$C\&C++$

24 באוגוסט 2011

סיכום הרצאות קורס הקיץ.
מחברת: נגה רוטמן.
צוות הקורס לא אחראי לתוכן, וכו', השימוש באחריות הקורא בלבד.

תוכן עניינים

חלק I
C

1 דברים בסיסיים

2/8/2011
4/8/2011

1.1 Switch

מבנה - שקופית 3.
בסוף כל מקרה - לשים break, מקרה דיפולטיבי תחת default. תמיד לתעד אם ויתרנו על ה-break!

1.2 המשך - שלבי קימפול

Main.c הופך ל-Main.o, שלא כולל את הישום של פונקציות חיצוניות אלא רק קישור אליהם. בשלב הבא יווצר קובץ executable שיכיל את הכל.
אם משתמשים בסיפריות חיצוניות יש להשתמש בפקודה lm- בסוף הפקודה gcc, לאחר הקומפילציה הראשונית של קובץ ה-c.

2 ניהול זיכרון

זיכרון המחשב מחולק למספר סוגי זכרון - אחד מהם הוא המחסנית - stack - first in last out.
stack overflow - אין מקום במחסנית לפריט המידע שניסינו להכניס.
כל המשתנים שהשתמשנו בהם עד עכשיו - בזיכרון במחשב הם נמצאים בצורה זו. בעזרת sizeof אפשר לדעת כמה מקום כל טיפוס תופס בזיכרון.
char הוא טיפוס הזיכרון הבסיסי ב-C - כלומר, תמיד הגודל שלו הוא 1 (בייט).
היכולת להמיר בייטים למספרים ניתנת על ידי טבלת ה-ASCI עליה דיברנו בשיעור הקודם.
בדוגמא בשקופית 11 - מניחים שהדאבל תופס 8 בייטים.
ב-struct מכיוון ואנחנו רוצים שיהיה יישור - alignment, לא רוצים שייוצר מצב שיקרא משתנה, ולמרות שהמשתנה קטן מהגודל הנקרא, לא הכל יקרא. לכן, יש בו יישור, כמו שדיברנו בשיעור הקודם - בעזרת פעולה זו, תמיד יקרא המשתנה המתאים ב-offset המתאים.

$$\text{Address} = \text{offset} \ \% \ \text{alignment}$$

לכל משתנה יש את ה-alignment שלו, וכאמור ערך זה תלוי מערכת הפעלה. דוגמא - שקופית 13.

2.1 מערכים

מגדירים בלוק של תאים המגיעים אחד אחרי השני. גודל התא בהתאם לגודל הטייפ של המערך.
דרך אחת לאתחל את המערך - בדומה לג'אווה, בשורת ההגדרה (שקופית 14), בעזרת סוגריים מסולסלים.

ישנם דרכים שונות להגדרות - דוגמאות בשק' 14 - הגישה היא לכתובת בזיכרון בה המידע שמור. במידה ולא ניתן לגשת לכתובת נקבל שגיאה, יש אפשרות גם לגשת למידע שאנחנו לא אמורים לגשת אליו ולדרוס אותו, ועוד, על כן יש חשיבות גדולה לניהול הזיכרון - בניגוד לג'אווה, שם מקרה כזה לא יתכן (התוכנית לא תתקמפל).

אין דרך לדעת מראש את הכתובת בזיכרון אליה נשמר המידע - זה נקבע בזמן הקומפילציה ובזמן הריצה (כלומר, בדוגמא, אין דרך לדעת שאכן המערך מתחיל בכתובת 40).

כמו כן אסור להשוות בין כתובות, כגון a=b, $\text{if } (a == b)$ - נראה דרכים להשוואה נכונה בהמשך. המשמעות של הביטויים הללו - הצבה של מערך בתוך מערך, וזה אינו אפשרי.

2.1.1 אתחול

דוגמאות - מצגת 17.

הקריאה השלישית אפשרית רק לערך 0.

קריאה רביעית - סטייל גרוע - יאתחל את שאר הערכים לאפס. אם משתמשים בזה בכוונה, יש לשים הערה בצד ולהבהיר את השימוש.

קריאה חמישית - אי אפשר להגדיר גודל מסויים למערך ולהגדיר לתוכו יותר ערכים מהמקום שהוגדר.

אפשר גם להגדיר מערכים דו מימדיים - אפשר לאתחל בשתי דרכים - קריאה שישית ושביעית במצגת. שתיהן שקולות זו לזו, ושקילות זו נובעת מהדרך בה מאוחסן המידע - בצורה רציפה.

דוגמא מהכיתה:

$$\begin{aligned}
&\text{int arr}[2]\,[3]\\
&\text{sizeof}(int)==4\\
&\text{arr } =40\\
&\Rightarrow \text{arr}[1]\,[0] = 40+(i\times 3)\cdot 4+\text{j}\times 4
\end{aligned}$$

כאשר i הוא מספר השורות, ו-j מספר העמודות. מכיוון והמערך הוא של int, אזי כל "תא" יהיה בגודל 4, ולכן על מנת למצוא את מיקום מסויים, כגון $\text{arr}[1]\,[0]$, נשתמש בנוסחא מעלה.

על כן, הקריאה הכמעט אחרונה שגויה! שהרי החלוקה של התאים אינה נכונה.

שימוש בסוגריים מסולסלים יתכן רק בעת ההגדרה, ולא בשלב מאוחר יותר.

2.2 פוינטרים

פוינטר - משתנה המצביע על מקום בזיכרון:

הצהרה על פוינטר - $*$. שתי צורות ההצהרה הבאות שקולות:

$$\text{int *x,y; int* x,y;}$$

הצורה השמאלית ברורה יותר, ועדיף להשתמש בה.

