# עקרונות שפות תוכנה

בקורס הזה נלמד עקרונות שונים ונראה איך הם באים לידי ביטוי בשפות שונות.

## קריטריונים להערכת שפה:

1. Readability – קריאות – עד כמה השפה קלה להבנה.
2. Writability – כתיבה – עד כמה קל לייצר תוכנות בשפה.
3. Reliability – אמינות – עד כמה השפה מאפשרת/דואגת לאמינות בקוד.
4. Cost – עלות – כמה עולה לייצר קוד בשפה?
5. ניידות – עד כמה (אם בכלל) השפה ניידת?
6. שפה מוגדרת היטב – האם השפה ברורה וחדה?

כל הסעיפים הנ"ל מושפעים מהרבה דברים, ולפעמים באים אחד על חשבון השני. נרחיב:

1. Readability
   - Control statements – מבנים סינטקטים שאפשר לשלוט באמצעותם על זרימת הנתונים (if, else).
   - Data types structures - האפשרות להגדיר טיפוסי נתונים.
   - Syntax considerations – האם סינטקס השפה מובן. (שפה מובנת מהווה בעצם סוג של תיעוד לעצמה).
2. Writability
   - Simplicity and orthogonality- קל ללמוד ולכתוב בשפה שהיא פשוטה ואותוגונלית (= יש מספיק דרכים לבטא הכל אבל לא יותר מדי לכל דבר).
   - Support and abstract- באיזה סוגי הפשטה השפה תומכת (פונקציות). זה תורם לכתיבות כי לא צריך לכתוב קטע קוד שחוזר על עצמו הרבה פעמים
   - Expressiblity - כוח ביטוי של השפה –האם יש דרך (והאם היא קצרה ופשוטה) לבטא כל דבר.
3. Reliability
   - Type checking – שפה שתצליח "לתפוס" בעיות של השמת משתנה אחד במשתנה אחר נחשבת ליותר אמינה.
   - Exception handling – שפה שתומכת במנגנון exception נחשבת ליותר אמינה.
   - Alising – שפה שמאפשרת ששני דברים יצביעו לאותו מקום נחשבת פחות אמינה. ( reference נחשב יותר אמין, כי בניגוד למצביע אין לו אפשרות תזוזה..)
4. Cost – כמה עולה להתשמש בשפה? לכתוב בה קוד? כמה עולה לכתוב את השפה עצמה?
   הערות :
   - העלות של כתיבת קוד בשפה קשורה לwritability – ככל שהשפה יותר כתיבה, יהיה יותר קל ללמוד אותה ולפתח בה קוד, ולכן פיתוח הקוד יעלה פחות.
   - העלות תלויה גם באמינות – שפה יותר אמינה תדרוש קומפיילר שעושה יותר בדיקות – וזה יהיה כמובן יותר יקר. (זו דוגמא למצב שבו מאפיין אחד בא על חשבון השני).
5. ניידות – האם אפשר לפתח שפה בפלטפורמה אחת ולהריץ באחרת? שפה שאינה ניידת היא תלויית חומרה. (C++ לא ניידת, java כן)

**הערה:** Readability & Writability - מושפעים הרבה זה מזה והרבה מהמאפיינים של אחד מהם משפיע גם על השני (למשל סינטקס מובן עוזר גם לכתיבה..)

## מנגנון הטיפוסים:

המחשב לא יודע להבדיל בין פקודות, נתונים, שלמים או שברים וכד' – ולכן אנו מגדירים מנגנון טיפוסים שנותן משמעות לאוסף ביטים הנמצאים בחומרה.מתוך כך, פעולות הופכות לגיטימיות ומובנות. (למשל: אחרי שהגדרנו אוסף ביטים כמחרוזת, הוא לא ייתן להכפיל/לחבר את אוסף הביטים הזה).

שני היבטים למנגנון טיפוסים:
היבט מבני – איזה מבנה נותנים לאוסף ביטים.
היבט התנהגותי – איזה פעולות מוגדרות על פירוש השטח הנתון.

מה מרוויחים?
- טעויות מתגלות בזמן קומפילציה ולא מגיעים לקריסות בזמן ריצה.
- אופטימיזציה – בכמות הזכרון להקצאה לכל משתנה וכו' (ניצול טוב של הזכרון).
- הפשטה – כאשר מגדירים משתנה מסויים זה אוטומטית קובע איזה פעולות חוקיות לאותו משתנה ואיך הן מתבצעות.
- תיעוד (לא מהיתרונות העיקריים, אבל בכל זאת ☺) – כאשר מגדירים int x אז ברור שx הוא מסוג int!

**טיפוסים דינאמים וסטטים:**
1. Static typing - סוג המשתנה "מוחלט" בזמן קימפול (C/C++, Java, Pascal, Ada).
2. Dynamic typing - סוג המשתנה "מוחלט" בזמן ריצה (Lisp).
3. Duck typing - סוג המשתנה אינו משנה כל עוד שהוא תומך בפעולות אנו רוצים לבצע (python).

| | יתרונות | חסרונות |
|---|---|---|
| מנגנון סטטי | *בודק טעויות לפני ריצה<br>*חסכון בעלויות פיתוח | מונע / מסרבל הרבה מטלות (גורם לcasting , template .. ) |
| מנגנון דינאמי | גמישות:<br>*מאפשר ליצור אב טיפוס מהיר – לא רוצים משהו בטיחותי רק זמני לראות שזה עובד, או לבדוק חלק מפונקציה באופן מהיר.<br>*מאפשר פונקציות eval (פונקציות המאפשרות בניית קוד תוכ"ד ריצה). עצם זה שהטיפוסים לא מוגדרים עד זמן הריצה מאפשר להחליף קוד בזמן ריצה.<br>*Hot swapping of code - החלפת קוד בזמן ריצה. | לא ידוע מה מתקבל, אקראי, הרבה בדיקות תקינות.<br>אין שום מנגנון שבודק אם יש שגיאות בזמן קימפול ולכן יש הרבה יותר נפילות בזמן ריצה. הבדיקות נעשות (כמעט) רק ע"י הרצת כל המקרים האפשריים.. |

**מנגנונים חלשים וחזקים:**
- מנגנון חלש(weak): מאפשר המרה מרומזת של משתנים (בתנאי שהדבר הגיוני).
- מנגנון חזק (strong): לא מאפשר המרה מרומזת של משתנים.

<u>לשים לב:</u> אין קשר בין strong\weak ל static\dynamic. כל הקומבינציות בינהם אפשריות.

**טיפוסי נתונים ומימושם:**
1. שלמים (Int)
2. מניה (enum)
   - שלמים: למשל מגדירים מס' מ-1 עד 7 שמייצגים את ימות השבוע
   - סוג נפרד – שלא ממומש ע"י טיפוס אחר, מפתחים מנגנון מיוחד כדי ליישם זאת כסוג תוים נפרד.
3. בוליאני
   - שלמים: 0 – false כל השאר – true
   - Enum - 0 – false 1 – true
   - סוג נפרד - למשל ב++C יש סוג נפרד שנקרא BOOL ובו 0 מייצג FALSE ו- 1 מייצג TRUE.
4. תוים
   - סוג נפרד: יש אפשרות להגדיר תו – char
   - שלמים – למשל ב-C מגדירים מס' בין 128- ל127+
   - Enum – ניתן מ-0 עד 127 שם לכל מס', וכל שם יהיה אות.
5. תת סוג – לוקחים סוג מסויים ומצמצמים אותו (PASCAL). למשל: לקחת מה-Int את כל המספרים מ-1 עד 100.

**<u>מצביעים:</u>**
- מצביעים ובעיותיהם:

משתנה (שאינו מצביע) הוא שם שמסמן תא בזיכרון.
שם המשתנה הוא סטטי, ונקבע בזמן קימפול. אין דרך, (אולי חוץ מהאינדקס למערך), ל"חשב" שם של משתנה.
לעומת זאת: מצביע הוא משתנה שערכו הוא כתובת של משתנה אחר. כשנרצה לשלוף ממשתנה זה את ערכו, נצטרך להשתמש באופרטור אחר (לדוגמא " * " ב++C). חישוב בעזרת מצביעים מטפל בכתובות ולא בערך בזכרון.

- פעולות על מצביעים:

שתי פעולות מרכזיות:
1. השמה (assignment) – אתחול המצביע:
   - משבץ כתובת שימושית בתא.

- בזיכרון דינאמי המתודה שמקצה זיכרון יכולה לתת ערך.
- במיעון עקיף, צריך אופרטור (או משהו דומה) שיכול להחזיר את הכתובת של משתנה. (ב..C האופרטור הינו &).

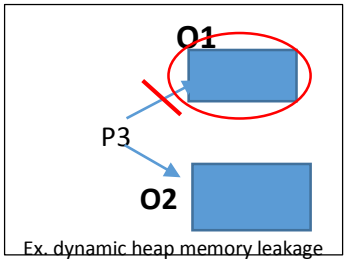
2. Dereferencing – אפשרות להגיע באופן עקיף למקום שהמצביע מצביע עליו.
יכול להיות ברור או מרומז:
<u>ברור</u>: בשפות בהן יש אופרטור מיוחד שמתייחס למצביעים( * , ->) [C]
<u>מרומז</u>: בשפות בהן מתייחסים אותו דבר לפוינטרים ומשתנים רגילים – אז רק ע"פ ההגדרה בהתחלה הקומפיילר יודע איזה ערך להציב. [ADA]

אפשר לתרגם שם של מצביע בשני אופנים:
- כמתייחס לתוכן שיש בתוכו (הכתובת של המשתנה השני):
במצב זה, השימוש בשם הינו זהה לשימוש בשם של כל משתנה מכל סוג שהוא.
- כמתייחס לתא עליו הוא מצביע:
במצב זה, שם המצביע הינו עוד דרך לגשת לנתון המוצבע – דהיינו מעון עקיף.
<u>זוהי הגישה שבד"כ מקובל לכנות כdereference</u>

- בעיות במצביעים:
1. מצביע מתנדנד (dangling pointer)
מצביע אשר איבד את הזיכרון עליו הוא מצביע (בדוגמה: P2)
למה זה מסוכן?
- יכול להיות שהזכרון שעליו הוא הצביע (ob) יוקצה למשתנה דינאמי אחר ואז:
1. אם הנתון החדש הינו מסוג אחר, ה - type checking לא יעבוד.
2. הערכים שלו ישתנו ללא קשר לעצם השימוש בו.
- יכול להיות שהמנגנון לניהול הזיכרון כרגע משתמש בזיכרון זה, ואז המנגנון יכול לקרוס (למשל אם יש שם מצביע לרשימת הבלוקים הריקים).

```
P1 = &ob ;
P2 = P1;
delete P1 ;
```
Ex. Dangling pointer

2. זליגת זיכרון דינאמי (dynamic heap memory leakage)
זיכרון דינאמי שהוקצה אך אינו נגיש בשום צורה (O1).
משתנים מסוג זה נקראים garbage (זבל), כיון שאינם שימושיים לתכליתם המקורית אך לא הוחזרו למאגר הזיכרונות הפנויים להקצאה מחודשת.
בעיה זו קיימת בשפות אשר דורשות שיחרור ידני של זיכרון שהוקצה (כמו ++C).
אם אין דרישה של שיחרור ידני של זיכרון, הבעיה לא יכולה להווצר כלל!
(כמו ב- Java, LISP ועוד)



