

תש"פ 2020

עקרונות שפות תוכנה

סוכם מהרצאותיו של ד"ר אריאל שטולמן



תוכן עניינים

2.....	קריטריונים להערכת שפות
5.....	מנגנון הטיפוסים
20.....	מצביעים
26.....	Scopes
32.....	Garbage Collection
38.....	פונקציות
46.....	תכנות מונחה עצמים

קריטריונים להערכת שפות

שפות אימפרטיביות – imperative

הפעלות מוגדרות ע"י מעבר ממצב למצב (state), הפרוצדורה היא מה שחשובה, כאשר הדבר היחיד שמשנה את המצב שאני נמצא בו זה פקודת ההשמה. כי עד אותה נקודה אתה יכול ללכת לפה או לשם, אבל בפועל מה שעובר ממצב למצב זה פקודת ההשמה (כשנתונים משתנים בזיכרון).

שפות לדוגמא: C, Ada, Pascal, Assembly.

שפות מונחות עצמים – OOP

האובייקט הוא ישות, מיפוי הקוד לעצמים בעולם, העצם עצמו הוא לא רק איזושהו תכלית להעברת מידע ממקום אחד למשנהו (כמו בשפות אימפרטיביות, כאשר נכתוב $i=j$ המטרה תהיה להעביר את המידע מ-j אל i). ה-data עצמו הוא מרכז האבסטרקציה (נפרט על כך בהמשך).

שפות לדוגמא: Java, C#, C++, go, D.

שפות פונקציונאליות – Functional

Functions as a first-class citizen - פונקציות מתפקדות כמשתנים. ישנם שתי תכונות לכך.

Referential Transparency – כל מקום שאפשר לרשום במקום הפונקציה את ערכה. הפונקציה היא שקופה, היא דרך לבטא ערך כלשהו. זה מאפשר לפרק את הקוד למעבדים שונים מבלי לאבד משמעות. לדוגמא - $f(g(), h())$, אני יכול לשלוח את g אל מעבד אחד ואת h לשלוח למעבד אחר, וזה לא יהווה בעיה. מכיוון שכיום התרבו המעבדים, אז הרבה שפות תכנות מנסות להכניס תכונות פונקציונאליות כדי שיהיה אפשר בצורה אוטומטית לפזר את הקוד שלך לכמה מעבדים (אם לא, אז הדבר היחיד שאתה מקבל זה שתהליכים שונים רצים על מעבדים שונים, אבל התהליך שלך ירוץ רק על מעבד אחד).

Lambda Reference – יצירת פונקציה אנונימית, שזה בעצם מה שדיברנו עד עכשיו, שהפונקציה מבטאת משתנה, ערך כלשהו, ואז לא צריך לתת לו שם (זה בעצם יחידת עיבוד שניתן להעבירה ממקום למקום).

שפות לדוגמא: Lisp, Pascal.

שפות לוגיות – Logic

הסקת מסקנות על סמך חוקים לוגיים.

שפות לדוגמא: Prolog.

שפות סקריפט – Script Language

שפות דור 5, הרעיון העומד מאחוריהן (הן עדיין לא שם, אבל זה הרעיון) שאני לא רוצה להגיד לך איך לעשות, אני רוצה להגיד לך מה אני רוצה. כלומר, אני אתאר מה אני רוצה ואתה תסתדר עם זה. אני לא רוצה להתעסק עם כל הפרטים, אני רוצה שאתה תעשה בשבילי. המון שפות סקריפט מנסות לעשות את זה, הן לוקחות חלקים מפה, חלקים משם, ומחברות וכו'. לא שפות לבניית טיל לחלל.

שפות לדוגמא: JavaScript, Python.

הערות:

- HTML אינה שפה, זה Markup language, שפה שמתארת משהו. עוד שפה כמוהו זה XML. הן לא עושות כלום ולכן לא שפות תכנות.
- PHP – ערבוב של Markup language ויש בפנים גם code.

אז מה השפה הכי טובה? באיזה שפה הייתי רוצה להשתמש?

אז התשובה הכי מדויקת היא – תלוי מה אתה רוצה לעשות. כעקרון אפשר לכתוב קוד בשפה כלשהו ואז להמיר את אותו לשפה אחרת (הוכחה לדבר היא שהשפה הראשונה מומרת ל-assembly ואז נמיר את ה-assembly אל השפה השנייה, וזה אפשרי כי אפשר להגיע מהשפה השנייה אל assembly אז מ-assembly ניתן להגיע לשפה השנייה). יש שפות שיותר קל בהן לעשות פעולות מסוימות ואילו בשפות אחרות הדבר יותר מסובך (לדוגמא, מצביעים ב-Java זה לא פשוט, ואילו ב-C++ זה הרבה יותר נוח).

אז אנחנו כעת מגיעים אל שאלה אחרת, איזה קריטריונים צריך לקחת בחשבון כאשר בוחרים שפה?

עלות - cost, עד כמה השפה קרובה למעבד, Low level. כמה עולה להכשיר מהנדסים בשפה. כמה עולה לי סביבת העבודה. מה המהירות של השפה. כמה עולים לי הקומפיילרים. כמה יעיל הקומפיילר. כמה עולה לי לתחזק את השפה הזו (ידוע גרף התחזוקה שעולה אקספוננציאלית, ככל שעובר יותר זמן, כך המחיר עולה אקספוננציאלית, ולכן שפה בעייתית בתחזוקה אז יעלה לנו יותר בהמשך).

קריאות - readability, עד כמה השפה קריאה. לדוגמא, C++ שפה לא קריאה בעליל (כמה שורות צריך לכתוב בשביל להדפיס מחרוזת, 3 לפחות), ולעומתה Python שפה קריאה (להדפיס מחרוזת דורש שורה אחת, `print(my_str)`).

פשטות - simplicity, שפה בעלת מספר מוגבל של חוקים ומבנים שעובדים אחד עם השני בצורה ברורה ומוגדרת.

לדוגמא, ב-C++: הפקודה הזו `cout<<ptr;` מדפיסה את הכתובת של המשתנה שעליו הוא מצביע. אבל זה לא נכון תמיד, כאשר `ptr` יוגדר כך `char *ptr = ...` אזי זה ידפיס את הערך שעליו הוא מצביע, ולא את הכתובת שעליו מצביעים. כלומר יש כאן פקודה בעלת 2 משמעויות. זה חוסר בקריאות.

Orthogonally – שיעשה משהו, אבל שתמיד יעשה אותו דבר. הדוגמא לעיל ממחישה את הדבר.

כתיבות - writability, עד כמה תומכת בזה שניתן לכתוב בה בצורה קלה. בדרך כלל קרירות וכתיבות באים אחד על חשבון השני.

לדוגמא ב-C++, הפעולה הזו – `for(;;)`, היה אפשר להסתדר בלי ה-`for` ע"י שימוש בלולאת `while` (לאתחל לפני הלולאה, לשים את התנאי, תוכן וקידום בתוך הלולאה). לכאורה לא צריך אותה, אלא שצריך אותה בשביל writability, קל לי לכתוב ככה כשאני שם את הקוד באותו מקום, זה בעיקר בשביל כתיבות השפה.

תמיכה באבסטרקציה (אני לא בטוח אם זה חלק מכתובות או שזה עומד בפני עצמו..) – כל הנושא של מחלקות, ירושות של פונקציות, של אובייקטים, הם כולם אבסטרקציה שמאפשרים כתיבות מאוד מעניינת, שנכתוב בצורה נוחה, בצורה טובה.

אמינות – reliability, אמינות של שפה, כגון – type checking (נדבר על כך המון, זה מונע מהמשתמש לעשות טעויות שהוא לא אמור לעשות), טיפול בחריגות, למשתנה יש שתי שמות (זה חסרון באמינות, כי יכול להיות שמשתנה א' ישנה משהו ומשתנה ב' לא ישנה אותו וזה אותו משתנה בעצם ואתה לא יודע מה השתנה לך). ככל שהשפה היא יותר קריאה וכתיבה זה מוסיף לאמינות של השפה שלך.

ניידות – portability, יש שני סוגים, האחד ברמת ה-source code (זאת אומרת שיש קומפיילר לכל השפות). השני ברמת ה-VM code, רמה יותר נמוכה. התכונה הזו גרמה לכך ש-Java עלתה (בכך שהיה לה את ה-bit code). Java הגיעה בזמן שהדרישה לניידות (וכן לשימוש חוזר) עלתה.

כלליות – generality, עד כמה ניתן להשתמש בשפה הזו ע"מ לפתור מרחב של סוגי בעיות. כפי שכבר אמרנו, אם אפשר לפתור בעיה בשפה אחת, ניתן לפתור בשפה אחרת. כלומר, ניתן לפתור בעזרת שפה כלשהי כל דבר, אז לכאורה כל שפה היא כללית. אלא, השאלה היא עד כמה השפה תומכת בכל הסיפור הזה, כמה היא נותנת גישה פשוטה להצליח לעשות את זה.

מזה נגזר גם כמה היא תומכת בכמה שיותר סוגי בעיות שרוצים לפתור. לדוגמא, יש לנו שפות שהן פותרות בדיוק משהו אחד מאוד ספציפי בצורה קלה, וכל השאר ממש מסובך לפתור בשפה הזו. לעומת זאת, יש שפות שאולי הן טיפה לא תומכות במשהו ספציפי אבל יש להן יכולת לפתור בעיות ממגוון תחומים בצורה פשוטה יחסית. עוד דוגמא, כאשר יש לי שני פרויקטים כאשר שפה א' טובה בשביל פרויקט א' ושפה ב' טובה בשביל פרויקט ב'. אז לא יהיה נכון לקחת את שפה א' לפרויקט א' ואת שפה ב' לפרויקט ב', מכיוון שכעת נצטרך לקנות 2 סביבות עבודה, כלומר המחיר יוכפל. לכן נעדיף שפה כללית.

מוגדר היטב – well defined, כאשר אני מסתכל על קוד כלשהו וברור לי מה כתוב וזה לא פתוח לפרשנויות.

דוגמא להמחשה ב-C++: `foo(g(), h());`

השפה לא מגדירה מה ירוץ קודם, הפונק' g או h. נעמוד כאן על ההבדל בין שפה לבין המימוש שלה. קומפיילר של visual studio זה מימוש של השפה, זה לא השפה. השפה מוגדרת באיזשהו סטנדרט. כלומר, בדוגמא שלנו יכול להיות שקומפיילר א' יריץ קודם את h ואז את g ואילו קומפיילר אחר יריץ קודם את g ואז את h. ולכן בקוד שלנו אנו לא יכולים להסתמך על כך שאחד ירוץ לפני השני, מכיוון שזה לא מוגדר.

מנגנון הטיפוסים

מה המטרה של מנגנון הטיפוסים:

המטרה המרכזית היא לתת משמעות לביטים בזיכרון. הזיכרון הוא אוסף של ביטים רציפים (כמו מערך של 0,1), הפרשנות של מה עומד מולנו זה מנגנון הטיפוסים. לדוגמא אם כתוב בזיכרון 100001, האם זה 65 או שזה A. נגזר מזה גם משמעות של פעולות. לדוגמא אם כתוב $a+b$, האם זה חיבור של מספרים או חיבור של מחרוזות. כמו כן, אם כתוב a/b , אם זה int אז זה חלוקה ללא שארית ואם זה float אז זה חלוקה עם שארית. מנגנון הטיפוסים היה חלק הרבה לפני שחשבו על הפשטה, קריאות (תיעוד), אמינות, אופטימיזציה וכו'. למה? כי היה צורך לתת פרשנות לביטים בזיכרון. זו הייתה המטרה הראשית של מנגנון הטיפוסים.

כמובן שלאחר שהתקדמו, אז ברור שזה נותן **אמינות**. לדוגמא, תוכנות לא יקראו מחמת שימוש לא נכון במידע, חוסר הבנה של פקודות, קומפילרים יכולים למצוא טעויות בשלב מוקדם יותר וכך גם נחסך כסף (הגרף מהנדסת תכנה שמראה שכל שמגלים שגיאה בשלב מאוחר יותר זה עולה כסף). כמו כן, **אופטימיזציה** נוגע לכמה מקום להקצות למשתנה, למה להקצות את המקסימום כשאפשר להקצות בצורה הרבה יותר יעילה. **הפשטה** – לא צריך להתייחס לדברים.

איזה מנגנוני טיפוסים קיימים:

יש פה שתיים שהם ארבע. יש פה שתי קבוצות שמתחלקות לעוד שתיים.

	strong	weak
Static		C++
Dynamic	Python	

קיים בלבול רב בנושא. רבים החושבים שאם שפה היא strong אז היא static או דברים כאלה. אבל זה לא נכון, אפשר להיות בכל אחד מארבעת הבלוקים. כלומר אפשר להיות static & strong, אפשר להיות static & weak וכו'. אין קשר בין שתי החלוקות.

בתחום של קומפילרים ושפות תכנות, **static תמיד פירושו זמן קימפול ו-dynamic תמיד פירושו זמן ריצה**. לדוגמא, ייצור משתנה דינאמי הכוונה לייצור משתנה בזמן הריצה. אם אנו אומרים על משתנה שהוא static typing, משמע שסוג המשתנה מוחלט בזמן הקימפול, זאת אומרת שבזמן הקימפול הקומפיילר צריך לדעת מה סוג המשתנה (לא בהכרח שאני אגיד לו, יכול להיות שהוא יסיק לבד). למשל יש שפות כמו C++ שצריך להגיד לו מה סוג המשתנה. לעומת זאת יש **type inferencing**, זה אומר שאני לא צריך להגיד בתוך הקוד מה סוג המשתנה, זה לא אומר שהקומפיילר לא צריך לדעת מה סוג המשתנה, זה לא אומר שהקומפיילר לא מחליט בזמן קימפול מה סוג המשתנה, זה רק אומר לא אני כמתכנת צריך להגיד לו מה סוג המשתנה. דוגמא קלאסית: ב-C# יש לנו את var, אני לא אומר לו מה סוג המשתנה, אבל זה לא אומר שהקומפיילר לא יודע (הראיה לכך שהוא יודע מה סוג המשתנה זה שאם נשתמש אחר כך במשתנה בצורה שגויה, אז הקומפיילר יצעק error!! אז זה גם יכול להיות סוג של static typing).

נסכם: ישנם שני סוגי **static typing**. הראשון שאני אומר לו מה סוג המשתנה כמו ב-C++. השני זה **type inferencing**, אני לא אומר לקומפיילר את הסוג אבל הוא יודע.

לעומת זאת, יש שפות שהן **dynamic typing**. שפות שסוג המשתנה מוחלט בזמן ריצה, כלומר בזמן קימפול הקומפיילר (אם בכלל יש אחד, בטח אם יש interpreter) לא יודע מה סוג המשתנה. רק בזמן ריצה (שזה אחרי הקומפיילר) יחליט מה סוג המשתנה וישתמש בו בהתאם. הדוגמא הקלאסית היא LISP וכן Python. ב-LISP, בה לא מגדירים סוגי משתנים, זה הוגדר בעצמו בזמן ריצה, ואחרי שהוא החליט הוא המשיך עם ההחלטה הלאה. שפות כמו Python, הן גם לכאורה דינאמיות, במובן שלא אומרים לו בשלב הקוד מה סוג המשתנה, בשלב ה-run time הוא מחליט מה סוג המשתנה ומתקדם הלאה.

לדוגמא, הקוד הבא יהיה חוקי ב-Python, אך ב-C++ למשל, הקומפיילר יצחק בשורה 3. חשוב לזכור, גם ב-Python, החולט בזמן ריצה של שורה 1 שסוג המשתנה a הוא int, ואין לנו בעיה עם הסקת המסקנה הזו, הנקודה היא שבשורה 3 הוא כבר לא int. אז איך זה יכול להיות? אז בזכות ה-dynamic typing שעד שורה 3 הוא היה int ובשורה 3, הוא יזרוק את a הראשון לפח, ויגדיר כעת a חדש, אחר לגמרי, ללא קשר ביניהם.

```
a = 7          a → int
```

```
print(a)       will print 7
```

```
a = "Hello"    a → string
```

```
print(a)       will print "Hello"
```

כעת נדון בהבדלים בין סוגי השפות, למה מישהו הבונה שפה, יחליט על דינאמי מאשר סטטי?

שפה סטטית

יתרונות:

אמינות - ניתן לתפוס טעויות בזמן קימפול (אחד הדברים הבסיסיים בנושא אמינות), צמצום טעויות המתגלות בזמן ריצה. זהו יתרון עצום לשפה.

מהירות ריצה - אמנם יותר איטי בזמן קימפול, אך יותר מהיר בזמן ריצה.

חסרונות:

כל הגמישות שיש בשפה דינאמית לא נמצאת פה. לדוגמא, הקוד הבא בשפת Python, יעבוד על כל דבר שיש לו איטרטור, אך בשפה סטטית נצטרך ליצור מלא עותקים עבור כל סוג טיפוסים.

```
def my_Len(L):
```

```
    x = 0
```

```
    for item in L:
```

```
        x += 1
```

```
    return x
```

אז יצרו את התבנית template, שזה מנגנון שהקומפיילר מייצר עבור עצמו את העותקים שהוא צריך בשביל לבצע את זה. Template לא אומר שיש קוד אחד שרץ על כולם, אלא הקומפיילר ישכפל את הקוד שלך עבור כל סוג משתנה, פשוט לא המתכנת כותב אלא הקומפיילר. מכאן אנו יכולים לראות עד כמה שפה סטטית היא לא גמישה

שפה דינאמית

יתרונות:

גמישות – ניתן לכתוב אב טיפוס (Proof of Concept), הוכחת היתכנות שניתן לכתוב את הרעיון הזה ולממשו). הרבה פעמים כאשר בונים פרויקט מסוים אז בתחילה כותבים אותו בשפה דינאמית, עושים תיאום ציפיות עם הלקוח, מגלים דרישות חדשות וכו', ולאחר מכן חוזרים אחרון וכותבים הכול בשפה סטטית.

גמישות מתבטאת גם בפונקציות eval שמקבלת כקלט קוד מסוג מחרוזת ובזמן ריצה הופכת אותו לקוד ממש ואין לקומפיילר אפשרות לגלות טעויות בקוד הזה והטעויות שם מתגלות בזמן ריצה. בשפות דינאמיות, אין שום הבדל בין קוד שבתוך eval לבין קוד אחר בתוכנית.

ניתן להרחיב את הרעיון הזה לתחום בשם Hot Swapping of Code, החלפת קוד בזמן ריצה, אני לא רוצה לעצור את התוכנית, להחליף קוד ואז להריץ מחדש אלא אני רוצה להחליף קוד בזמן הריצה. הדוגמא הכי קרובה ב-windows זה DLL, Dynamic Linked Libraries, שהתוכנה שלי הולכת ולכאורה מטעינה תוכניות אחרות כשהיא מתחילה לרוץ ואז אני יכול ללכת ולכאורה להחליף ספריות תוך כדי (אבל זה לא מדויק, כי התוכנה מטעינה את ה-DLL לפני, לא בזמן הריצה היא מחליפה).

חסרונות:

כל הגמישות הזו באה על חשבון של: אמינות, מהירות ריצה וטעויות המתגלות בזמן ריצה במקום בזמן קימפול.



כעת נדון בתת קבוצה של מנגנונים דינאמיים – Duck typing. השם הגיע מהמשפט: "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck". במשפט הזה מונח שזה לא באמת משנה אם זה ברווז או לא, זאת אומרת שלא אכפת לי אם זה ברווז או לא, אם הוא מתנהג כמו אחד אז הוא ברווז. לדוגמא אם יש לנו class Duck ובו יש פונקציה של quack(something) (לגעגע), כל עוד אני יכול בזמן ריצה להריץ את הממשקים או הפונקציות שיש לנו במחלקת "ברווז" על המשתנה שקיבלנו אז מבחינתי זה תקף, זה לא משנה לי אם המשתנה something הוא ברווז או לא.

עד כה עסקנו בסטטי ודינאמי, כעת נעסוק ב-weak ו-strong.

:Weak & Strong

חלש וחזק הפירוש עד כמה ישנה הקפדה על הטיפוסים בשפה. מנגנון חזק מאוד מקפיד על הטיפוסים, למשל – ימנע המרה מרומזת.

נרחיב קצת על המרה: כאשר באים לקומפילר ומבקשים ממנו שיתייחס למשתנה מסוג int כ-float, אז אין שום פלא שהוא יעשה את זה. נגיד במקרה שאתה מבקש מהקומפילר להמיר מ-employee ל-worker, אז אם הוא לא יודע – הוא לא יעשה, אך אם הוא יודע – הוא ימיר. זה נקרא המרה.

ישנם שני סוגי המרות – המרה מרומזת (**implicit conversion / cast**) והמרה מפורשת (**explicit conversion / cast**). אם אני במפורש אומר לקומפילר להמיר – אז אין שום פלא שיעשה כך אם הוא יודע, ואין לזה קשר לשפה strong או weak.

Strong או weak זה שאלה שמתחילה כשהוא בעצמו צריך להמיר. לדוגמא קטע הקוד הבא:

```
Int foo(int b, char c)
{
    //Do something...
}
float w = 7.4;
int x = 65;
foo(w, x);
```

גישה ראשונה – תגיד זה לא מתאים וזהו.

גישה שנייה – אין בעיה להפוך float ל-int (נהפוך מ-7.4 ל-7) ואין בעיה גם להפוך int ל-char (נהפוך מ-65 ל-A).

בצורה פשוטה זה השאלה האם שפה היא strong או weak. אם היא strong אז היא לא תסכים לעניין, ואם היא weak אז היא תגיד למה לא להמיר. זה מקרה אחד, אבל ניתן לראות את ההבדל בעוד מקומות.

לדוגמא, ב-C++ הדבר אכן ירוץ כי היא שפה חלשה.

נראה עוד דוגמא מעניינת בשפת Python.

```
a = 7
print(a)          //will print 7
print(a + "b")    //will throw error
```

```
a = 7
print(a)          //will print 7
print(a + ord("b")) //will print 105
```

התוכנית הראשונה (משמאל) לא תעשה את ההמרה ולא תסכים לחיבור שבין מחרוזת ל-int. בתוכנית השנייה (מימין) כאשר אנו עושים המרה מפורשת אנו רואים שאכן התוכנית מצליחה. כלומר, למרות שהיה ניתן להמיר ללא המרה מפורשת, השפה לא מוכנה לעשות המרה מרומזת. כמו כן, חשוב להשים לב ש-Python היא Dynamic & Strong. לעומת זאת C++ היא Static & Weak או weakly type static language. כלומר השאלה עד כמה הוא מוכן לעשות בלי שאגיד לו.

נראה עוד דוגמא בשפת C++:

```
Class DC
{ // DC = Don't - Care
Public:
    DC(int x);
    ...
Private:
    Int date;
    ...
};
```

```
int foo(DC dc)
{
    //Do something
}
```

```
foo(3);
```

השאלה כעת – האם קטע הקוד כאן יעבוד או לא. מצד אחד ניתן להגיד שלא יעבוד מכיוון שהפונקציה מקבלת משתנה מסוג DC ולא מסוג int ולכן לא יעבוד. מצד שני הבנאי של המחלקה (מוקף בעיגול) יודע להמיר משתנה מסוג int ל-DC אז אולי זה אפשרי.

התשובה היא – כן! קטע הקוד יעבוד! הוא ימיר מ-`int` ל-`DC`. חשוב לדעת שהבנאי המסומן יש לו עוד שם – **conversion constructor**. בנאי זה בעצם מלמד את הקומפיילר איך ממירים מ-`int` ל-`DC`, ולכן הקומפיילר ידע איך לעשות את המעבר המסומן בחץ האדום, זו המרה מרומזת. התהליך שהקומפיילר עשה בהמרה בחץ האדום שקולה להמרה הנצרכת במקרה הבא:

```
DC my_dc(4);
```

נמצאנו למדים כלל מאוד מעניין ב-C++, כל בנאי שיש לו פרמטר אחד הוא גם **conversion constructor**.

חשוב לזכור, זה עובד רק בגלל ש-C++ היא שפה **weak typing**.

נשנה ונוסיף קצת על הדוגמא הקודמת ונראה נקודה נוספת:

```
Class IRDC
{ // IRDC = I Really Don't - Care
Public:
    IRDC(DC dc);
    ...
Private:
    ...
};
```

```
int foo(IRDC dc)
{
    //Do something
}
```

```
foo(3);
```

הוספנו `class IRDC` ושינינו את הפונקציה כך שתקבל משתנה מסוג `IRDC`. השאלה היא – האם הקריאה כאן לפונקציה תעבוד או לא. מצד אחד, אולי זה לא יעבוד מכיוון ש-3 זה לא מסוג `IRDC`. מצד שני, אולי הקומפיילר יגלה שניתן להמיר את 3 ל-`IRDC` בצורה כזו – `IRDC(DC(3))`.