בדוגמא בשקופית 18, המשמעות של הפעולה עם החץ היא שבתוך x יש כתובת בזיכרון של int. מומלץ להצמיד את הכוכבית ל-x.

השורה x=&i משמעה - x מכיל את הכתובת של הערך של i - כלומר, בדוגמא - 0.

$x = \&j$ - כלומר x הופך להיות הכתובת בזיכרון בה שמור j. השורה בסוף משנה את הערך של j - כי $(*x) = j$.

נשים לב כי לא נוכל לכתוב $j = x$, מכיוון ואלו שני סוגים שונים של משתנים.

דוגמא לשימוש בפוינטרים - שק' 23 - ללא שימוש בפוינטרים, הערכים לא ישתנו כשנחזור לתוכנה המקורית (כמו בג'וואה). לעומת זאת, אם נשתמש בפוינטרים, הערכים באמת יוחלפו כפי שנרצה. בקריאה לפונקציה - אנו מעבירים את הכתובות של הערכים.

2.3 פוינטרים ומערכים

בשק' 24 - אפשר "לשאוב" בעזרת פוינטרים מידע מתוך מערכים.

קיימת "אריתמטיקה של פוינטרים" - שורה 4 - ניתן לקחת את p ולהוסיף לו 1 - כך אפשר לגשת לכתובת הבאה. יש לשים לב לשים סוגריים במקום המתאים.

לגבי העתקה של $struc$-אים:

ההשמה $c2 = c1;$ לוקחת את הקטע בזיכרון שמופיע $c1$ ומעבירה ל-$c2$ - זה פשוט כמו טיפוס פרימיטיבי.

לכן, בעת ההחלפה בין סטראקטים, יש להשתמש גם בהחלפה כפי שמופיעה בשק' 24.

בשק' 25 - למעשה a ללא הסוגרים המרובעים היא כתובת בזיכרון, ולכן השורה השלישית חוקית. ההשמה ההפוכה **אינה חוקית**, כי a אינו מסוג פוינטר. כמו כן, אי אפשר לקדם את a, כי זוהי כתובת קבועה.

בשק' 26 - הגודל של p הוא קבוע - זוהי הרי כתובת בזיכרון. כאמור, גודל זה תלוי במערכת ההפעלה. נשים לב להבדל בין הגודל של p לגודל של a - גודל זה משתנה בהתאם לסוג המשתנה ולמספר התאים.

נשים לב לשיוויון בשק' 29 - ברגע שאנחנו מעבירים מערך כמשתנה לפונקציה, הערכים משתנים - כבר לא נקבל את גודל המערך ב-$sizeof$, אלא את הגודל של תא - מידע זה "אובד" בדרך.

לכן, אם נרצה לעבור על מערכים, יש להעביר גם את גודל המערך גם כן לפונקציה.

2.4 אריתמטיקה של פוינטרים וקאסטינג

שק' 31 - ניתן לבצע קאסטינג על ידי סוגריים. במקרה זה, הקאסטינג בשורה השלישית לא טוב, אבל מותר "לכפות" זאת (אם לא היינו עושים קאסטינג היתה שגיאת קומפילציה).

לאחר פעולה זו, ב-q וב-p ישנו אותו ערך. עם זאת, ב-p אנחנו משתמשים ככתובת של int, וב-q אנו משתמשים ככתובת של $char$ - לכן כאשר נקדם (או נחסיר) את כל אחד מהמשתנים הללו, נקבל כתובות שונות.

דוגמא נוספת בשק' 32 - ריצה על מערך בעזרת אריתמטיקה של פוינטרים.

עם זאת, עדיף להשתמש בצורה התחתונה בשק' בתכנות, אך חשוב להבין את הצורה המופיעה מעלה בעזרת אריתמטיקה של פוינטרים.

דוגמא אחרונה בשק' 33 - יש לשים לב כי אריתמטיקה של פוינטרים עובדת רק עם פוינטרים!

מכיוון ו-a היא כתובת (במערת 64 ביט היא בגודל 64 ביט = 8 בייט), ולכן ניתן לבצע את ההצבה ל-i ול-j, שניהם מסוג long. מכיון ומשתנים אלו מסוג זה, ההבדל המחושב יהיה גודל של int, ולא כתובת.

2.4.1 $void*$

שק' 34-35 - אין צורך בקאסטינג מפורש, מתבצע קאסטינג אוטומטי בין $void*$ לפוינטר. ההפך אינו אפשרי, ויש להשתמש במקרה זה בקאסטינג מפורש.

עבור $void*$ לא ניתן להשתמש באריתמטיקה של פוינטרים, ולא ניתן לגשת לערך של הפוינטר.

מה כן אפשר? לאחר קאסטינג (מפורש) אפשר להשתמש בסוג זה כפוינטר, כפי שמופיע בשורה האחרונה.

את הכוח בשימוש בסוג זה נראה בהמשך.

2.4.2 NULL pointer

דרך לציין שהפוינטר לא מצביע על שום דבר. מוגדר ב־stdlib.h. יש לשים לב שניתן לבצע פעולות על פוינטר רק אם הוא אינו NULL. אם לא - כפי שמופיע בשק' 37 - יעבור קומפילציה, אבל יגרום לבעיות בריצה.

2.4.3 פוינטרים לפוינטרים

שק' 38.

סימון - $\%p$ - ידפיס את הכתובת בצורה נוחה (הקסדצימלי).

2.4.4 מחרוזות ב־C

אין סוג מיוחד למחרוזות ב־C. במקום זאת, סטרינגים הם מערך של $char$ (שק' 39). תמיד התא האחרון יכיל את הערך $\backslash 0$ - כלומר, אם נגדיר את המערך להיות "$hello$" - אזי הוא יהיה בגודל 6 ולא 5 (בשביל מקום לתא הנוסף).