Ex. dynamic heap memory leakage

<u>דוגמאות לפתרונות בשפות שונות:</u>

- **Pascal** - יש טיפול מפורש בשחרור זיכרון:dispose
בעיית המצביעים המתנדנדים: יש מימושים שמתעלמים מהפקודה כשהיא מופיעה- כלומר: אין מצב שמצביע יצביע על זיכרון ששוחרר והוקצה לשימוש אחר.[הסבר יותר מפורט (לפי הדוגמא הראשונה): המצביע שרוצים לשחרר(P1) הופך לnull, ובעקבות כך – כל המצביעים שמצביעים לאותו אובייקט (ob) הופכים גם לnull (P2)].
בעיית זליגת הזכרון: אין פיתרון.
- **ADA** – המצביעים נקראים: access types (זהו סוג שתפקידו להצביע על משתנה אחר).
בעיית המצביעים המתנדנדים: כשהמצביע מגיע לסוף דרכו (כלומר יוצא מה-scope), הזכרון שהוקצה דינאמית משוחרר ומוחזר לשטח הפנוי בצורה אוטומטית. (זה מצמצם את הבעיה כיון שהמקור המרכזי ליצירת הבעיה נובע משחרור זיכרון בצורה שאינה נכונה).
בעיית זליגת הזיכרון: אין פיתרון.
- **C\C++** - המצביעים חופשיים מאוד וללא הגבלות! אפשר להגיע (כמעט) לכל מקום, מה שיכול להיות מסוכן מאוד. אין בשפות אלה פתרון לאף אחת משתי הבעיות הנ"ל, ויותר מזה – היכולת לבצע פעולות אריתמטיות על מצביעים מאדירה את הבעיה!
אפשר לייצר מצביע גנארי שיכול להצביע על כל סוג: (זה שימושי בעיקר בפונקציות שמבצעות פעולות על קטעי זיכרון). **void *vPtr**
**= someOtherPtr;** (זה כמו להעתיק הרבה זכרון מבלי להתייחס לסוג).
הערה: זה לא יהרוס את הtype checking כי א"א לעשות dereferencing למצביע מסוג זה (כמובן שאפשר לעקוף את המנגנון, אבל...)

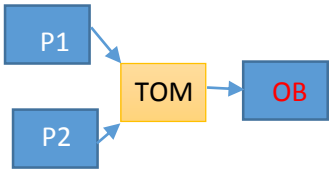
**<u>סוגי היחס - REFERNCE:</u>**

- ++C תומך בסוג נוסף של מצביעים: REFERNCE. זהו בעצם מצביע קבוע.
המצביעים האלו תמיד עוברים dereferencing לפני השימוש בהם, כלומר:
מאתחלים אותם בזמן יצירתם ומאותו רגע א"א לשנות את התוכן שלו.
(זה בעצם מה שמונע את שינוי הערך המוצבע).

```
int result = 0;
int &ref_result=result;
```

העברת פרמטרים לפונקציה כמצביעים מורידה את האמינות (כי אפשר לשכוח אופרטורים) והקריאות של הפונקציה, ולא מאפשרת תקשורת דו כוונית.
לכן, השימוש המרכזי בreference נעשה בעת העברת פרמטרים לפונקציות, ובכך מאפשרים קריאות,אמינות, ותקשורת דו כוונית.

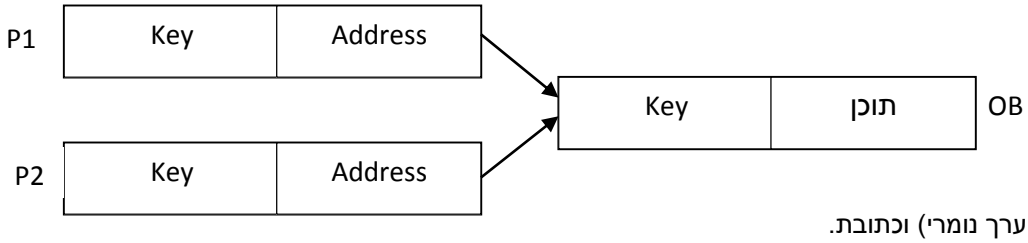
- בJAVA אין מצביעים בכלל – reference מחליף את השימוש במצביעים לחלוטין! כל המצביעים מוגדרים כאילו עברו כבר Dereferencing – כלומר אפשר להתייחס רק לתא המוצבע ולא למצביע בעצמו.
זה מונע אריתמטיקה על כתובות (כי זה לא הגיוני), זה לא מונע השמה (דרך הפוינטרים).
- #C - לרוב משתמשים בreference, אבל קיימים גם מצביעים "רגילים" – דורש הגדרת קטע unsafe. (הם הוכנסו לשפה בערך רק כדי לתמוך ב++C/C).




**פתרונות לבעיית המצביעים המתנדנדים (dangling pointer):**

1. **Tombstone** -
לכל משתנה דינאמי יש עוד תא, הנקרא tombstone, שהוא עצמו מצביע למשתנה.
המצביע האורגינאלי (זה שהקצאנו בשפה) אינו מצביע על הזיכרון עצמו אלא על ה – Tombstone
כאשר משוחרר משתנה דינאמי, הוא ממשיך להצביע על ה- tombstone שעכשיו כבר אינו מצביע כל המשתנה אלא ערכו nil. כך, מצביע לעולם אינו מצביע על זיכרון שכבר אינו קיים, ואפשר לגלות גישה ל..tombstone שהושם ל..nil ולטפל כיאות.
זו שיטה יקרה בזמן ובזיכרון. כל גישה למשתנה דינאמי דורשת עוד רמה של מיעון (בד"כ עוד cycle של המכונה), הזיכרון שהוקצה ע"י השיטה לעולם אינו משוחרר (באמת), אי אפשר להקצות את הזיכרון למישהו אחר כי הtombstone חייב להשאר nil (כלומר א"א לשחרר את תא הtombston),**אין דרך למחזר את הזיכרון ששוחרר!** (אנחנו מאבדים את תא הזכרון ברגע כי הtombstone כבר לא מצביע אליו).
קיצר, אין שפה פופולארית היום שמימשה אלגוריתם זה – כנראה העלות לא שווה את האמינות...

2. **Locks-and-keys**




כל מצביע מיוצג ע"י 2 ערכים: מפתח (ערך נומרי) וכתובת.
בכל פעם שמוסיפים מצביע לOB, נותנים לו את הkey של OB. לפני כל פעולה שנרצה לבצע דרך P1/P2 נוודא שהמפתחות עדיין זהים. אם הם זהים – הפעולה תתבצע. אחרת – שגיאה! הפעולה לא תתבצע.
ברגע שמוחקים את OB – שמים בkey שלו ערך לא חוקי, וכל נבטיח שהמצביעים לא יוכלו להגיע אליו..
כל העתק של המצביע למצביע אחר דורש העתקה של המפתח ג"כ.
באופן זה, אפשר לייצר כמה העתקים של המצביע לזיכרון מסויים
אם מצביע (אחר) מנסה לגשת לזיכרון לאחר ששוחרר, למרות שהכתובת שבתוכו תקיפה, המפתח כבר לא, והגישה תדחה.

## Scope:

```
int g()
{
   int x = 1;
}
```

```
int f()
{
...
}
```

בלוק המגדיר את המובן של ישויות (ביטויים, משתנים וכד') הנמצאים בתוכו.
בד"כ משתמשים במרחב זה על מנת להשיג הכמסת מידע
(ע"י הגבלת טווח הראיה והנגישות של נתונים מחלקים אחרים של התוכנית).
הscope של משתנה בעצם מגדיר איפה אפשר לראות אותו.
למשל: אם אנחנו נמצאים בפוקנציה f, אנחנו לא נכיר את המשתנה x שמוגדר בפונקציה g.(בהנחה שf וg זרות).

Scope דינמי וסטטי:

- סטטי – כבר בזמן קומפילציה (וע"פ המבנה הטקסטואלי של התוכנית) מזהים את המשתנים (לאיזה x מתכוונים..) .
- דינמי – רק בזמן הריצה מזהים את המשתנים.

קינון של תתי תוכניות
יש שפות המאפשרות לקנן הגדרות של תתי-תוכניות (פונקציה בתוך פונקציה).
משפחת מבוססי C אינה מאפשרת לקנן תוכניות.
לעומתן יש שפות אחרות שכן מאפשרות (lisp, python ועוד).
אם אין אפשרות קינון – אין מה לדבר על scope דינמי וסטטי.

דוגמא לקינון של תתי תוכניות:

```
define (f1, x, y)
    define (f2, z)
    ...
    )
    F3(x)
```

מדובר בx הזה
כי הם באותו scope

מימוש תוכניות מקוננות:

### Static scope:

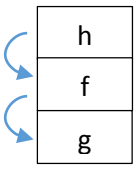
תמיד מחפשים קודם כל אם הנתון נמצא לוקלית(ואז אין בעיה לזהות אותו). אם לא – מחפשים בשרשרת הסטטית, כמו שיתואר להלן:
כאשר הנתון אינו מקומי (מובן שקיימת אפשרות לגשת אליו רק בשפות בהן יש אפשרות לקינון תתי תוכניות) יש צורך בשני שלבים:

1. מציאת הרשומה בה נמצא הנתון - הנתון חייב להיות במחסנית, אחרת הוא לא נגיש. ניתן לגשת לתת תוכניות רק אם תוכנית האב פעילה. (למשל בדוגמה למטה – אם h() פעילה אז וודאי שגם f() פעילה, ולכן וודאי שגם g() פעילה (כלומר כולן נמצאות במחסנית אחת אחרי השניה בסדר יורד)).
מחפשים במחסנית באב הסטטי ובודקים אם המשתנה מופיע. אם לא – מחפשים בסבא הסטטי. וכו'. כלומר – נחפש כל פעם ברשומה מעל, ונעצור ברשומה שבה מופיע המשתנה. נשים לב שהנתונים הנגישים הינם אלו שמוגדרים במעלה ההיררכיה (=שרשרת) הסטטית (הגדרת שרשרת סטטית בהמשך).

2. גישה לנתון הנכון ברשומה – בעזרת הoffset שמוגדר מראש נוכל לדעת איפה המשתנה מוגדר ולגשת אליו. (דבר זה דומה לגישה לנתונים לוקאלים).
הoffset מוגדר כזוג (chain,local), כאשר chain = כמה זזים בשרשרת כדי להגיע לרשומה הרצויה, ו-local = איפה הנתון נמצא בתוך הרשומה. (הערה: אם chain=0 הכוונה לרשומה הנוכחית).

איך נדע איפה האב הסטטי במחסנית ואיך ניגש אליו? בעזרת הstatic link.
השרשרת של כל הstatic links נקראת static chain.

static chain - שרשרת סטטית - רשימה משורשרת של מצביעים סטטיים
שכל אחד מהם מצביע מהבן לרשומת ההפעלה (כלומר: הנתונים) של אביו הסטטי.
(בראש השרשרת [=תחתית המחסנית] נמצא הmain והstatic link שלו מצביע לnull).




```
int g()
{ ...
   int f()
   {
      int h()
      { }
   }
}
```

Ex. Static chain

סורקים את השרשרת הסטטית מהרשומה הנוכחית עד לסוף המחסנית,
ועוצרים כשמוצאים את הרשומה שמכילה את הנתון הרצוי.
בניית השרשרת זה בעצם מה שעושים בזמן קימפול כדי שנדע לאן ללכת בזמן ריצה.

אפשר, בשעת קומפילציה, לדעת את אורך השרשרת שיש לעבור על מנת להגיע לרשומה הנכונה:

עומק קינון - רמת הקינון של הגדרת התת-תוכנית.

אורך השרשרת (chain offset) שיש לעבור הינו ההפרש בין עומק הקינון של תת-התוכנית המפנה לנתון לבין העומק של תת-התוכנית בתוכה הוגדר הנתון.