התשובה היא – לא! הקומפיילר לא בודק יותר מהמרה ישירה בין המשתנים. לבדוק יותר ממעבר אחד זה יותר מידי עבודה (לבדוק האם יש מסלול זה המון עבודה). אם הוא ימצא בתוך `IRDC` את ההמרה הישירה אז הוא לא יחפש עוד.

אם יהיה כתוב `foo(IRDC(DC(3)))` אז זה היה עובד. אפילו אם היינו כותבים כך `foo(DC(3))` זה גם היה עובד מכיוון שהקומפיילר יודע כיצד להמיר מ-`DC` אל `IRDC`. אלה יעבדו כי זו המרה מפורשת (אבל עיקר הנושא שלנו ב-**Strong** ו-**weak** זה על המרה מרומזת).

מנגנון הטיפוסים - המשך

איך נתונים ממומשים מאחורי הקלעים:

השלמים – integer:

Integer ספציפית הוא מאוד מעניין, בהתחלה הוא הוגדר כ-word בזיכרון – תלוי מכונה כמובן, כלומר הוא הוגדר כפונקציה של מיפוי הזיכרון. כאשר המחשבים היו 16bit אז integer היה מוגדר להיות 2 בתים (16 ביטים), וכאשר יצאו מחשבים של 32bit אז Integer הפך ל-4 בתים בזיכרון (32 ביטים). לכאורה נגזר מזה שכאשר המחשבים עברו ל-64bit אז גם integer יתקדם. אבל זה לא קרה, החליטו להקפיא את המצב ו-integer היה half-word בזיכרון. ולכן גם לעיתים יש בקוד int32 או int64, כלומר לא תלוי קומפיילר ככה ש-Int יהיה קבוע.

מנייה – enumeration:

ישנן שתי דרכים לממש enum, הראשונה היא לעשות אותו סוג בפני עצמו, כמו ש-int הוא סוג כך enum יהיה סוג, ואז פשוט מייצרים לו זיכרון משלו. הדרך השנייה (גישת C, C++) היא שמנייה היא סוג של סימבולי ל-int, כלומר אם כותבים `enum{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}`, מה שיקרה מאחורי הקלעים הוא שיהיה לנו מספרים מ-0 עד 6. לדוגמא נוכל לשאול כמה זה ונוכל לשאול כמה שווה `thuesday + 1` וזה יהיה שווה Wednesday. כמו כן, `Tuesday * 3 = Friday (= 6)`, ואין בזה הגיון, אבל זה עובד. בנוסף, היום לפני ראשון יהיה 1-, והיום אחרי שבת יהיה 7. אנו רואים שלממש מנייה כשלם לכאורה מייצר בעיות, ומצד שני ייצר פשטות ליישום.

יש שפות כמו Ada למשל, שהגדירו מנייה כסוג נפרד, אפילו אם זה ברמה הסינטקטית סוג נפרד (כמובן שמאחורי הקלעים אולי הוא מיוצג ע"י כל מיני מספרים, אבל לנו כמשתמשים של שפה, לא נוכל להשתמש בהם ככה). למשל, אפשר לשאול מה בא אחרי שלישי והתשובה תהיה רביעי, אפשר לשאול איזה מספר כרונולוגי הוא רביעי והתשובה תהיה 4, אבל אי אפשר לשאול כמה שווה שלישי * 2 – אלו פעולות לא חוקיות. כלומר, אנו רואים שאם מממשים אותו כסוג נפרד זה יהיה הרבה יותר הגיוני, ולא רק בזיכרון מאחורי הקלעים של איך שזה נעשה.

משתנה בוליאני – Boolean:

ישנן 3 דרכים לממש בוליאני.

הדרך הראשונה, שהיא הדרך ש-C בחרה בה היא דרך של שלמים. כלומר, Boolean זה פשוט integer. ידועה השגיאה בשפת C: `if(cond = 0)`, השגיאה היא שיש = ולא ==. הקוד הזה יחזיר שקר. מה שקורה כאן הוא שיש כאן השמה `cond=0` ואז מה שקורה הוא שעכשיו בתנאי יש 0 = שקר. אם Boolean היה סוג בפני עצמו, אז לכאורה הקומפיילר היה מוציא שגיאה על זה, שיש כאן השמה אבל אין כאן תנאי בוליאני. בשפות כמו C# אם נרשום את התנאי הזה הוא ינסה לעשות את ההשמה ואז יצעק שאין תנאי בוליאני, אבל בשפות כמו C, C++ לא תהיה בעיה אם נרשום `if(cond = 1)` זה תמיד יהיה אמת, מכיוון ש-boolean הוגדר כ-int. מהגדרה זו מסתעפים יתרונות וחסרונות, לדוגמא: `7*18` זה מספר גדול אבל זה עדיין אמת, `1*0` יהיה שקר, `false + 1` יהיה שווה אמת.

הערה חשובה: Boolean לא מומש כ-bit!! הסיבה לכך היא מכיוון שאין דרך להקצות בזיכרון bit, ההקצאה המינימלית בזיכרון הוא בית.

הדרך השנייה היא ע"י enum, נייצר סוג חדש כך: `{false, true}`. אוקיי, אבל איך enum יהיה ממומש? זה כבר תלוי במה שראינו לעיל, האם enum ממומש כ-int או כסוג בפני עצמו. למעשה, בשלב שבו Boolean היה Int, הרבה מתכנתים

כתבו בעצמם `enum{false, true}`, ואז יצא להם איזשהו סדר במילים, זה לא עזר הרבה כי עדיין הקומפיילר התייחס לזה כ-0,1, אבל זה נתן סוג של סדר.

בשפות אחרות, Boolean הוא סוג בפני עצמו. כלומר, במקום שמצפים לקבל Boolean ושמת ערך אחר – הקומפיילר יצחק. לדוגמא, בשפת C#: אם תשים בתנאי משהו שהוא לא בוליאני, הקומפיילר צועק, הוא מצפה ל-boolean, אבל הוא קיבל ערך אחר.

תו – Charter:

גם פה, char יכול להיות איזשהו משחק על Int, לדוגמא בשפת C++. הוא יכול להיות סוג בפני עצמו. הוא יכול להיות בר מנייה, כי ב-char אין יחס של כפל או חילוק ביניהם, יש יחס של פערים ביניהם (3 אותיות לפני W וכו').

כפי שכבר הזכרנו, אין דרך להקצות bit אחד עבור Boolean, אבל זה לא מדויק. למשל, אם אני אקצה מערך של 8 משתני Boolean, אז קומפיילר חכם לא יקצה 8 בתים, אלא יקצה לכולם בית אחד. לזה קוראים compaction, אם אני מזהה שאני יכול להכניס הרבה לבית אחד אז אני אעשה זאת, כי בכל מקרה אני צריך להקצות בית. רק כאשר אני לא יכול אז אני אקצה יותר.

נראה כעת דוגמא להמחשת העניין, בשמאל נראה את קוד התוכנית, ובימין נוכל לראות כיצד הזיכרון נראה:

```
#include <iostream>

typedef struct
{
    int i;
    char c2;
    int j;
    char c1;
    bool a;
} DC;

int main()
{
    DC dc = { 0x11111111, 0x55, 0x77777777, 0x66, true};
    size_t size = sizeof(dc);
    std::cout << size;
}
```

0x0023F98C	11	11	11	11	...
0x0023F990	55	cc	cc	cc	Uooo
0x0023F994	77	77	77	77	www
0x0023F998	66	01	cc	cc	f.oo

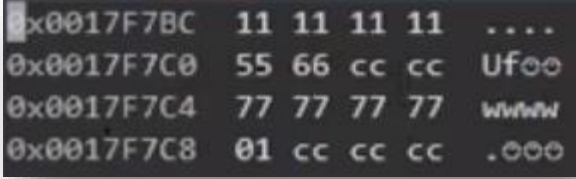
Int תפס word שלם, 4 בתים, כפי שציפינו. לאחר מכן, ה-char תפס word שלם, כאשר הוא השתמש רק בחלק מהבית שהוקצה לו. לאחר מכן, j תפס מקום של בית כמו i. ולבסוף התו והבוליאני נמצאים באותו בית. כלומר, כל אחד תפס את המינימום שהוא יכול לתפוס, קרי, זה לא שנעשה פה compaction, פשוט עשה את המינימום.

כעת נעדיכן את הקוד שבדוגמא ונעבור על זה שוב:

```
#include <iostream>

typedef struct
{
    int i;
    char c2;
    char c1;
    int j;
    bool a;
} DC;

int main()
{
    DC dc = { 0x11111111, 0x55, 0x66 0x77777777, true};
    size_t size = sizeof(dc);
    std::cout << size;
}
```



0x0017F7BC	11 11 11 11
0x0017F7C0	55 66 cc cc	Ufoo
0x0017F7C4	77 77 77 77	www
0x0017F7C8	01 cc cc cc	.ooo

נשים לב לזיכרון בשורה 2, ברגע שנהיה 2
char-ים, אז הוא עשה להם compaction.

מצביעים ובעיותיהם:

מטרת המצביעים:

שם של משתנה מייצג מיקום בזיכרון (memory location) ואפשר לעשות חישוב בזמן ריצה על תוכן שממוקם בזיכרון. לדוגמא, כאשר יש לי משתנה $a = 8$ בזיכרון, אז אם אני אחשב את $a * 3$ אני אקבל 24. חשוב להשים לב, שלא a חושב אלא ערכו, ה-8, חושב, a ניתן לי. כאשר עשיתי את החישוב אז לא שאלתי מיהו a , כי את זה טבלת הסמלים אמרה לי, אלא שאלתי מה ערכו של a .

```
cin >> a;
if(a > 0)
{
    cout << a;
}
else
{
    cout << b;
}
```

בדוגמא כאן, הכנסנו ערך למשתנה ושאלנו על **ערך** המשתנה. אבל אם הייתי רוצה לשאול את המשתמש האם להשתמש ב- a או ב- b , אז לא הייתי יכול להדפיס את מה שהוא בחר (לדוגמא אם הקלט מהמשתמש היה b , אז לא יכולתי לכתוב `cout<<user_input`). מה שכתוב פה הוא שהמשתמש הכניס ערך ל- a , ואז אנחנו מדפיסים את מה שהמשתמש הכניס ולא את a אלא את תוכנו.

```
cin >> i;
cout << arr[i];
```

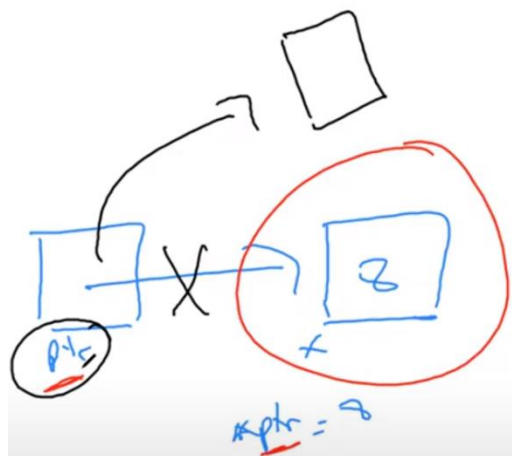
ניתן לראות את הדקויות הללו בצורה בולטת במערכים. הכנסנו לתוך i ערך ואז התייחסנו אליו בשורה השנייה (**לערך**). מה שמעניין פה הוא הגישה למערך. במערך יש לנו כמה מקומות ואנו צריכים לבחור מבניהם את מי אנו רוצים, כלומר, מה שמחושב כאן זה באיזה משתנה אנו רוצים להשתמש לא ערכו של המשתנה. כלומר, `arr[i]` עוד לפני שהוא מחזיר את הערך שנמצא במקום הזה במערך, הוא קודם אומר לי על מי מדובר. לאחר מכן אנו נבדוק מה ערכו. זאת אומרת, המטרה של מצביעים דבר ראשון היא **לחשב משתנים**, בשונה ממה שעשינו עד היום **שחיבנו ערכים**.

נעשה תזכורת קצרה על L-values ו-R-values.

L-values: ערכים שיכולים להיות בצד שמאל של משוואה, כלומר משתנים וכו', אבל לא ערכים (כי לא ניתן לכתוב $x = 3$)

R-values: ערכים שיכולים להיות בצד ימין של משוואה, כל מה שבעצם יש לו ערך (משתנים, מספרים ממש וכו').

המטרה של משתנים היא בעצם להפוך שם של משתנה ל-L-values, זאת אומרת שאני יכול להגיע אליו בצורה חישובית. זה בעצם מה שקורה פה, יש את כתובת הבסיס של המערך – לדוגמא 1000 ונניח שזה מערך של `int` (כלומר שלכל מקום מוקצה 4 בתים) אז ע"מ למצוא את המקום `arr[i]` נלך לזיכרון במקום: $1000 + i * 4 = 1012$. `arr_base + i * sizeof(int)`. כפי שהזכרנו, מספר הוא לא L-value אלא R-value, ולכאורה לא יכולתי להשים אותו בצד שמאל של משוואה, לעומת זאת אני כן יכול לכתוב `arr[i] = 8`. אז איך בעצם 1012 זיכרון הפך להיות L-value? אלא מדובר פה במצביעים. כלומר, מצביעים נתנו לנו את היכולת לחשב את שמות המשתנים (ובמילים אחרות את המשתנים) ולא את ערכם. השלב אחרי זה (במקרה שלנו של ההדפסה) יהיה לבדוק מה ערכו (שזה לבדוק מה יש במקום הזה בזיכרון). אבל דבר ראשון חישבנו במי אנו רוצים להשתמש, ובלי מצביעים זה לא היה אפשרי. בדוגמא של ה-`if-else` מתחילת העמוד, הכל היה קבוע מראש, אם הקלט הוא גדול מאפס אז אנו נדפיס a ואם לא אז b . בנוסף, בדוגמא שם קלטנו ערך לתוך המשתנה a והדפסנו אותו אם הוא היה גדול מ-0, כלומר הערך שנכנס הוא זה שיצא אבל a לא חושב. הקומפיליר הסתכל בטבלת הסמלים בזמן הקימפול וכבר החליט על איזה זיכרון מדובר, על הזיכרון ששמור ע"י השם a , התוכן שלו ייוודע בהמשך מהמשתמש, אבל לא a עצמו חושב, הוא ידוע מראש.



לא כך במצביעים, שם אנו מחשבים את שם המשתנה.

כאשר נכתוב $*ptr = 8$ (ב-C++), אנו נשים 8 (במשתנה המוקף באדום, נקרא לו x לשם הנוחות, ה"מוצבע"). נשים לב שאנו לא יודעים מה הכתובת של x, זה גם לא מעניין אותנו, אבל עדיין הצלחנו לחשב אותו. לעומת x, המצביע ptr הוא קבוע מראש, כלומר בטבלת הסמלים יש סמל ptr ויש לו כתובת כלשהי. אבל ל- ptr אין זכות קיום משל עצמו, כל מטרת חייו היא לייצג את x. אבל את x אנו לא יודעים, אנו מחשבים אותו בזמן ריצה, ולכן שום דבר לא מונע מאיתנו לשנות את ptr שיצביע למישהו אחר, לדוגמא שיצביע למשתנה השחור (למעלה באיור). נשים לב שהקוד שכתבנו $*ptr = 8$, עדיין יעבוד, כי אין קשר בין יעד ההצבעה לבין טבלת הסמלים, ptr עצמו לא השתנה (הוא באותו כתובת הזיכרון), מה שהשתנה זה למי אנו מצביעים, על איזה משתנה אנו עובדים כעת. לכן יצא שיש לנו דרך לחשב בזמן ריצה שמות של משתנים (הכוונה לחשב איזה משתנה אנו רוצים עוד לפני שאני מחשב את ערכו של המשתנה הספציפי הזה).

יוצא אם כך, שמצביעים זה דבר ששונה במהותו משאר סוגי המשתנים.

```
int a = 7;
int &b= a;
b++; /*or a++ */
```

```
int main()
{
    int a = 7;
    foo(a);
}

int foo(int &b){ b++; }
```

כעת נדון בדבר נוסף, ב-reference שבו עסקנו בשפת C++. נראה כמה דוגמאות (הדוגמאות אותו דבר בפועל). בשתי הדוגמאות כאן קורה אותו דבר, ומה שלימדו אותנו זה שכל מה שיקרה מעכשיו ל-b יקרה גם ל-a. אך מה שלימדו אותנו הוא לא מדויק, אותנו לימדו בעצם שלמשתנה a כרגע יש שם נרדף והוא b וכל מה שקורה ל-b יקרה ל-a מכיוון שהם אותו משתנה. בלוגיקה, ברמה התאורטית, b הוא אכן שם נרדף ל-a, אך בפרקטיקה זה **מצביע** לכל דבר ועניין (הלם!!). b הוא בעצם מצביע על a. כאשר אנו כותבים b++ מה שבעצם קורה זה ++(*b), וזה מקביל לכך שהיינו כותבים:

```
int foo(int *b) { (*b)++;}
```

כלומר, אם נחזור לדוגמא העליונה, בשורה 2 אנו יצרנו מצביע ל-a, ולא שם נרדף. וזהו reference, הוא מצביע. ההבדל היחיד הוא של-reference יש automatic de-reference. כלומר, כל מקום שנכתוב b הוא יהפוך את זה ל-b*. וזה בשונה ממצביע ב-C++ שאין את ההמרה האוטומטית הזו.

חשוב לדעת את זה, מכיוון שיש שפות שיש להן reference ואין להן מצביעים. זה לא שאין להן מצביעים, אלא בשפות אלו יש automatic de-reference. זה הגיוני כי למצביע בפני עצמו אין זכות קיום, זכות קיומו הוא שהוא עוזר לי לחשב את יעד ההצבעה. אז חלק מהשפות אמרו לעצמן – מה ההיגיון להגיד אני רוצה את מה ש-b מצביע עליו, הרי אנו אף פעם לא נרצה את b אלא תמיד נרצה את יעד ההצבעה של b, ולכן במקום לכתוב כל פעם *b, תרשום פשוט b וה- automatic de-reference ימיר את זה ל-b*. לעומת זאת, שפות כמו C++ אמרו שזה לא נכון, מכיוון שאנו רוצים לעשות מניפולציות עם מצביעים ולכן אני צריך גישה אליהם. כלומר, אני יכול לגשת למצביע ואני יכול לגשת למה שהמצביע מייצג ואז נרצה de-reference ואז נוצרו לנו שתי טיפוסים. הטיפוס הראשון הוא המצביע, והטיפוס השני הוא ה-reference שזה גם מצביע רק שהוא עושה automatic de-reference.

היתרון בשימוש עם reference (כלומר לא בכתיבה הזו: int foo(int *b) { (*b)++;}) זה שלא יכולות לקרות לנו טעויות בסגנון של לשכוח לכתוב את הכוכבית, הקומפיילר לוקח את כל העבודה הקשה עליו, וזה reference type. כאשר אנו לא רוצים להתייחס למצביע כמצביע, אלא להתייחס למצביע כמה שהוא מצביע עליו (ככה עובדות C#, Java, אנו לא מייצרים בשפות אלו מצביעים אלא reference לאובייקטים, אבל זה עדיין מצביעים).


```
int main(){
    int a = 7;
    int& r = a;
    int* ptr = &a;
    a = 18;      //one assembly line
    r = 19;      //two assembly lines!!
    *ptr = 20;   //can you tell the difference

    //show how arrays work
    int b[12] = { 0 };
    b[3] = 4;
    return 0;
}
```

```
int a = 7;
00EE17E2 mov     dword ptr [a],7
int& r = a;
00EE17E9 lea     eax,[a]
00EE17EC mov     dword ptr [r],eax
int* ptr = &a;
00EE17EF lea     eax,[a]
00EE17F2 mov     dword ptr [ptr],eax
a = 18;      //one assembly line
00EE17F5 mov     dword ptr [a],12h
r = 19;      //two assembly lines!!
00EE17FC mov     eax,dword ptr [r]
00EE17FF mov     dword ptr [eax],13h
*ptr = 20;   //can you tell the difference
00EE1805 mov     eax,dword ptr [ptr]
00EE1808 mov     dword ptr [eax],14h

//show how arrays work
int b[12] = { 0 };
00EE180E push     30h

//show how arrays work
int b[12] = { 0 };
00EE1810 push     0
00EE1812 lea     eax,[b]
00EE1815 push     eax
00EE1816 call     _memset (0EE10F5h)
00EE181B add     esp,0Ch
b[3] = 4;
00EE181E mov     eax,4
00EE1823 imul    ecx,eax,3
00EE1826 mov     dword ptr b[ecx],4
return 0;
```

כעת נראה את זה מאחורי הקלעים של הקוד. למעלה אנו נוכל לראות את הקוד שלנו, ולמטה את הקוד ב-assembly המתאים.

בתרגום של הגדרת המצביעים יש 2 שורות, בשורה הראשונה אנו טוענים לתוך `eax` את הכתובת של `a`, בשורה השנייה הוא טען את הכתובת לתוך המצביע.

כאשר אנו עושים השמה לתוך `a` מתבצעת שורה אחת בלבד, הכנסת הערך לתוך `a`. לעומת זאת, כאשר אנו עושים השמה למשתנה דרך המצביע שלו, אז קודם אנו קודם מכניסים ל-`eax` את התוכן של המצביע ואז בשורה השנייה אנו שמים בכתובת הזו את הערך 20 (או 19). אנו רואים שהתזוזה של ערך לתוך משתנה רגיל זה שורה אחת, וכאשר משתמשים במצביעים אז התזוזה היא שתי שורות. שורה אחת לקחת את התוכן של המצביע לתוך רגיסטר ואז כמו כל דבר אחר אנו יכולים לגשת לכתובת שנמצאת ברגיסטר. נשים לב לפער בין משתנה רגיל לבין מצביע. כמו כן, נשים לב שאין הבדל בקוד ה-assembly בין השימוש במצביע לבין השימוש ב-reference.

כעת נחקור את המשך הקוד. בשורה הראשונה בקוד הוא מציב אפסים בכל המערך (לא נכנס לקוד ב-assembly העושה את זה). כעת נסתכל על השורה השנייה למטה, ונראה שמה דמיון עצום למצביעים. דבר ראשון הוא דחף 4 (ה-4 הזה הוא מכיוון שמדובר על מערך מסוג `int` ולכן לכל משתנה הוקצו 4 בתים) ל-`eax`, אחרי זה הוא הכפיל ב-3 כי אנו מחפשים את המקום השלישי ואת זה הכניס ל-`ecx`. כעת בתוך `ecx` יושבת הכתובת של המשתנה שנרצה. כלומר, בשתי השורות הראשונות חישבנו את הכתובת ובשורה השלישית דחפנו לשם את הערך 4. זה מקביל למה שעשינו למעלה עם מצביעים (בשורה הראשונה חישבנו את הכתובת, הוא לא היה צריך לעשות פעולה אריתמטית מכיוון שהיה לו את כל הערכים מול העיניים, ואז הוא דחף את הערך פנימה).

וזה מביא אותנו למסקנה שכבר אמרנו מקודם, שאין הבדל מהותי בין מצביעים ל-reference ואפילו בין מערכים. זה הכול סגנונות שונים של אותו דבר בדיוק. ההבדל הוא שבמערכים זה עוזר לקריאות, וההבדל בין מצביעים ל-reference זה ה-automatic de-reference (שגם עוזר לקריאות וכתובות, שלא צריך כל שנייה לכתוב * כל שנייה, וגם זה נראה יותר ברור כאשר אין את כל הבלגן הזה).

נסכם, תפקידם של מצביעים הוא לחשב שמות, ו-reference type הם פשוט מצביעים עם automatic de-reference שזה

מוסיף לקריאות וכתובות ועוזר מאוד לטפל בסיפור של המשחקים של "חישוב השמות", חישוב של משתנים – לא ערכו אלא מי הוא (ובמילים אחרות ה-L-value שלו).

ישנן 2 משמעויות ל-de-reference. מצד אחד הוא יכול להיות כמו ההגדרה שלו ב-C++, שהכוונה שהשם של המצביע זה המצביע וצריך לעשות את ה-de-reference, זה לא מתבצע ישירות, צריך לעשות *ptr. מצד שני יש שפות כמו Ada, Fortran, בשפות אלו אם נגיד ptr אז הוא יבין שהכוונה היא de-reference למרות שמדובר על מצביעים. לדוגמא בשפות אלו אם נכתוב R.field = something, אז אין לנו דרך לדעת האם R הוא האובייקט עצמו או reference לאובייקט, כי תמיד נעשה אוטומטית de-reference.

כעת נעסוק בבעיות במצביעים, נתמקד בעיקר בשתי בעיות.

בעיית המצביע המתנדנד – Dangling Pointer:

מצביע מתנדנד זה מצביע שאיבד את הזיכרון עליו הוא מצביע אך המצביע לא מודע לעניין (הוא עדיין מצביע לשם). כאשר כתוב אצל המצביע שיעד ההצבעה שלו זה null, אזי הוא לא מצביע מתנדנד.