דרך נוספת להגדרה נמצאת בשק' 40 - צריך לשים גרש לפני כל אות. מכיוון ולא השתמשנו בגרשיים המיוחדים, זה יכול להיות מאוד בעייתי בעת שימוש. לכן, אם משתמשים בשיטה זו לאיתחול מחרוזת, יש לשים לב תמיד לשים בסוף תא נוסף הם הערך המיוחד $\backslash 0$.

כל המניפולציות על מחרוזות יפעלו היטב רק אם באמת הוגדר התא האחרון הנ"ל! את הפעולות על מחרוזות נראה בתרגול.

2.5 קבועים

ראינו איך מגדירים קבועים. נרצה להגדיר קבועים שלא ניתן לשנות - בדומה לfinal בג'וואה.

נשתמש במילה הקבועה const, והכלל - שומר על הצד השמאלי בלבד. אם אין שום דבר בצידו השמאלי - ישמור על הצד הימני משינויים (שק' 42). דוגמאות לשימוש - שק' 43-44.

נשים לב כי ללא הקאסטינג בשק' 44 היינו מקבלים טעות בגלל ה־const - אחד המשתנים הוא מסוג זה והשני לא.

מומלץ לא להשתמש ב"הסרה" הזו של ה־const, אבל נראה בהמשך מקרים מיוחדים בהם נשתמש.

מאוד מומלץ ואף דרוש להשתמש ב־const בעת קריאה לפונקציות.

אם הכוונה היא שהפונקציה לא תשנה את הפרמטרים שימוש ב־const יעזור לנו לוודא כי תנאי זה אכן מתקיים. כמו כן, שימוש בסוג זה הופך את הקוד ליותר קריא.

נשים לב בשק' 45 - בדוגמא הראשונה, לא ניתן לשנות את הדברים עליהם p מצביע, אך ניתן לשנות את p עצמו. בדוגמא השניה - ההפך!

אם נרצה להגן גם על הכתובת וגם על הפוינטר עצמו, פשוט נשתמש פעמיים ב־const.

בשק' 46 - מלכתחילה לא ניתן לשנות מערכים.

בשק' 47 - האפשרויות השניה והשלישית שקולות - מגינות על הערך אליו הפוינטר מצביע.

בשק' 48 - דרך חדשה לאתחול struct - הצבה של הערכים בעת האתחול, עם const. לעיתים ניתן לשנות את הערכים בצורה עקיפה אם הגדרנו כך - על ידי גישה ישירה לזיכרון, אך עלול להיות בעייתי.

בשק' 49 - האובייקט מוגן אז לא ניתן לשנות אותו, אבל ניתן לשנות ערכים בו! (בניגוד לג'אווה).

שק' 50 - האורך של סטרינג לא משתנה, אז חשוב להשתמש ב־const. נראה שימושים נוספים ב־$C++$.

2.5.1 עוד קצת על $headers$

8/8/2011

ראינו בהרצאה קודמת את הוספת הdefine ל־header. אבל, מה קורה בשק' 4? יש לנו הגדרה כפולה (למרות שיכול להיות שזו בעצם אותה הגדרה, את ה־preproccessor זה לא מעניין) - הפתרון בשק' הבאה. התגית #ifndef - "אם לא מוגדר", וכך נמנע מהגדרה כפולה - אם לא היתה הגדרה של המבנה, תכלל ההגדרה החדשה. אחרת, החלק הזה בקוד ימחק בשלב זה של הקומפילציה. יש להוסיף את התגית הזו הכל קובץ header על מנת להמנע מטעויות שכאלו.

מה כדי לעטוף בתגית זו? כדי לעטוף את כל הקובץ, ולא רק פונקציה או מבנה מסויים.

עוד מקרה בעייתי - מה קורה בהגדרה מעגלית? כלומר, שני קבצי h שכל אחד רוצה להכיל את השני. בדר"כ דבר זה מעיד על בעיה בתכנון, נרצה להמנע ממצב שכזה, ונרצה להוציא את הדברים המשותפים לשני הקבצים לקובץ חיצוני.

2.6 מבנים - $struct$

ראינו כי הזיכרון הוא זיכרון רציף - מערך יופיע בצורה רציפה על הזיכרון, למרות שאולי יהיו תאים "שמורים" על מנת לשמור על גישה נוחה למידע שבמערך.

כמו כן, נשים לב לפוינטר - שימוש ב־$(p*)$ או $p->$. בצורה הראשונה, יש לשים לב לסוגריים!

2.6.1 העתקת struct

שק' 6 - העתקה בעזרת " = " תעתיק רק את הערכים!

לכן, בצורה זו, המערך יועתק כולו.

אבל - מה קורה עם פוינטרים? דוגמא בשק' 12 - הפוינטר מצביע לכתובת של המערך. אזי, בעת העתקה שכזו, הכתובת של הפוינטר של העצם השני יצביע לכתובת של המערך של הראשון a - לא כפי שרצינו!

אזי, במקרים של פוינטרים, יש להגדיר פונ' clone (כמו בג'וואה) בעת ההעתקה - כמו שק' 16, שתעתיק את המידע ואת הפוינטרים בצורה הנכונה. נדבר בהמשך על המיקום של הפונ' הזו - היכן כדאי למקם אותה על מנת לשמור על יעילות גבוהה. הלולאה הראשונה שוות ערך לפעולה עם האופרטור " = ", בעוד השורה האחרונה מעדכנת את הפוינטר לערכו הנכון.