לדוגמא: אם רוצים להגיע מh לg (בדוגמא למעלה):

| | |
|---|---|
| רמת הקינון של h | 3 |
| רמת הקינון של g | 1 |
| אורך השרשרת שיש לעבור | 2 |

הגעה לנתון כלשהו בתוכנית תהיה בסדר גודל O(1).
אם המשתנה לוקלי – אז הגישה אליו פשוטה וזה ברור.
אם המתשנה נמצא במעלה ההיררכיה הסטטית – אנחנו יודעים כבר בזמן קומפילציה כמה צעדים נצטרך לעשות (chain), ומס' הצעדים הזה הוא מקסימום מס' רמות הקינון שיש בתוכנית שהוא מספר קבוע, ולכן ההליכה לרשומה הזאת היא O(1), וההליכה בתוך הרשומה היא כמו הגישה למשתנה לוקלי... סה"כ: O(1).

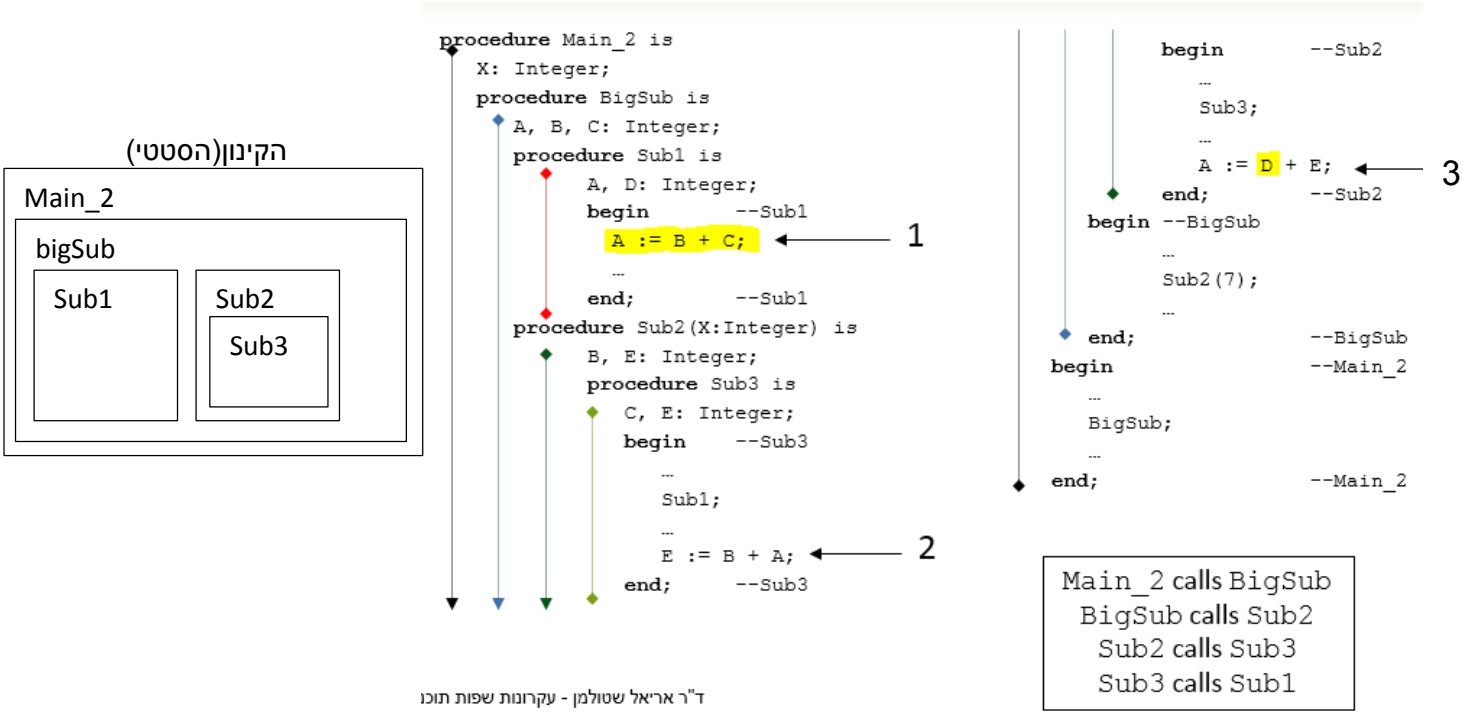
**רשומת הפעלה** - כל המידע שאנחנו שומרים במחסנית לפני קריאה לפונקציה כדי שנדע לאן לחזור (בתרגילים – LCL, ARG וכו'). ה static link הוא חלק מרשומת ההפעלה.

מה עושים כשיוצאים מפונקציה? פשוט מורידים (pop) את רשומת ההפעלה מהמחסנית והכל חוזר לקדמותו.

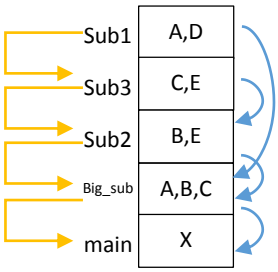
מבחינת קומפילציה וריצה - מה עושים כשקוראים לפונקציה? (זה קצת חוזר על מה שאמרנו קודם, רק כדי להדגיש!)

- בזמן קומפילציה -בודקים מהי רמת הקינון של תת-התוכנית הקוראת ביחס (ההפרש, כפי שהסברנו למעלה) לאב של תת-התוכנית הנקראת.
- בזמן ריצה - עוברים על השרשרת הסטטית של התוכנית הקוראת כערך עומק הקינון שחושב בשלב הקודם עד למציאת הרשומה הרצויה.

ההבדל בין שרשרת סטטית לשרשרת דינמית:




ד"ר אריאל שטולמן - עקרונות שפות תוכ




מחסנית הקריאות בשורה המסומנת ב-1 (וצבועה בצהוב) נראית כך:

השרשרת הסטטית (מסומנת בחיצים כחולים מעוגלים) בנויה לפי סדר הקינון בקוד.

ואילו השרשרת הדינמית (מסומנת בחיצים צהובים מרובעים) בנויה לפי סדר הקריאות.

אם נחפש את המשתנים C,B,A הנדרשים לשורה זו, נקבל משתנים שונים לפי צורת החיפוש:

| | סטטי | דינמי |
|---|---|---|
| A | Sub1 | Sub1 |
| B | Big_sub | Sub2 |
| C | Big_sub | Sub3 |

בשורה המסומנת ב3: כשנחפש את D – לא נמצא אותו לאורך כל ההיררכיה, ולכן זו שורה שגויה.

הסבר על הדינאמי – אנחנו נמצאים בsub2 ומחפשים את המשתנה D (D נמצא בsub1). כאשר אנו עולים המעלה ההיררכיה הדינאמית של sub2 (כלומר: מסתכלים ממנו ומטה במחסנית) sub1 לא מופיע שם! ולכן לא נמצא את D!!

חיסרון של השרשרת הסטטית - זמן הגישה למשתנים אינו קבוע. ככל שהמשתנה הרצוי רחוק יותר – יקח יותר זמן להגיע אליו..
<u>אבל</u> – לפי מחקרים גילו שבד"כ לא ניגשים למשתנים רחוקים ולכן זה בסדר, <u>ורוב התוכניות משתמשות בקינון סטטי</u> ולא דינמי.
שינוי באופן הקינון של תתי-התוכניות משנה את זמן הגישה למשתנה מסויים.
זהו חיסרון בעיקר למי שעוסק במערכות זמן אמת. קשה להעריך את זמן הביצועים ע"י הסתכלות על הקוד.

<u>קינון של בלוקים</u>

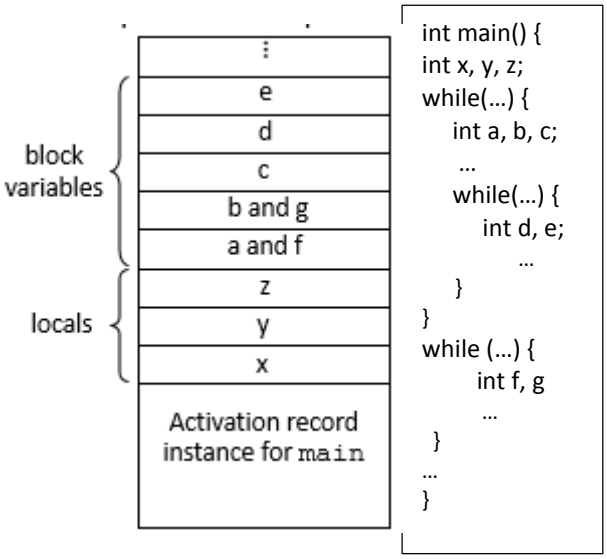
יש שפות שמאפשרות קינון בלוקים. במבוססות C זה קיים אע"פ שאין אפשרות לקנן תתי-תוכניות. הסמנטיקה של בלוק:

- בעת כניסה לבלוק יש גישה לכל המשתנים של הבלוק המכיל (בתנאי שלא נדרסו ע"י הגדרה מחודשת של משתנה בבלוק המוכל).
- כל המשתנים המוגדרים בתוך הבלוק משוחררים בסופו.

כיון שסמנטיקה זו היא בעצם אותה סמנטיקה כמו תוכניות מקוננות,
אפשר לממש אותה ע"י רשומת הפעלה לכל בלוק בתוספת קישור סטטי.

ישנה דרך פשוטה ויעילה יותר למימוש:
אפשר לקבוע בזמן קומפילציה את מקסימום הזיכרון שבלוק מסויים צריך (כולל הנתונים שהבלוקים הפנימיים שבתוכו צריכים).
זה מבוסס על העובדה שנכנסים ויוצאים מבלוקים ע"פ סדר טקסטואלי.
ע"י הגדרת היסט נכון, אפשר לגשת לכל המשתנים כאלו היו לוקאלים.

בדוגמא בצד –קוד, ואיך שהזכרון נראה עבור הקוד.




## dynamic scope

תמיד מחפשים קודם כל אם הנתון נמצא לוקאלית.
אם לא – מחפשים בשרשרת הדינאמית, כמו שיתואר להלן:
הפניה לנתון שאינו מקומי מצריכה שני שלבים:

1. מציאת הרשומה בה נמצא הנתון - הנתון חייב להיות במחסנית,
אחרת הוא לא היה נגיש.הנתונים הנגישים הינם אלו שמוגדרים במעלה ההיררכיה הדינאמית. הביאור הסמנטי של dynamic scope מחייב שאפשר לגשת רק לנתונים בתת-תוכנית שאקטיבית באותו הזמן.
2. היסט ברשומה לנתון הנכון (דבר זה דומה לגישה לנתונים לוקאלים).

מימוש הפניה לנתון שאינו מקומי נעשה באחד משני אופנים:

1. גישה עמוקה (deep access)

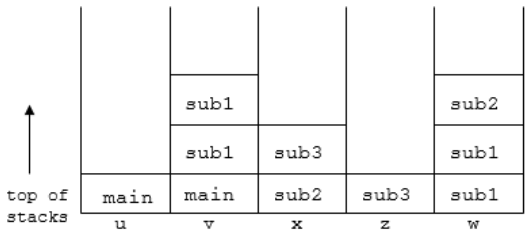
בשיטה זו, עוברים על השרשרת הדינאמית עד למציאת הנתון שמחפשים.
השרשרת הדינאמית מגדירה במדוייק את סדר הקריאות, ולכן מעבר עליה בצורה הפוכה נותן את הנתון הרצוי.
אין דרך לחשב את אורך המעבר הנצרך בזמן קומפילציה – צריך לעבור על הרשימה ולחפש בכל הרשומות את הנתון הרצוי.
הערה: ביחס לשפות סטטיות - מציאת נתונים שאינם לוקאלים דורשת יותר זמן . כמו-כן, יש צורך לאחסן ברשומות ההפעלה גם את שמות כל הנתונים.