עיקר הבעיה במצביע מתנדנד הוא שיכול להיות שהזיכרון שעליו המצביע הצביע, הוקצה למישהו אחר. אם הוא לא הוקצה למישהו אחר אז אין לנו בעיה, כי אף אחד לא ישנה אותו, הבעיה מתחילה כאשר הזיכרון הוקצה למישהו אחר והמישהו הזה חילק את הזיכרון המדובר בצורה אחרת והמצביע חושב שהזיכרון מחולק לפי השיטה הישנה. דבר ראשון, יכול להיות שתשנה ערכים של מישהו אחר ולא שלך, דבר שני יכול להיות שתקבל ערכים ממישהו אחר ואלה לא יהיו הערכים שאתה מצפה לקבל. כמו כן, יכול להיות שה-type checking ישבר, כי מבחינת הקומפיילר זה מעולם לא קרה והזיכרון המדובר עדיין שלך ומה שקורה הוא שהקומפיילר בודק את הקוד שלך (את הפקודות, האם הפקודות חוקיות ביחס לטיפוס שהוא מצפה שיהיה שם, ולמעשה יש שם טיפוס אחר), אז יכול להיות שכל מנגנון ה-type checking ישבר. בנוסף, יכול להיות שתצא לך שגיאה על כך שאתה חורג ממגבלות הזיכרון שלך, זה שאתה יכול לגשת אליו.

אז למה שנעשה שטויות כאלו שיגרמו למצביע מתנדנד? למעשה, ניתן להגיע לטעויות כאלו בקלות ממש, לדוגמא הקוד הבא:

```
int main() {
    DC a;
    if (true) { DC b = a; }
}

class DC {
public:
    DC() { arr = new int; }
    ~DC() { delete arr; }
private:
    int* arr;
};
```

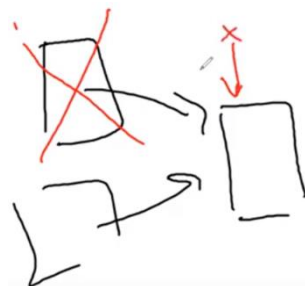
בדוגמא הזו, יצרנו את a שבתוכו יש מצביע arr וכאשר אנו מגדירים את b, מכיוון שלא בנינו בנאי של העתקה עמוקה ולכן מתקיימת כאן העתקה רדודה (כפי שלמדנו ב-C++), b יצביע לאותו משתנה ש-a מצביע עליו (-). a.arr מצביע, כלומר a.arr = b.arr. ולכן, ברגע שנצא מה-scope שבו b הוגדר, מה שיקרה הוא שנמחק את b (ולכן נמחק גם את arr של b, שזה אותו arr של a) ואז a.arr יהיה מצביע מתנדנד. אפשר על אותו עיקרון לשנות את הדוגמא כך ש-b יהיה מוגדר באותו scope של a, ואז נעשה ידנית delete b ונגיע שוב לאותו מצב. a.arr עדיין חושב שהוא מצביע על ערך int כלשהו, הוא לא מודע למה שקרה.

```
int main(){
    int* p = foo();
}

int* foo() {
    int x = 3;
    return &x;
}
```

בדוגמא פה בשמאל, יצרנו מצביע ל-x וזה מה ש-foo החזירה, כך ש-p לכאורה מצביע ל-x. אך, ברגע שיצאנו מה-scope של foo, הזיכרון של משתנה x שוחרר והנה הגענו למצב של מצביע מתנדנד. מזה נגזר הכלל המפורסם שלעולם לא מחזירים מפונקציה reference ל-local data, כי זה מייצר מצביע מתנדנד.

איור להמחשת העניין של מצביע מתנדנד:



במקרה שבו אנו מוחקים את המצביע העליון ואיתו אנו מוחקים גם את הזיכרון שאליו מצביעים (הריבוע בצד ימין) ואז המצביע התחתון יצביע על זיכרון שכבר לא מוקצה עבורו.

בעיית זליגת זיכרון דינאמי מהערמה – Dynamic heap memory leak: ובקיצור mem.leak.

```
{
  int* ptr = new int[13];
}
```

זליגת זיכרון זה זיכרון שמוקצה לנו אך לא נגיש בשום צורה. בדוגמא כאן בשמאל, כאשר אנו יוצאים מה-scope, המשתנה ptr מנוקה מהמחשנית והזיכרון שעליו ptr הצביע עדיין קיים בזיכרון. מבחינת המערכת, הזיכרון הזה מוקצה לנו, ולכן הוא לא יכול להשתמש בו. השם המקצועי שניתן לזיכרון הזה הוא garbage (בהמשך נדבר על אלגוריתמי איסוף זבל). הזבל הזה הוא לא נגיש, וככל שנעשה כך יותר ויותר כך יגמר לנו המקום בזיכרון, עד שהמערכת תקרוס ותגיד שנגמר לה הזיכרון (נגמר ה-free memory).

```
DC* dc = new DC;
dc = another_dc;
```

נראה עוד דוגמא: בשורה הראשונה יצרנו מופע של DC שאליו dc מצביע, ובשורה 2 אמרנו ל-dc להצביע על another_dc (מופע אחר של dc). בזיכרון כעת יש לנו את המופע הראשון של dc שיצרנו בשורה הראשונה, אבל הוא לא נגיש, כי המצביע שהיה אליו, כעת מצביע על מישהו אחר.

הבעיה הזו קיימת רק בשפות שדורשות שחרור ידני של זיכרון (למשל ++C), כלומר, שהזבל הוא באחריות המתכנת בשפה לנקותו.

אז איזו בעיה יותר בעייתית? כלומר, אם יכולתי לפתור אחת על חשבון השנייה (קרי, אם הייתי צריך לפתור את בעיית המצביע המתנדנד ע"י זה שאני איצור זליגת זיכרון, או לפתור זליגת זיכרון ע"י שאני איצור מצביע מתנדנד), את מי היה כדאי לפתור?

מצביע מתנדנד – יש חוסר באמינות, וגם אנו לא יודעים שזה קורה (oblivious).

זליגת זיכרון – אם יש הרבה מזה אז תוכניות לא יכולות לרוץ בכלל, ומצד שני אנו מודעים לעניין (detected).

בעולם אוטופי, שלכאורה אנו צועדים לכיוונו, בעיית זליגת הזיכרון תהיה פחות בעייתית ככל שנתקדם, מכיוון שהזיכרון עם הזמן גדל וגדל, ולא אכפת לי שיש זבל כי יש מלא זיכרון. ולעומת זאת, בעיית המצביע המתנדנד תישאר אותו דבר (כמובן בשפות שבהן המתכנת צריך לנקות זיכרון).

נקודה נוספת, זה עצם הידיעה שהבעיה קיימת, כאשר אני יודע שיש בעיה אז אני יוכל לטפל בה. בעיית המצביע המתנדנד אני לא מודע אליה עד לרגע שבו אני נכווה ממנה, כלומר עד הרגע שבו יש ניסיון להשתמש בזיכרון שם וזה לא עובד. לעומת זאת, אנו יכולים לטפל בבעיה של זליגת זיכרון לדוגמא כאשר אנו מנסים לעשות new וזה לא עובד כי אין לנו זיכרון פנוי, אז נוכל לגרום שהתוכנית לא תקרוס, אלא נעשה graceful degradation, כלומר לסגור את המערכות בצורה יפה (במקום לקרוס), או להגיד שאין מקומות נגשים יותר (ועוד דברים בסגנון במקום לקרוס).

ולכן, בגלל שתי הנקודות שהועלו פה, כדאי יהיה לפתור את בעיית המצביע המתנדנד על חשבון בעיית זליגת הזיכרון.

בשפת Pascal, יש פונקציה בשם dispose (מקבילה ל-delete בשפת ++C), ברוב הגרסאות של Pascal, הפונקציה הזו לא תעשה כלום. כלומר, המתכנת קורא ל-dispose ע"מ לנקות את הזיכרון ולהחזיר אותו לזיכרון הפנוי, והשפה לא עושה כלום. הרעיון מאחורי, הוא בדיוק הנקודה שאנו עוסקים בה כעת, Pascal העדיפה שיהיה לה זליגת זיכרון אבל שלא ייווצר

לנו אף פעם מצביע מתנדנד (כי לא נגיע למצב שבו אנו מוחקים מצביע אחד לזיכרון כלשהו, והמצביע השני יחשוב שהוא עדיין מצביע אל הזיכרון הזה אבל הוא בפועל נמחק. אם זה לא היה ברור, אז ניתן להסתכל באיור בסוף העמוד הקודם להמחשה). המנגנון הזה של Pascal יוצר זליגת זיכרון אוטומטית, העדיפו זליגת זיכרון מאשר מצביע מתנדנד. באיור כאן, ניתן לראות כיצד Pascal יוצרת זליגת זיכרון, הוא מוחק את המצביע (כאשר המתכנת מבקש dispose, אבל האיבר המוצבע לא ימחק אלא ישאר למקרה שיש עוד מישהו שמצביע עליו).



מצביעים - בעיות ופתרונות

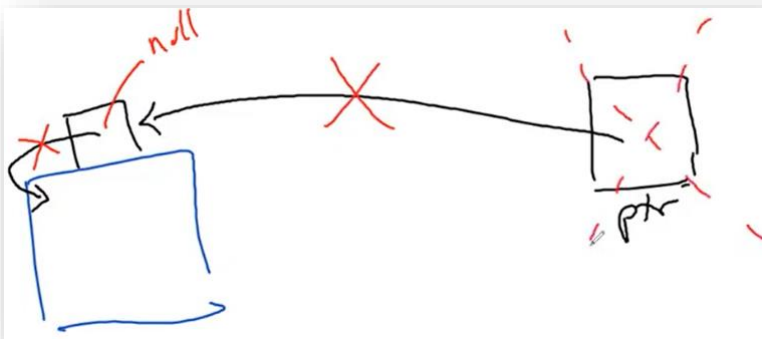
מחיר של מצביעים:

נתבונן בדוגמא שכבר ראינו (זו עם התרגום ל-assembly), החיסרון המרכזי שישי במצביעים הוא שכל גישה היא double access, כל פקודה הופכת לשתיים (הטענת מצביע, תוכנו של המצביע, ונתקדם הלאה) וזה גורם לחוסר יעילות מסוימת.

בהמשך למה שראינו בשיעור שעבר על ההשוואה של זליגת זיכרון מול מצביע מתנדנד וראינו את הפתרון של Pascal (להתעלם מה-dispose), כעת נראה עוד פתרון.

Tombstone - מצבה:

הרעיון הוא שאנו נוסיף "מצבה" לזיכרון עליו אנו רוצים להצביע (המצבה תצביע על הזיכרון המדובר), ואז כל מצביע יצביע על המצבה במקום על הזיכרון עצמו.



כאשר נרצה למחוק את הזיכרון, אזי נשנה את המצבה כך שהיא תצביע על null (כלומר מנתקים את המצבה מהזיכרון) ואז גם מנתקים את המצביע מהמצבה (ובשלב זה נוכל למחוק את המצביע עצמו אם נרצה). איור להמחשה:

הרעיון המרכזי הוא כאשר יש עוד מצביע לאותו

מקום, נכנה אותו dptr (d - dangling), על פי מה שלמדנו כעת הוא מצביע מתנדנד מכיוון שהוא לא יודע שהזיכרון שלו השתחרר. בזכות ה"מצבה", מה שיקרה הוא שתצא שגיאה (לפני כן יצאה שגיאה שהוא מנסה לגשת לזיכרון שלא ברשותו, וזה לא יקרה עם מצבה). השגיאה היא שננסה לגשת ל-null כמצביע, ואז הוא יזרוק שגיאה.

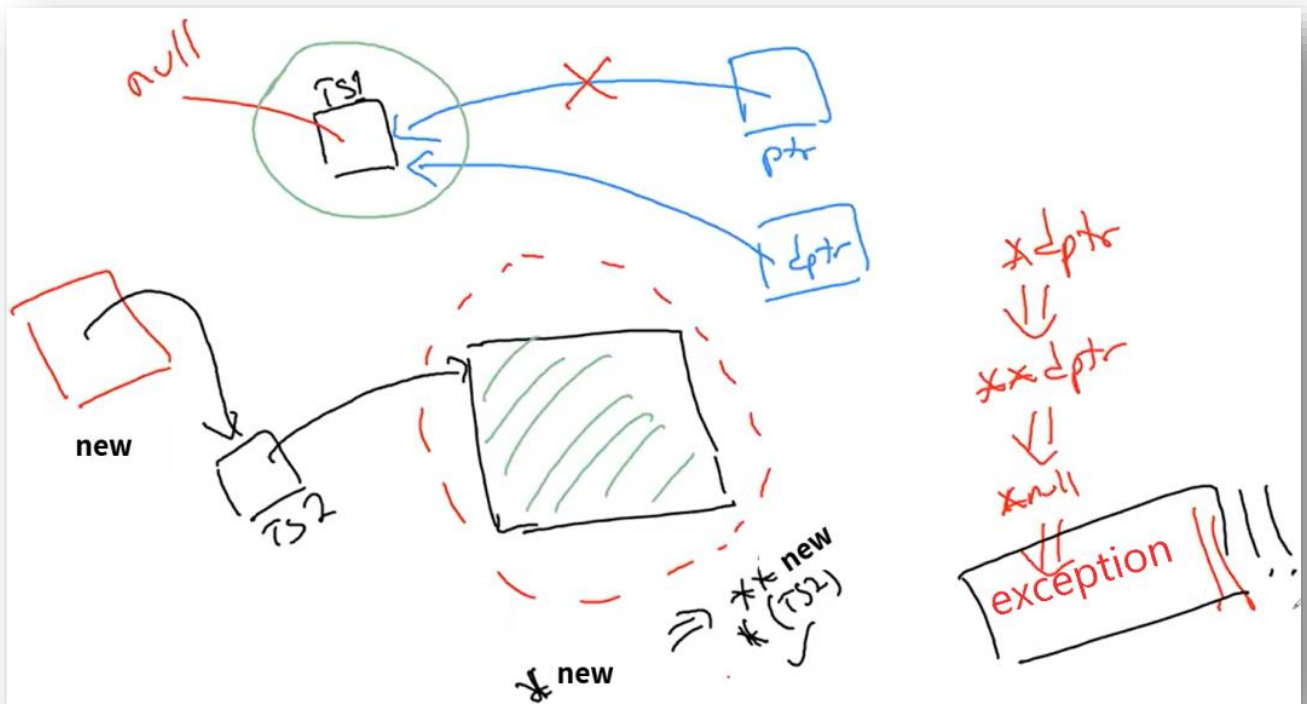
חשוב להדגיש, עצם זה שיש שגיאה זה לא בעיה, הבעיה היא שאני אפול בו, כלומר, אם אני מגלה את השגיאה בדרך כלשהי גם טוב. כפי שכבר הזכרנו את ה-graceful degradation, אמרנו שאחד מהיתרונות של זליגת זיכרון על גבי מצביע מתנדנד היא שאני יכול לדעת על קיום הבעיה ולטפל בה, וזה מה שקורה פה. כאשר אנו מנסים לגשת ל-null הוא יפגוש את ה-null וזה טוב לנו, הוא יבין שיש בעיה וירתיע בפנינו על כך. זה פתרון ולא בעיה. בכך שיצרנו לכל מצביע מצבה, אז פתרנו את בעיית המצביע המתנדנד.

גישה סטנדרטית נעשית ע"י כך שכאשר אנו כותבים $*ptr$, בפועל קורא $**ptr$, כלומר, יש כאן מצביע כפול. כפי שצינו בתחילת השיעור, הבעיה המרכזית מצביעים היא הגישה הכפולה, שכל פקודה הופכת לשתי פקודות. פה יותר גרוע, כל פקודה הופכת להיות שלוש פקודות. כאשר נעשה $dptr$, בפועל יקרה $dptr \leftarrow **dptr \leftarrow *null \leftarrow$ שזה יביא לשגיאה.

בעיית המצביע המתנדנד נפתרה, אך זליגת זיכרון לא, אלא להפך – הוחמרה. אם הזיכרון נעשה זבל, ע"י כך שכל המצביעים אליו נותקו ואנו לא יודעים איפה הוא נמצא. עכשיו יש לנו יותר זבל, את המצבה צריך להשאיר במקומה תמיד (ואז המצבה מצטרפת לזבל). לכאורה היינו מציעים שנמחק את המצבה כאשר אנו מנתקים ממנה את המצביע אליו, אבל (!!!) לא נוכל לעשות זאת מכיוון שאנו לא יודעים אם יש עוד מצביע אל המצבה ולכן היא תישאר במקומה עם ערך null. ע"מ שלא יהיה לנו מצביע מתנדנד אנו משאירים את המצבה במקומה ואז אנו מוסיפים זליגת זיכרון. בפתרון זה ניתן לראות שבחרנו לפתור את בעיית המצביע המתנדנד על פני זליגת זיכרון.

ניתן לייעל את המצבה במקצת. מה שנעשה הוא שנפריד בין המצבה לבין הזיכרון שלה. לפני כן המצבה הייתה ליד הזיכרון ולכן אם המצביע הופך להיות Null, אז גם המצבה וגם הזיכרון יהפכו לזיכרון זולג (זבל כאילו).

כאשר נשחרר את הזיכרון אנו נשים null במצבה הראשונה והיא לא תשחרר, אבל הזיכרון עצמו ישוחרר (!!!). מה שחשוב הוא שאם נרצה להקצות את הזיכרון עצמו שוב נצטרך רק להוסיף מצביע חדש לזיכרון הזה. למרות השינוי הזה, עדיין לא ישתנה מה שראינו עד עתה, כפי שניתן לראות באיור המצורף בצד ימין, עדיין $dptr$ יוציא שגיאה כמו שאנו רוצים, וגם אנו מצליחים לגשת לזיכרון הזה ע"י מצבה חדשה. הדבר היחיד שלא משתחרר הוא המצבה, אז סה"כ העלות היא רק מצביעים ולא כל הזיכרון שהוקצה לאובייקט כלשהו. אנו עדיין גורמים לזליגת זיכרון על פני מצביע מתנדנד, אבל אנו מרוויחים שזליגת הזיכרון תהיה קטנה יותר.

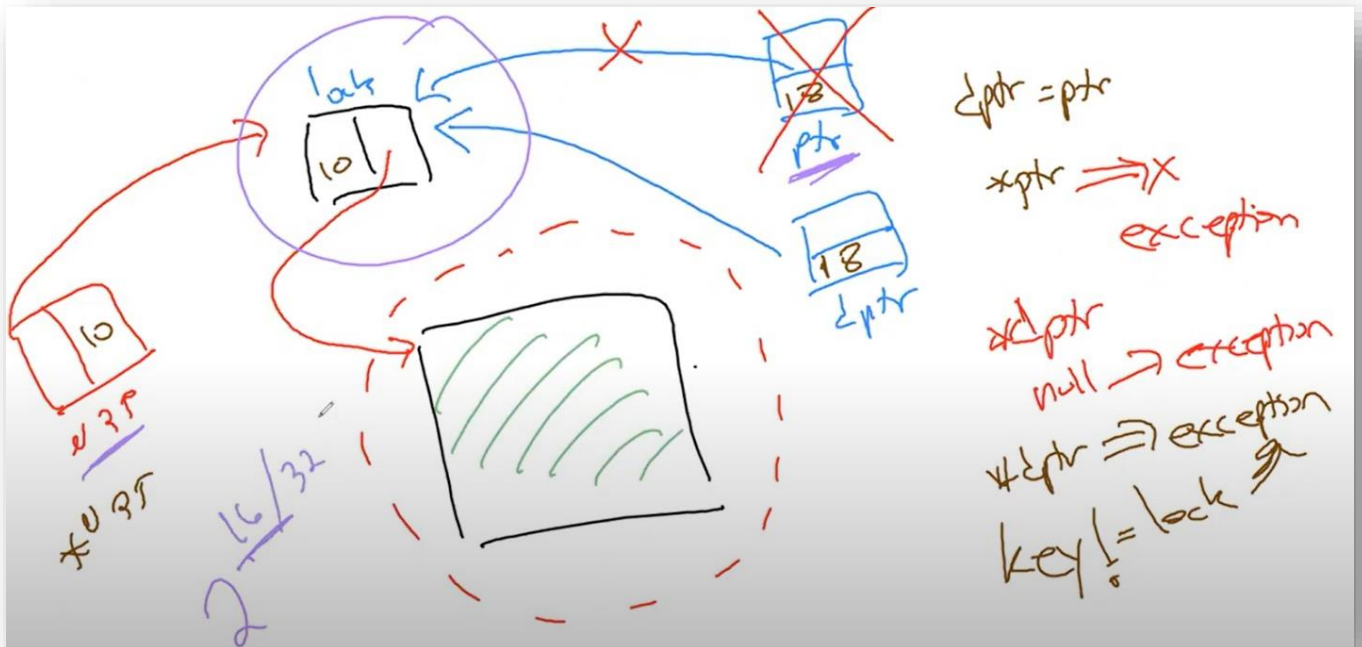


:Locks and keys

הרעיון דומה למצבה, במקום מצבה אנו נייצר מצבה שמורכבת משני חלקים. כאשר יהיה מצביע שירצה להצביע על הזיכרון הדינאמי, אז הוא יצביע על המצבה, כמו שכבר ראינו. מה שיקרה הוא שבחלק השני של המצבה יהיה מספר רנדומלי שהוגרל, וכאשר המצביע יוגדר להצביע על המצבה, הוא גם ישמור את המספר – המפתח בעצם. נניח אם יש לנו מצביע ptr אז כאשר נעשה $dptr = ptr$ אז $dptr$ יהיה כעת גם את המפתח (שבאיור למטה הוא 18). כאשר נעשה $*ptr$ (כמובן שזה מתורגם כפי שראינו ל- $**ptr$ וכו'), אז ייבדק האם המפתח שיש אצל המצביע שווה למנעול (שנמצא בחלק השני של המצבה), אם כן אז נמשיך הלאה – יש לנו גישה לזיכרון, אם לא – אז הוא לא יוכל לגשת לזיכרון ותיזרק שגיאה (כפי שכבר הזכרנו, שגיאה זה מצב טוב, זה אומר שאנו מודעים לבעיה ועושים graceful degradation).

ישנם שני מצבים שתיזרק שגיאה. מצב אחד שבו ptr נמחק ולכן המצבה נשארה עם אותו מפתח אבל היא מצביעה על null (כמובן שהזיכרון עצמו, מסומן באיור בירוק, השתחרר וכעת פנוי), ולכן כאשר ננסה לעשות $*dptr$ הוא יגלה שאנו מצביעים על null ויוציא שגיאה. המצב השני הוא, שלאחר ששחררנו את הזיכרון ו- ptr , אז בא מישהו חדש ומשתמש במצבה שלנו (באיור הוא קיבל גם את אותו זיכרון, אבל זה לא מחייב, הוא יכול לקבל כל זיכרון כי אמרנו שאין תלות בין המצבה לזיכרון). ולכן, יוקצה לו מנעול רנדומלי חדש (באיור למטה – 10), ואז כאשר ננסה לעשות $*dptr$ הוא יראה שאין ל- $dptr$ את המפתח המתאים למנעול ולכן יוציא שגיאה, כי $dptr$ הוא מצביע מתנדנד, שהרי הוא חושב שהזיכרון ההוא עדיין ברשותו.

שיטה זו מומשה שאחת מהגרסאות של Pascal, בשביל למנוע את בעיית המצביע המתנדנד. נשים לב שלכאורה הם פתרו את בעיית זליגת הזיכרון, שאפשר להשתמש באותו מנעול (אותה מצבה) עבור איבר חדש. כמובן, תמיד יכול להיות שהמצביע החדש הגריל את אותו מספר שהיה מצביע הישן, הסיכוי שזה יקרה זה 2 בחזקת הביטים של המנעול, ככל שכמות הביטים במנעול יותר גדולה אז הסיכוי קטן.



Smart pointers:

הפתרון של CPP לבעיות מצביע מתנדנד ולזליגת זיכרון.

ראשית נדבר על ההיסטוריה, ע"מ להבין את הרעיון טוב. בהתחלה המצביעים החכמים היו חזקים מאוד ולכן הם סורסו. קיימים כיום שלושה סוגי מצביעים חכמים שכל אחד לקח חלק מהיכולות של המצביע החכם הקדום.