כאשר מעבירים מערך - כמו שעושים שווה למערך - מועבר רק הפוינטר לערך הראשון במערך. כלומר, אם יש פונק' שמקבלת מערך, היא מקבלת למעשה את הכתובת של האלמנט הראשון במערך. במקרה של מערך דו מימדי, יש לרשום ספציפית את הגדלים של המערך (הפרמטרים), כדי שהקומפיילר "ידע" איך לגשת לכתובת הנכונה בזיכרון. כאשר מעבירים struct, מועברים הערכים, כמו הטיפוסים הבסיסיים.

דוגמא - שק' 18 - הפונקציה g לא משפיעה על x, אלא על המשתנה המקומי החדש s, בעוד f מקבלת מערך, כלומר את הכתובת של האלמנט הראשון של המערך של x, ולכן

השינוי תקף ל־x. נשים לב שיש כאן שגיאת קומפילציה - צריך להגדיר *typedef*. "בטוח שיהיו שאלות בסגנון זה במבחן".

2.7 Bit fields

דיברנו בשיעורים האחרונים על כך שמבנה הנתונים הקטן ביותר שאנחנו מעבירים הוא char. מצד שני, היתרון בעבודה ב־C הוא עבודה יעילה.

אז, איך נעביר בתים בודדים? שק' 21.

ניתן לעשות or לוגי - מבצע אותו int int. השורה הראשונה של הקוד "מדליקה" (אם נחשוב על הביטים כפי שהם מופיעים פיזית בזיכרון - מתג שהוא דלוק או כבוי). את הביטים שהם external ו־static, השניה - "מכבה" אותם - פעולת and.

ישנן מספר שיטות הצגה - שק' 22.

מה נעשה עם זה? שק' 24. ניתן להגדיר סטראקט כלשהו, שיכיל unsigned int אחד - במערכת שלנו 4 ביטים. לכל אחד מהמשתנים שם מגדירים ביט אחד במבנה - הראשון הוא הימני ביותר.

השורה השניה בפעולה "מדליקה" את שני הביטים העליונים.

השורה האחרונה - האדומה - לא תעבוד.

משתמשים בשיטה הזו על מנת להעביר מידע בצורה חסכנית.

2.8 union

לא משתמשים בזה הרבה (שק' 25) - יוקצה לו הגודל הגדול ביותר במבנה - double, אבל ניתן להשתמש בו, אם מתייחסים למשתנה הקטן, כמו בגודל הקטן - דוגמא לשימוש נמצאת בשק' 26 - שימוש ב־int "ידרוס" רק את ארבעת הביטים הראשונים.

2.9 סטאטיות

כאן יש מספר סוכים, חלקם דומים לג'אווה, חלקם שונים ויש להזהר.

2.9.1 static heap

המיקום שלו גלובלי, ונשמר תמיד - לא נמחק בסיום הריצה של הפונקציה. בשק' 28 - רק ביצירת המשתנה יתבצע האתחול, בפניות הבאות לא, על כן בפניה השניה נקבל 2.

סוג זה דומה לג'אווה.

2.9.2 משתנה גלובלי סטאטי

דומה למשתנה גלובלי - הוא private לפונקציה מסויימת, למשל בשק' 29 - בשימוש של קובץ אחד במשתנה של קובץ אחר, נוכל "לשאוב" משתנים מהקובץ האחר בעזרת השימוש במילה השמורה extern. אולם, מכיוון ו־x סטטי, לא נוכל לשנות אותו (הטעות תהיה בשלב ה־linkage).

2.9.3 פונקציות סטאטיות

שק' 30 - בדיוק כמו המשתנים, רק עם פונקציות.

2.10 עבודה בפרוייקט עם מספר קבצים

לזכור לבצע linkage בין הקבצים השונים!
טעויות אפשריות:

- חסר אימפלמנטציה.
- אימפלמנטציה כפולה.

אם שינינו רק קובץ אחד, אין צורך לקמפל מחדש את שאר הקבצים. נרצה כלי חכם שידע לעדכן רק את הקבצים שתלויים בקובץ שעדכנו (שוב, יעילות). ה־Make הוא כלי כזה. מומלץ מאוד להשתמש בו, גם אם לא מבקשים זאת בצורה מפורשת. שימוש וסינטקס - שק' 37 והלאה.

קובץ ה־makefile הוא מאוד מורכב - מכיל חוקים מפורשים, חוקים לא מפורשים, הגדרת משתנים, הוראות והערות. ניתן להשתמש בו גם למעקב אחר עדכוני קבצים.

הפעולה עצמה פשוטה - מוצאים את החוק המתאים, בודקים אם הדרישות לחוק מעודכנות, אם אחד הדרישות חדשה יותר מהמטרה - מריץ את הפקודות המופיעות בגוף החוק.

בדוגמא בשק' 41 - אם חתימת הזמן של prog1 חדשה יותר מזו של הקבצים האחרים - לא יקרה דבר. אחרת, תתבצע ההוראה המופיעה לאחר מכן: באופן רקורסיבי, הוא יבדוק את הקבצים הרלוונטיים. אם אחד הקבצים בכלל לא קיים, יקמפל אותו.

הפעלה - make prog1. אם לא פירטנו - יבצע את הפקודה הראשונה.

הכנסת הערות ב־makefile - בעזרת סולמית

2.10.1 משתנים

case sensetive. כיצד נשתמש בהם? $(varname)$ (שק' 43).

בשק' הבאה - רשימה של פעולות על המשתנים.

*.o - לא בעת הגדרת משתנים! פשוט יחליף את האובייקט בסטרינג (שק' 45).

2.10.2 חוקים לא מפורשים

שק' 46 והלאה.

למשל - נוכל להגדיר תלות של כל קובץ o. בקובץ המתאים c., וכשנכתוב את התלויות של הקובץ o. לא יהיה צורך לכתוב את c. - בשל החוק הנ"ל, הוא יבדוק אותו בכל מקרה.