2. גישה רדודה (shallow access)

בשיטה זו, משתנים המוכרזים בתוכנית אינם מאוחסנים ברשומת ההפעלה של התוכנית (אין מחסנית כללית לכל ה activation records) במקום זה, יש מחסנית נפרדת לכל אחד משמות הנתונים.
בין כה וכה אין אפשרות לגשת לנתון עם אותו השם מ scope רחב יותר.
בכל פעם בה מוגדר נתון בתת-תוכנית מקוננת, הנתון שלה מושם (push) במחסנית של השם הספציפי.
כאשר התת-תוכנית מסתיימת, הנתונים שהיא הגדירה מוצאים (pop) מהמחסניות והמצב חוזר לקדמותו כמו שהיה קודם הקריאה.

```
void sub3() {
    int x, z;
    x = u + v;
    ...
}
void sub2() {
    int w, x;
    ...
}
void sub1() {
    int v, w;
    ...
}
void main() {
    int v, u;
    ...
}
```

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3



דוגמא לסיכום: ההבדל בין static scope ל dynamic scope:

**Static scoping**

```
const int b = 5;
int foo()
{
   int a = b + 5;
   return a;
}

int bar()
{
   int b = 2;
   return foo();
}

int main()
{
   foo(); // returns 10
   bar(); // returns 10
   return 0;
}
```

**Dynamic scoping**

```
const int b = 5;
int foo()
{
   int a = b + 5;
   return a;
}

int bar()
{
   int b = 2;
   return foo();
}

int main()
{
   foo(); // returns 10
   bar(); // returns 7
   return 0;
}
```

בקריאה ל-()bar – הסטטי מחזיר 10 כי foo לא מכיר את ה-b של bar, ולכן הוא מתייחס לb const שמוגדר למעלה.
לעומת זאת הדינאמי מחזיר 7, כי הוא כן מכיר את המשתנים של "אבא" שלו – bar, ולכן במעלה ההירכיה הדינאמית הוא יפגוש קודם את ה- b של bar, שערכו כרגע 2.

**<u>מנגנונים לאחזור זיכרון:</u>**

טיפול בזיכרון – הגדרת הבעיה:

גישת התא הבודד – אם נניח שאפשר להקצות ולשחרר רק תאים בגדלים קבועים, נפשט את בעיית הטיפול בזכרון. ואם נניח שלכל תא יש מצביע (במקום קבוע) הבעיה תהיה עוד יותר קלה.
זהו הפיתרון שהרבה מימושים של LISP בחרו - כל תוכניות LISP ורוב נתוני LISP הינם תאים המקושרים באופן של רשימה משורשרת.
הגישה היא ע"י שרשור כל התאים לרשימה אחת המהווה את מרחב הזיכרון הקיים.
הקצאת זיכרון (דינאמית) נעשית באופן פשוט ע"י לקיחת כמות התאים הנצרכת מהרשימה.
שחרור זיכרון זה דבר מסובך - עצם זה שמצביע אחד מנותק, לא אומר שאין עוד מצביעים שמצביעים על התא המדובר. וחוץ מזה - קשה לדעת מתי תא כבר אינו מכיל נתונים הנצרכים לתוכנית.
ישנם שתי דרכים שמומשו בLISP.. (השפה הראשונה שהחליטה להעביר את מיחזור הזיכרון מידי המתכנת לידי המערכת):

- Reference counters – מיחזור הנעשה באופן מדורג
- Garbage collection – מיחזור הנעשה בעת מצוקה (כשנגמר הזיכרון).

**<u>איסוף זבל – garbage collector</u>:**

<u>בכל אלגו' GC צריך לעשות שני דברים:</u>

1. זיהוי האשפה – זיהוי כל מרחב הזכרון שהשתמשו בו ועכשיו הוא "מת".
2. מעבר על הזכרון הפנוי ושחרור כל הקטעים שזוהו כאשפה.

[מה היה קורה אם לא היה GC?]המנגנון המקביל (ואולי ההפוך) - מקצה זיכרון כשצריך ומנתק זיכרונות (איפוס מצביעים) כשצריך, ללא התייחסות למיחזור. **שיטה זו גורמת לזבל להאסף.** בעקבות כך: מרחב הזיכרונות הפנויים מתדלדל, ובשלב כלשהו הקצאת זיכרון חדשה תכשל למרות שאנו לא משתמשים (בזמן ההקצאה) בכל מרחב הזיכרונות האפשריים.

יש לנו שלושה סוגים של אלגוריתמי GC:

א. <u>Reference counting - מוני יחס</u> (שיטת מניית התייחסות היא סוג של GC )
לכל תא בזיכרון שומרים מונה שסוכם את מספר המצביעים שמצביעים אליו. בכל הקצאה או השמה של מצביעים, מקדמים את המונה. בכל שחרור של זיכרון, מפחיתים מהמונה.
כאשר המונה מתאפס, אין לתוכנית צורך בזיכרון זה יותר והוא משוחרר.
בעייתיות: אם גודל ה"תא" (המוצבע) קטן יחסית, העלות של שמירת מונה גבוהה מאוד (וזה לא משתלם).
כמו-כן, ניהול המונים דורש זמן.
בשפה כמו LISP שכל פעולה, כמעט, משנה מצביעים, זוהי עלות כבידה, אך אם המצביעים אינם משתנים לעיתים קרובות - העלות יורדת.
עוד קאצ' ממש בעייתי: אם יש אוסף תאים שיוצרים מעגל - בעצם כולם מיותרים, אבל אין למערכת דרך לזהות את זה.

ב. <u>Mark & Sweep – סימון ומחיקה:</u>
כשאין יותר זיכרון להקצות, מופעל אלגוריתם לאיסוף אשפה (garbage collector) שתפקידו לאסוף את כל אותם תאים שאינם שימושיים יותר.
לכל תא יש ביט המשמש כאינדקטור.
בתחילה כל הביטים מסומנים כזבל (למרות שיש תאים שזה לא נכון לגביהם בהתחלה).
כל קטע שניתן להגיע אליו דרך ה root set (מצביעים בתוכנית) – "נדליק" את הביט שלו, ואז בסוף כל התאים שנשארו לא פעילים נאספים לערימה חדשה כאזור של תאים פנויים שניתן להשתמש בהם.

בעיות בשיטה:

- הכי צריך אותו כשהתוכנית משתמשת ברוב הזכרון שקיים, ודווקא אז הוא לא עובד טוב, כי במצב הזה לוקח לאלגוריתם הרבה זמן למצוא את הזיכרונות הפנויים, וגם אז הוא מוצא רק מעט מקום פנוי.
- תגי הסימון גם תופסים מקום (אך בכמות הזיכרון שקיים היום ובתוספת זיכרון ווירטואלי זו פחות בעיה..).
- יש עלות זמן ריצה.
- בלוקים בגדלים משתנים - אם אפשר להקצות זיכרון ע"י לקיחת בלוקים בגדלים שונים, יש את כל הבעיות של הקצאה לא קבועה, וזה מוסיף עוד בעיות:
  1. סימון הבלוק כזבל (תחילת התהליך) - קשה לסרוק זיכרון כשלא יודעים את גודל הבלוק.
  2. סימון בלוק כ'לא זבל' - איך אפשר לעקוב אחרי הבלוקים אם אין דרך לדעת היכן המצביע שמקשר בין בלוק לבלוק??
  3. שמירת רשימה של בלוקים ריקים - לאט לאט הבלוקים מפוצלים עוד ועוד (תלוי בהקצאות) עד שבשלב מסויים יש הרבה בלוקים רקים קטנים, ואין אחד שמתאים לגודל הנצרך.

← יש פתרונות לכל השאלות: כמו הוספת גודל הבלוק בראש כל בלוק, הוספת מצביע מערכת לכל בלוק במקום קבוע, אפשרות לקיפול בלוקים רקים צמודים, אבל לכולם עלות.

שימוש ב-reference counting (מידע על הגודל של כל בלוק) פותר את שתי הבעיות הראשונות אבל לא את הבעיה השלישית.

ג. אלגוריתם צ'ייני (Cheney):

מחלקים את מרחב הזיכרונות לשני חלקים:

a. to – אליו כותבים.
b. from – ממנו מעתיקים בעת איסוף זבל.

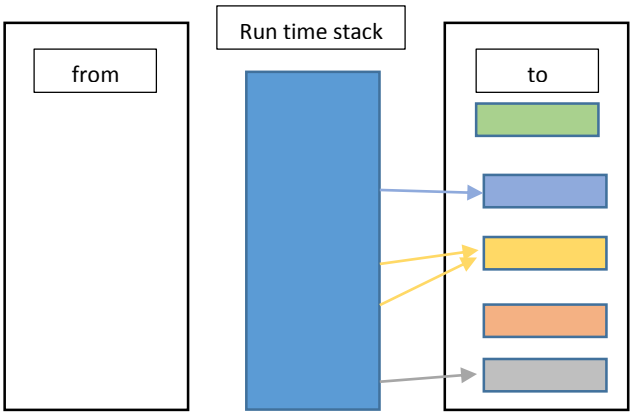
כשנגמר הזיכרון (בחצי שבשימוש – to)
מפעילים את האלגוריתם:

- הופכים את to ל from ולהיפך.
- עוברים על ה Run-time Stack ועוקבים אחר הפניות לזיכרון.

לכל אובייקט שנמצא במחסנית מבצעים:

i. אם האובייקט עדיין לא הועבר מfrom לto -
   1. מעבירים אותו.
   2. מעדכנים את הזיכרון עם הכתובת החדשה (אם יש עוד מצביע אליו - נדע לאן לפנות).
   3. מעדכנים את הזיכרון ב run-time stack.
ii. אם הוא כבר הועבר (כלומר יש יותר ממצביע אחד לאובייקט הזה) - מעדכנים את הרפרנס אליו (ב RTS) עם הכתובת החדשה.

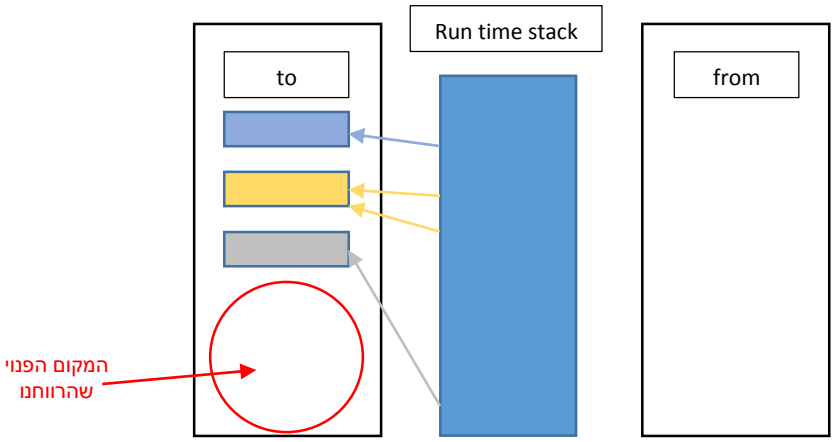
השמות באלגוריתם הם אחרי החלפת השמות מto ל from ולהיפך!!



הזכרון התמלא

לכל אובייקט שנמצא בזיכרון to שהפך לfrom מבצעים את הפעולות שתוארו לעיל (כך שאם יש הפניה לאובייקט שמפנה לאובייקט זה יתפס ג"כ).
ואז מה שקורה בעצם הוא שיש לנו חצי זיכרון בשימוש, שלא כולו מלא, כי העתקנו רק את מי שבשימוש, ומן הסתם היו חלק שלא בשימוש והם לא הועתקו.