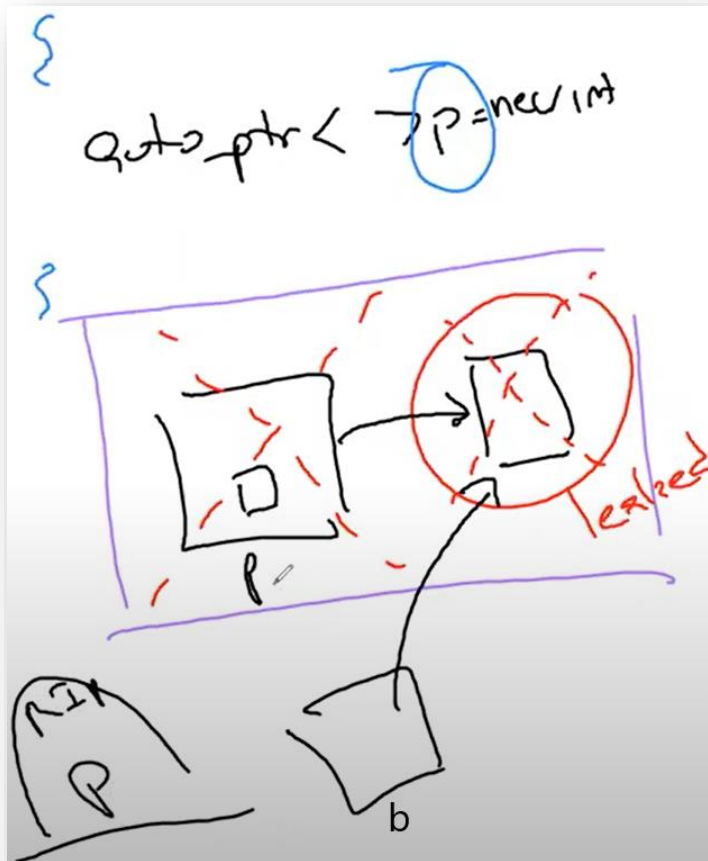
Auto - ptr (לא קיים יותר): בתוך STL של השפה היה class (כמובן שהוא היה template, אך בשביל הנוחות לא התייחסנו לזה כאן). כאשר יצרנו איבר מסוג המחלקה, אז נוצר איבר שבתוכו קיים שבפועל מצביע. כאשר אנו יוצרים איבר אז בסופו של דבר (לא משנה אם עשינו `new int` או `new int(&a)`, בכל מקרה כך זה יראה), כפי שניתן לראות בצד שמאל באיור (מוקף בסגול). הרווח של פתרון זה הוא מה-distractor.

נניח שהגדרנו את המצביע האוטומטי בתוך scope כלשהו, אז כאשר נצא מה-scope, אוטומטית נהרוס את האובייקט שיצרנו (באיור נקרא p), ה-distractor בעצם ימחק את האיבר שעליו מצביעים, כלומר ימחק את הזיכרון ממש ולא רק את המצביע. מכאן יוצא שפתרנו את בעיית זליגת הזיכרון (בעיה זו הייתה כאשר לנו זיכרון שהצבענו עליו ומחקנו את המצביע ולא יכולנו יותר לגשת לזיכרון הזה ומצד שני גם לא יכולנו להקצות אותו למישהו אחר, כעת זיכרון זה נמחק עם המצביע ולכן נפתרה הבעיה של זליגת זיכרון).

אבל, פתרון זה פתר את בעיית זליגת זיכרון על חשבון בעיית מצביע מתנדנד, הפתרון הזה יוצר מצביע מתנדנד (לדוגמא, כאשר b מצביע על a ומחקנו את a אז כעת b

עדיין חושב שהוא מצביע על a, אבל p מחק את האיבר a כאשר הוא קרא ל-distractor). יש כאן תמיהה גדולה על CPP שהם בחרו לפתור את בעיית זליגת הזיכרון ולייצר מצביע מתנדנד, כי ראינו עד עכשיו שכולם העדיפו אחרת. גם יוצרי auto-ptr לא רצו לייצר מצביע מתנדנד על פני זליגת זיכרון, ולכן יצרו מושג בשם "העברת בעלות" (Transfer of Ownership). ולכן, הוסיפו עוד 2 פונקציות למחלקה (כפי שניתן לראות במחלקה בשמאל).

הדרך להגיע בעצם אל המצב שמופיע באיור למעלה, זה ע"י כך שכותבים כך: `b = p` (אובייקטים מסוג auto-ptr), ולכן אנו נדאג בפונקציות החדשות שכאשר ירצו לעשות דבר כזה אז מה שיקרה הוא שיהיה רק אובייקט אחד שהוא הבעלים של הזיכרון כלומר, כאשר אנו עושים `b = p`, אז b כעת יצביע לזיכרון האמיתי, ו-p יצביע ל-null ולכן יהיה רק מישהו אחד שיוכל להשתמש בזיכרון כל פעם. תכונה זו



```
class auto_ptr {
public:
    auto_ptr(int* a) { ptr = a; }
    ~auto_ptr() { delete ptr; }
    operator* ...
    operator& ...

    // הוספו מאוחר יותר
    operator= ...
    auto_ptr(auto_ptr &a)...
private:
    T* ptr;
};
```


דואגת לכך שלא נגיע למצב שבו אם מוחקים מצביע אחד אז יישאר מצביע אחר מתנדנד (שהוא לא מודע למחיקה של הזיכרון).

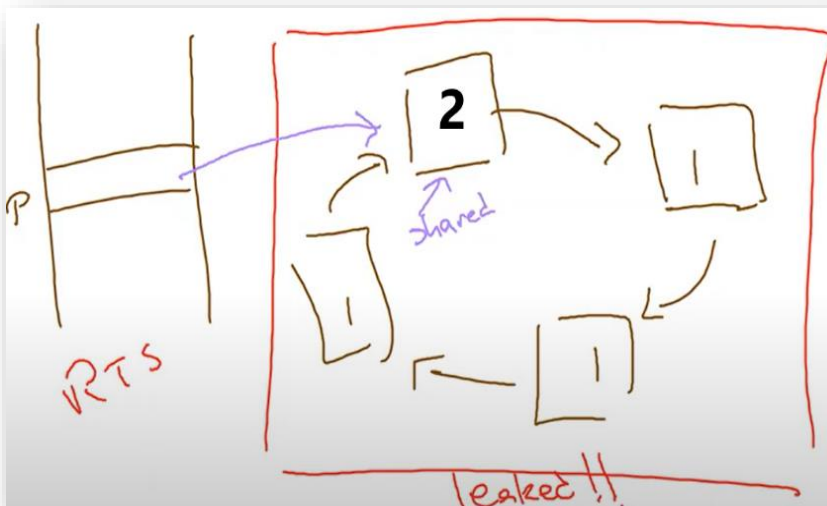
הערה: מימשנו גם copy constructor, לדוגמא לפעמים נראה: $p = \text{foo}(p)$, אז מה שקרה הוא שהפונקציה קיבלה את האובייקט p שלנו ועשתה לו העברת בעלות אל האובייקט שהיא מקבלת, ולכן אם נרצה שהבעלות תחזור אל p אז נכתוב בצורה הזו, וכן נדאג שבסוף הפונקציה יוחזר האובייקט שבתוך הפונקציה (ואז שוב תעבור הבעלות מהאובייקט שבתוך הפונקציה אל האובייקט p שלנו).

אין לנו בעיה שכעת אחד מהאובייקטים ינסה למחוק את הזיכרון, כי אם p ינסה למחוק אז הוא לא יוכל כי הוא מצביע על Null, ואם b ימחק אז לא תהיה בעיה מכיוון שהוא היחיד בעל גישה לזיכרון.

כפי שכבר אמרנו, auto-ptr לא קיים יותר, אבל קיימים unique-ptr (שקול ל-auto-ptr), share-ptr (שניים יכולים להחזיק זיכרון), weak-ptr (כמו share רק יותר חלש).

Unique – Pointer: שקול ל-Auto-ptr, כלומר, יש העברת בעלות (יש רק אחד שמחזיק בו זמנית את הזיכרון) וכו'.

Share – Pointer: יש פעמים שאנחנו כן נרצה שיהיו שנים שיצביעו לאותו זיכרון, ולכן נשנה קצת את הגדרת ה-distractor. יהיה counter שיגיד כמה מצביעים עליו ברגע זה (כאשר עוד מישהו יצביע עליו, שזה יכול לקרות ב-copy constructor אז נעשה counter++. כאשר מצביע אחד ישתנה ויצביע על אובייקט אחר אז נעשה counter--). כאשר מוחקים אובייקט אז נעשה counter--). ב-distractor יהיה כתוב שאם המונה יגיע לאפס אחרי שנוריד אחד (כאשר אנו באים למחוק מצביע ספציפי) אז delete ptr; כלומר, רק כאשר המונה יתאפס אנו נמחק לגמרי את הזיכרון. המנגנון הזה מונע זליגת זיכרון וגם מצביע מתנדנד.



נראה כעת מקרה בעייתי כאשר משתמשים ב-share-ptr: כאשר יש לנו מצביע p מתוך המחסנית (run time stack), ואנו נמחק את p אז החץ הסגול יתנתק והמונה יקטן באחד, אך כל שאר המצביעים שבמעגל שם יישארו, ולנו לא תהיה גישה אליהם והנה לנו זליגת זיכרון. מכאן אנו רואים שיש מקרים שאנו נרצה שזה יהיה משותף אך שהמונה לא יגדל (כי אז נוכל להגיע למצב של זליגת זיכרון כמו באיור כאן, שכל המצביעים במעגל ההוא בלתי נגישים). ולכן המציאו את ה-weak- ptr .

Weak – Pointer:

זה מצביע משותף שלא מגדיל או מקטין את ה-counter. אמנם יכולים להיווצר מצביעים מתנדנדים, אבל כל הרעיון הוא להשתמש במצביע מסוג זה רק כאשר אתה יודע שזה מה שאתה רוצה. לדוגמא באיור כאן, אנו נרצה שכל המצביעים במעגל יהיו weak, ולכן מראש המונה לא יהיה שווה ל-2 אלא ל-1, ואז כאשר אנו ננתק את p כל המעגל ייפול כמו מגדל קלפים (שהמונה של הראשון יהיה אפס ואז הוא יהרוס את עצמו ולכן גם המונה של השני יתאפס ולכן גם אותו נהרוס וכן הלאה). הרעיון הוא בעצם לקחת את auto- ptr ולחלק אותו לשלוש מחלוקת שונות שלכל אחת יש יכולות מסוימות שנצרכות במקרים שונים.

Move semantics בשפת C++:

```
a = b
string a(b) //copy-ctor of a
```

כאשר אנו מריצים את הקוד שבדוגמא בשמאל, אז אנו מצפים שבזיכרון a יצביע על איזה מחרוזת כלשהי, ו-b יצביע על העתק של אותה מחרוזת. עד לכאן הגיוני ופשוט. במקרה זה b הוא L-value, ולכן יהיו שני עותקים.

```
b = "hello"
c = "world"
string a(b + c)
```

נניח שנריץ את הקוד בשמאל, אז לפי מה שלמדנו כאשר אנו נקרא לבנאי העתקה הוא ייצר עותק של "hello world" ועל העותק הזה הוא יעבוד כקלט. הבנאי יעתיק את התווים מהמחרוזת הזמנית הזו (שהרי היא נועדה אך ורק בשביל הבנאי) אל זיכרון חדש שיהיה המחרוזת של a. ה-(b + c) הוא זמני (העותק שייצרו בכניסה לבנאי) ולאחר שנסיים את פעולת הבנאי לא תהיה דרך לגשת אליו, הוא חסר שם בזיכרון. במילים אחרות הוא R-value.

הרעיון של move semantics מציע שבמקום לייצר מחרוזת חדשה בשביל a, פשוט נגיד ל-a להצביע על הזיכרון הזמני ההוא ולא נמחק אותו בסוף הבנאי (כי כעת הוא בשימוש). למה לייצר זיכרון חדש, בואו נגנוב את הזיכרון הזמני ונשתלט עליו. במקרה שיש לנו זיכרון שהוא R-value בלבד אנו נדע שלא נצטרך אותו בהמשך, כי אין לנו דרך לגשת אליו בהמשך. רואים כאן את הרעיון של העברת בעלות.

```
class DC {
public:
    DC(DC &rhs); //copy-ctor
    DC(DC &&rhs); //move-ctor
    operator=(DC rhs);
};
```

ניקח לדוגמא את המחלקה כאן בשמאל. המימוש של בנאי העתקה אומר תייצר זיכרון נוסף ותעתיק מ-rhs. **יש עוד בנאי העתקה.** ההבדל במימוש הוא שהוא לא ייצר זיכרון חדש ויעתיק לתוכו ואז ימחק את הזיכרון הזמני שנשלח אליו (כמובן כאשר זה מה שנשלח, ועל זה כעת אנו דנים), הוא פשוט יגיד לו תצביע לזיכרון הזמני שקיבלנו.

כאשר הקומפיילר רואה פקודה כגון: DC(DC a), הוא צריך להחליט האם הערך הנשלח הוא R-value או L-value. אם זה R-value הוא יקרא לבנאי השני, ואם לא אז הוא יקרא לבנאי העתקה רגיל.

רעיון זה יהיה נכון גם **לאופרטור =**. המימוש תמיד יהיה תיקח ל-Rhs את הזיכרון. נניח אם נכתוב a = b, a הוא L-value, אז מראש נקרא לבנאי העתקה הרגיל, ואז נוצר העתק זמני (כי יש קריאה לבנאי העתקה, מסומן בעיגול בקוד). אם הקריאה לאופרטור הייתה עם R-value, אז היינו קוראים לבנאי העתקה השני ואז כשנתחיל את האופרטור יהיה לנו R-value, ולכן נעתיק אותו ולא נייצר אחד חדש. רואים שבכל מקרה אנו מקבלים בכניסה לאופרטור = ערך שהוא R-value, ולכן במקום לייצר זיכרון חדש פשוט נגנוב את הזיכרון ממה שקיבלנו, ובמילים אחרות **move semantics**.

כל זה הגיע מ auto-ptr כשייצרנו את המושג "העברת בעלות", גילו שיש לזה שימוש נוסף מאוד מעניין, שאפשר במקום בנאי העתקה ואופרטור שווה הפשוטים, אפשר לייצר בנאי חדש בשביל R-values, ומכך נוצר נושא חדש של move semantics, שמייצל את כל התהליך – שלא צריך לייצר להעתיק ולמחוק, אפשר פשוט לקחת את השני וזה עדיין יעבוד בצורה נכונה.

SCOPES

מה זה scopes:

בא לנהל גישה לזיכרון. מוסיף קריאות לזיכרון. יעילות – ניתן למחוק דברים שאנו לא רוצים שיהיו.

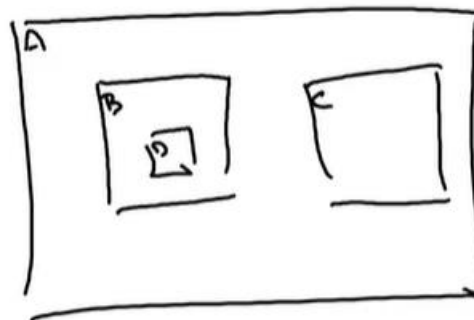
הדבר המרכזי של scopes הוא להגדיר את המובן של ישויות וביטויים. כלומר, כאשר אומרים לי a זה reference למיקום בזיכרון, אז על מי מדובר. משתמשים במרחב הזה להגבלת גישה למשתנים כאלו ומענק גישה למשתנים אחרים.

בנוסף, יש שם את הנושא של ניהול מורכבות ומנושא זה נגיע בהמשך לנושא של פונקציה (סוג של scope). הרעיון הוא לייצר יחידה סגורה ואז אני יכול להתייחס ליחידה הזו מבלי להתייחס למה שמחוץ לה. מה שאנו נקבל זה סוג של קופסא שחורה שלה יש קלטים ופלטים מסוימים (לדוגמא, פונקציות). יש בפונקציות ניהול מורכבות, אנו לא צריכים יותר לקחת בחשבון מלא דברים מהרבה כיוונים אני יכול להסתכל על ה-scope כנקודת התחלה וסוף ולהסתכל עליו כיחידה עצמאית (ברור כמובן שאני יכול גם ביחידה עצמאית זו לגשת למשתנים מבחוץ, והם יכולים להשפיע עלי ועוד כל מיני כאלה). נתייחס לנושא זה יותר כאשר נעסוק בפונקציות, מכיוון שקשור יותר לפונקציות ומחלקות. לדוגמא לולאה אני יודע שהיא חיה לבד, בלי קשר למה שמסביב, יש לה את המשתנים שלה, את התנאים שלה, לכאורה מאוד מנותקת מסביבה. רוב הנושא של scope הוא בלוק המגדיר את המובן של ישויות הנמצאים בתוכו, מגביל את טווח הגישה וכו'.

סוגי scopes:

נקודה ראשונה – הציור מימין שקול להגדרה בקוד משמאל. כאשר מדברים על הכלה של scope בחברו מבחינת פונקציות, הכוונה להגדרה של הפונקציה ולא לקריאה.

```
function A() {
  function B() {
    function D() {}
  }
  function C() {}
}
```



נקודה שנייה – כאשר יש לנו לדוגמא בתוך הפונקציה D, משתנה x ואנו מדפיסים אותו אז ברור שהכוונה אליו, לא משנה סוג ה-scope, מכיוון שמדובר על קישור לוקאלי (מקומי).

אבל, במקרה של הציור כאן משמאל (ראה עמוד הבא), השאלה שאותנו מעניינת היא – על איזה x מדובר. נניח שסדר הקריאות לפונקציות הוא כפי שמתואר בתחתית האיור. A יכול לקרוא ל-C, מכיוון שהוא כמו משתנה מקומי של A. C יכול לקרוא ל-B מכיוון שהם אחים. C לא יכול לקרוא ל-D ישירות, מכיוון שהוא מוכר רק בתוך B.

התשובה לשאלה שלנו תלויה בסוג ה-scope. עבור סטטי אזי מדובר על ה-x של A. ואם עוסקים בדינאמי, אז מדובר על x של C.

Static scope: מוכר לנו מרוב השפות.

Dynamic scope: נמצא יותר ב-LISP, APL, PERL.

נזכיר ש-Static בקורס שלנו הכוונה בזמן קימפול, ו-Dynamic הכוונה בזמן ריצה. אם הכוונה בזמן קימפול, הכוונה כשהקומפיילר רואה את ה-x בקריאה אז הוא צריך להחליט על איזה x מדובר. הקומפיילר רואה את התמונה הסטטית, אך לא מודע לסדר הקריאות, ולכן לא יכיר את ה-x שנמצא אצל C ולכן יבחר ב-x שנמצא ב-A.

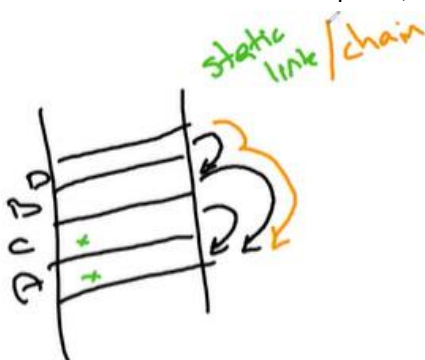
מציאת נתון שאינו מקומי:

אז השאלה היא, איך נמצא את המשתנה הנכון? דבר

ראשון, תמיד נשים לב שהמשתנה הנכון, אם הוא קיים, אז הוא נמצא על ה-run time stack (RTS). יכול להיות שהוא נמצא קרוב אלינו, ויכול להיות שהוא רחוק מאוד. אם לא נמצא אז שגיאה. במקרה שלנו, לא יכול להיות מצב שבו הפונק' D חיה ו-A לא חיה (כי D מקומי עבור A), גם אם A מוקפא הוא עדיין נמצא במורד המחסנית. ע"מ למצוא את השתנה, אנו צריכים הפנייה לנתון לא מקומי, ובשביל זה אנו צריכים לעשות שני דברים. דבר ראשון, אנו צריכים למצוא את רשומת ההפעלה הנכונה, שבו נמצא המשתנה שלנו. אנו לא יודעים איפה היא נמצאת או כמה צריך לרדת למטה, אין דרך לדעת זאת (לדוגמא, קריאה רקורסיבית אז עבור כל קריאה יש רשומה אחת במחסנית). בזמן הקימפול אנו לא יודעים כמה קריאות יהיו לפונקציה (כלומר, כמה רשומות של הפונק' הרקורסיבית יהיו ב-RTS). דבר שני, לאחר שמצאנו את רשומת ההפעלה, נצטרך למצוא גם את ה-offset בתוך רשומת ההפעלה הזו בשביל למצוא את המשתנה שלנו (רשומת ההפעלה מתחילה במקום כלשהו ב-RTS, ובמיקום כלשהו יחסית לתחילת הרשומה נמצא המשתנה. הרעיון דומה למשתנה מקומי, כיוון שהקומפיילר יודע את המבנה של רשומת ההפעלה של scope כלשהו, אז הוא יכול להגיד לנו גם מה ה-offset הנדרש ע"מ למצוא את המשתנה.

קישור סטטי:

נתייחס כעת למציאת רשומת ההפעלה אנו בעצם מייצר ברשומת ההפעלה נוסף בשם Access Link Static. תפקיד משתנה זה הוא להצביע על האבא הסטטי של רשומת ההפעלה. למשל, עבור רצף הקריאות שתיארנו לעיל, כך יראו הצבעות המשתנים static link של רשומות ההפעלה. אוסף לינקים כאלו, יוצר שרשרת, ולה קוראים Static Chain.

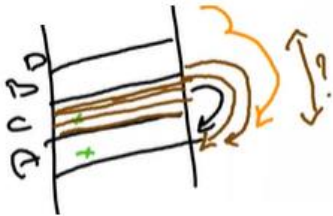


תפקיד שרשרת זו הוא לקשר את כל האבות הסטטיים של frame מסוים. אז כאשר אני מחפש את המשתנה הנכון, מה שאני צריך לעשות הוא לסרוק את ה-static chain ולחפש עבור המשתנה הזה. במקרה שלנו, אנו נסרוק ונקפוף מ-D אל B, ומשם אל A, ושם נמצא את המשתנה x שלנו. נשים לב, שלא נגענו ב-frame של C, דילגנו עליו בכלל. נגיע ישירות ל-A, ושם נמצא את x. חשוב לציין, בזמן ריצה אנו לא עומדים להתחיל לסרוק ולחפש, גם בגלל שהשם x לא נמצא על המחסנית (הוא נעלם בזמן קימפול, הוא משמש רק כ-reference, התייחסות לכתובת בזיכרון) אז קצת קשה לסרוק כאשר אנו לא יודעים את מי לחפש. מה שקורה הוא, שהקומפיילר בזמן קימפול כבר יכול להגיד לנו כמה לינקים צריך לעבור בשרשרת הסטטית כדי לקבל את רשומת ההפעלה הנכונה. אנו נגדיר לכל קיבול ולכל scope משהו בשם עומק קיבול. לדוגמא, אם עומק הקיבול של A הוא 1, אז עומק הקיבול של B ו-C יהיה 2, ושל D יהיה 3. הקומפיילר יכול להגדיר לכל scope כמה הוא מקובל מה-scope החיצוני הגדול. ע"מ למצוא את רשומת ההפעלה הנכונה, אנו מסתכלים על הפרשי עומקי הקיבול בין הפונקציה הקוראת לבין המקום שבו מוגדר המשתנה שאנו מחפשים. בנידון דידן, הפונקציה הקוראת היא D שעומקה הוא 3, והמשתנה שאותו אנו

מחפשים (הקומפיילר יודע ש-x נמצא ב-A) נמצא בעומק 1, ולכן הפרשי עומקי הקינון יהיו 2. לכן, הקומפיילר ישתול קוד כך שבמקום הקריאה ל-x, הקומפיילר יוסיף את הפקודה הבאה "תצעד פעמיים על ה-static chain", וכאשר נעשה זאת נגיע ל-A ושם נמצא את x. כלומר, בזמן קימפול הקומפיילר יכול להגיד לי כמה אני צריך לצעוד בתוך ה-static chain ע"מ למצוא את רשומת ההפעלה הנכונה. כפי שכבר אמרנו, ע"מ למצוא את המשתנה בתוך רשומת ההפעלה, זה כבר ידוע לקומפיילר, מכיוון שהוא בונה את רשומת ההפעלה (בדיוק כמו שמשתנה מקומי יודע את ה-offset הנכון). ולכן, כבר בזמן קימפול, הקומפיילר שותל קוד שאומר "תצעד k צעדים על ה-static chain ושם תמצא את רשומת ההפעלה הנכונה; לאחר שמצאת אותה, ב-offset כלשהו נמצא המשתנה שלך". כאשר k מוגדר להיות ההפרש בין עומק הקינון של הפונקציה הקוראת לבין המיקום בו הוגדר המשתנה. וכך יוצא שבזמן ריצה, אנו לא צריכים לחפש את המשתנה, אלא לעשות פעולות כלשהן ובכך למצוא את המשתנה הרצוי.