אם אנחנו לא יודעים אלו חוקים יש built in - פשוט לבדוק.

לכן, כדאי לכתוב פעם אחת makefile מפורט כל האפשר - ולהשתמש בו בתרגילים הבאים (בשינויים קלים).

2.10.3 תלויות אוטומטיות

gcc יודע לפתור תלויות (שק' 48) - מוסיף בצורה אוטומטית תלויות ל־makefile בעזרת הפקודה gcc -MM *.c » makefile - יוסיף אותם לסוף ה־makefile. עדיף לנסות לבד ואז להשוות עם התוצאה של הפקודה הזו בשביל תרגול. על מנת לבצע פקודה זו מחדש - למחוק את הקטע שכבר הוסף (אם נריץ את הפקודה כמה פעמים, הכל יתווסף כמה פעמים).

כמו כן, ניתן להכניס את הפעולה הזו כחוק ב־makefile עצמו. כדאי לקרוא את ה־manual של ה־makefule - בעזרת:

Read: man makedepend

2.10.4 חוקים נוספים

- all - בדר"כ תהיה הפקודה הראשונה ברצף.
- אם נרצה למחוק - נוסיף את הפקודה phony. (באותיות גדולות) - כלומר, לא שם קובץ אמיתי - מחיקה של ה-executables וקבצי object.

2.10.5 makefile אוטומטי

לאקליפס לדוגמא יש makefile אוטומטי. עם זאת, עלינו לכתוב את הקבצים שלנו לבד.

2.11 ארגון מידע

בעת ריצה, משתנים יכולים להשמר בשלושה מאגרים:

- stack
- מיקום גלובלי
- ערימה דינמית

המימוש של הסוגים הללו היא ברמת הקומפיילר (תוכנה).

2.11.1 Stack

מנהל מידע במהלך ריצת הפונקציה (שק' 53).
הגודל מוגדר בזמן הקומפילציה. כשמכניסים דברים למחסנית, מקדמים את הפוינטר. קטעי הזיכרון במחסנית נמחקים לאחר סיום הפעולה. כיצד מתבצעת המחיקה? לא מחיקה אמיתית, כי אם העברה של הפוינטר למיקום הנכון (למקום הראשון בו מופיע המידע אותו נרצה למחוק). אם לא ימחק - לבסוף יגמר המקום, ונקבל הודעת שגיאה.
על כן, כלל מאוד חשוב - אף פעם לא להחזיר מפונקציה כתובת של משתנה מקומי! - הרי הוא ימחק בסוף הריצה של הפונקציה.

2.11.2 איזור גלובלי

(שק' 63) מוקצה בזמן הקומפילציה. ראינו שנגיש לא רק לפונקציה אחת, אלא לכל אחד (אלא אם כן הגדרנו משתנה מסויים כסטאטי).
דוגמא - שק' 65. לעיתים - מבוזבז, לעיתים - לא מספיק.
דיברנו על סטרינגים מיוחדים, ונראה עוד בהמשך - ברגע שכותבים כתובת:

```
char *str="hello"
```

הקטע "hello" יהיה חלק מהקוד - באיזור גלובלי, ולא ניתן לשנות אותו.

2.11.3 ערימה דינאמית

(שק' 67) ניתן להקצות מקום תוך כדי ריצה, כלומר, לא חייבים לדעת בהתחלה כמה מקום צריך. גם יכול להגמר, אבל יש לנו יותר שליטה עליו. יש לדאוג לשחרור זכרון זה בסוף הפעולה.

כיצד יוקצה המקום? בעזרת ה פקודה malloc* (שק' 68) - הפקודה מחפשת מקום בזיכרון שמתאים לגודל שאנו מחפשים, מסמנת אותו עבור המשתנה שביקשנו, ומחזירה לנו את המיקום - לא "מעניין" אותה לאיזה סוג של מידע נשתמש במקום, אלא רק הכמות. כיצד משתמשים? יש לבצע את הקאסטינג המתאים.

צריך לבדוק תמיד את הפונקציות שאנו מבצעים, ובפרט את פונקציה זו - האם הפעולה הצליחה, כלומר, במקרה זה, אם מצאה מקום מתאים.

דוגמא לשימוש בשק' 69 - מחפשים כתובת, מבצעים לה קאסטינג לסוג המבוקש - במקרה שלנו, *int.

באותה צורה - אפשר לבצע לכל משתנה, גם לסטראקט.

לאחר סיום העבודה - יש לשחרר את הזיכרון, כלומר, לבצע מחיקה - בעזרת הפקודה:

$$\text{free}(iptr);$$

10/8/2011

overloading

לא קיים ב-C! כלומר, אפשר להשתמש בשם של פונקציה פעם אחת בלבד.

(שק' 12) - יש לזכור לאתחל פוינטר ל-$NULL$ לפני שמתייחסים אליו. כמו כן, ליצור פונקציית אתחול לסטראקטים - דוגמא בשק' 13. בדוגמא זו לא נעשית בדיקה עבור malloc (או כל פונקציה אחרת) אם הפקודה התקיימה או לא - כלומר, במקרה זה, אם נמצא מקום בגודל שאנחנו צריכים, אבל כדאי תמיד לבצע בדיקה שכזו.

(שק' 14) - הסטרינג הוא חלק מסגמנט של הקוד, ולכן ההתייחסות אליו כקריאה בלבד[1]. אזי, איך נעתיק סטרינג? שק' 15!