הזכרון לאחר הפעלת האלגוריתם:




לאלגוריתמים העובדים בשיטה זו קוראים:
**two finger collector**
(כי בעצם צריך רק "2 אצבעות" שיצביעו לto ולfrom).
חיסרון: רק חצי מהזיכרון בשימוש ברגע נתון (בשונה מ mark & sweep).
יתרון: סורקים את הזיכרון רק פעם אחת (ב mark & sweep סורקים פעמיים).

כמובן שהפיתרון הכי טוב הוא שניהול הזיכרון יהיה בידי המערכת ולא בידי המתכנת. אם המתכנת לא יכול לשחרר זיכרון (כמו בשפות: c#,java ועוד) – לא יהיו בעיות של מצביע מתנדנד. (כדי שהמערכת תוכל לתמוך בשיטה כזו היא צריכה לדעת לזהות מתי זיכרון כבר אינו בשימוש ולשחרר אותו – אלגוריתמים של GC).

## תתי תוכניות – Sub-Programs:

ישנם שני סוגים של אבסטרקציה (הפשטה) שקיימים בעיצוב תוכנה:

1. הפשטה פרוצדוראלית (הפשטת תהליך) **procedural abstraction**– נותנים שם לקטע קוד שרוצים להשתמש בו הרבה פעמים כדי לחסוך ולא לכתוב אותו בכל פעם. למשל: פונקציה. (כבר קיים מזה שנים בשפות תוכנה, והוכר כחלק מרכזי בכל שפות התכנות).
2. הפשטת נתונים **data abstraction** - רק בשנות השמונים הוכר כחלק מרכזי בעקרונות התכנות (גרם להתפתחות תכנות מונחה עצמים).

<u>הפשטה פרוצדוראלית = תת–תוכניות - תכונות:</u>
לתת תוכנית יש נקודת כניסה אחת ובד"כ גם יציאה אחת. הערה: גם אם יש תת-תוכנית שיש בה כמה return למשל ע"י switch – ניתן לכתוב אותה כך שתהיה רק נק' יציאה אחת.
הקריאה לתת תוכנית משעה את הביצוע של התוכנית הקוראת לה. כלומר: יש רק תוכנית אחת שרצה בכל זמן נתון. השליטה תחזור לתוכנית שביצעה את הקריאה לאחר סיום ביצוע תת התוכנית.

יש 3 עקרונות לתכנות מובנה:
1. Sequence – סדרה של דברים לביצוע, בעלת כניסה אחת ויציאה אחת.
2. Selection – לכל סדרה יכולה להיות זרימת נתונים לפי תנאי כלשהו.
3. Iteration – כל סדרה יכולה להכיל איטרציות (while, do while, repeat).

כל תוכנית שנכתוב ניתנת לכתיבה ע"י שלושת המבנים העקרוניים האלו. (לא כולל exception שהוא מקרה מיוחד לטיפול בבעיה).

<u>הפרמטרים המועברים לתת התוכנית:</u>
תת-תוכנית מבצעת חישוב מסויים, ויש שתי דרכים להעביר לה פרמטרים:
1. משתנים גלובליים – לתת-התוכנית תהיה גישה למשתנים אלו, ואז הפונקציה לא מקבלת את הפרמטרים, אלא ניגשת ישירות למשתנים הגלובליים הרצויים.
חסרון: מוריד באמינות של תוכנית כיון שהם גם נגישים ממקומות אחרים. (יכול לתת פלט שונה לאותו קלט אם החישוב תלוי במשתנה הגלובלי והוא משתנה בינתיים).
הערה: בתכנות פונקציונאלי הערכים הם מסוג unmutable, לא ניתנים לשינוי לאחר הצבת ערך התחלתי – ואז אין בעיה של אמינות.
2. רשימת פרמטרים (נתונים שהועברו כפרמטרים נגישים דרך שמות מקומיים).
יתרון: גישה זו גמישה יותר (ויותר אמינה) - היא יכולה לקבל נתונים שונים מקריאה לקריאה ללא צורך בשינוי ידני של נתונים גלובלים.

המשתנים המוגדרים בתת התוכנית נקראים פורמאליים, והמשתנים שמועברים נקראים ממשיים.

שיטות לחבר בין הפורמאלים לממשיים:
- שיטת positional parameters - הקישור בין הפורמאליים לממשיים נעשה ע"פ **המיקום** – הפורמאלי הראשון מקושר לריאלי הראשון וכו'.
חסרון: שיטה זו טובה רק כשיש רשימה קצרה של פרמטרים. כאשר הרשימה ארוכה אפשר להתבלבל במיקום וכו'.
- שיטת keyword parameters - הקישור בין הפורמאליים לממשיים נעשה ע"פ **שם** הפרמטר הפורמאלי.דוגמא:
יתרון: אפשר להעביר את הפרמטרים בכל סדר שנמצא לנכון.
חסרון: חייבים לדעת את השמות הפורמליים בכדי לקרוא לפונקציה.

```
f(length=>myLength, list=>myArray, sum=>mySum)
```

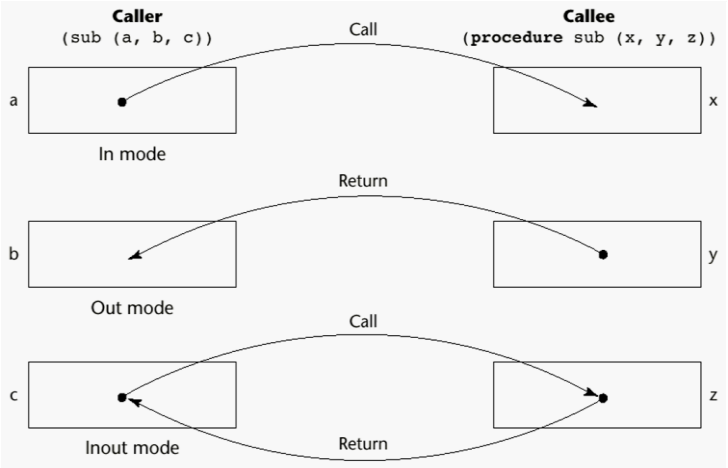
- חיבור בין שתי השיטות הקודמות: העברת פרמטרים ע"פ **שמם ומיקומם**.
הגבלה: אחרי שהועבר פרמטר ע"פ שמו כל שאר הפרמטרים יועברו גם ע"פ שמם.
- יש שפות שמאפשרות לתת ערכי ברירת מחדל לפרמטרים – זה מאפשר השמטת חלק מהפרמטרים מרשימת הפרמטרים הריאליים. יש שפות שמחייבות שהפרמטרים שמועברים כברירת מחדל יופיעו בסוף רשימת הפרמטרים הפורמאליים.

← דוגמאות בכמה שפות:
- Ada: אפשר לשים את ערכי ברירת המחדל בכל אחד מהפרמטרים.
הגבלה: אחר שאחד מהפרמטרים הושמט, כל שאר הפרמטרים חייבים לעבור על פי שמם.
הערה: אין צורך בפסיק (,) לציין פרמטר חסר.
- C++: כשאין קריאה על פי שם, החוקים חייבים להיות שונים:
1. פרמטרים עם ערכי ברירת מחדל חייבים להופיע בסוף הרשימה.
2. אין דרך להשמיט חלק מהפרמטרים - לאחר שהושמט פרמטר אחד , כל שאר הפרמטרים מושמטים ג"כ, ומקבלים ערך ברירת מחדל.
- ברוב השפות שאין ערכי ברירת מחדל, על הרשימות להתאים באופן מוחלט, למעט יוצאי הדופן: C,C++,Perl,JavaScript - בהם תפיקדו של המתכנת לדאוג שהפרמטרים שהושמטו לא יהרסו את הביצועים.
- #C מאפשר קבלת מספר לא מוגדר של פרמטרים, ובתנאי שכולם מאותו הסוג – לצורך כך צריך להגדיר את הרשימה כמערך מסוג params.

## העברת פרמטרים:

שתי דרכים להעברת פרמטרים:

1. By value - העברת ה**העתק** של הערך של הפרמטר (לפונקציה, בחזרה או לשני הכיוונים..).
2. By result - העברת **מסלול הגישה** לנתון המועבר כפרמטר (בד"כ הדבר מתבצע ע"י העברת מצביע עם הכתובת של הנתון)




**צורות להעברת פרמטרים:**

In – מקבלים פרמטרים

Out – מחזירים פרמטרים

In-Out – מעבירים דרכו ערכים ממי שקרא לו ומשתמשים בו כדי להחזיר ערך.

דרכים למימוש (הצורות הנ"ל) העברת פרמטרים:

- **Pass-by-Value** – מממש את שיטת in להעברת פרמטרים. בד"כ נעשה ע"י העתקת הערכים. אפשר היה להעביר כתובת אבל אז צריך לבדוק שלא כותבים אליה. בדר"כ תומך בטיפוסי נתונים בסיסיים (ב++C יש copy constructor כדי לאפשר העברה by value גם של אובייקטים).
  יתרון: גישה יעילה יותר לנתונים. שינוי ערך של משתנה בתוך הפונקציה של משפיע מחוץ לפונקציה.
  חסרון: המקום שנתפס ע"י המשתנים הנוספים, הזמן שלוקח להעתיק אותם.

- **Pass-by-Result** - מממש את שיטת out להעברת פרמטרים. התוצאה שמחושבת בפונקציה תוצב בפרמטר הזה. לא מועבר נתון לפונקציה. בשיטה זו,הפרמטר הפורמאלי מתנהג כפרמטר רגיל, ורק בסוף הבלוק הנתון מועבר חזרה לפרמטר הריאלי. בד"כ פרמטר התוצאה הריאלי מועבר על ידי העתקה (למרות החסרונות הקשורים). יש לוודא שלא ישתמשו בפרמטר לפני שנותנים לו ערך.
  בעיות:
  - אחת הבעיות המרכזיות קורית כאשר מעבירים את אותו הנתון הריאלי לשני פרמטרים פורמאליים שונים: sub (p1, p1) – איזה ערך יקבל p1? – את הערך שקיבל הפרמטר הראשון בsub או השני? האחרון מבין השנים שיועתק חזרה לפרמטר הריאלי ישמור את ערכו. ← כיון שהסדר אינו מוגדר, דבר זה מוריד בניידות של קוד.
  - יכול להיות שמימושים שונים יבחרו זמנים שונים לחשב את הכתובת של הנתון אותו יש להחזיר - נניח שחלק מחשבים את הנתון בעת הקריאה וחלק בעת ההעתקה, אם הנתון משתנה בתוך הפונקציה יחושבו שני דברים שונים. ← מוריד בניידות.

- **Pass-by-Value\Result** - מממש את שיטת in-out להעברת פרמטרים (חיבור של שתי השיטות הקודמות) – הפרמטר יועבר by value אבל גם ישמש להחזרת ערך.
  נעשה ע"י העתקה בתחילת ובסוף הפונקציה – אין דרך אחרת לממש את זה.
  חסרונות (זה בעצם החסרונות של שתי השיטות..): דורש זיכרון נוסף, תלוי בסדר העתקת הפרמטרים חזרה לנתונים.

- **Pass-by-Reference** - מממש את שיטת in-out להעברת פרמטרים. בשיטה זו אנחנו לא מעתיקים נתונים (פנימה והחוצה) אלא מעבירים את הכתובת שלהם. באמצעות כתובת זו אנו ניגשים למשתנה עצמו שנשלח לפונקציה (ולא להעתק שלו!) ומשתמשים בו לכל קריאה/כתיבה.
  יתרון: העברת נתונים יעילה בזמן ומקום. (אפשר להעביר אובייקט מורכב או מבנה ע"י מצביע כאילו מעבירים טיפוס בסיסי).
  חסרונות:
  - כל גישה לנתון הרבה יותר איטית (כי יש עוד רמה של מיעון).
  - אפשר לתת לשני פרמטרים פורמליים את אותו נתון ריאלי ובכך נוצרים שמות נרדפים לאותו נתון. ← מוריד את הקריאות ואת האמינות.