יצירת קישור סטטי בזמן ריצה:

עד עכשיו פעלנו בהנחה שיש static link, ואם כן אז נציל אותו ע"מ למצוא את המשתנה. כעת נעסוק בדרך בה אנו יוצרים את ה-static link. נניח לדוגמא, ש-C הייתה רקורסיבית, אז לא נכון להגיד "תרד 2 למטה ושם נמצא A", כי אם C רקורסיבי והוא רץ k פעמים, אז צריך לרדת k+1 רשומות למטה, ולקומפיילר אין שום דרך לדעת בזמן קימפול את הדבר הזה. ולכן, חייבת להיות שיטה אחרת שכאשר נבנה את ה-frame הבא נדע איפה נמצא האבא הסטטי. לבינתיים, הנחנו שהאבא הסטטי קיים, עכשיו השתמשנו בו, רצנו על השרשרת, מצאנו את הרשומה הנכונה, אבל איך מייצרים את ה-static link.



הקומפיילר בזמן קימפול שואל את עצמו – מה הם הפרשי הקינון בין הפונקציה הקוראת לאבא של הפונקציה הנקראת. למשל, אצלנו C קרא ל-B (במקרה ש-C רקורסיבי, אז הקריאה העכשווית בוצעה מהקריאה ל-C האחרונה ביותר במסגרת, כלומר העליונה ביותר), מה הפרשי הקינון בין הפונקציה הקוראת – C, לבין האבא של הפונקציה הנקראת (הפונקציה הנקראת-B) – A. הפער יהיה $2-1=1$, וכאשר נבנה את רשומת ההפעלה B הקומפיילר יגיד שהאבא הסטטי של B נמצא צעד אחד על ה-static chain כשאתה יוצא מ-C. לגבי C אנו יודעים איפה הוא, הוא העליון כרגע, אז האבא הסטטי נמצא במרחק קפיצה אחת כאשר אתה יוצא מ-C. כלומר, נלך ל-C ואצלו נקפוץ קפיצה אחת על השרשרת הסטטית – שזה A. נשים לב שלא משנה לנו כל הקריאות הרקורסיביות ל-C והצלחנו להגיע ישר למקום הנכון.

נראה עוד דוגמא להבנה, נחפש את האבא הסטטי של D. עומק קינון הפונקציה הקוראת, שזה $B=2$, עומק הקינון של הפונקציה הנקראת, כלומר אבא של D שזה $B=2$. ההפרש ביניהם יהיה אפס, שמשמעותו היא: האבא הסטטי של D נמצא 0 מעברים על ה-static link כשיוצאים מ-B, כלומר B עצמו. הקומפיילר עזר לנו להגיע מה-farme החדש שאנו בונים לאבא הסטטי שלו, ולא משנה בכלל כמה frames יש ביניהם. וכך נבנה את ה-static link וניצור static chain כאשר אורך המעברים שצריך לעבור על השרשרת הוא מקסימום כגודל הקינון ולא יותר מזה, ולא תלוי במספר הפונקציות הנקראות באמצע.

כאשר נצא מ-scope ונרצה למחוק static link אז פשוט נמחק את ה-farme ואוטומטית נמחק ה-link. הוספת link, ראינו עתה. מציאת ה-frame הנכון, ראינו מקודם כיצד הקומפיילר משתמש בשרשרת הסטטית ע"מ למצוא זאת בזמן קימפול (והשתמשנו במידע מהקומפיילר בזמן הריצה).

חסרונות הקישור הסטטי:

החסרון הגדול ביותר של ה-static chain, נעוץ בעובדה שזמן הגישה למשתנים הוא לא קבוע. ככל שהמשתנים רחוקים יותר מבחינת עומקי קינון, כך זמן הגישה גדול יותר. כלומר, הפרשי קינון של 5 יגרמו לזמן גישה גדול יותר מאשר גישה למשתנה לוקאלי, וזו תכונה לא רצויה. אנו משתמשים במשתנים אלו כאילו הם לוקאליים אבל למעשה זמן הגישה אליהם הרבה יותר גדול. מחקרים שעשו גילו שאין עומקים גדולים מידי כי מתכנתים לא תופסים ראש עמוק מידי, אבל זה לא פותר את הבעיה של זמני הגישה.

בעיה נוספת היא, ששינוי מבנה התוכנית משנה את זמני הגישה למשתנים מסוימים. כלומר, אם נשנה את המבנה (refactoring וכו' לקוד) אזי זה ישנה את זמני הגישה לנתונים, וזו גם כן תכונה לא טובה. כי אם אנו מתעסקים במערכות זמן אמת, אנו לא רוצים שהזמנים יהיו תלויים במבנה וצורת הקוד, ואלו בעיות ששפות כמו C++ העדיפו שלא להתמודד איתן.

Un-named scopes:

כלומר, יש חסרונות בשרשרת סטטית ולכן שפות מסוימות לא יסכימו לממש אותה בשפה שלהם. לדוגמה, שפת C, לא מוכנה לאפשר קינון scope אחד בתוך השני במובן שאי אפשר להגדיר פונקציות כמו בדוגמה שדנו עליה.

מה שמעניין בשפת C, היא שאע"פ שאי אפשר להגדיר scopes בצורה רגילה, ניתן לקנן scopes. כלומר, לא ניתן להגדיר

```
void foo()
{
    int DC()
    {
        ...
        ...
    }
}
```

```
void foo()
{
    int x, y, z;
    {
        int x;
        cout<< x+y;
    }
}
```

scope ולקנן (ראה דוגמה בקוד השמאלי). אבל לקנן scopes זה כן אפשרי (כפי שניתן לראות בדוגמה הימנית). ה-x הפנימי יותר ימות כאשר נצא מה-scope שבו הוא הוגדר, בעוד ש-x,y,z ימותו בסיום ה-scope הגדול יותר.

כלומר, ניתן לראות מכאן, שפונקציות אי אפשר לקנן, אבל scopes רגילים ניתן לקנן. ע"מ להבין זאת, ראשית נשאל את עצמנו מה ההבדל המרכזי בין הקוד ימני לשמאלי.

ההבדל המרכזי הוא שה-scope השמאלי הוא named, ניתן לו שם, ואילו הימני הוא un-named. זה חשוב, מכיוון שזה

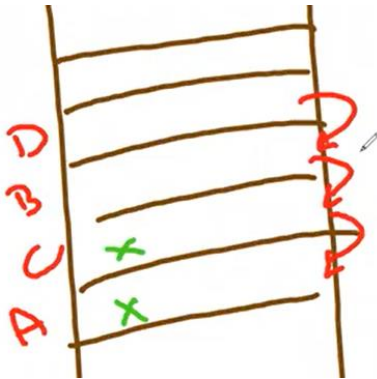
אומר שלקוד בצד שמאל ניתן לקרוא בצורה א-סינכרונית, כלומר הקוד לא חייב לרוץ לפי הסדר, אלא אפשר לרוץ ולדלג על DC ואז בסוף foo לקרוא ל-DC. מכיוון שיש לו שם אזי ניתן לגשת אליו בצורה לא סידרתית, א-סינכרונית. לעומתו, בקוד בצד ימין לא קיימת האפשרות הזו, ותמיד נרוץ באותו הסדר, ולכן אנו לא חייבים לממש את ה-scope הפנימי בימין כמו זה השמאלי. את ה-scope השמאלי אנחנו חייבים לייצר כ-frame חדש ולחזור אחורה לכל מה שדיברנו עד עכשיו. אבל, את צד ימין ניתן לממש אחרת, נתעלם מהסוגריים של ה-scope והקומפיילר ישנה כל מקום שכתוב x ל-x' (אך ורק בתוך ה-scope הפנימי), ואז יצרנו לנו משתנה מקומי רגיל ואין שום הבדלים בזמן הגישה ל-x ול-y. בפועל, או שהקומפיילר באמת משנה את המשתנה ל-x' (ואז כל ניסיון גישה ל-x יומר להיות גישה ל-x'), או שהוא פשוט ישתמש במשתנה שלא חיים בו זמנית (לדוגמה, אם יש שני scopes אחד אחרי השני, אז הוא יכול להשתמש במשתנה מה-scope הראשון בשביל השני). הנקודה היא שהמשתנים המקומיים בתוך ה-scope הפנימי הם יהיו אותו דבר כמו המשתנים המקומיים של ה-scope החיצוני (פכי שכבר אמרנו, זמן הגישה אליהם יהיה אותו דבר).

ולכן, שפות מבוססות C יתנו את האופציה ל-scopes, כי זה נותן לנו דברים רבים (כפי שכבר הזכרנו – מורכבות, ניהול וכו'). אבל כל זה רק בגלל שסוג ה-scope במקרה הימני הוא un-named, כלומר שאי אפשר לגשת אליו בצורה לא סינכרונית. אם היינו יכולים לגשת אליו בצורה לא סינכרונית אז הייתי חייב לייצר לו שם וגם farne וכו', ואלו דברים ששפות המבוססות על C לא עושות.

קישור דינאמי - Dynamic Scopes:

הפנייה לנתון שאינו מקומי מצריך גישה ל-frame כלשהו ב-RTS (Run Time Stack), ולאחר שמצאתי את ה-frame צריך למצוא את ה-offset המתאים בתוך ה-frame. כלומר, גם פה, כמו בסטטי, צריך לעשות אותו דבר. צריך למצוא את הרשומה הנכונה ולאחר מכן למצוא את ה-offset בתוך הרשומה.

הבעיה היא, שלמצוא את הרשומה במחסנית כעת יותר מסובך. נעסוק באותה דוגמא ממקודם, שהיה משתנה x ב-C

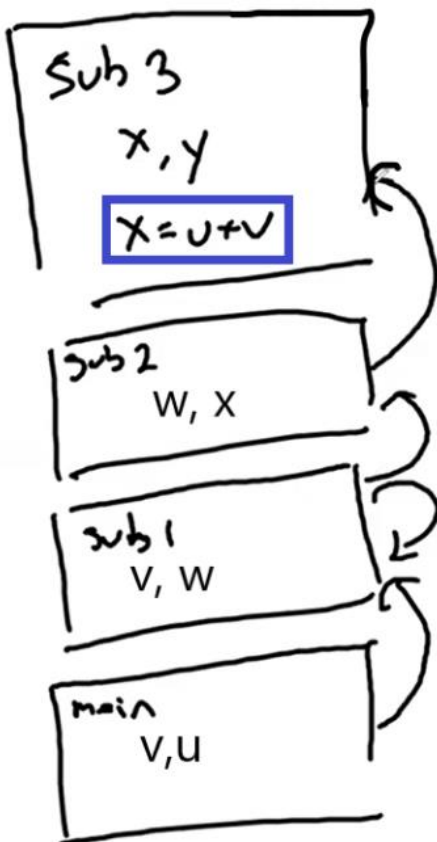


ומשתנה נוסף x ב-A. כאשר נכתוב $cout < x$ ב-D, אז יהיה מדובר על ה-x של C, מכיוון שאנו מדברים על scope דינאמי, אנו מסתכלים על סדר הקריאות, D יסתכל

על B ואז יסתכל על C וימצא שם x. **סדר החיפוש הוא לפי סדר דינאמי, סדר קריאות, ולא לפי הסדר הסטטי.** ניתן לראות את מבנה המחסנית באיור המצורף בשמאל. ברור שכאשר הוא יחפש במורד ה-dynamic chain הוא ימצא את ה-x שנמצא ב-C לפני זה שנמצא אצל A.

כדי למצוא את הרשומה יש שני דברים - shallow access ו-deep access.

ב-deep access מתחילים לעבור על השרשרת הדינאמית (האדומה, מגדירה את סדר הקריאות בצורה הפוכה) ומחפשים את הנתון הרצוי. אין דרך בזמן קימפול לדעת כמה מעברים נצטרך לעבור על השרשרת. נניח ש-B רקורסיבי, אז אין לנו דרך לדעת כמה צריך לרדת עד שנגיע - C, ולכן בזמן ריצה נלך לחפש את הנתון וזה לא טוב לנו. חסרון נוסף, בדר"כ אנו רגילים לחשוב שבתוך ה-RTS אין שמות אלא כתובות, וב-deep access אנו חייבים לשמור את שמות המשתנים. כאלטרנטיבה לזה, משתמשים ב-shallow access. ראשית, נציג את המושג בצורה אחת ע"מ להבין ולאחר מכן נגדיר בצורה מדויקת יותר.



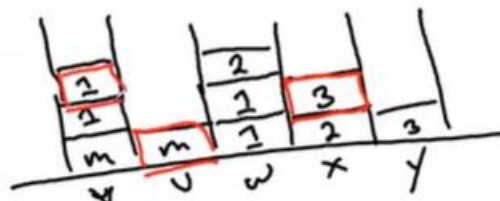
כל פעם שאנו נכנסים ל-scope אנו מגדירים מחסנית שבה ישמרו המשתנים. המקרה שנעסוק בו כעת, הוא הקוד בשמאל וכן סדר הקריאות הבא:

$sub3 \rightarrow sub2 \rightarrow sub1 \rightarrow sub1 \rightarrow Main$

השאלה שנשאלת היא, עבור השורה ב-sub3, על איזה u ו-v מדובר, כי יש כמה כאלו בקוד. כאשר מייצרים משתנה בשם כלשהו נוצרת מחסנית בשביל משתנים בעלי השם הזה. אז כאשר אנו נראה את המשתנה u, v ב-Main, אנו נייצר שתי מחסניות, אחת עבור משתני v ואחת עבור משתני u. למחסנית אנו נכניס את המשתנים הללו ונסמן אותם שהם שייכים ל-main. לאחר מכן נגיע לפונקציה sub1, ונראה את w, v, אז את v נכניס למחסנית הקיימת עבור משתנים בשם v, ועבור w אנו נייצר מחסנית חדשה וכן הלאה. כאשר ניכנס שוב ל-sub1 (שימו לב, sub1 הוא רקורסיבי לפי סדר הקריאות הוא קורא לעצמו ורק אז קוראים ל-Sub2), אז אנו נכניס את המשתנים של הקריאה השנייה של sub1 למחסנית המתאימה לשם המשתנה. וכן עבור Sub2 ועבור sub3. אז כעת, כאשר נבדוק עבור הפקודה $x = u + v$ מיהם המשתנים

המדוברים, התשובה

תהיה ה-x העליון וכן ה-u וה-v העליונים במחסנית. כלומר אלו יהיו המשתנים שנבחרו. נשים לב,



ה-v שנבחר הוא זה של sub1 השני. את המציאה של המשתנה כעת עשינו מבלי לחפש ולהתאמץ, רק הלכנו למחסנית ולקחנו את העליון. כל פעם שמישהו חדש נכנס הוא מסתיר את אלו שהיו לפניו. נשים לב שזמן הגישה עבור כולם הוא שווה, הגישה למשתנה המקומי x הייתה שווה לזמן הגישה למשתנה u שנמצא בכלל ב-main.

אז איך מנהלים את המחסניות הללו? כל פעם שנכנס ל-scope חדש ונגדיר משתנים חדשים אז נבדוק אם קיימת מחסנית עבור המשתנים בשם הזה, אם כן אז נדחוף אותו למחסנית, ואם לא קיים אז תיצור מחסנית ותדחוף אותו. כל פעם שנצא מ-scope אנו נלך ונמחק את כל המשתנים של ה-scope המדובר, ואם המחסנית התרוקנה אז נמחק אותה.

חסרונות הקישור הדינאמי:

חסרון גדול מאוד הוא הגישה. ייצור המתשנים, מציאת שמות המחסניות, האם קיימת מחסנית בשם הזה או לא, לייצר מחסנית חדשה וכו' יקח כמובן הרבה יותר זמן.

בנוסף, יש לנו בעיה מבחינת סוגי המשתנים. מי אמר שסוג המשתנה x שיש ב-sub3 הוא אותו סוג שיש ב-sub2? יכול להיות שהם מסוגים שונים ולכן יהיו בעלי גדלים שונים ויוצא מכך שהמחסנית תצטרך להכיל משתנים שונים מסוגים שונים וזו כבר בעיה, ניהול המחסנית הזו כבר יהיה מסובך.

ע"מ לפתור זאת, מה שמקובל לעשות ב-shallow access הוא שלא נשים את המשתנים במחסנית, אלא אותם נשמור על ה-RTS. במחסנית נשים מצביעים ל-RTS, למיקום של המשתנים. אז עדיין זה נכון שכאשר ניכנס ל-scope חדש עם משתנה חדש עדיין נצטרך לנהל את המחסניות וגישה וכו', אבל עדיין נוכל פשוט לגשת למיקום העליון במחסנית וניגש ע"י המצביעים למשתנה הרצוי. וכעת, פתרנו אתה הבעיה שיש לנו כמה סוגי משתנים במחסנית, כי כעת יש לנו מחסניות מסוג אחד בלבד, מצביעים באותו גודל ואז אין יותר את הבעיה של סוגים שונים. אב, עדיין יש את הבעיה של הזיכרון, כי עבור כל משתנה אנו נחזיק שני דברים – המשתנה עצמו (ב-RTS) ומצביע למשתנה (במחסנית שעליה למדנו כעת). כמו כן, כעת הגישה נהיה איטית יותר, כי כעת ניגש למצביע ומשם נלך למשתנה, כלומר כעת אנו ניגשים בגישה כפולה (double access) במקום גישה ישירה. אז הרווחנו שלא נצטרך לטפל בסוגים שונים ושילמנו בגישה ותוספת זיכרון. ולזה קוראים shallow access (זו ההגדרה המדויקת עבור shallow access).

GARBAGE COLLECTION

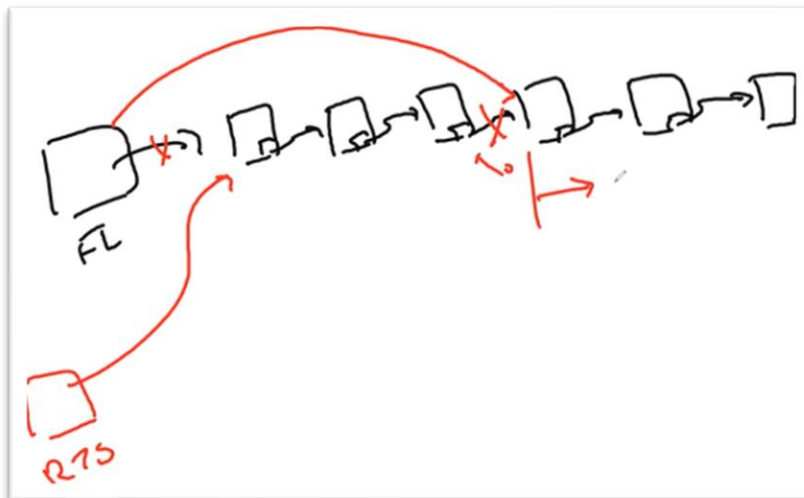
מה זה scopes:

דיברנו על scopes ו-type system וראינו שיש בעיות של זליגת זיכרון. עיקר הזמן שבו זיכרון זולג הוא בזמן יציאה מ-scope. כאשר אנו יוצאים מ-scope אז נוצר זיכרון שלכאורה משוחרר והוא הופך להיות זבל.

אז יש כמה אלגוריתמים שמטפלים בנושא, אנו נעסוק בבסיסיים שבהם, ומשם נתקדם לראות מה קורה כיום.

זבל – זיכרון שאינו נגיש. למשתנה ניתן לגשת רק מה-RTS, כלומר כל משתנה בעל שם נמצא על ה-RTS, והגישה לשם היא ע"י כתובת מהקומפילר (לא משנה אם סטטי או דינאמי), כאשר אנו אומרים משתנה – בסופו של דבר נגיע לאיזה כתובת על ה-RTS. מה-RTS יש גישה ל-heap, כלומר אנו צריכים לעבור דרך ה-RTS בשביל להגיע ל-heap אל המשתנה עצמו. אז להגדיר משתנה לא נגיש, אז לכאורה ההגדרה הכי פשוטה היא משתנים שאין reference אליהם מה-RTS (כמובן, אם יש מישהו שה-RTS לא מתייחס אליו ישירות, אלא יש ביניהם "מתווך", הוא עדיין נגיש, כלומר לא צריך גישה ישירה מה-RTS). כלומר, זבל – משתנה שאין גישה אליו דרך ה-RTS (← אין גישה אליו בכלל). איסוף זבל זה סוג של פתרון עבור memory leak שעסקנו בו, יש פה דליפת זיכרון, רק שבמקום להשים מצבה (ראה פרק 4) אז נראה פתרון אחר.

ישנם שני סוגי Garbage collectors והם הפוכים אחד מהשני. הראשון – Reference counting, השני – GC (= Garbage Collector). כאשר אומרים GC מתכוונים לאחד מהם, ולא דווקא לשני. ראשית נעסוק ב-reference counting. נניח

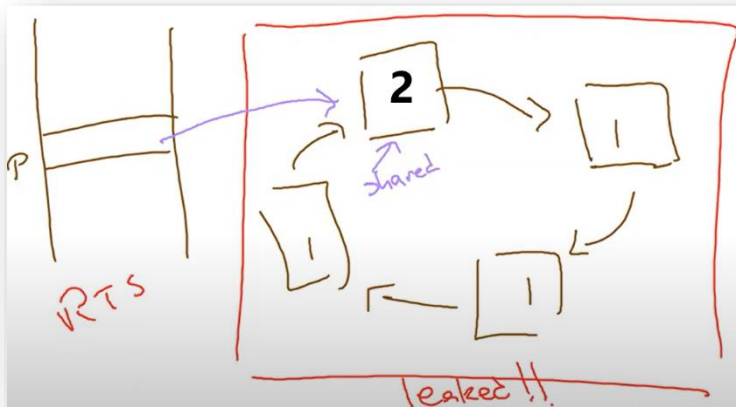


שאפשר להגדיר ולשחרר זיכרונות רק בגדלים קבועים (כלומר, כעת נעסוק בגדלים קבועים), כלומר, ניתן להקצות ולשחרר רק גדלים קבועים. הנחה שנייה – יש לנו איזשהו free list שמצביע על כל הזיכרון, ונניח שלכל תא בזיכרון יש את המידע שלו וכן מצביע ל-next, ואז אם נרצה נגיד להקצות פי 3 מהגודל הסטנדרטי אז נגדיר מצביע מה-RTS אל מקום כלשהו ב-free list והוא יצביע על 3 מקומות (לתא השלישי נמחק את המצביע וכן נשנה את ה-free list שתצביע אל התא הבא הפנוי בזיכרון). איור מצורף להמחשה. כלומר, נוכל להקצות איזה גודל שנרצה בצורה הזו. ההנחות נועדו להפשטה (יש שפות שאכן עושות כך, כמו LISP). אז הקצאת זיכרון – ראינו, אז נצטרך לעסוק בשחרור זיכרון, ובנקודה הזו נתעסק בשני הסוגים שציינו לעיל.

מוני היחס – Reference Counting:

דומה ל-smart pointers שראינו. כל תא בזיכרון יהיה לה next (כפי שציינו) ובנוסף יהיה לה counter – שמה נסכום כמה מצביעים אליו. כל פעם שעוד מישהו מצביע אז ++counter וכן אם מישהו עוזב אז --, וכאשר המונה מתאפס אז המערכת יודעת שאין עוד צורך בו, הוא הפך להיות זבל ולכן המערכת תאחזר אותו. Python עובדת כך.

בעיות בשיטה:



אחת בעיה שהזכרנו ב-smart pointers של מעגל ההצבעות (פירושט לבעיה ניתן לראות שם, פרק 4, smart pointers). בעיה זו כה חמורה, ששפות כמו Python מריצים אלגוריתם של Reference Counting ופעם בכמה זמן רק בשביל לפתור את הבעיה הזו הם מריצים אלגוריתם אחר (mark and sweep, נעסוק בו בהמשך).

יש כאן מניפולציה של מצביעים וזו עלות מאוד כבדה של העלאת המונה, חיסור, ניהול המצביעים, מישור עומד על אפס אז לשחרר אותו, כל זה עולה המון ולוקח זמן.

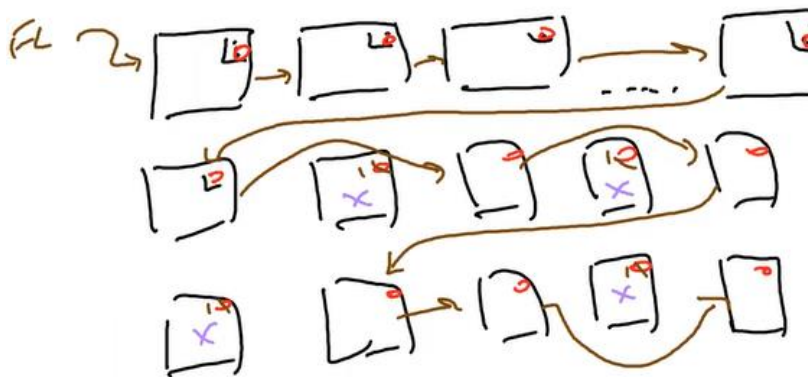
נשים לב שאיסוף הזבל נעשה כל הזמן, אין נקודת זמן שבה אנו עוסקים באיסוף זבל.