נקבל בשקופית זו שגיאה - כי הסטרינג מסתיים ב-$\backslash 0$ ולא ב-m, לכן יועתקו 7 תווים במקום ה-6 שהוקצו למשתנה $p2$. ניתן להשתמש בגרסא ה"בטוחה יותר" להעתקה שראינו בעבר - strncpy, שדואגת שמספר התווים שיועתקו לא גדול מהמספר שהוקצה ליעד. אולם, במקרה זה לא נקבל סטרינג, מכיוון ולא מופיע תו הסיום $\backslash 0$, ולכן בעת הדפסה לא ניתן לדעת מתי הוא מסתיים.

הדרך הנכונה אם כך להעתיק את הסטרינג היא לחבר 1, כפי שמופיע בשק' 17.

יצירת עותק של סטרינג - בשק' 18, בעזרת פקודה מהספריה הסטנדרטית. חשוב לזכור לשחרר את הזיכרון בעת שימוש בפקודה זו. כיצד משחררים הזיכרון? בעזרת הפקודה free (שק' 20). נשים לב כי לפקודה זו אין בדיקת שגיאות - כלומר, אם למשתנה p לא הוקצה זיכרון, ההתנהגות של הפונקציה לא צפויה. אם מעבירים לפונ' הזו NULL, לא עושה כלום.

דוגמאות לשחרור - שק' 21-22.

שק' 25 - יש חשיבות גדולה לסדר - קודם כל יש לשחרר את הראשון, ורק אחרי כן את v - אחרת כנראהמאת שבזמן הריצה היתה טעות סגמנטציה. כמו כן, חשוב להוסיף את הבדיקה שמופיעה בשק' 26.

[1] זוהי צורה שונה לקריאה לסטרינג מהקריאה:

char str[]="hello"

VLA 2.11.4

מערכים שהגודל שלהם משתנה באופן דינמי. כיצד להגדיר? שק' 28. ב־C99 ניתן להגדיר מערך שכזה גם על ה־stack, אבל לא קיים ב־ANSI. לא בכל המקומות משתמשים עדיין ב־C99, ולכן עדיף לא להשתמש בזה (או בכל פיצ'ר אחר של C99 בלבד). בתרגילים בקורס זה אסור להשתמש במערך מסוג זה.

program design 3

Interfaces 3.1

(שק' 32) הגדרה של סט של פונקציות הספק מודול קוהרנטי (או ספריה). נראה בשיעורים הבאים איך יוצרים קבצי ספריה. שימוש באינטרפייס לשימושים שונים.

מודולריות 3.1.1

יהיה באינטרפייס - "מה", לא יהיה - "איך" - כלומר, היישום של הפונקציות.

שמירה על האינקפסולציה והסתרה של המימוש מאוד חשובה. בצורה זו, כל התוכנות הנמצאות באינטרקציה עם התוכנה שלנו לא משנות את השימוש אם שינינו את היישום. אם נשדרג את המערכת, אפשר לעדכן את האינטרפייס. לכן, את המבנה עצמו של הסטראקטים עדיף "להחביא" כדי שנוכל לשנותו בהמשך.

דוגמא לאינטרפייס - שק' 34 - StrStack. פעולות - המפורטות במצגת. יישום - בקובץ מצורף למצגת.

שימוש ב־*const* בחתימות של פונקציות - מדוע?

- תיעוד - הצהרה שהפונקציה לא משתנה את המשתנה שהיא מקבלת.
- מניעת שגיאות - אם בטעות נעשה שינוי בעת היישום, נקבל טעות קומפילציה, ואז נוכל להמנע מכך.
- הטיפוס שאנו מקבלים הוא const - אם לא היה, לא יכולנו להעביר אותו.

לאחר מימוש האינטרפייס, ישנן החלטות פנימיות של היישום (כמובן שיכולות לשנות את זמן הריצה, וכו') - שק' 35־36:

- מבנה הנתונים - באיזה סוג נשתמש? במקרה זה הוחלט על רשימה מקושרת בשביל פשטות.
- העתקה של סטרינג - לשמור עותק, ואז יש לנהל את הזיכרון, או להחזיק רק פוינטרים? במקרה זה הוחלט להחזיק פוינטרים.

דוגמא חשובה: יישום

נגיד ושתי כתובות שונות - p1,p2 מצביעות לאותו מקום בזיכרון - נסמן:

$$pp1=\&p1;\ pp2=\&p2$$

אזי:

$$*pp1==*pp2$$

- בקובץ מצורף למצגת. בפונקצית שחרור הסטאק - משחררים כל ערך בסטאק, ואז מאפסים את הכתובת של הסטאק עצמה.

3.1.2 עקרונות

- אנקפסולציה:
  - החבאת ה-data structures.
  - לא לתת גישה ל-data structures שעלולים להשתנות במימוש אלטרנטיבי.
  - פרט "חשוף" לא יכול להשתנות בשלב מאוחר יותר.
- השתמש בסט קטן של פעולות "פרימיטיביות":
  - get, set...
- don't reach behind the back! (שק' 41).
- שמור על קונסיסטנטיות:
  - חתימות דומות לפונקציות.
- ניהול המשאבים
  - ניהול באותה הרמה - אם יש פונקציה שמקצה מקום, ליצור פונקציה שמשחררת את המקום. אם המודול לא מקצה, הוא גם לא ישחרר.

4 פוינטרים לפוינטרים ומערכים רב מימדיים

4.1 פוינטרים לפוינטרים

שק' 45 והלאה.

נשים ♡:

הגודל של כתובות בזיכרון הן אותו הגודל - כלומר, בשק' 47, הגודל של $*p2$ ושל $**ipp$ הוא אותו הדבר.

הפקודה *ipp=&k משנה את המצביע מ-j ל-k.

ראינו את פונקציה ההחלפה swap, מתי היא עובדת ומתי לא - תזכורת בשק' 49.

שק' 50 - דוגמא של פוינטר לפוינטר.