- **Pass-by-Name** – שיטה לא מצויה! אבל בכל זאת: הרעיון הוא שמחליפים כל איזכור של פרמטר פורמאלי בתוכנית עם הפרמטר הריאלי שלו. (הקישור נעשה בעת הגישה לנתון). צורת עבודה כזו מאפשרת lazy evaluation – פרמטר יוחלף רק כאשר יהיה בו צורך, וזה כמובן חסכוני בזמן.

נדגים בכמה שפות את הצורות להעברת פרמטרים:

1. Fortran - תמיד השתמשו ב inout אבל לא ציין האם יועבר value או reference.
   (רוב המימושים לפני 77 הועברו by reference. במימושים מאוחרים יותר פרמטרים פשוטים מועברים by-value-result)
2. C - נתונים מועברים by value. אפשר להשיג גם by reference ע"י העברת מצביע לנתון (זה קצת by value, כי הכתובת מועתקת ושינוי שלה בתוך הפונקציה לא ישפיע בחוץ).
3. ++C - דומה ל C, אך מוסיף עליו את האפשרות להעביר נתונים by reference ע"י שימוש באופרטור reference.
4. Java - נתונים מועברים by value. אך בכל זאת שתי נקודות חשובות על by reference בג'אווה:
   - כיון שלמעשה מועבר reference לנתון, מתקבלת תוצאה של העברה by reference.

- כיון ש reference ב java אינו יכול להצביע על נתון פשוט (scalar) ואין מצביעים, אין דרך להעביר נתונים פשוטים by reference.

5. Ada - ניתן להעביר בכל אחת משלוש הדרכים:

```
procedure Adder (A: in out Integer;   B: in Integer;  C: out Float)
```

- ערכים שהוגדרו in אפשר לקרוא מתוכם אבל לא לכתוב אליהם.
- ערכים שהוגדרו out אפשר לכתוב אליהם אבל לא לקרוא מהם.
- ערכים שהוגדרו in out אפשר הן לקרוא מהם והן לכתוב אליהם.

בשפות בהן יש כמה דרכים להעברת פרמטרים – מה צריכים להיות השיקולים שינחו אותנו מה לבחור?

1. **האם צריך העברה חד כיוונית או דו-כיוונית?**

בעקרון אנחנו רוצים לצמצם גישה למשתנים מחוץ לתת-תוכנית. לכן:

- נשתמש בהעברה פנימה (in) כאשר אין צורך להוציא נתונים.
- נשתמש בהעברה החוצה (out) כאשר אין צורך להכניס נתונים.
- נשתמש בהעברה פנימה והחוצה (in out) רק כאשר יש צורך בשני הכיוונים.

2. **יעילות:**

יש מקרים בהם היעילות מצדיקה שימוש בשיטת in out למרות שהתת-תוכנית אינה משנה את הנתונים (מקרה קלאסי בו צריך שימוש ב..in בלבד). כאשר הנתון המועבר הינו גדול במיוחד ושיטת in גורמת להעתקה (כמו ב pass-by-value), מקובל להעביר בשיטת in out (pass-by-reference) בכדי ליעל את התוכנית.

הערה: אם אפשר (וב ++C אפשר) יש לקבע את הנתונים באופן בו הקומפיילר ימנע שינוי לנתונים בתוך הפונקציה.

```
void fun (const int &);
```

## בדיקת סוג הפרמטרים:

כיום מקובל שיש לבדוק התאמת הפרמטרים הממשיים לפרמטרים הפורמאלים, כי ללא בדיקת התאמה, טעויות איות קטנות יכולות לגרום לבאגים שקשה מאוד למצוא (בעצם זה עניין של אמינות).
(שפות "עתיקות" דוגמת Fortran77 ו C לא בדקו התאמת פרטמרים, רוב השפות שהופיעו אח"כ בודקות, אבל יש אוסף של שפות חדשות ששוב אינם בודקות (Perl, JavaScript, PHP)).

התייחסות מיוחדת לבדיקת פרמטרים במשפחת ה-C:

בהתחלה C לא בדקה לא את מס' הפרמטרים או את סוגם.בC89 אפשר להגדיר את הפרמטרים עם או בלי בדיקה. (יש לזכור שגם אם הפרטמרים אינם תואמים לגמרי, עדיין יש אפשרות לקרוא לפונקציה, וההקומפיילר מבצע המרה אם הדבר מתאפשר. יש המרות שיגרמו לטעות – למשל float שיומר לint – כשנעשה פעולה חשבונית נקבל "שטות" כי לint ולfloat יש חלוקה שונה של ביטים).

```
C89:
                    ללא בדיקה:
double foo(x)
double x;
{...}
                    עם בדיקה:
double foo(double x)
{...}
```

ב C99 ו ++C כל הפרמטרים נבדקים הן בסוג והן בכמות.
אפשר להתגבר על בדיקת כמות הפרמטרים ע"י שימוש בערכי ברירת מחדל.
אפשר להתגבר על בדיקת סוג הפרמטרים ע"י שימוש בelipses .

## חפיפת תת תוכניות:

הקדמה: אופרטור חפוף הוא אופרטור שיש לו כמה מובנים. המובן של האופרטור ייקבע ע"פ הנתונים עליהם הוא פועל.
לדוגמא: אם האופרטור " / " יקבל שני integers, הוא יבצע חילוק ויחזיר את השלמים. לעומת זאת, אם הוא יקבל שני float (בעצם מספיק אחד בכמה שפות), הוא יבצע חילוק מלא.
איך זה נעשה? הקומפיילר בונה טבלה שמכילה מידע על ההעמסות השונות ומצביע לקוד הרלוונטי לכל אחת (חילוק של 2 int – לך לפה, חילוק של 2 float – הקוד נמצא שם..)

תוכנית חפופה: תת-תוכנית שיש לה את אותו השם כמו תת-תוכנית אחרת.
אנחנו צריכים לדאוג שכל תת תוכנית תזוהה באופן מוחלט! ניתן לעשות זאת ע"י:

1. רשימת הפרמטרים – סוג, כמות, סדר.
2. ע"י בדיקת סוג הערך המוחזר (אם מדובר בפונקציה).

ב Ada אפשר להבדיל בין הגירסאות של פונקציה ע"י הסוג המוחזר כיון שבתוך ביטוי אין אפשרות לערב בין סוגים.
ב C,C++,Java ועוד, שמאפשרות ביטויים עם סוגים שונים של נתונים, אין אפשרות להבדיל בין גירסאות של פונקציות ע"י הערך המוחזר.

```
void fun (float b = 0.0);
void fun();
...
fun();    //(דו משמעי) ambiguous call – compiler error
```

הערה – קאצ': תת-תוכניות חפופות עם ערכי ברירת מחדל לפרמטרים יכולות לגרום לקריאות בלתי ברורות – טעות קומפילציה.

## תת תוכניות גנאריות:

**תוכנית גנארית** הינה תוכנית שיכולה לקבל פרמטרים מסוגים שונים ולבצע את המטלה באותו אופן על כולם.
מה הקטע? שימוש חוזר בקוד הינו אחד מהתורמים המרכזיים להגדלת הייצור של תוכנה.
אחת מהאפשרויות להגדיל שימוש חוזר הוא ע"י הפחתת הצורך לכתיבת תת-תוכניות שונות לאותו אלגוריתם הפועל על סוגי נתונים שונים.
(למשל: אין צורך שיכתבו פרוצדורת מיון לשלמים, שברים, וכו' – אפשר לכתוב לכולם את אותו הדבר!!).
בשפות בהם הקישור לפונקציה מתבצע באופן דינאמי, אין צורך להגדיר את סוג הנתונים.

- דוגמא: פונקציות APL

בשפות בהם הקישור לפונקציה הינו סטטי, צריך מנגנון אחר על מנת לממש תיכנות גנארי.

```
template <class T>
T max (T first, T second)
{return first>second ? first : second; }
```

דוגמא לתכנות גנארי ב++C :
בכל מקום שבו נכתב בקוד T הכוונה לtemplate שהגדרנו בהתחלה.
בקוד הזה, ההנחה היא שהסוג שיוצב בT תומך באופרטור " < ".
אם נקרא לפונקציה max עם integers – אז T יהיה int , וכו'.
הערה: הקומפיילר מייצר קוד שונה לכל קריאה עם סוג משתנה שונה (מס' העותקים כמספר **סוגי** הקריאות).

אפשר היה לממש את הפונקציונאליות הזאת ע"י מאקרו: #define max(a,b) ((a)>(b)) ? (a) : (b) (אם a גדול מb תחזיר את a, אחרת תחזיר את b). זה תיכנות גינארי שכמובן יעבוד על כל סוג.
הבעיה היא שהוא לא יעבוד כראוי אם יש תופעות לוואי בפרמטרים,
למשל: max(x++,y) יתורגם ל: ((x++)>(y)?(x++):(y)) דבר שיגרום ל x להתקדם פעמיים בכל פעם שהוא גדול מy.

ב Ada ישנה אפשרות לתכנות גינארי. ל Java ו #C נוספה האפשרות לתיכנות גינארי לאחרונה.

## פונקציות – נושאי עיצוב:

1. תופעת לוואי של פונקציות:
שפה מאפשרת תופעות לוואי של פונקציה שלא רק מחזירה ערך, אלא עושה עוד דברים מאחורי הקלעים ומשתמש הפונקציה לא יודע מזה (בלי קשר לקלטים שמועברים לפונקציה).
למשל: אם פונקציה משתמשת במשתנה גלובלי, והשפה מאפשרת שינוי שלו – זה גורם לבעיות, כי בכל פעם שנקרא לפונקציה עם אותו קלט נקבל פלטים שונים!
כדי למנוע תופעות כאלה: אם הצורה היחידה להעברת פרמטרים לפונקציה היא in וההעברה by value, מה שקורה בפונקציה לא משפיע החוצה ולכן אין בעיה של תופעות לוואי.
כמו כן, אם משתמשים במשתנים גלובליים שלא ניתנים לשינוי – הפונקציה תהיה ללא תופעות לוואי, כלומר **פונקציה טהורה**.
אם בקריאה לפונקציה, חוץ מהחזרת פרמטר הפונקציה גם עובדת עם קובץ (קוראת/כותבת) – יש שמחשיבים זאת לתופעת לוואי (כי תופעת לוואי = כל דבר שהפונקציה גורמת חוץ מהערך שאותו היא מחזירה).
יש להעביר את הנתונים לפונקציה בשיטת in בכדי למנוע תופעות לוואי בביטויים שמשתמשים בפונקציות. רוב השפות אינן דורשות העברה כזו, ותופעות לוואי יכולות לקרות. יש שפות שמצריכות העברה כזו (למשל: Ada).
2. סוגי הערכים המוחזרים:
רוב השפות (האימפרטיביות) מגבילות את סוגי הנתונים שפונקציה יכולה להחזיר.
   - C מאפשרת החזרת כל סוג מלבד מערך ופונקציה (אפשר לטפל בשני המקרים ע"י החזרת מצביע).
   - ++C מאפשר גם החזרת משתנים שהמשתמש יצר (class).
   - Ada מאפשרת החזרת כל סוגי התונים (אך כיון שפונקציה אינה "סוג" ב..Ada, אין אפשרות להחזיר פונקציות).