שיטת אלגוריתמי Garbage Collectors:

ב-reference counting אנו לא נותנים לזבל להיווצר, ברגע שמישהו נהייה זבל אז מחקנו אותו. בשיטות מהסוג הזה, אנו ניתן לזבל להיווצר (ולכן קוראים לאלגוריתמים GC). אנו לא נתעסק בזליגת זיכרון, אנו ניתן לזבל להיווצר עד שיגיע הרגע שמישהו יבקש ליצור משתנה (new) ולא יהיה לנו מקום בזיכרון עבורו. ברגע זה, נכנס ה-GC. זו השיטה הכללית של כל השיטות שנראה כעת, וכל אחד יוסיף איזה יעילות כזו או אחרת.

השיטה הראשונה – Mark & Sweep:

בכל תא בזיכרון יהיה bit indicator שיקבל ערך 0 או 1. כולם בהתחלה יהיו מאתוחלים להיות זבל, כלומר הביט יהיה עם ערך 0 (0 = זבל, 1 = לא זבל). הנחה זו,



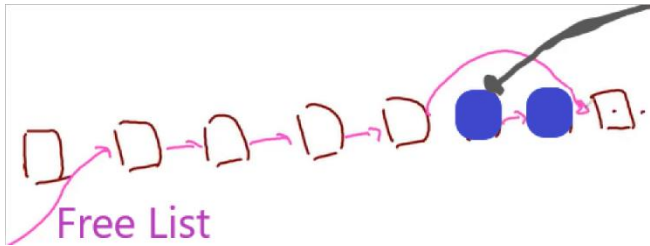
שכל הזיכרון בהתחלה זבל, היא לא תמיד נכונה. כל הקצאת זיכרון תהפוך את הביט להיות 1. עבור כל תא שהוא לא זבל חייב להיות מצביע ב-RTS אליו. ולכן, השלב השני יהיה לעבור על ה-RTS ושמה עבור כל תא בזיכרון שמצביעים אליו מה-RTS, אנו נשנה את הביט מ-0 ל-1 (בהתחלה הנחנו שכל הזיכרון זבל, וזה לא נכון ובכך טיפלנו בבעיה). השלב הבא יהיה להגדיר את ה-free list

(השתמשנו בה לעיל ע"מ להקצות זיכרון) - נעבור על הזיכרון וכל תא שהביט שלו 0 אנו נשרשר אותו ל-free list, וכל מקום שהביט 1 אנו נדלג (כלומר, נאסוף את כל הזיכרון הפנוי ל-free list). ולכן מגיע השם של האלגוריתם: mark - מסמן את כולם כזבל ואז מסמן את מי שלא זבל, sweep - נאסוף את כל ה-free list.

חסרונות השיטה Mark & Sweep:

אנו עוברים על הזיכרון בשביל לסמן את כולם, אחר כך כדי לסמן את כל מי שלא אפס (לא זבל) ואחרי זה שוב בשביל לאסוף את כל מי שאפס. כלומר, אנו עוברים על הזיכרון בין 2 ל-3 פעמים (כי בשלב השני לא בהכרח שנעבור על כל הזיכרון). במקרה הטוב, שאין אף אחד חי וכולם זבל, אנו עוברים על כל הזיכרון פעמיים. במקרה הגרוע, שיש הרבה חיים, אנו נעבור על הזיכרון 2 פעמים. וזו בעיה, כי השיטה רצה באמצע הפעולה. נגיד באמצע משחק הזיכרון מתמלא ואז מפעילים את השיטה אז פתאום המשחק יתקע עד שהשיטה תסיים את עבודתה ותחדש את ה-free list. נשים לב שבמקרה הגרוע שרצים 3 פעמים, זה המקרה שבו אתה לא מצליח לנקות הרבה זבל, והמקרה הטוב, שבו אתה מנקה הכי הרבה זיכרון, אתה רץ פחות. כלומר, זה הפוך מההיגיון – אם אני עובד יותר אז אני אוסף פחות (ככל שנעבוד בשלב 2 יותר כך יש יותר דברים חיים ופחות זבל). עד פה ציינו שני חסרונות – לא יעיל, והפוך מההיגיון.

בנוסף, יש חסרון של fragmentation. כאשר נקצה זיכרון, אז נחלק אותו לגושים שאותם נקצה (נקצה גודל לפי הגודל שאותו מבקשים מאיתנו). כאשר נשחרר את הזיכרון, אנו לא נחבר בין גושים אלא הם יישארו כגושים, וכאשר נעסוק במרחב רציף של זיכרון (הנחנו שהזיכרון שלנו מחולק לתאים) נראה שמה שקורה כאן זה פרגמנטציה. כל פעם שנקצה נחלק את הזיכרון לעוד תאים. השיטה גורמת לפיצול הזיכרון. במקרה שלנו, שהנחנו שהזיכרון מחולק לתאים, זה פחות

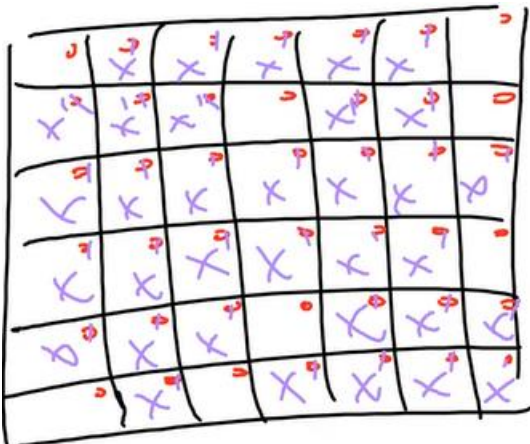


בעיה כי אין מושג כזה של זיכרון רציף או לא. אבל גם בבלוקים זה בעיה במובן של עוד תכונה, שנעסוק בה והיא Locality of Reference. נניח ושני התאים הכחולים יהיו מוקצים, ואז בשלב כלשהו נשחרר אותם ונחבר אותם חזרה ל-free list. הזכרונות הללו לא יחברו מהמשבצת שנמצאת לפניהם לרשימה, אלא ע"י מצביע מהסוף של הרשימה, איפשהו בהמשך (ראה חץ אפור באיור).

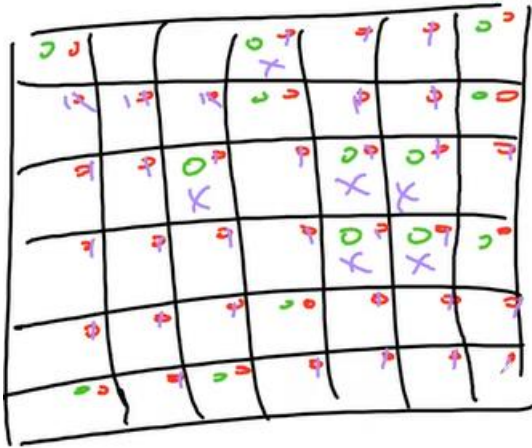
התכונה Locality of Reference אומרת שזיכרון קשור נמצא אחד ליד השני, לא נרצה להקצות זיכרון מפוצל למישהו אחד, אני מעדיף להקצות לו זיכרון שנמצא באותו מקום, רצוף. כי בדר"כ זיכרונות שהוקצו באותו זמן גם יגשו אליהם באותו זמן, ואז אם הם יהיו אחד ליד השני אנו לא נצטרך להיכנס לבעיה של להביא זיכרונות ו-virtual memory וכו'. מה שיקרה שאם משתמשים בשיטה הזו (Mark&Sweep), גם אם נשתמש בשיטה שבה הזיכרון מחולק לתאים אנחנו נאבד locality of reference. כלומר, יש בעיה של פרגמנטציה בכל מקרה, גם אם מדוב על זיכרון רציף וגם אם לא, וחץ מזה יש בעיה של locality of reference (שהוא בדר"כ תופעת לוואי של פרגמנטציה).

שיפור זמן האיסוף של Mark & Sweep:

ננסה לשפר את האלגוריתם כך שיהיה קצת יותר יעיל במחינת פיזור הזמן. אמרנו שהאלגוריתם מתחיל לעבוד ברגע שאתה מבקש זיכרון ואין לו להקצות לך, כל הפעילות תיעצר, הוא יאסוף כמה זיכרון שיוכל ורק אז תוכל להמשיך בפעילות, וזו בעיה עבורנו. הוצע עבור בעיה זו כמה פתרונות ואנו נעסוק כעת באחד מהם. הגדרנו שאם הביט 0= אז הזיכרון זבל, ואם 0= אז לא זבל. נשנה את ההגדרה, נגדיר זבל=1, ולא זבל 0=0. באיטרציה הבאה נעשה ההפך וכן הלאה.



נראה דוגמא ע"מ להבין יותר טוב. נניח בהתחלה כל הזיכרון הוגדר להיות זבל ולכן כל הביטים 0=. לאחר מכן, כל תא בזיכרון שנקצה, נשנה את הביט שלו להיות 0=. כל תא שהקצנו מסומן ב-x. עכשיו לא נשאר לנו מקום בזיכרון להקצות, אז נפעיל את האלגוריתם, ונשנה את ההסתכלות שלנו, כלומר כעת נניח זבל 1=. כעת לא נצטרך לעשות את הסיבוב הראשון של האלגוריתם, כי רוב הזיכרון כבר מסומן ב-1, יש מעטים שמסומנים באפס ואותם היה ניתן להקצות, אבל כל השאר כבר מסומנים באחד – חלק זבל וחלק לא (כמו שהיה עד עכשיו).



ואז נתחיל בשלב 2 שהוא יסמן את כל מי שלא זבל ב-0 (מסומנים באפס ירוק). ואז בשלב השלישי כל מי שלא סומן באפס אנחנו נאסוף אותו ל-free list (ראה איור משמאל).

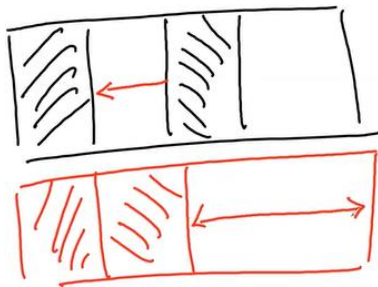
ואז באיטרציה הבאה נהפוך, את כל מי שנקצה אנו נסמן ב-0. יוצא שכל מי שמוקצה מסומן ב-0 וגם כל האלו המוקצים מהסבב הקודם יהיו מוקצים באפס (כל אלה שלא אספנו בסבב הקודם). כלומר רוב ככל הזיכרונות יהיו מסומנים באפס (חלקם חיים וחלק לא). יהיו חלק שלא יהיו מסומנים באפס (כמו מקודם) והפעם זבל = 0 ולא זבל = 1. ואז נעבור על הזיכרון ונבדוק מי מהם לא זבל (מי חי ומי לא) ונסמנם. כלומר, לא יעלנו במובן שנעבור על פחות. אלא, יעלנו כך שאת כל השלב הראשון של השיטה עשינו תוך כדי הקצאת הזיכרון, כלומר

פיזרנו את השלב הראשון על פני יותר זמן ובכך חסכנו זמן שכאשר יגיע הזמן להרצת האלגוריתם אנו נצטרך לעשות רק את שלב 2 ו-3 (נצטרך לחכות פחות). במכלול קיצרנו ממש במעט את ריצת האלגוריתם (כי יש כאלו שלא סימנו ולא אספנו), אבל בתמורה חסכנו 1/3 שליש זמן מריצת האלגוריתם בזמן של ההרצה.

נקודה נוספת, אם יש כאלו שלא אספנו בזמן האיסוף, אז נאסוף אותם בסבב הבא כשנהפוך בין 0 ו-1.

יעילויות של Mark & Sweep:

- Generational Mark&Sweep: מחקר הראה שיש הרבה משתנים שנוצרים ומתים מיד אחרי, ולעומתם משתנים הקיימים לאורך זמן הם מעטים מאוד. בנוסף, בדר"כ מי שלאורך זמן אז הוא לאורך זמן. כלומר, בדר"כ מי שהצליח לעבור סיבוב אחד של השיטה, אז ההסתברות שהוא יצליח לעבור את הסיבוב הבא היא יותר גדולה. זה אומר שמשתנים זמניים בדר"כ יאספו בסיבוב הראשון ומי שלא נאסף בסיבוב הראשון, אז ההסתברות שיאסף בסיבוב השני קטנה. זה חשוב כי אם אני יכול לתת לכל זיכרון איזשהו מספר שאומר כמה סבבים הוא קיים, אז ההסתברות שהוא ישרוד את הסיבוב הבא היא גדולה יותר ואין סיבה שאני אבדוק אותו שוב. אז נחלק את הזיכרון לדורות, לכל זיכרון ניתן את מדד הדור שלו. ואז, שנבוא להפעיל את האלגוריתם נבדוק את הזיכרון לפי דורות, קודם את הדור האפס, כל אלו שעדיין לא עברו אפילו סיבוב אחד, ושם נמצא הרבה מאוד זבל. אם זה לא יספיק או שעבר הרבה זמן אז נלך לבדוק את הדור הבא וכן הלאה. בדר"כ נגיע עד הדור השני ונפנה מספיק זיכרון. כלומר, אין סיבה לבדוק דור שני, אלא אם כן אין לך זיכרון או שעבר הרבה זמן (מידי פעם נעשה "ניקיון יסודי"). ומה שיוצא הוא שברגע הקריטי שבו יגמר לנו הזיכרון אנו לא נבזבז הרבה זמן אלא נפנה זיכרון במהירות. ניתן לראות זאת ב-C# או ב-Java ספרייה של GC ושם ניתן לראות את הדורות של משתנה מסויים (כמו כן, ניתן להריץ את ה-GC, כלומר אם אני עומד להיכנס לקטע קריטי אז אני יכול מראש להריץ את ה-GC ולא לחכות לרגע האחרון שבו יגמר לי הזיכרון).



- Compaction: אם יש לנו זיכרונות רציפים, וזיכרונות התפצלו (הקצנו זיכרונות ולכן פיצלנו, לאחר מכן כאשר הם התפנו, הם נשארו מחולקים, פירוט על כך יש למעלה ב-fragmentation), אז אנו נחבר ביניהם ובצורה כזו אנו נצליח לבצע איחוד מהפרגמנטציה. ולא רק שהוא מאחד בין שני זיכרונות פנויים סמוכים, הוא גם מזיז את הזיכרונות התפוסים אחד ליד השני ובכך מגדיל את השטח הפנוי הרצוף בזיכרון (ראה איור, העליון הוא ההתחלתי והתחתון הוא לאחר ה-compaction).

- Generational Mark and Compact: השיטה הזו היא חיבור של שני היעוליים הקודמים. JVM מריץ אותו, וגם CLR, ובצורה כזו הם חופרים רק במקומות שמראש מצפים לקבל מהם הרבה תועלת, וגם הם עושים

compaction כדי לנסות לטפל בנושא של פרמנטציה. כמו כן, הם מרוויחים את ה-locality of reference כאשר עובדים בצורה כזו אבל זה לא מוכרח (תלוי באיזה סדר ובאיזה מקום הם הוקצו ומתי בשלבי השיטה הם יוצרו).

שיטת צ'ייני לאיסוף זבל:

נקראת two finger approach. בשיטה זו לוקחים את מרחב הזיכרון ומחלקים אותו לשניים, לאחד נקרא to ולשני נקרא from. כל פעם שנקצה על ה-heap, אנו נקצה מקום על ה-to (חלק מההקצאות כמובן בהמשך יהפכו לזבל), ובשלב כלשהו יגמר המקום על ה-to להקצאות. כפי שכבר אמרנו, כל מי שחי משמע שיש אפשרות לגשת אליו דרך ה-RTS. כעת, מה שנעשה הוא, נשנה את השם של to ל-from וההפך עבור ה-from. נשים אצבע אחת על הזיכרון הריק הבא ב-to (החדש). נתחיל מתוך ה-RTS וכל מי שחי אנו נעביר אותו מה-from ל-to. אז במקרה שלנו נעביר את 8, ונעדכן לאחר מכן ב-RTS את הכתובת שלו, וכן בכתובת שבה 8 היה אנחנו נשים מצביע אל המיקום החדש של 8, ונקדם את האצבע לשורה הבאה. כעת יש לנו 2 אצבעות – אצבע אחת על השורה הבאה וכן אצבע נוספת על השורה הזו. כעת, נמצא גם את הכתובת של 9 (מסומן ב-9&), וגם עבורו נעשה את אותו תהליך – נעתיק אותו אל ה-to, נשנה את המצביע ב-RTS להצביע אל המיקום החדש של 9& וכן את המיקום הישן של 9& (שזה ב-from) אנו נעדכן להצביע אל המיקום החדש של 9&. נמשיך לעשות זאת עבור כל ה-RTS, ונגיע למצב שבו יש על ה-to את כל מי שניתן לגשת אליו ישירות מה-RTS. כל מי שאין גישה ישירה אליו מה-RTS וכן ערכי זבל נמצאים ב-from. כעת ניזכר שוב בשתי האצבעות, אצבע אחת בהתחלה של to ואצבע שנייה מצביעה על השורה הריקה הבאה. נתחיל מהאצבע בהתחלה ונעבור ונבדוק האם יש באחד מהתאים מצביע אל מיקום ב-from. עבור 8 זה לא יתקיים אז נקדם את האצבע הראשונה אל 9&. כעת נבדוק עבור 9& ואכן הוא מצביע אל תוך ה-from. אז נעשה עבורו את אותו תהליך כמו מקודם – נעתיק אותו למיקום הריק הבא ב-to, נעדכן את הכתובת ב-RTS, נשנה את הכתובת של 9 ב-from לכתובת שלו ב-to ולקדם את האצבע לתא הריק הבא. כעת נבדוק עבור התא הבא האם הוא מצביע אל כתובת ב-from, כלומר נבדוק את 9 שאותו הוספנו כעת, וזה לא מתקיים עבור 9 ולכן נקדם את האצבע הראשונה. וכעת שתי האצבעות מצביעות לאותו מקום, ברגע זה נסיים את התהליך. כעת, ה-RTS מצביע אל משתנים ב-to בלבד, כל מי שחי נמצא ב-to וכן הערכים שיש ב-from הם זבל מבחינתנו. כעת, כל מה שיש בהמשך ה-to הוא free list רציפה בזיכרון, קיבלנו גם Locality of Reference (צמצמנו את המשתנים למקום אחד בזיכרון).

למה השארנו את החיצים האדומים מה-from שיצביעו אל הכתובת של המשתנה ב-to? נניח ויש עוד מצביע מה-RTS אל 8. אנחנו העברנו את ה-8 מה-from ל-to. כאשר נגיע אל המצביע ב-RTS אל 8, הוא יצטרך לעדכן אצלו את המצביע, ולכן הוא יעתיק את הערך שיש כעת במיקום הישן של 8, שהחלפנו את הערך שם להצביע אל המיקום החדש של 8, ובכך הוא יעדכן את המיקום השני ב-RTS שמצביע אל 8. בעצם השארנו מאחורה דרך מעקב, דרך להגיד למשתנים שאולי גם מצביעים על אותו נתון איך למצוא את הנתון הנכון, וזו חשיבותם של החיצים האדומים בימין. לא אכפת לי מהמצביעים הללו, כי כולם יהפכו להיות זבל בכל מקרה.

וכעת, חזרנו לאותה תמונת מצב כמו שהייתה בהתחלה, ה-from ריק, אנו מקצים על ה-to וכאשר הוא יתמלא אנו נחזור על התהליך הזה שוב.

יתרונות של השיטה:

נוגע רק באובייקטים החיים, לא מתעסק עם מי שאין אליו גישה. נגזר מזה שזמן הריצה שלו פרופורציונלי לכמות החיים והמתים, ולא "אנטי פרופורציונלי", זאת אומרת ככל שיש יותר חיים כך הוא ירוץ יותר זמן וכן להפך.

אין בו פרגמנטציה. הזיכרון כולו מעבר למה שהקצנו, הוא באופן רציף נהיה free list.

לגבי Locality of Reference, זה לא מוכרח כי זה תלוי גם בסדר של הדברים, ואיך הם עובדים. לא הבכרח שדברים שעוקבים ישארו עוקבים (כאשר נעביר אותם מה-from ל-to). עושה compaction, וזה לא בהכרח שומר על locality of reference.

חסרונות של השיטה:

החסרון העיקרי הוא שכל הזיכרון המסומן ב-from מבוזבז לחלוטין ואנו לא משתמשים בו. אנו צריכים להריץ אץ האלגוריתם אחרי 50% ניצול זיכרון. יש עוד חצי ריק ולמרות זאת נגמר לנו הזיכרון. יש שיטות שמייעלות את השיטה הזו, כגון לחלק ל-1/3 וכו', אך לא בזה עסקינן, אנו למדנו את השיטה הבסיסית.

עצם זה שנתונים זזים זה גם חסרון, אך לא כזה קריטי.

שיטה זו שונה לגמרי משיטת Mark & Sweep, וכולם נכנסים תחת הכותרת Garbage Collection algorithms. בשונה מ-Reference Counting, שעובד בשיטה אחרת לגמרי. הוסיפו על זה עוד שיפורים וכו', אבל בסופו של דבר, כל האלגוריתמים מבוססים על אחד משלושתם - Reference Counting, Mark&Sweep, Two Finger algorithm (לקראת המצגת, לבדוק על ה-GC של השפה ולראות על מי מהם ה-GC מבוסס).

פונקציות

מה זה פונקציה:

ראשית, נדבר על אבסטרקציה (Abstraction). פונקציה זה הפשטה של תהליך, סוג של קופסא שחורה המתאימה לקלט פלט כלשהו. חוץ מזה ממיפוי של קלטים לפלטים, אנו מקבלים הפשטה של התהליך, לא מעניין אותנו כיצד ההתאמה בין קלט לפלט התבצעה. לדוגמא, לא מעניין אותנו כיצד עובדת הפונקציה sqrt . כלומר אנו מקבלים הפשטה של תהליכים (בעוד שב-OOP אנו מדברים על הפשטה של נתונים ולא של תהליכים, נעסוק בזה בפרק הבא).

כמו כן, `code-reuse` מאוד שימושי, ברגע שיש לנו קופסא שחורה אנו יכולים להשתמש בה שוב ושוב.

נניח שלכל פונקציה יש נקודת כניסה אחת (לא חובה לעבוד ככה, אבל מחקר גילה שזה לא מוסיף ליכולות וזה רק מסבך את המחשבה וההבנה אז זה די נדיר שנראה דבר כזה), כלומר אנו מתעסקים עם פונקציות שהן לא `parallel` (כאשר אחד מתחיל לפעול אז השני מושהה).

קישור פרמטרים:

פונקציה היא תיאור של חישוב, או של מיפוי של קלטים לפלטים והיה אפשר לעשות זאת ע"י גישה למשתנים גלובליים או דברים אחרים שלא קשורים לפונקציה עצמה. אבל מקובל שלכל פונקציה יש את הפרמטרים הפנימיים שלה, השמות של הקלטים שלה. נגיד יש לנו את התוכנית המחשבת את $f(x) = x^2$, אז עבור $f(3) = 9$. ל- x קוראים פרמטר פורמלי ול-3 קוראים פרמטר ריאלי. כלומר, בתוך הפונקציה x מייצג משהו פורמלי, הוא לא ערך ממשי. כאשר שלחנו לפונקציה את 3 אז שלחנו פרמטר ריאלי. השאלה שלנו היא – איך מקשרים בין ה-3 לבין ה- x , ישנן כמה דרכים. חלק משהפוט בחרו בשיטת `Positional Parameters`, בשיטה זו הקישור נעשה על סמך מיקום, כלומר הראשון לראשון וכו', שפות כמו C עובדות כך. לעומתן, יש חלק שבחרו בשיטת `Keyword parameters`, בשיטה זו צריך להגיד לכל פרמטר פורמלי מה הפרמטר הריאלי שיתאים לו (לדוגמא, ADA). יש שפות כמו Python שמאפשרות או זה או זה, וברגע שעברת ל-`Keyword` אין דרך לחזור ל-`Positional`, כי המיקומים כבר חסרי משמעות. כלומר, ניתן להתחיל מ-`keyword` ואז לעבור ל-`positional`.