4.2 מערכים רב מימדיים

4.2.1 אוטומטית

כששולחים לפונקציה - לא חייבים לשלוח את המימד הראשון, חייבים את הבא.
דוגמאות לקריאות לפונקציות חוקיות ולא חוקיות - שק' 52.

4.2.2 מערך סמי-דינמי

(שק' 54) מגדירים מערך של פוינטרים. כל אחד מהפוינטרים האלו מצביע לבלוק בגודל מסויים, בדוגמא - כל בלוק בגודל 7. בזיכרון, הוא יהיה מסודר בצורה שונה לגמרי.

4.2.3 מערך דינמי

הגדרה - לפי שק' 55, בשני שלבים:

- מגדירים מקום עם malloc חמש שורות בגודל של int* - כלומר כתובת של מערך של אינטים.
- עוברים על חמשת השורות - ובכל שורה מגדירים את המצביע להיות למערך של שבעה אינטים. בקוד יש טעות - יש לציין int*. כמו כן, יש לבדוק שה-malloc עבד, כלומר, שלא החזיר NULL.

בדוגמא זו - רק הכתובת של המערך היא על ה-Stack.
פעולת הגישה היא יותר יקרה, כי המקום לא רציף בזיכרון.
לא לשכוח לעבור על כל התאים בזיכרון ולשחרר אותם! כיצד? שק' 56 - לולאה על השורות - לשחרר שורה שורה, ואז לשחרר את הפוינטר הראשון.
עדיף לרשום בסוף הפקודה array=null.

4.2.4 יותר יעיל!

נרצה לשמור על הדינמיות, אך גם על היעילות שנקבל מכך שהמידע רציף. כיצד? שק' 56.
אפילו יותר מסובך - מערך של מטריצות - תלת מימדי - שק' 58.

4.2.5 פוינטרים למערכים

כיצד מגדירים? שק' 60. שתי דרכים - שני יישומים שונים!
char (*arr)[5] - פוינטר לתחילת המערך. char *arr[5] - מערך של מצביעים ל-char.
איך זה "נראה"? שק' 61. שימושים לשתי הדרכים - שק' 62.
מה נעשה?
נייצג במערך חד מימדי במקום דו מימדי. יישום זה יותר נוח ויותר יעיל, אך פחות קריא.
אופטימיזציה - מצגת 5 שק' 3 - עדיף להשתמש באופציה השניה - כך הקריאה של המידע רציפה (המידע נמצא קרוב). 15/8/2011
בשק' 4 - הכרזה על מבנים (סטראקטים) - הצורה השניה בהערה לא תתקמפל (כי במקרה כזה על התוכנה להכיר בדיוק מה מוכרז - גדלים וכו'). על כן, חייבים ממש לפרט את המבנה על ידי פוינטרים.

4.2.6 *goto*

במקום *if* ו-*break* אפשר להשתמש ב-*goto* - עדיף לא להשתמש בזה כי קשה לעקוב אחרי זה, אבל כדאי להכיר (דומה לאסמבלי - שק' 6).

4.2.7 קבועים

תיקון לתרגול - אמרנו שאם לא נרשום *const* נקבל הערת קומפילציה (שק' 8) - לא נכון! ה-*const* לא רקורסיבי. לעומת זאת, הקוד המופיע מתחת כן יתן הערת קומפילציה, וחייבים לשים *const*.
שאלה מהפורום - למה אי אפשר להעביר char** לפונקציה שמחכה ל-const char**?
אם אנו מכריזים על const char (שורה ראשונה) הקומפיילר יכול לבחור (אך לא חייב) לשים את המידע באזור שהוא read only. אם לא היינו מקבלים אזהרה בפעולה 3, היינו יכולים, על ידי הפעולות הבאות, לשנות את הערך של p2**, בסתירה להגדרה של c כקבוע.

4.2.8 volatile - לא בחומר הבחינה

שני $qualifiers$ שנכיר - הראשון הוא ה-const, השני - volatile - נשתמש בקורס אחר (מערכות הפעלה).

4.3 מצביעים לפונקציות

C מרשה לתת פוינטר לפונקציה (הפונקציה עצמה הרי נשמרת בזיכרון במקום מסויים). דוגמא - שק' 12. חייבים להשתמש בסוגריים - אחרת היה הקומפיילר חושב שאנו משתמשים על תוצר של פונקציה מסוג int (בתרגיל הרביעי נתרגל את הפיצ'ר הזה לעומק). ההשמה - לא חייבים לציין & - יתן את אותו הדבר גם בלי.

אפשר, ולפעמים אפילו הכרחי, להשתמש ב-$typedef$. סינטקס בשתי השקופיות הבאות. לא ניתן להשתמש בסינטקס השלישי בשק' 14 (13), לא יתנו לנו לפרש במבחן דברים כאלה.

דוגמא - חישוב אינטגרל. נרצה לחשב אינטגרל לא לפונקציה אחת ספציפית, אלא לשלוח פונקציה ולקבל את האינטגרל שלה. איך? נבדוק באינטרוולים קטנים.

שולחים את המצביע לפונקציה - זו פשוט כתובת בזיכרון, יותר נכון ויעיל לשלוח כך (במקום את הפונקציה עצמה).

4.3.1 MAPCAR

דוגמא לשימוש בשקופית 18\19 - אם אנחנו רוצים שפונקציה אחת תחזיר סוגים שונים של משתנים - אפשר להגדיר כמו במצגת פשוט את הכתובת של המידע, ללא התחייבות לסוג המידע שיהיה שם.

דוגמא נוספת ספציפית יותר - שק' 20\21.

למה בפונקציה הראשונה אנחנו לא קוראים ל-data מסוג ListStats - כי אחרת נהרוס את הכלליות של הקריאה (כפי שהגדרנו אותה במצגות הקודמות).

