void addAccount(std::string ownerName, int initialBalance) override
{
 if(!govAPI::getIdentityApproval(ownerName)){
 throw UnapprovedAccount();
 }
 return BankSystem::addAccount(ownerName,initialBalance);
 }
};
```

נוסיףvirtual במחלקה BankSystem

```
class BankSystem {
....
public:
 virtual int addAccount(const std::string& ownerName, int initialBalance = 0);
....
};
```

# סעיף ה'

לגרסה הבאה נדרשת תוספת הדרישות הבאה:

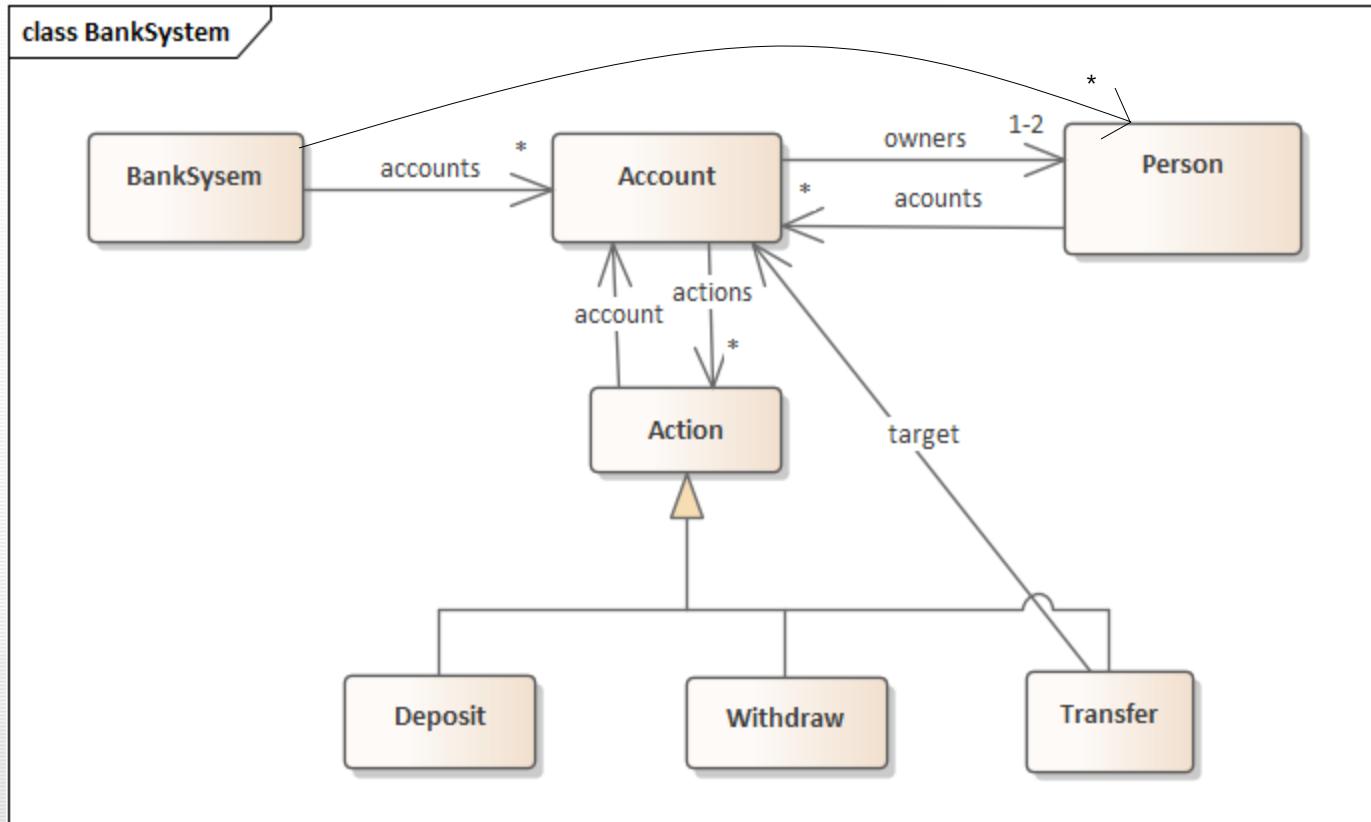
נדרש להוסיף למערכת ניהול של לקוחות מאופיין על ידי מספר ת"ז ייחודי. לקוחות יכול להיות בעליים של יותר מחשבון אחד. לחשבון יש בד"כ בעליים ייחדים אולם לעיתים כאשר חשבון משותף נדרשים שני בעליים.

נרצה לעדכן את התכנן כך שיתמוך בשאלות הבאות על המערכת:

1. מי הם לקוחות הבנק.
2. אילו חשבונות שייכים לכל לקוח.

תארו את התכנן החדש באמצעות UML.

# סעיף ה'



# שאלה בonus הכללת תכו ב-++

# שאלות בונוס

בשאלה זו נרצה לתכנן ולממש חלק ממשחק "פוקימון".

פוקימונים הם יצורים דמיוניים בעלי כל מיניסוגים של יכולות.

במשחק שלנו, כל מאמן פוקימונים (השחקן) יכול להציג קבוצה של פוקימונים, איתם יוכל להילחם נגד מאמנים אחרים.

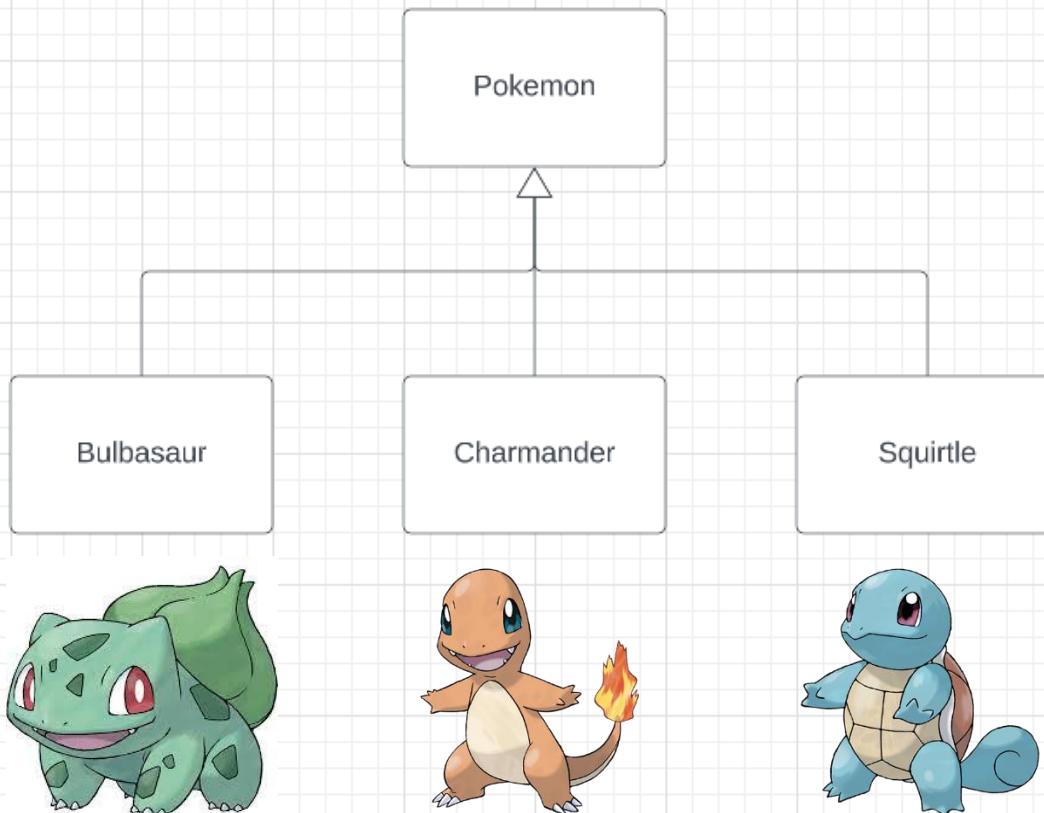


# שאלות בונוס

לאורך השאלה נמשך מספר מחלוקת, בין היתר את המחלוקת Pokemon. עברו כל מין ספציפי תוגדר מחלוקת היורשת מ-ho-ם ומצדירה את ההתנהגות של פוקימון זה.

# שאלות בונוס

לדוגמה:



# סעיף א (1)

במהלך קרב, פוקימון יכול לתקוף את הפוקימון היריב בעזרת "מהלך" (מתќפה) על ידי קריאה למתודה:

```
void Pokemon::makeMove(Pokemon& opponent);
```

בזמן יצירת הפוקימון יבחר עבורו מהלך אחד אותו הוא יוכל לבצע באופן רנדומלי מתוך כל המהלךים האפשריים (למשל Growl, Tackle, ...). לכל מהלך יש אפקט שונה, המושפע מסוג המהילך, מנוטוני הפוקימון התקוף ומנתוני המותקף.

כיצד נממש את makeMove?

# סעיף א (1) - פתרון

נגיד מחלוקת אבסטרקטית Move המייצגת מהלך וממנה ירשו  
מחלקות נוספות הממשות מהלכים קונקרטיים:

```
class Move {
 Pokemon& m_pokemon;

public:
 Move(Pokemon& p): m_pokemon(p) {};
 void apply(Pokemon& opponent) = 0;
};
```

כל מחלוקת היורשת מ-Move תשתמש את apply בהתאם למה מהלך  
יעשה.

# סעיף א (1) - פתרון

נוסיף ל-Pokemon שדה המחזיק את המהלך ונפעיל אותו בעת קרייה :makeMove-ל

```
class Pokemon {
 unique_ptr<Move> m_move;

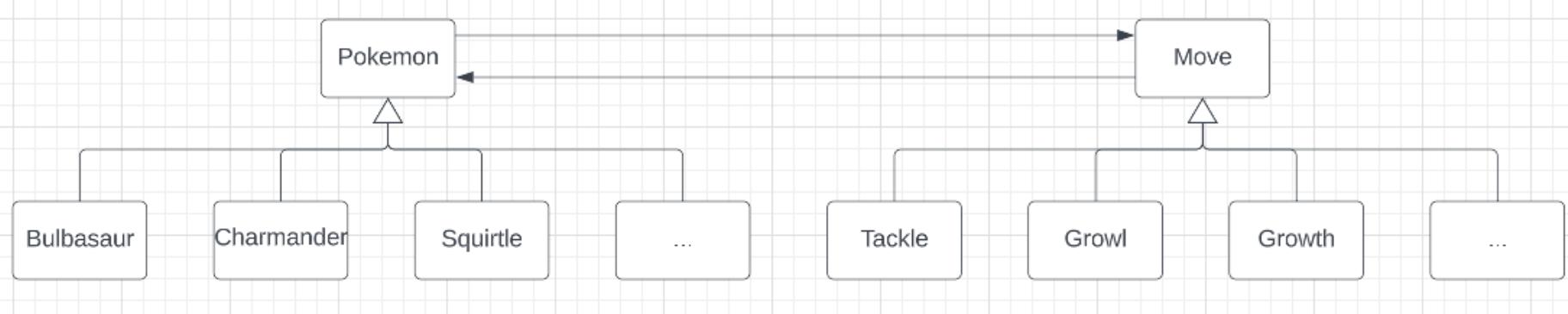
public:
 void makeMove(Pokemon& opponent) {
 m_move->apply(opponent);
 }
};
```

# סעיף א (2)

באיזה Design Pattern השתמשנו?  
צייר UML של המערכת.

# סעיף א (2) - פתרון

.Command Design Pattern שבו השתמשנו הוא



# סעיף ב (1)

נגיד מחלקת Trainer המממשת את השחקן במשחק. לכל שחקן יש סוג אופי המשפיע על אופן הפעולה שלו בכל מיני אופנים, למשל על אופן בחירת הפוקימון שישלח קרב:

- אימפולסיבי – יבחר תמיד את הפוקימון הראשון בקבוצה.
- חמדן – יבחר תמיד את הפוקימון בעל הרמה הכى גבוהה.
- כאוטי – יבחר באופן רנדומלי.

כיצד תממשו את המתודה drawPokemon של Trainer המממשת פונקציונליות זו?

# סעיף ב (1) - פתרון

בדומה לסעיף א', נגידר מחלוקת Character ומחלות הירושות ממנו:

```
class Character {
 Pokemon&
 drawPokemon(const vector<unique_ptr<Pokemon>>& pokemons) = 0;
};

class Impulsive: public Character {
 Pokemon&
 drawPokemon(const vector<unique_ptr<Pokemon>>& pokemons) {
 return *pokemons[0];
 }
};
```

# סעיף ב (1) - פתרון

```
class Trainer {
 vector<unique_ptr<Pokemon>> m_pokemons;
 unique_ptr<Character> m_character;
public:
 Pokemon& drawPokemon() {
 return m_character->drawPokemon(m_pokemons);
 };
}
```

# סעיף ב (2)

באיזה Design Pattern השתמשנו?

## **סעיף ב (2) - פתרון**

הפעם ה-Design Pattern שבו השתמשנו הוא Strategy (מהו ההבדל?)

# סעיף ג

נניח ומוגדרת עבורינו מחלקה `InfoProvider` המספקת לנו כל מיני סוגי של מידע על פוקימונים. בין היתר מוגדרת במחלקה מתודה המייצרת ומחזירה לנו תמונה של פוקימון לפי השם שלו (בעזרה אובייקט שנקרא `Picture`):

```
class InfoProvider {
public:
 const Picture& getPokemonPicture(const string& name,
 bool is_shiny);
}
```

# סעיף ג

דוגמה לשימוש ב-InfoProvider במחלקה Pokemon שכבר הגדרנו קודם:

```
class Pokemon {
protected:
 InfoProvider& m_prov;
public:
 Pokemon(const InfoProvider& p): m_prov(p){}
 const Picture& getPicture() {
 return m_prov.getPokemonPicture(
 this->getName(),
 false);
 }
}
```

## סעיף ג

לאורך השנים התווסף המונח פוקימונים לעולם שלנו, אולם המפתחים של המחלקה InfoProvider לא תחזקו את המודול ובעת לא ניתן להשתמש בו יותר (למשל לא ניתן לבקש תמונה של פוקימון מהדור החדש).

כדי להוסיף תמיכה בתמונות של פוקימונים חדשים, נרצה להשתמש במחלקה אחרת הנקראת `getPokemonPicture`, לה מתודה `getPokePic` דומה ל-`NextGenProvider`. שכבר ראינו.

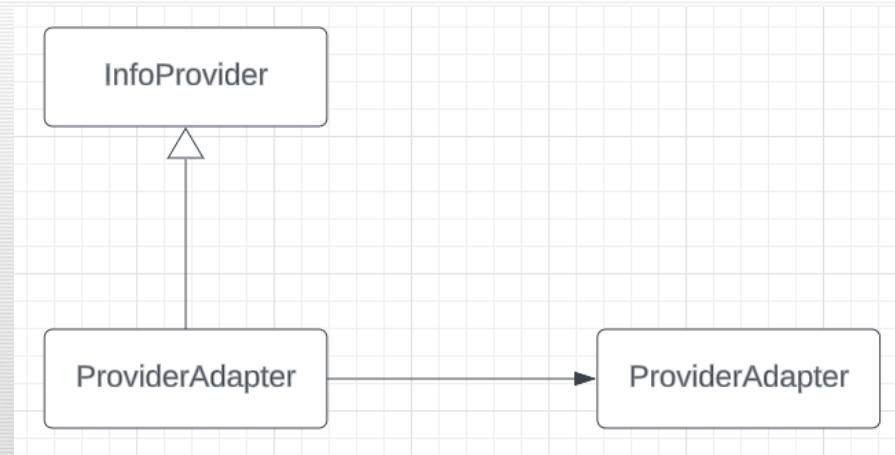
# סעיף ג

כעת נוצרה לנו בעיה – בכל מקום במערכת בו רצינו להשתמש בתמונות השתמשנו באובייקט מהמחלקה `InfoProvider`. אם נרצה להחליף אותו באובייקט מסווג `NextGenSupplier` נדרש לשנות את כל המKENOT היללו.

כיצד נפתרת את הבעיה? חשבו על **Design Pattern** מתאים שבעזרתו נוכל לשנות מעט מקומות בקוד הקיים.

# סעיף ג - פתרון

נשתמש ב-`Adapter`.  
נדיר מחלקה בשם `InfoProvider` שתירש מ-`ProviderAdapter` ותחזיק אובייקט מסווג `NextGenProvider`.  
השינוי היחיד שנצרך לעשות לקוד קיים, הוא להחליף את היצירה של האובייקט `provider`, כך שהיא מסווג `ProviderAdapter`, ולא רק `InfoProvider`.



# סעיף ג - פתרון

ימוש של המחלקה :ProviderAdapter

```
class ProviderAdapter : public InfoAdapter {
 NextGenProvider m_prov;
public:
 const Picture& getPokemonPicture(const string& name,
 bool is_shiny) {
 return m_prov.getPokePic(name, is_shiny);
 }
}
```

# סעיף ד (1)

פוקימונים זוחרים הם מוציאה נדירה של פוקימונים רגילים. ההבדל היחיד בין פוקימונים רגילים הוא שיש להם תמונה שונה (ניתן לקבל אותה מהפונקציה `getPokemonPicture` על ידי העברת `true` בפרמטר `is_shiny`).



כיצד נוכל למש פוקימונים זוחרים בקלות?

# סעיף ד (1) - פתרון

נוסיף את המחלקה הטמפליתית Shiny:

```
template<class P>
class Shiny : public P {
public:
 const Picture& getPicture() {
 return m_prov.getPokemonPicture(this->getName(),
 true);
 }
}
```

# סעיף ד (2)

באיזה Design Pattern השתמשנו?

## **סעיף ד (2) - פתרון**

בו השתמשנו הוא Design Pattern – הגדרנו דרך להרחיב או לשנות את אופן הפעולה של אלמנטים קיימים במערכת בצורה גמישה ומודולרית.

# סיכום

- בשאלות תcen נקבל דרישות מערכת באמצעות סיפור/טקסט כלשהו או בעזרת API – אוסף פונקציות שצרכות להיתמן על ידי המערכת.
- ננסה להבין מה הם ה"עצמים" בסיפור. נחפש מילים המיצגות יישות, וקשרים בין היחסות.
- האם זה ירשה – a zo, או הכלה – אפיון, ניהול, בעלות וכדומה.
- נחשב האם המקרה מתאים לpatterns design שראינו בהרצאות.
- לרוב נתבקש לציר UML לפי הכללים שלמדו.