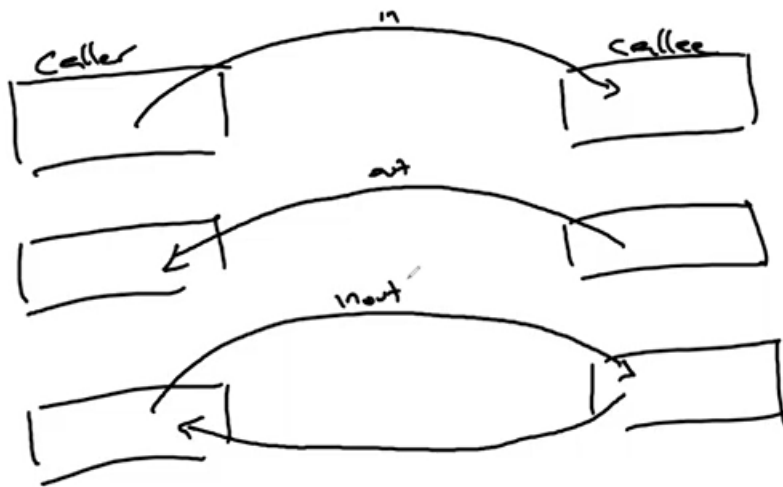
Keyword parameters	Positional Parameters
Order doesn't matter	IDK IDC (name of formals)

יתרון של `Keyword` הוא שהסדר לא מעניין אותנו. נניח שיש לנו פונקציה עם 50 פרמטרים ונרצה להכניס ערכים רק לפרמטר ה-1, ה-2 וה-49. בשיטת `Positional` נצטרך להשים מלא פסיקים עד שנגיע למיקום ה-49 (גם אם יש ערכי ברירת מחדל, מכיוון שאני רוצה להשים ערך במקום ה-49 אז אני צריך לספק ערכים עד המקום ה-49 [כמו ב-CPP שאומרת שכל המשתנים בעלי ערכי ברירת מחדל צריכים להיות אחרונים וצריך לספק את כל המקומות עד למקום שאתה רוצה ומשם זה יהיה ערכי ברירת מחדל]), ואילו בשיטה זו נוכל ללכת ישר למקום ה-49.

יתרון של `Positional` הוא שלא מעניין אותי מה השם הפורמלי (אף אחד לא יודע מה השם הפורמלי של המשתנה ב-`sqrt`, זה לא מעניין), אבל בשיטת `Keyword` נצטרך לדעת מה השם הפורמלי של הפרמטר.

מודל סמנטי להעברת פרמטרים:

ישנן 3 דרכים שבהן פונקציה יכולה לקרוא לפונקציה, 3 מודלים סמנטיים להעברת פרמטרים.



המודל הראשון (in) – העברת פרמטרים פנימה.

המודל השני (out) – העברת פרמטרים החוצה. ב-C# יש את המילה out

המודל השלישי (in out) – העברת פרמטרים פנימה והחוצה.

ניתן לממש כל אחד מהם בכמה דרכים. לדוגמא, ניתן לממש ע"י: העתקת הנתונים, העברת הנתון בעצמו (או מצביע אליו). זה עדיין אותו מודל וממש את אותו הדבר.

מימוש ששמענו בעבר לשיטת ה-in הוא Pass by Value – לרוב מבוצע ע"י העתקת ערכים (לא מחייב, ניתן לבצע בשיטות אחרות כגון- void foo(const int a), במקרה זה לא ניתן להעביר נתונים החוצה, אי אפשר לשנות את a שקיבלנו. החיסרון המרכזי הוא זמן העתקה וכן הזיכרון שנתפס להעתקים.

מימוש נוסף הוא Pass by Result, בא לממש את ה-out. לא נעביר לפונקציה נתון אלא מיכל למילוי, כלומר אם נכתוב ב-C# כך: void foo (out int a) אז העברנו פרמטר שאותו צריך למלא. ניתן לראות שאכן מדובר על פרמטר שהוא out

```
int a = 7;
foo(a);
void foo(out int a) {
    a++; // שורה בעייתית !
}
```

במקרה שננסה להשתמש בפרמטר הזה לפני שאתחלנו אותו, כמו במקרה המצורף פה, אז הקומפיילר יצחק, אבל ל-a יש ערך אז מה הבעיה? אלא, בגלל שעבדנו בשיטת Pass by Result אזי a לא הועבר פנימה אלא הועבר מיכל למילוי. כאשר נגיע לסוף הפונקציה foo, אז רגע לפני שהמשתנים המקומיים ימותו, הקומפיילר יעתיק אותם החוצה.

שיטה שלישית היא Pass by Value Result, מימוש של in וגם out, מעתיק פנימה את הפרמטר ולבסוף מעתיק אותו החוצה. לא מעביר את הכתובת של המשתנה האמיתי אלא יוצר עותק ובסיום הפונקציה הוא יעתיק אותו אל תוך המשתנה האמיתי.

בנוסף, Pass by Reference גם מממש את שיטת ה-in & out, נעביר reference למשתנה החיצוני ואז כל מה שקורה לו בתוך הפונקציה קורה לו גם בחוץ. החיסרון שלו לעומת המימוש הקודם הוא יעילות זמן ריצה. כל פקודה שקוראת בפנים דורשת מאיתנו 2 רמות של הפניה (כפי שלמדנו בפרק על מצביעים). לעומתו, השיטה הקודמת רק בתחילת הריצה ובסיום הריצה אני אעתיק (בהתחלה פנימה, בסיום החוצה), כלומר, נשלם על 2 העתקות בתמורה לזה שהמשתנים יהיו מקומיים. הבחירה במי להשתמש תלויה בקוד של הפונקציה, אם יש פניות רבות למשתנים אז כבר כדאי את השיטה הקודמת, ואם יש קצת אז אעדיף את השיטה הזו.

אז באיזה מודל נבחר להשתמש? היינו אומרים להשתמש תמיד ב-in & out כי זה לא משנה לנו, אבל זה לא נכון. לדוגמא, לא נרצה לתת לכל אחד הרשאה (תכונת least privilege) לשנות לנו את המשתנים ולכן לא נשלח לו בשיטה זו אלא רק ב-in. במקרה שלא נעביר כלום פנימה אבל נרצה שתוציא לנו ערך, אז הכי נכון יהיה להשתמש ב-out. נצטרך להתחשב מבחינת שני תכונות. הראשונה היא least privilege, כפי שכבר הזכרנו שיש רמות של הרשאות. השנייה היא יעילות, לפעמים היעילות מגדירה שנעבוד בצורה אחת ולא אחרת, לדוגמא אם מעתיקים פנימה אובייקט ענק אז נעדיף

להעביר מסלול ע"י Pass by Value (לא נרצה שיהיו לנו 2 אובייקטים) ומצד שני נעביר עם const. כל פעם נבדוק מה עדיף לנו ולפי זה נבחר, כמובן שכל שפה בחרה את הדרך שבה היא מממשת את העניין.

מחסנית זמן ריצה ורקורסיה:

לכל קריאה לתת תוכנית יש סמנטיקה כלשהי, מוסכמה מה מעבירים לה, כגון – מעבירים פרמטרים פנימה או החוצה (כפי שראינו לעיל), האם המשתנים הלוקאליים הם סטטיים או לא, שמירת מצב התוכנית אותה הפסקנו כרגע, להעביר שליטה לפונקציה הקוראת וכו'. בנוסף, יש סמנטיקה ליציאה מתת תוכנית, אמורה לטפל בדברים כמו – איך מחזירים ערכים, איך משחזרים זיכרון שהוקצה, איך מחזירים שליטה לתת תוכנית הקודמת, איך משחזרים את הסביבה וכו'.

לפני 60 שנה היה, לפני שהיה RTS, לכל תוכנית יצרו כמה חלקים. חלק אחד – הקוד לביצוע, וחלק שני – כל ה-data שאולי נצטרך עבור התוכנית, לשני אלה קוראים activation record. עבור כל פונקציה היה activation record וכאשר הרצנו את הפונקציה אז נהיה מופע של ה-activation record. מה שעשו הוא הקצו מקום ל-activation record של כל פונקציה שנריץ בהמשך (מקצים data מראש לכל פונקציה). הבעיה בשיטה זו היא בגלל שאין לנו שום דרך לייצר זיכרון נוסף בזמן ריצה. ואז, העלו את הרעיון של ה-RTS, שבעצם הוא אמר להגדיר נתונים אוטומטית בזיכרון. כלומר, אני לא הולך ומקצה מראש עבור כל פונקציה data בזיכרון, אלא אני מייצר איזשהו RTS וכל פעם נכניס את המופע של ה-activation record של הפונקציה שעומדים להריץ עכשיו. הרווחנו מזה שאפשר לעשות רקורסיה, שניתן להכניס ל-RTS את המופע של ה-activation record של פונקציה כלשהי ואז להכניס אותה שוב למחסנית. זה לא היה ברור בעבר שניתן לעשות רקורסיה, כי היו מקצים עבור כל קריאה את ה-data שלה מראש, כלומר יכול להיות מופע אחד של activation record עבור כל פונקציה. כל היכולת לייצר frame בזמן ריצה, לחזור מ-frame וכו', זה נוצר אז בזכות ה-RTS.

דמו של קריאה לפונקציה ומה שקורה מאחורי הקלעים:

נראה את ההרצה של הדוגמא הזו וכן את הזיכרון בשעת ההרצה. נסתכל על ה-disassembly (התרגום של הקוד ל-assembly) ואותו נריץ.

בהתחלה נכניס 7 ו-a ו-8 ו-b בהתאמה.

לאחר מכן, אנו נדחוף את הערכים הללו למחסנית (נדחוף אותם לתוך רגיסטר ואז אותו נדחוף למחסנית).

לאחר מכן, יש קריאה ל-foo. מכיוון שכעת אנו יוצאים מהקוד שלנו אל foo, אנו צריכים לדחוף למחסנית את הכתובת חזרה אל הקוד שלנו (שזו השורה המתחילה ב-D9). ולכן, כעת יידחף למחסנית ה-address.

כך המחסנית תיראה עד כה: דחפנו את 7, 8, כתובת חזרה.

0x003AFB8C	d9 17 24 00	..\$.
0x003AFB90	07 00 00 00
0x003AFB94	08 00 00 00

נשים לב, שה-stack pointer הצביע לפני דחיפת הערכים הללו על השורה 3AFB98.

```
int main()
{
    int a = 7, b = 8;
    foo(a, b);
    return 0;
}

void foo(int x, int y)
{
    y = y + 1;
    x = x + 1;
}
```

```
int main()
{
002417A0 push     ebp
002417A1 mov      ebp,esp
002417A3 sub      esp,0D8h
002417A9 push     ebx
002417AA push     esi
002417AB push     edi
002417AC lea      edi,[ebp-0D8h]
002417B2 mov      ecx,36h
002417B7 mov      eax,0CCCCCCCCh
002417BC rep stos dword ptr es:[edi]
    int a = 7, b = 8;
002417BE mov      dword ptr [a],7
002417C5 mov      dword ptr [b],8
    foo(a, b);
002417CC mov      eax,dword ptr [b]
002417CF push     eax
002417D0 mov      ecx,dword ptr [a]
002417D3 push     ecx
002417D4 call     foo (0241375h)
002417D9 add      esp,8
    return 0;
002417DC xor      eax,eax
}
```

כעת, הגענו אל הפונקציה foo. כפי שאמרנו, יש צורך לשמר את ה-environment, וזה מה שנעשה בתחילת הפונקציה -

```
void /*_stdcall*/ foo(int x, int y)
{
002416F0 push     ebp          ≤ 10ms elapsed
002416F1 mov     ebp,esp
002416F3 sub     esp,0C0h
002416F9 push     ebx
002416FA push     esi
002416FB push     edi
002416FC lea     edi,[ebp-0C0h]
00241702 mov     ecx,30h
00241707 mov     eax,0CCCCCCCCh
0024170C rep stos    dword ptr es:[edi]
    y = y + 1;
0024170E mov     eax,dword ptr [y]
00241711 add     eax,1
00241714 mov     dword ptr [y],eax
    x = x + 1;
00241717 mov     eax,dword ptr [x]
0024171A add     eax,1
0024171D mov     dword ptr [x],eax
}
00241720 pop     edi          ≤ 13ms elapsed
00241721 pop     esi
00241722 pop     ebx
00241723 mov     esp,ebp
00241725 pop     ebp
00241726 ret
```

דחופים למחסנית את edi, esi, ebx, ebp וכן ביציאה מהפונקציה נהפוך את הסדר הזה ונוציא את הערכים מהמחסנית חזרה לרגיסטרים.

נשים לב, שלאחר ששימרנו את ebp, שמנו ב-ebp את הערך של הפונקציה שלנו, כלומר את הערך של נקודת העוגן של ה-scope שלנו, שזה בדיוק איפה שנמצא ה-SP (stack pointer). נייצר מקום למשתנים ע"י כך שנגדיל את ה-SP, נעלה אותו למעלה, ובכך יצרנו מקום למשתנים המקומיים. את המשתנים המקומיים נגדיר שם כאשר ניצור אותם. ולכן ב-CPP כאשר אנו מגדירים משתנה בלי לאתחל אותו אנו מקבלים ערכי זבל, מכיוון שיש במחסנית ערכי זבל. וזה בניגוד לקומפיילר שאנו בנינו הסמסטר, שבו אנו דחופים למחסנית אפס עבור כל משתנה מקומי (כלומר "אתחלנו" את המשתנים המקומיים לאפס).

כל הקוד באמצע של מימוש הפונקציה לא מעניין אותנו כעת.

כפי שאמרנו בהתחלה, בסדר הדחיפה כך סדר ההוצאה. דחפנו את edi אחרון ולכן הוא הראשון שנצטרך לדחוף אותו חזרה לרגיסטר מהמחסנית. וכן הלאה עבור שאר משתני הסביבה.

את esp הגדלנו ע"מ להקצות מקום למשתנים מקומיים ולכן נצטרך להקטין אותו חזרה, להחזיר אותו אחורה.

לאחר מכן, נגיע לשורה ret ומה שהיא תעשה זה היא תחזיר אותנו ל-return address שמופיע בראש המחסנית.

```
0x003AFB80 cc cc cc cc
0x003AFB84 cc cc cc cc
0x003AFB88 7c fc 3a 00
0x003AFB8C d9 17 24 00
0x003AFB90 08 00 00 00
0x003AFB94 09 00 00 00
0x003AFB98 00 00 00 00
```

אבל עדיין אנו מצביעים במחסנית על המיקום 3AFB90 וזה לא איפה שהוא היה בפונקציה החיצונית, ב-main, ולכן נצטרך להחזיר אותו למיקום שהיה לפני שקראנו לפונקציה ולפני שדחפנו לה את 7 ו-8. ולכן, השורה לאחר הקריאה ל-foo (ראה בקוד בעמוד הקודם) היא add esp, 8. כלומר, כעת esp = 3AFB98, שזה בדיוק הערך שלו לפני הקריאה לפונקציה foo (סוף העמוד הקודם רשום את ערכו).

נשים לבד שהנתונים שהכנסנו למחסנית עדיין שם, והם לא נעלמו (זה למה יש ערכי זבל) ומכאן והלאה אנו נדרוס את הערכים שיש שם (ערכי הזבל).

מוסכמות קריאה לפונקציה – Calling Conventions:

אלו מוסכמות שהפונקציה הקוראת והנקראת צריכות להסכים עליהן ע"מ שהקריא תעבוד כראוי.

הדברים שצריכים לקחת בחשבון: סדר הדחיפה של הפרמטרים למחסנית (לדוגמא, עבור הפונקציה "foo" האם קודם יידחף a או b). מי ינקה את המחסנית, הקוראת או הנקראת (clean – up). איפה נשים את הערך המוחזר (בתרגיל 2 הוא היה במחסנית, אפשר להשים ברגיסטר). איך קוראים לפונקציות (naming convention), כלומר האם נקרא לה רק עם שמה (בעייתי כי יש כאלו עם אותו שם אבל עם פרמטרים שונים וכו'), צריך שמות שלכל פונקציה הם יהיו שונים. שמירת הסביבה (environment), מי שומר את משתני הסביבה (לדוגמא, הקוראת תשמור חלק והנקראת את השאר), האם שומרים וכו'.

:CDECL - C Declaration



במוסכמה זו, כל הארגומנטים נכנסים על המחסנית, נכנסים מימין לשמאל (לדוגמא עם foo לעיל, אז נכנס קודם את b ואז את a) וזה גורם לכך ש-a יושב מעל b שזה מאוד הגיוני. המחסנית נראית כך לאחר דחיפת המשתנים וכל משתני הסביבה וכו' (בדוגמא לעיל של foo). כאשר נרצה לגשת אל a אז נעשה $ebp + 8$, וכאשר נרצה לגשת ל-b אז נכתוב $ebp + 12$, וזה הגיוני שאם נרצה לגשת למשתנה השני בסדר אז נצטרך להוסיף מספר יותר גדול (ebp מסומן ע"י חץ באיור בשמאל).

מבחינת ה-clean up, תפקיד הפונקציה הקוראת היא לנקות את המחסנית.

ה-return value יהיה על הרגיסטר eax.

Naming convention – נרחיב על כך בהמשך.

שמירת הסביבה – אין שמירה של הרגיסטרים eax, ecx, edx אין שמירה, ולכן אם הקוראת רוצה לשמור אותם אזי היא צריכה לשמור אותם ידנית. כל שאר הרגיסטרים נשמרים.

:STDCALL – Standard Call


דוחף את הערכים למחסנית בסדר הפוך, מימין לשמאל. תפקיד הפונקציה הנקראת הוא לנקות אחריה. שאר הדברים אותו דבר כמו CDECL.

:FASTCALL

תפס חזק כאשר עברו ל-64bit. בשיטה זו, שני הארגומנטים (שני הראשונים משמאל) הראשונים של הפונקציה הנקראת יעברו בתוך רגיסטרים וכל השאר על ה-RTS. כל שאר הפרמטרים יעברו בסדר הפוך מימין לשמאל. קוראים לו כך בגלל שיש לו ערכים על הרגיסטרים ולכאורה הגישה אליהם יותר מהירה.

דמו של מוסכמות קריאה:

כאשר ניקח את הקוד שראינו לעיל, ונמחק את המימוש של foo, נקבל את השגיאה הבאה:

 LNK2019 unresolved external symbol "void __cdecl foo(int,int)" (?foo@@YAXHH@Z) referenced in function _main


הוא מחפש פונקציה foo המקבלת שני משתנים מסוג int, ובסוגריים אנו רואים את השם שלה, השם הייחודי שניתן לה.

כעת, נשנה את סוג הערכים ש-foo מקבלת ונראה שניתן לה שם אחר:

```
int main()
{
    int a = 7, b = 8;
    foo(a, b);
    return 0;
}

void foo(int x, int y);
```

```
void foo(int x, char y);
```

 LNK2019 unresolved external symbol "void __cdecl foo(int,char)" (?foo@@YAXHD@Z) referenced in function _main

כלומר, שתי פונקציות בעלות סוגי פרמטרים שונים יקבלו שמות שונים. מכיוון שיש להם שמות שונים אז אנו נקרא לפונקציה אחרת לגמרי ולכן ברור איך עובד Function overloading (-שאנחנו יכולים לתת לשתי פונקציות אותו שם כאשר יש להן ארגומנטים שונים). זה עובד ע"י כך שבפועל אין להם את אותו השם. השם משתנה כתלות בפרמטרים. וזה name convention (שלא דיברנו עליו מקודם). לכל שפה יש name convention כזה או אחר, הרעיון הכללי אותו דבר.

דבר שני, אנו רואים שהוא מחפש פונקציה של CDECL, כלומר יש פה הסכמה כלשהי כיצד הדברים עובדים ומה הוא אמור לחפש.

```
void __cdecl foo(int x, int y)
{
    x = x + 1;
    y = y + 1;
}
```

בנוסף, אני יכול להגדיר לו כיצד הפונקציה תהיה מוגדרת, כלומר באיזו מוסכמה היא עובדת. כלומר, foo יקומפל במוסכמה זו גם אם בכללי זה לא עובד ככה. כעת, זה יעבוד בצורה נכונה וטובה.

```
void __stdcall foo(int x, int y)
{
011C16E0 push     ebp          ;6ms elapsed
011C16E1 mov     ebp,esp
011C16E3 sub     esp,0C0h
011C16E9 push     ebx
011C16EA push     esi
011C16EB push     edi
011C16EC lea     edi,[ebp-0C0h]
011C16F2 mov     ecx,30h
011C16F7 mov     eax,0CCCCCCCCh
011C16FC rep stos dword ptr es:[edi]
        y = y + 1;
011C16FE mov     eax,dword ptr [y]
011C1701 add     eax,1
011C1704 mov     dword ptr [y],eax
        x = x + 1;
011C1707 mov     eax,dword ptr [x]
011C170A add     eax,1
011C170D mov     dword ptr [x],eax
}
011C1710 pop     edi
011C1711 pop     esi
011C1712 pop     ebx
011C1713 mov     esp,ebp
011C1715 pop     ebp
011C1716 ret     8
```

אם אנו נעשה אותו דבר רק עם stdcall, אז הוא ירוץ אבל יש דברים שצריכים להשתנות לשם כך. התרגום של הפונקציה ל-assembly נראה כך (ראה בשמאל). ניתן לראות שכעת השורה האחרונה השתנתה, כתוב שם 8 ret, ולא סתם 8. תוספת ה-8 אומרת שכעת בהחזרה הוא גם יוסיף 8 ל-esp, כלומר ניקוי המחסנית יבוצע ע"י הפונקציה הנקראת.

אם הפונקציה הקוראת גם תנקה את המחסנית, כלומר תוסיף עוד 8, אז אנו ניכנס לבעיה,

```
foo(a, b);
011C175C mov     eax,dword ptr [b]
011C175F push     eax
011C1760 mov     ecx,dword ptr [a]
011C1763 push     ecx
011C1764 call     foo (011C1370h)
        return 0;
011C1769 xor     eax,eax
}
```

במקום לעלות 8 אנו נעלה 16. ולכן, הפונקציה הקוראת לא תעשה 8 add esp, כמו מקודם. ניתן לראות פה, איפה שהיה אז את השורה הזו, כעת היא לא נמצאת, כלומר היא מתנהגת ע"פ מוסכמת stdcall (ברירת המחדל היא

cdcel, כלומר מה שראינו למעלה הוא ע"פ מוסכמת cdcel).

היתרון של `cdcel` על `stdcall` הוא שאם אנו לא יודעים כמה ארגומנטים יש (כמו `printf`), אז אין שום דרך לפונקציה הנקראת לדעת כמה מקומות לנקות במחסנית, לעומתה הפונקציה הקוראת יודעת כמה היא דחפה ולכן הרבה יותר קל לה לנקות את המחסנית. בשפה שבה מספר הפרמטרים לפונקציה הוא קבוע, נעדיף את `stdcall` מכיוון שהוא הרבה יותר פשוט: תהיה שורה אחת שבה יהיה כתוב `ret k`, ולא 2 שורות כאשר האחת היא `ret` והשנייה היא `add esp, k`. נעדיף פקודה אחת של `assembly` מאשר שתי שורות.

תכנות מונחה עצמים

מה זה object oriented programming:

לפני כן נשאל שאלה נוספת, למה כאשר התחילו לדבר על OOP התחילו לדבר על דברים כמו information hiding, encapsulation, ובנאים. למה זה לא היה קיים עד אז? התשובה היא – Abstraction! עד OOP דיברנו על פישוט (אבסטרקציה) של תהליכים (לדוגמא, כאשר אנו מדברים על sqrt לא מעניין אותנו איך זה קורה). ב-OOP יצרו פישוט של מידע (Abstraction of Data). כעת אנו יכולים להתייחס למידע כדבר מופשט. לדוגמא – מחסנית, זה לא מערך או רשימה, זה סוג שמתנהג ע"פ חוקים כלשהם. מכיוון שהמידע הוא מופשט אנו צריכים להגדיר מה מייצג את המידע, ואת זה להפריד מצורת המימוש שלו. צורת המימוש שלו אינה חלק אינטגרלי מסוג המידע. לדוגמא, אם הגדרנו data type שהוא stack, כלומר LIFO, אז זה לא משנה לי איך אני אממש אותו בפועל (כמעריך או רשימה מקושרת). יש פה משהו מופשט בשם מחסנית, שהוא מתנהג בצורה כלשהי, ואני צריך לממש אותו – וזה לא קשור למושג מחסנית. ולכן, כעת צריך לדבר על information hiding – הרעיון הוא לנתק את המושג המופשט מהמימוש הספציפי. לדוגמא, אם אני שם במחלקה (שזו דרך לממש את ה-data type) ב-private רשימה משורשת, זה עדיין לא קשור למה שמשתמש במחלקה שלי. שזה דומה למה שראינו עבור sqrt, לא מעניין אף אחד איך זה פועל, כך גם אצלנו, לא מעניין איך זה מומש בפועל. אם לא נסתיר את המימוש עצמו אזי מישהו יוכל להשתמש במימוש הספציפי ובכך הרסנו את ה-abstract data type (המושג מחסנית, תור וכו' ממבנה נתונים לכאורה הם אבסטרקטיים, לאחר מכן מימשו אותם בצורה כזו או אחרת).