הערה: אין אפשרות לכתוב פונקציות ב Java וב #C. המתודות של האובייקטים דומים לפונקציות והן יכולות להחזיר כל סוג שמוגדר.. כיון שמתודות אינן "סוג", אין דרך להחזירם ממתודה.

## חפיפת אופרטורים:

יש שפות המאפשרות לחפוף את האופרטורים של השפה. ההבדל בין אופרטורים לפוקציות רגילות מתבטא באופן הקריאה - פונקציה רגילה הינה prefix c = foo(a,b); ואילו אופרטור הינו infix c = a*b;.
השאלה היא עד כמה לחפוף אופרטורים? כמה חפיפה עדיין נחשבת "טובה", או האם יש חפיפה "יותר מדי"?
יש חפיפות שיגרמו לתוכנית להיות פחות קריאה. למשל: c = a * b; האם מדובר בכפל של משתנים רגילים, כפל מערכים או אופרטור נקודה של מערכים?

## רקורסיה וארכיטקטורת מחסנית:

סמנטיקה של מימוש תתי-תוכניות:
קריאה לתת-תוכנית בד"כ כוללת:
- העברת פרמטרים (בכל שיטה שנבחרה).
- אם המשתנים הלוקאלים אינם סטטיים צריך להקצות זיכרון.
- שמירת מצב התוכנית הקוראת.
- העברת השליטה לפונקציה הנקראת ובניית מנגנון שיוכל להחזיר את השליטה בסוף למקום הנכון.
- בניית מנגנון גישה לנתונים שאינם לוקאלים (רלוונטי בעיקר אם אפשר לקנן הגדרות לפונקציות).

ביציאה מתת-תוכנית צריך:
- להחזיר ערכים (ובאם הפרמטרים הוגדרו בשיטת out צריך להעתיק אותם לפרמטרים הממשיים ע"י העתקתם מהפרמטרים הפורמאלים).
- שיחרור הזיכרון שהוקצה.
- החזרת השליטה לתוכנית הקוראת (כולל החזרת מצב המשתנים, פקודה לביצוע, ועוד).
- אם תומכים בהגדרות מקוננות - החזרת שרשרת ההגדרות לקדמותה (כמו שהיתה קודם הקריאה).

"תוכנית פשוטה" - תוכנית שאי-אפשר לקנן הגדרת תתי-תוכניות וכל המשתנים הינם סטטיים.
על מנת לבצע קריאה/חזרה בתוכניות פשוטות צריך זיכרון בשביל:
- שמירת נתונים של התוכנית הקוראת.

- פרמטרים.
- כתובת חזרה (של התוכנית הקוראת).
- ערכי פונקציה (מה שהיא מחזירה).

תוכנית פשוטה מורכבת משני חלקים:
- הקוד לביצוע - חלק סטטי, שאינו משתנה.
- הנתונים הלוקאלים ואלו שהוגדרו קודם - יכולים להשתנות בעת ביצוע התת-תוכנית.

שני החלקים הינם קבועים בגודלם, ואפשר לחשב אותם כבר בשלב הקומפילציה.

**activation record** - מבנה וסידור הנתונים. הנתונים **מתארים רק מה שרלוונטי בעת ההפעלה של התוכנית**. הצורה (מבנה) של רשומה זו היא סטטית.
**activation record instance** - מופע של רשומה זו למקרה ספציפי.

בזמן נתון, (בתוכנית פשוטה), יכולה להיות רק רשומה אחת מכל תת תוכנית פשוטה (ולכן כמובן שאין תמיכה ברקורסיה).

מבנה רשומת ההפעלה:

| |
|---|
| משתנים מקומיים |
| פרמטרים |
| כתובת החזרה |

הקדמה קטנה: **משתנים דינאמיים** [אוטומטיים] = משתנים שיוצאים ונכנסים מהמחסנית – הם מוקצים רק כשצריכים אותם [כשמגיעים לבלוק שבו הם מוגדרים] ונמחקים כשגומרים להשתמש בהם [ביציאה מהבלוק].
לעומת זאת, **משתנים סטטיים** = משתנים שמוגדרים מההתחלה, וקיימים לכל אורך התוכנית.
בתוכנית שלא תומכת במשתנים דינאמיים, יש רק עותק אחד לכל רשומת הפעלה, ואילו בתוכנית שכן תומכת במשתנים דינאמיים, אפשר להגדיר כמה עותקים של רשומת הפעלה, ולכן אפשר לעשות רקורסיה.

<u>מימוש תתי תוכניות עם משתנים דינאמיים:</u>
נושא זה רלוונטי רק לשפות שתומכות במשתנים דינאמיים בזכרון.
יתרון גדול בשפות כאלה – אפשרות לרקורסיה.
הקישור בין תתי-תוכניות בשפות אלו מורכב יותר כי:
1. הקומפיילר צריך לייצר קוד להקצאה ושחרור משתנים בזמן ריצה.
2. רקורסיה מוסיפה את האפשרות שאותה תת-תוכנית רצה כמה פעמים בו זמנית, ולכן צריך יותר מרשומת הפעלה אחת לכל תת-תוכנית. (לכל רשומת הפעלה צריך העתק של הנתונים - הן הפרמטרים הפורמאליים, הן הנתונים הלוקאלים, והן כתובת החזרה).

מבנה רשומת ההפעלה לתת-תוכנית נתונה בד"כ (כשתת התוכנית לא תומכת במשתנים דינאמים) ידוע בשלב קימפול, אך יש שפות (כמו ADA למשל) בהן הגודל לא ידוע, כי המשתנים יכולים להיות בגדלים שונים (מערכים) ונקבעים בזמן ריצה.
את רשומת ההפעלה אנחנו מייצרים בזמן ריצה באופן דינמי.

מקובל להשתמש במחסנית(run – time - stack) למימוש הפעולות (כי אנחנו רוצים לחזור למקום ממנו יצאנו).

בכל קריאה לתת-תוכנית יוצרים מופע חדש של רשומת ההפעלה במחסנית (push), וכך מאפשרים שמירת העתק נתונים נפרד לכל הפעלה של התת תוכנית.

בסוף ההפעלה, מוציאים את רשומת ההפעלה (pop) מהמחסנית, ובכך המצב חוזר לקדמותו כפי שהיה לפני הקריאה.

| <u>**מבנה רשומת ההפעלה של תתי תוכניות דינאמיות:**</u> |
|---|
| משתנים מקומיים |
| פרמטרים |
| קישור דינאמי – מצביע על רשומת ההפעלה הקודמת * |
| כתובת החזרה – כוללת: מצביע לקוד של התוכנית הקודמת והיסט בתוך הקוד להצביע על הפקודה הבאה |

*יכול להיות שיש עוד דברים שהושמו בדרך על המחסנית (כמו משתנים לניהול המנגנון) וע"כ צריך לדעת היכן הרשומה הקודמת

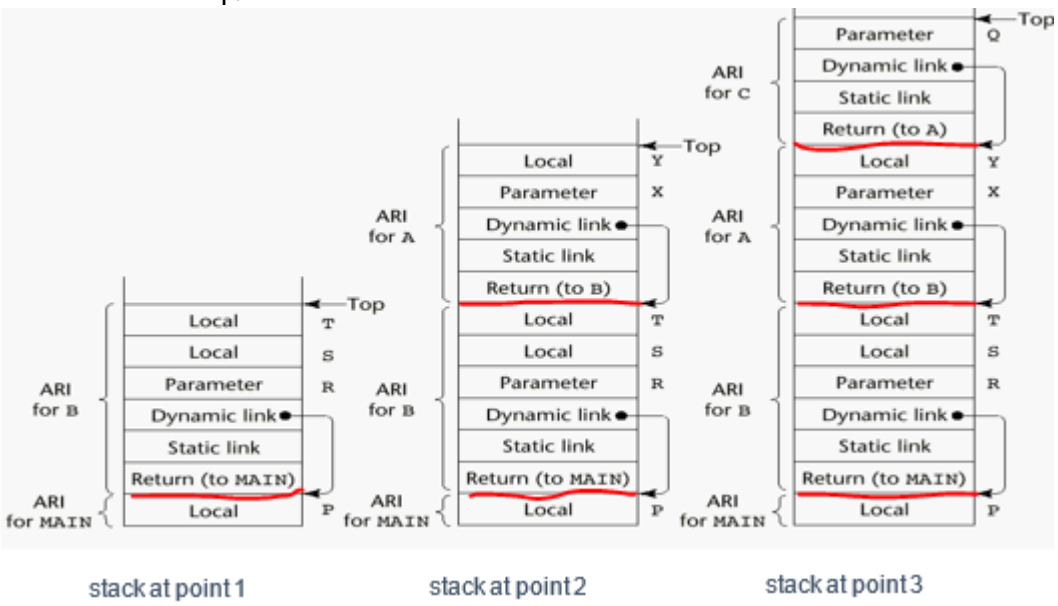
דוגמא לרשומות הפעלה בזמן ריצה של קוד:

```
void A(int x)
{
      int y;
      ...   <---- 2
      C(y);
      ...
}
void B(float r)
{
      int s, t;
      ...   <---- 1
      A(s);
      ...
}
```

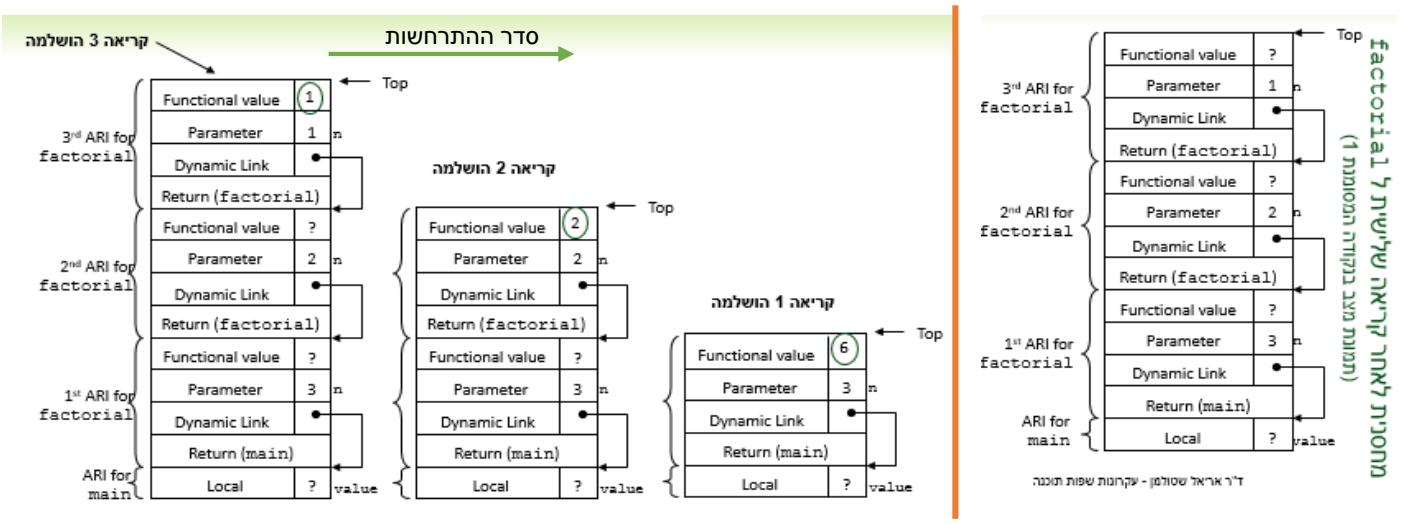
```
void C(int q)
{
   ...   <---- 3
}
void main()
{
   float p;
   ...
   B(p);
   ...
}
```

main calls B
B calls A
A calls C

מצב המחסנית בכל אחת מהנקודות המסומנות - רואים שכל קריאה לפונקציה הוסיפה רשומת הפעלה מתאימה.:




במקרה של פונקציה רקורסיבית, אופן הפעולה של רשומת ההפעלה דומה:




אנחנו חוזרים כאן שוב לשרשרת הדינאמית (dynamic chain) – שמייצגת את ההסטוריה הדינאמית של התוכנית ומראה כיצד התוכנית הגיעה לנקודה הנוכחית.