לדוגמא, ה-SDL (Standard Template Library של C++), לא מגדיר איך לממש מחסנית, הוא מגדיר מהי מחסנית, מגדיר את הפונקציונליות שלו (איך פונים אליו) ומגדיר את הסיבוכיות של האלגוריתמים שצריך להיות. המימוש, מוטל עליך ואתה יכול לבחור איזה. הקומפיליר/המתכנת לא אמור לגעת במימוש הספציפי. ולכן נכנס רמות של information hiding, לפה אתה יכול לגשת (Public) ופה מחוץ להרשאה שלך (private). וזהו פישוט של מידע!

Encapsulation – שמים ביחד את המימוש ואת הפעולות שעליו (ה-API). אנו צריכים שכלפי חוץ זה יראה אחד וכלפי פנים הוא ידע להתממש עם המימוש הספציפי שמימשנו, ולכן נהיה חייבים להשים ביחד את הפעולות וה-data, כדי שהכל יעבוד ביחד, שיהווה טיפוס מידע כלשהו – ADT (Abstract Data Type). גם אם בפנים נממש בצורה כזו או אחרת, ההתנהגות החיצונית שלו לא אמורה להשתנות. לכן כאשר מגדירים ADT (טיפוס מידע) אז נוציא API, כלומר מה הוא אמור לעשות ואיך להתנהג, ולא מה המימוש הספציפי (לדוגמא, למחסנית נגדיר שצריך להיות push, pop וכו'). ולכן הפעולות והמידע אמורים לבוא יחדיו, כדי שנקבל את הממשק הנכון עם המימוש המתאים בלי לחשוף את המימוש כלפי חוץ.

ולכן, כאשר באנו לדבר על פישוט מידע, היינו חייבים לדבר על מושגים אלו ונוספים. לא מעניין אותנו כיצד int פועל וכיצד מומש. וכן גם כאשר אנו נכתוב טיפוס מידע, אנו נצטרך להפריד בין המימוש בפועל לבין המושג. כך גם היה בקורס מבנה נתונים, קודם הגדרנו מבנה נתונים המתנהג בצורה כלשהי, ולאחר מכן מימשנו את זה בצורה כלשהי (לעיתים ראינו שלאותו דבר יש כמה מימושים).

נחזור לשאלה הראשית שלנו, מה זה OOP. OOP זה סגנון תכנות המאפשר למדל ADT, שהולכים ומייצגים רעיונות מופשטים מהעולם האמיתי – לוקחים את העולם וממדלים אותו בצורת מידע ואז מגדירים איך הוא עובד, ואיך המימוש והרעיון המופשט יתקשרו. פישוט מידע (Abstraction of Data) זה מה שמסתתר מאחורי תכנות מונחה עצמים, שלא היה קיים לפני 60 שנה. הרווחנו כאשר נשתמש בזה (מבחוץ) אז לא נצטרך להתעסק בצורה שזה מומש בפועל.

פולימורפיזם והקשר לטיפוס נתונים אבסטרקטי:

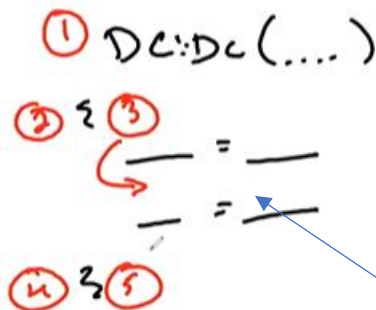
ירושות לא קשור לנושא שלנו, ל-ADT. נסביר, מימוש OOP ללא information hiding יפגע במושג ADT, כי כולם יכלו לגשת למימוש. מימוש OOP לא הכמסה (encapsulation), יפגע במושג, כי כבר לא יהיה ADT. אבל, מימוש OOP ללא ירושות, לא יפגע במושג ADT, אז נצטרך לכתוב עוד מחלקות וכדומה אבל עדיין לא יפגע פה האבסטרקציה. זה יותר בחירה של מימוש יעיל של הנושא מאשר הגדרתו.

עקביות הנתונים:

הבנאים נועדו לייצר עקביות נתונים (Data consistency). כאשר אנו מדברים על ADT, אנו מייצרים משהו אבסטרקטי בעל תכונות כלשהן, ואנו מצפים שיהיה לו את התכונות הללו. למשל, נניח ש-`int` חייב שתמיד יהיה לו ערך כלשהו, מישהו חייב לדאוג לזה. יכול להיות שהתכונה הפיזיקלית של הביטים בחייבים שתמיד יהיה בהם ערך ולכן תמיד גם ב-`int` יהיה ערך, וזה יכול להיות מישהו אחר שישים שם אפס. כלומר, אם `int` משמע שיש פה משהו בעל תכונות כלשהן, אזי מישהו חייב לדאוג שאכן הם יתקיימו, ולא יכול להיות מצב שנתחיל לעבוד והתכונות לא מתקיימות.

נניח שלביטים היו 3 מצבים: אפס, אחד וריק. כאשר נקצה `word` בזיכרון עבור `int`, אז הביטים נמצאים באחד משלושת המצבים. אבל `int` לא יכול להיות במצב של ריק ולכן מישהו יהיה חייב להשים בביטים האלו אפס או אחד במקום הריק. ולכן, חובה שיהיה מישהו שידאג לכך שטיפוס הנתונים שייצרנו יעמוד בכל התכונות שאמורות להיות לו. וזה התפקיד של בנאים - לייצר עקביות נתונים. ה-`distractor` זה כבר מנגנון, מכיון שיש בנאי אז יש גם הורס.

פונקציות בונות ועקביות הנתונים - מתי:



באיזה נקודת זמן בהרצת הבנאי אנו נעמוד בתכונה של טיפוס הנתונים, באיזה נקודה נוכל להגיש שהאובייקט שיצרנו הוא מוכן עם עקביות הנתונים. יש סה"כ 5 אופציות (כאשר אופציה 3 היא לאו דווקא בהתחלה, אלא איפשהו בין שורות הקוד של הבנאי). לכאורה נקודת הזמן שבה אנו מצפים שהאובייקט יהיה עקבי היא 5 - בנקודה זו, הבנאי סיים את עבודתו. נקודה 1, 2, 3 לא הגיוניים כי עדיין לא עשינו את פעולת הבנאי אז וודאי שלא יכול להיות שהאובייקט מוכן כבר. לגבי נקודה 4 ניתן לדון, אבל עיקר הנקודה היא שהבנאי סיים את עבודתו אז האובייקט צריך להיות עקבי.

ניתן לראות זאת בדוגמא הבאה: נניח יש לנו מחלקה `DC` ונייצר אובייקט כך - `const DC a()`; השאלה היא באיזה שלב ניתן להשים ערכים באובייקט שיצרנו. נסביר כאשר נכתוב `const int x = 7`, אז רק בשורה הזו נוכל להשים ערכים ב-`x` ולאחר מכן כבר לא נוכל לשנות את ערכו. כלומר, רק בשעת היצירה ניתן להשים בו ערך, והוא עדיין לא `const` עד שיצרנו אותו. וכן גם אצלנו, בשלב היצירה האובייקט `a` לא יכול להיות `const` כי צריך להשים בו ערכים, אז באיזו נקודת זמן הוא נהיה `const`. נחזור לבנאי - בתוך הבנאי ישנן שורות השמה, כלומר האובייקט עדיין לא `const` כי אנו יכולים להשים ערכים. משמע, שהאובייקט נהיה `const` בנקודה 5, בנקודה בה האובייקט סיים את עבודתו, שמה הוא ננעל.

נרחיב את הדוגמא, נניח והמחלקה נראית ככה:

```
class DC
{
private:
    const DC2 b;

public:
    DC::DC(int x, int y):b(x)
    {
        b = new DC2(x);
    }
};
```

במקרה זה, יש משתנה `const` ולא נוכל לכתוב את השורה הזו מכיון שהוא `const`. אז באיזה שלב הוא מאותחל? בשביל זה יצרו את ה-`member initialization list`. הרעיון הוא - רגע לפני שנכנסים לבנאי של `DC` אנו מניחים שהאובייקטים שבהם הוא משתמש מאותחלים, ולכן `b` צריך להיות מוכן לפני הכניסה לבנאי. בשורה הזו אנו מריצים את כל הבנאים של כל החלקים של `DC`, ואם לא נריץ את הבנאי, אז בנאי ברירת המחדל ירוץ. רואים פה שהאובייקט `b` ננעל ברגע שהבנאי של `DC2` סיים לעבוד.

נושאי עיצוב בתכנות מונחה עצמים:

ראשית, נעסוק בבלעדיות של אובייקטים. כלומר, האם שפה שהחליטה שהיא OOP, האם כל דבר הוא אובייקט. לדוגמא, Small Talk, האב הקדמון של OOP, כל דבר שם הוא אובייקט, וכן Scala. היתרונות – יש מבנה אחיד לשפה, כולם עובדים אותו דבר ונראים אותו דבר, פשטות מבנית של הקוד, קל ללימוד ולתפעול. החיסרון – גם דברים פשוטים שלרוב מתורגמים ל-assembly בפשטות, כעת יתורגמו בצורה יותר ארוכה ומסורבלת – העלאת אובייקטים, העברת הודעות וכו' (כמו בתרגיל 5 שרצינו להריץ member function).

יש שפות שהלכו על גישה אחרת, שפות כמו Ada ו-CPP, ברמת העיקרון השפה תומכת בשני הדברים. מי שרוצה יכול לקחת את שפת C הישנה ולעבוד בצורה האימפרטיבית והישנה, ומי שרוצה יכול לעבוד עם אובייקטים. היתרון – לצורכי יעילות ומורכבות ניתן לעבוד עם האפשרות המתאימה לאותו רגע. החיסרון – המבנה של הסוגים מבלבל, חלק מהסוגים עובדים בצורה אחת וחלק עובדים בצורה אחרת, השפה קשה ללמידה, מורכבת, לא ברורה, דברים לא נראים אותו דבר. יש מחיר שצריך לשלם אם רוצים הרבה אפשרויות.

הגישה השלישית היא, גישה שבה בחרו שפות כמו C#, Java, בגדול השפה היא OOP, אבל היא לא pure OOP. כלומר, יש סוגים ספציפיים כמו char, int, שהם עובדים בצורה פרימיטיבית ולא בצורת אובייקט, כל דבר אחר יהיה אובייקט. הרעיון – לדברים פשוטים ניתן לבצע תרגום מהיר ל-assembly, דברים פשוטים יעבדו בצורה מהירה יותר. החיסרון – יש דברים שכאשר נעבר בין סוגים, למשל לעבור מ-int לאובייקט לא יעבדו בצורה פשוטה וחלקה אלא נצטרך להשתמש במנגנונים למעבר הזה (זה שיש int ו-INT, אחד זה הפרימיטיבי והשני זה המחלקה, צריך כי יש מקרים שנשלח int ונצפה לקבל אובייקט של מחלקה).

הנושא השני שנעסוק בו הוא האם תת מחלקה הוא תת סוג. יש שפות שניתן לייצר תת סוג. לדוגמא, אפשר לייצר מאוסף כל ה-char רק אלו שהם אותיות גדולות, זה לא אובייקט חדש אלא זה פשוט סוג חדש, תת סוג. כל מקום שנצפה לקבל את הסוג החדש הזה אז נוכל לקבל רק אותו, אבל מקום שנצפה לקבל char, נוכל גם לקבל אובייקט מהסוג החדש. ניתן להשים אותו שם, כי הוא char, אבל הוא מוגבל. השאלה היא, האם גם באובייקטים אנו מייצרים כזה יחס. האם בכל מקום שנצפה לקבל אבא נוכל להשים בן. בדרך כלל Derived class is a Base class, כלומר כל מחלקת בן היא גם מחלקת אב.

```
class base
{
public:
    virtual void foo(int x, char c);
};

class derived:public base
{
public:
    void foo(int y){...}
};
```

הנושא השלישי הוא, האם יש בדיקת התאמת סוגים בעת ריבוי צורות (-פולימורפיזם). נניח ויש לנו את המחלקות הנתונות בשמאל וכתבנו את השורה הבאה: `B* bptr = new derived();` בהנחה שיש לנו בנאי. כמו כן, נכתוב גם `bptr->foo(...)`. כמובן שיש פה פולימורפיזם, כי אנו קוראים לפונקציה foo אבל אנו לא יודעים בזמן קימפול לאיזה foo אנו נקרא, כי יש שתיים כאלו, אחת אצל האב ואחת אצל הבן. אם לדוגמא נשלח שם int כלשהו, אז וודאי שזה יהיה חוקי כי יש foo אצל הבן שמקבלת int, ובזמן ריצה לא תהיה בעיה, אבל בזמן ריצה אנו לא יודעים האם bptr מצביע על מישור מסוג האב או מסוג הבן.

אז יש לנו פה בעיה, מצד אחד פולימורפיזם זה משהו שנבדק בזמן ריצה, כאשר בדיקת הסוגים מתבצע בזמן קימפול (כמובן ב-static typing).

הפתרון של שפות כמו CPP, הדבר הזה יחזיר שגיאה, בזמן קימפול נסתכל על הסוג של bptr ונראה שהוא מסוג base*, ויחפש foo אצל base שמקבל int אחד (נניח ושלחנו int כלשהו). אז בשביל שיהיה פולימורפיזם נצטרך שבזמן קימפול תהיה התאמה בין הפונקציה הווירטואלית אצל base לבין זו התואמת ב-derived, מה שנקרא – override, ולכן נצטרך להוסיף פונקציה foo כך:

```
void foo(int x, char c){...}
```

ללא זה, לא היה עובד הפולימורפיזם. לא בגלל שלא היינו יכולים לקרוא ל-

foo אצל הבן, כי לכאורה היינו יכולים. אלא, הבעיה היא שלא נוכל בזמן קימפול לבדוק את הסוגים ולכן בזמן קימפול כבר נוציא שגיאה שלא קיימת התאמה בין הסוגים. לא נוכל להגיד – בזמן ריצה זה ירוץ אז הכל בסדר, כי בזמן קימפול זה

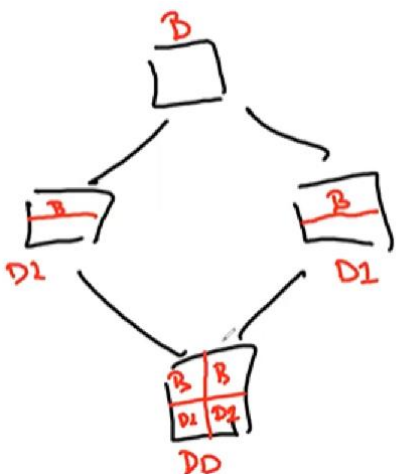
יחזיר שגיאה. ולכן יצרו לדוגמא כלל של בדיקת התאמת סוגים בעת ריבוי צורות (אנו צריכים שהסוגים יתאימו כדי שיהיה ריבוי צורות). וכל זה נגזר מהנושא של static typing עליו דנו.

ירושה מרובה:

כל שפה צריכה להחליט האם השפה מתירה ירושה מרובה או לא. ב-CPP ניתן לרשת משתי מחלקות. ב-Java, C# אי אפשר, אבל אפשרי לרשת ממחלקה אחת וממשק אחד. כלומר, קיים הבדל בין ממשק למחלקה, ובמילים אחרות, ישנם חסרונות בירושה מכמה מחלקות שלא קיימים בירושה מכמה ממשקים. יש יתרון לרשת מכמה מחלקות, לדוגמא, אם יש מחלקת סטודנט ומחלקת עובד, אז זה מאוד הגיוני לרשת משניהם אם אתה רוצה לבנות מחלקה של סטודנט-עובד, ואת זה יכולנו לעשות בזכות ירושה מרובה. לעומת זאת, יש לזה חסרונות גדולים. למשל, מה קורה אם במחלקת האב-1 פרמטר בשם name וגם אצל האב-2 יש פרמטר בשם name, איזה name יהיה מדובר עליו אצל הבן, אז נצטרך לכתוב כל פעם name וליד זה את שם האב שאליו אנו מתייחסים.

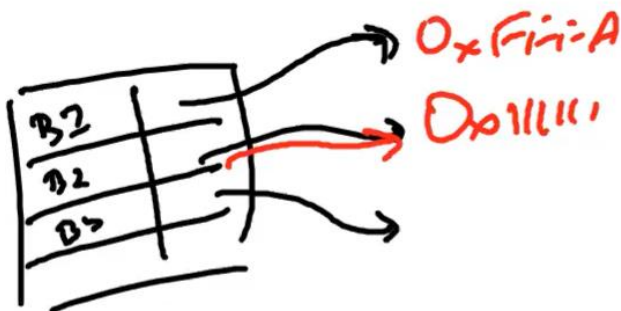
יש גם בעיית ירושת יהלום (זה לא שמישהו מראש יוצר ירושה כזו, פשוט כאשר יוצרים היררכיה ענקית אז זה יכול לקרות בדרך), איור מצורף בשמאל. ב-D1 ו-D2 יש את מחלקת B, ואז אצל הבן DD יש ארבע חלקים: B, D1, B, D2, ויכולים להיווצר מכך דברים מוזרים. ב-CPP פותרת בעיה זו ע"י virtual inheritance, כלומר, כאשר ניצור את D1 שירש מ-B, אז נוסיף לו בירושה את המילה virtual כך: `class D1: virtual public B`, וכך גם עבור D2. ואז רק בשעה שנגדיר את DD הוא ייצר base אחד עבור D1 ו-D2, ועוד כמה דברים, מנגנון שלם בשביל לפתור את הבעיה הזו.

באה שפה כמו Java ואמרה לא, אסור ירושה מרובה אלא אם כן מדובר על ממשק. ממשק זה מחלקה ללא data members, רק פונקציונאליות. הבעיה לא תיווצר כאשר נירש מכמה ממשקים ומחלקה אחת, כי תמיד יהיה ממשקים שיביאו פונקציונאליות, ועבור ה-data יהיה קו אחיד.



מימוש פולימורפיזם:

היה אפשר לממש ע"י שנשמור ב-activation record instance להכניס מצביעים עבור כל אובייקט איפה נמצאת הפונקציה שאותה הוא אמור להריץ, אך זה לא יעיל. נשים לב, כל האובייקטים מאותו סוג יריצו את אותו יריצו את אותן פונקציות, כלומר, אם יש במחלקה derived פונקציה כלשהי, אז כל מופעי המחלקה יריצו את הפונקציה הזו, ולכן לא נרצה לכל מופע לשמור את המצביע לפונקציה אותה הוא צריך להריץ, יותר פשוט לייצר טבלה עבור כל מחלקה וכל אובייקט יתאים את עצמו למחלקה לה הוא שייך. נניח לדוגמה D יורשת מ-B. כל מחלקה שיש לה פונקציה וירטואלית (לפחות אחת), נייצר עבורו V table - טבלה שיש בכל שורה את שם הפונקציות הווירטואליות (עוד מעט נראה שלא צריך את השם) ומצביע אל המיקום של הפונקציה (function pointer). כאשר נייצר אובייקט מסוג B, חוץ מה-data שלו, יש לו גם v_ptr, שהוא יצביע אל v table של B. אם בקוד של B כתוב שרוצים להפעיל את B1 (ו-B1 היא פונק' וירטואלית, ולכן היא בטבלה), אז הקומפיילר שותל קוד ששולח אותו לטבלה הווירטואלית לשורה הראשונה ושם יש את B1. אותו דבר יהיה גם עבור מחלקה D, תהיה טבלה וירטואלית עם כל הפונקציות הווירטואליות, אותו תהליך כמו שראינו עכשיו עבור B. נבחן מה יקרה עבור המקרה הבא: $bptr = \text{new } D();$ ובנוסף נקרא לפונקציה: $D2 \rightarrow bptr$. במקרה זה נקבל שגיאה בזמן קימפול. הקומפיילר רואה שהמצביע מסוג B, ולכן ילך לטבלה של B ויחפש שם את הפונקציה D2, אך היא לא נמצאת שם (היא נמצאת בטבלה של D, כפי שניתן לראות באיור). ולכן, הקומפיילר יוציא שגיאה.



V table - B



V table - D

לעומת זאת, כאשר נקרא לפונקציה $bptr \rightarrow B2$, אז הקומפיילר לא יוציא שגיאה. מה שיקרה הוא שהקומפיילר ילך לטבלה של B ויראה שיש שם את הפונק' B2 ולכן ישתול קוד האומר - לך לטבלה הווירטואלית שלך, ובשורה השנייה תמצא את הפונקציה. בזמן ריצה מה שיקרה הוא שאנו נלך לטבלה הווירטואלית של D, מכיוון שהאובייקט בסופו של דבר הוא מסוג D (!!!) ולא מסוג B. כלומר, נלך לטבלה הווירטואלית של D ונלך לשורה השנייה. נשים לב שיש כאן שני מקרים, אם D מימשה את הפונק' B2 אזי הכתובת שתהיה בטבלה של D בשורה השנייה תהיה שונה מזו שיש בטבלה של B בשורה השנייה, כלומר אנו נצביע לכתובת שונה, לכתובת של המימוש של הפונקציה B2, המימוש של המחלקה (נצביע ל-0x2222) ולא של המחלקה B (ולא ל-0x1111). אבל, אם המחלקה D לא מימשה את הפונק' אזי הכתובת תהיה אותה כתובת שיש בטבלה של D (כפי שניתן לראות באיור עבור הפונקציה B1, יש אותה כתובת כי D לא מימשה את הפונק'). כלומר, הקומפיילר התכוון לטבלה אחת (לטבלה של B) ובפועל בזמן ריצה הגענו לטבלה של D.

בנוסף, נשים לב שמבחינת הקומפיילר זה חוקי לגמרי שהמצביע יהיה מסוג B ובפועל יצביע על D מכיוון שגם ה-data של D וגם הטבלה הווירטואלית של D, הן הרחבה על אלו של B (באיור ניתן לראות שהחלק העליון של D הוא כל הטבלה של B, מסומן בסגול בהיר).

```
class base
{ public:
    virtual void foo(int x, char c);
};
class derived:public base
{ public:
    void foo(int y){...}
};
```

נובלות מאופן מימוש פולימורפיזם:

כפי שאמרנו, פונקציה שיש לפניה virtual אדי היא תיכנס לטבלה, ואם לא אדי היא לא תיכנס, ועבור פונקציה כזו הקומפיילר כבר בזמן ריצה יגיד לך את מי אתה צריך להריץ (הוא לא ישלח אותך לטבלה וכו', התאמה ישירה). כמו כן, אם יש לנו כמו במקרה לעיל, מחלקה B ומחלקה D היורשת ממנה (מצורף קוד להמחשה), אדי אם foo היא ווירטואלית אצל האבא אדי גם אצל הבן היא תחשב פונקציה ווירטואלית ← היא תיכנס לטבלה הווירטואלית של D גם אם לא כתוב אצלו virtual.

```
class base
{ public:
    virtual void foo(int x, char c) = 0;
};
class derived:public base
{ public:
    void foo(int y){...}
};
```

נתעסק כעת במושג pure virtual function. לא ניתן לייצר אובייקט מסוג מחלקה שיש לה פונקציה ווירטואלית טהורה (לפחות אחת). כמו כן, כל מחלקה היורשת ממחלקה זו, לדוגמה D, היא חייבת לממש את הפונקציה הזו, - את foo. זה נכון כי פונקציה ווירטואלית טהורה משמעת שבטבלה הווירטואלית של המחלקה המצביע למימוש יהיה null, כלומר אין מצביע למימוש כלשהו, ולכן לא ניתן לייצר אובייקט מסוג המחלקה מכיוון שהוא יצביע על טבלה ווירטואלית שתביא לנו שגיאה כאשר ננסה לגשת אל הפונקציה הזו. ולכן, חובה לדרוש - לממש את

הפונקציה הזו, וכך בטבלה הווירטואלית של D אכן יהיה מצביע עבור כתובת מסוימת שתכיל מימוש של הפונקציה foo. עבור מחלקה ללא פונקציות ווירטואליות, היא אמורה להיות final class (כמו ב-Java), כלומר זו מחלקה שאנו לא מצפים שיירשו ממנה, אין לה טבלה ווירטואלית, ולכן כל המנגנון של פולימורפיזם שראינו לא יכנס לפעולה ולכן אין סיבה שנירש ממנה. הסיבה היחידה שנירש ממנה זה בשביל ה-data שלה. מקודם אמרנו שלא צריך את השמות בטבלה הווירטואלית, אך אנחנו רשמנו לצורך ההבנה, בפועל אין שום צורך בשמות, הקומפיילר יוסיף קוד שישלח אותנו לאינדקס הנכון, השם לא ישנה לנו. כלומר, הטבלה מורכבת רק ממצביעים למימוש (function pointers). נשים לב לנקודה חשובה, שימוש בפולימורפיזם דורש מאיצנו עוד הפנייה, כלומר כל גישה תהיה כעת הגישה עצמה ועוד גישה.