יש לנו מצביע רק לתחילת הרשומה, אז איך נגיע למשתנים הלוקאליים? מחשבים את ההיסט (local offset) מתחילת רשומת ההפעלה. בד"כ אפשר לחשב אותו בזמן קומפילציה.

**כאשר הפונקציה מחזירה ערך, רשומת ההפעלה צריכה לכלול אותו:**

| |
|---|
| **Functional value** |
| **Parameters** |
| **Dynamic Link** |
| **Return Address** |

בסיום ביצוע הפוקציה יועתק הערך ממקום זה לזיכרון של הרשומה הקודמת.

# תכנות מונחה עצמים OOP:

כדי לאפשר הפשטת נתונים עלינו לכלול (ביחידה אחת) נתון מסוים ואוסף הפעולות שאפשר לבצע עליו.
על ידי שליטה על הגישה, אפשר להסתיר פרטים מישויות חיצוניות. (ישויות חיצוניות יכולות להגדיר נתונים ולהשתמש בהם באופנים שהוגדרו מבלי לתת את הדעת על צורת שמירת הנתונים מאחורי הקלעים).

לנתונים מסוג זה קוראים **abstract data type** או בקיצור **ADT**.

היתרון המרכזי מאחורי הפשטת מידע הוא האפשרות להשתמש בו מבלי לדעת כיצד הוא פועל. עצם זה שלקוח של נתון מופשט אינו "רואה" את המימוש, מונע ממנו מלכתוב קוד שתלוי באותו מימוש. דבר זה מאפשר שינוי מימוש פנימי מבלי לפגוע בלקוחות של הנתון.
שמירת כל חלקי הנתון במקום אחד מאפשרת תמיכה קלה בטיפוס הנתונים - כל עדכון דורש שינוי במקום אחד בלבד.

## נושאי עיצוב – טיפוסים מופשטים:

ישנם שני נושאי עיצוב:

1. האם לאפשר טיפוסי נתונים אבסטרקטים עם פרמטרים?
2. איזה אפשרויות גישה לחשוף, וכיצד מפעילים גישות אלו?
   - יש פונקציונאליות שנצרכת ע"י רוב טיפוסי הנתונים:
     - השמה
     - שוויון או אי-שוויון (צריך להפריד בין שוויון עמוק [ = האם תוכן האובייקטים זהה] לשוויון רדוד [ = האם הכתובות של 2 האובייקטים זהות])
     - פונקציות בונות, הורסות, איטרטורים, ועוד - פונקציות אלו שונות מטיפוס נתונים אחד למשנהו, וע"כ קשה (אם בכלל אפשרי) לכתוב מימוש גינארי לכולם.

דומאות בשפות שונות:

**Ada:**

ב Ada אין תמיכה פרטנית לטיפוסי נתונים מופשטים. יש חבילות (package) שמאפשרות להגדיר טיפוסים חדשים ולהסתיר מימושים (כפי שצריך ב ADT), אבל זה כלי רחב יותר. אפשר להגדיר כמה טיפוסים בתוך חבילה אחת.
החבילות מחולקות לשנים:

1. חבילת הגדרות (specification package).
2. חבילת מימושים (body package).

חבילת ההגדרות עצמה מחולקת לשניים:

א. החלק הנגלה:
- מגדיר את שם הנתון החדש.
- מכריז עליו כטיפוס נסתר. (אם לא מסתירים (מכמיסים) את המימוש, זה לא ADT)

ב. החלק הנסתר:
מגדיר את מבנה הטיפוס החדש (למה צריך שזה יופיע גם בחבילת ההגדרות? כדי שקומפיילרים יוכלו לקמפל ביחד עם הלקוח של הטיפוס, ללא הסתכלות בחבילת המימוש).

ישנם שני סוגי הכמסות:

1. private
2. limited private

ההבדל ביניהם הוא האם הקומפיילר מגדיר (כחלק מהטיפוס) את האופרטורים להשמה ובדיקת שוויון (ואי-שוויון) -

1. ל private הקומפיילר מגדיר נתונים אלו.
2. ל limited private הוא לא מגדיר אותם וצריך להגדיר אותם לבד.

זה שימושי במקרה שההתנהגות הטיפוסית לא מתאימה - אובייקטים שאין דרך לבצע פעולות אלו (מחסנית) ואובייקטים שצריך העתקה עמוקה.

```
package Stack_Pack is
        type stack_type is limited private;
        max_size: constant := 100;
        function empty(stk: in stack_type) return Boolean;
        procedure push(stk: in out stack_type; elem:in Integer);
        procedure pop(stk: in out stack_type);
        function top(stk: in stack_type) return Integer;
        private  -- hidden from clients
        type list_type is array (1..max_size) of Integer;
        type stack_type is record
                list: list_type;
                topsub: Integer range 0..max_size) := 0;
        end record;
end Stack_Pack
```

**C++:**

- ארגון נתונים (encapsulation):
C++ חושפת כלי יעודי לארגון ה ADT: מנגנון המחלקה.
  - נתונים במחלקה נקראים data members.
  - המתודות במחלקה נקראים member functions.
  - לכל אחד מהאובייקטים של המחלקה יש העתק של הנתונים של המחלקה (data members), אך לכל האובייקטים יחד יש רק העתק אחד של הפונקציות של המחלקה.( זה מספק כדי לתפוס את השוני בין 2 אובייקטים ללא בזבוז זיכרון מיותר).

השפה מאפשרת יצירת אובייקט הן על המחסנית והן בערימה.
- הכמסת נתונים:
השפה מאפשרת חידוד הגישה לנתונים לשלשה רמות:
  - Public - מאפשר גישה לכל קוד בכל scope.
  - Protected - מאפשר גישה רק לקוד שהינו בשרשרת הירושה, פונקציות חברות של המחלקה או friends.
  - Private - מאפשר גישה לפונקציות חברות של המחלקה או friends.
- פונקציות בונות והורסות:
אפשר לכלול פונקציות בונות (חפופות) שדואגות להכניס את הנתונים למצב עמיד (stable). פונקציות אלו נקראות אוטומטית בעת הגדרת האובייקט. כמובן, לכל גירסה צריכה להיות חתימה ייחודית (תלויית פרמטרים).
פונקציה הורסת תפקידה "לנקות" אחרי האובייקט. היא נקראת אוטומטית בעת שחרור האובייקט ומשחררת משאבים שנתפסו ע"י האובייקט. בד"כ משתמשים בה גם כדי להדפיס ערכים לצורכי מעקב.

```
class stack {
        private:
                int *stackPtr, maxLen, topPtr;
        public:
                stack() { // a constructor
                        stackPtr = new int [100];
                        maxLen = 99;
                        topPtr = -1;
                };
                ~stack () {delete [] stackPtr;};
                void push (int num) {…};
                void pop () {…};
                int top () {…};
                int empty () {…};
}
```

**השוואה בין Ada ל++C:**

- שניהם תומכים ב ADT עם אותם יכולות.
- שניהם חושפים יכולות הכללה והכמסה.
- C++ תומך יותר בצמצום הגישה למאפיינים לפי הצורך.
המאפיינים אינם סגורים (private) או פתוחים (public), יש גם אפשרויות ביניים.
כמו-כן, יש אפשרויות גישה לישויות מחוץ למחלקה (אם צריך) ע"י friend functions.

**Java:**

התמיכה של java ב ADT דומה ל C++ בשינויים קלים:

- כל האובייקטים מוקצים על הערימה ונגישים דרך reference variables.
- לכל ישות במחלקה (מאפיין או מתודה) מצורף מאפיין שמגדיר את אפשרויות הגישה.
- אפשר לייצר רמת גישה אמצעית (דומה ל friend) ע"י שימוש בחבילות (packages). כל ישות שאין לה מציין גישה, נגישה לכל מי שהוגדר בחבילה.
- אין פונקציה הורסת (אין צורך לשחרר נתונים).

<u>תכנות מונחה עצמים – נושאי עיצוב:</u>

- **בלעדיות של אובייקטים** – כמה אפשרויות:
  - כל ישות בשפה הינה אובייקט (Smalltalk) - מ integer הכי קטן ועד האובייקטים הכי גדולים.
  יתרונות: מבנה אחיד לשפה ופשטות מבנית (קל ללמוד ולתפעל).
  חסרונות: גם פעולות פשוטות (שבד"כ אפשר לממש ע"י פקודות מכונה) צריכות להתבצע ע"י מנגנון העברת הודעות. בד"כ זה יוצא איטי יותר.
  - הוספת מבנים לשפה אימפרטיבית שכבר קיימת (C++, Ada).
  יתרונות: אפשר לבחור (לצורכי יעילות וניהול מורכבות) בין שתי אפשרויות.
  חסרונות: שפה גדולה - מבנה הסוגים מבלבל, וקשה ללמידה.
  - לאפשר גישה רגילה רק לסוגים בסיסיים, כל השאר אובייקטים (Java).
  יתרונות: נותן את מהירות הביצועים לסוגים פשוטים שמצפים משפה אימפרטיבית רגילה.
  חסרונות: ערבוב של סוגים (בסיסיים ואובייקטים) גורם לבעיות (צריך ליצור מחלקת "עטיפה" כדי שיעבוד כמו שצריך).
- האם תת-מחלקה (מחלקה שיורשת ממחלקה אחרת) מתפקדת גם כתת-סוג?
  - בד"כ קיים יחס של IS-A בין יורש ומוריש (למשל: A circle IS-A shape, A car IS-A vehicle).
  - האם יחס זה מכריח שבכל מקום בו אפשר להשתמש באובייקט המוריש נוכל להשתמש במקומו באובייקט היורש?
  כמובן, מבלי שייווצרו טעויות בהקשר להתאמה לסוגים. sub type של Ada מהווה דוגמא לזה (ללא אובייקטים):
  **subtype** Small_Int **is** Integer **range** -100..100;

## עד כאן מה שנלמד בהרצאות כנראה.... ☺

- בדיקת התאמת סוגים בעת ריבוי צורות
  - ריבוי צורות גורם לכך שאין דרך לדעת בצורה סטטית (בעת קימפול) איזה פונקציה תקרא למעשה. דבר זה גורם לכך שאם רצוננו לאכוף בדיקת סוגים, זה צריך להתבצע באופן דינאמי, וזה יקר יותר – מתבצע בזמן ריצה וטעויות נמצאות מאוחר יותר.
  - אפשר לעקוף את הבעיה אם נכלול סייג שפונקציה פולימורפית צריכה את אותו השם ואת אותה החתימה ביורש כמו במוריש, ואת זה אפשר לבדוק באופן סטטי (כבר בעת קימפול).

- ירושה יחידנית או ירושה מרובה
  ירושה מרובה מאפשרת למחלקה לרשת מכמה מחלקות בו זמנית.
  יתרון: יש מקרים (ראה תרגיל) שיכולת זו לא תסולא בפז.
  חסרונות: מוסיף לסיבוכיות של שפה -
  - התנגשות בין שמות של ישויות במחלקות.
  - יכול לגרום לירושת יהלום.
  - מימוש של קישור דינאמי קצת פחות יעיל (עוד רמה של מיעון).

- הקצאה ושחרור של אובייקטים

- קישור דינאמי או קישור סטטי