דוגמא שלישית - quicksort.

המימוש של ה-compare - שלנו. כך אפשר להשתמש בפונקציית ה-qsort להשוואה בין ints, doubles, ועוד - פשוט נעביר לפונקציה את הקומפרטור המתאים לסוג של המידע אותו אנו מנסים לסדר.

4.3.2 פולימורפיזם בעזרת פוינטרים לפונקציות

דוגמא - בקבצים המצורפים להרצאה shapes.c.
נשים לב:

- בשחרור הצורה - משחררים את המשתנה עבורו השתמשנו ב-malloc, ולאחר מכן משתחררים את ה-shape עצמו.
- מכיוון ואנו מניחים כי בפונקציה circleArea אנחנו הולכים להשתמש רק עבור מעגלים, r מוגדר באמצעות CircleParam (אותו דבר כפי שראינו בדוגמא הקודמת).
- ב-main - הגדרת shapes כמערך של כתובות של צורות, יצירת הצורות והדפסתן - בהגדרה היינו יכולים לבחור shapes[i] ולאחר מכן shapes[j] למשל - יש תמיד לבחור בהצבה הנכונה - זהו שימוש נכון בפולימורפיזם. בעצם אנחנו מקבלים הרבה פוינטרים לאותו המקום. לאחר מכן - הזזה.

4.4 טיפול בשגיאות

שק' 26-25.

4.4.1 שימוש ב-asset

ראינו בעבר - טוב לבדיקות לוגיות, פחות לבדיקות פרמטרים. מחזירה פירוט מאוד מסודר.
באפשרות זו התוכנית מסתיימת, ולכן לא מתאימה למקרים בהם נרצה שהתוכנית תמשיך לרוץ.

4.4.2 enum של קודי שגיאה

נהוג שמספרים שליליים משומשים לכישלון (שק' 32). נהוג להשתמש בקבוע לתיאור הטעות - להתאים את הערך של ה-enum תיאור השגיאה. החיסרון - שימוש כזה מונע החזרה של ערך אחר מהפונקציה. כמו כן - דורש בדיקה של ערך ההחזרה של הפונקציה.

4.4.3 שימוש במשתנה גלובלי

ראינו בתרגול 2 - errno - משתנה גלובלי, הוספה של האפשרויות ע"י הוספת הקובץ errno.h.
דרך אחת להדפסה - שק' 36\38 - בעזרת strerror ע"י העברה של errno.

4.4.4 ספריות נוספות

ניתן לגגל ולמצוא ספריות נוספות שנוצרו על מנת להתמודד עם אקספשנים - ניתן להשתמש בהם בתרגילים.

4.5 Valgrind

החומר - לא במצגת השיעור, אלא במצגת נפרד. כפי שהזכרנו בעבר, זוהי התוכנה שבה משתמשים הבודקים לבדיקות של דליפת זיכרון ושגיאות אחרות. מותקנת במחשבי האוניברסיטה.
שימוש:

- קמפול בעזרת הדגל g-.
- מתקבל פלט ארוך המפרט את הטעויות המחולק לשני חלקים - error summery ו-leak summary.

הטעות בשק' 9 - ניגשים למקום במערך שלא הגדרנו - הגדרנו 10 מקומות במערך, כלומר הכתובת האחרונה היא 9, 10 אינה מוגדרת כלל.
אם אין באגים - עובד מאוד מהר. אחרת - לוקח יותר זמן, אבל עוזר במציאת הבאגים.

4.6 ספריות

ב-C אפשר לקחת אסופה של קבצים ולהפוך אותם לספריה.
שני סוגי ספריות:

- ספריות סטטיות - מקושרות קובץ האקזקיוטבל בזמן קומפילציה.
- ספריות דינמיות (shared) - מועלות בזמן הריצה.

4.6.1 ספריות סטטיות מול ספריות דינמיות?

איך יוצרים ספריות סטטיות? סינטקס בשק' 43:
הפקודה הראשונה ar - כמו tar - ארכיב של קבצי אובייקט.
הפקודה השניה $ranlib$ - יוצר, מעין טבלה של קבצי אובייקט לשימוש ה-$linker$.
היתרונות של ספריות סטטיות:

- עצמאי מהמיקום הנוכחי של הספריות.
- פחות לינקינג בזמן ריצה.

היתרונות של ספריות דינמיות:

- אקזקיוטבל קטנים יותר.
- מספר תהליכים שונים חולקים קוד.
- אין צורך בקומפילציה מחדש לאחר שינוי בספריות.
- אותו אקזקיוטבל יכול לרוץ עם ספרים שונות.
- ניתן להשתמש ב-dll.

כמו כן, ניתן להשתמש ב-makefile - סינטקס בשק' 45.
דוגמא נוספת לשימוש - שק' 46.

4.6.2 שימוש באובייקטים מול ספריות סטטיות בשלב הלינקוג'

אובייקטים:

- כל הקובץ מחובר לאקזקיוטבל.
- אם יש שני מימושים לפונקציה - נקבל טעות.

ספריות:

- יקושרו רק פונקציות שלא נמצאו בקבצי הקובייקט.
- אם יש שני מימושים לפונקציה - המימוש בקובץ האובייקט קודם לזה בספריה.

4.6.3 ספריות דינמיות

סינטקס ליצירה ושימוש - שק' 49.
תמיד שמות של ספריות, בין אם סטטיות ובין אם דינמיות, יש לקרוא בשם המתחיל ב-lib.
נשים לב כי בשימוש הלינקוג' הוא רק לכאורה (שלב דמה) - שהרי בספריות דינמיות הקישורים נעשים רק בזמן הריצה.
חשוב שהספריות יופיעו באיזה משתנה מערכת של מערכת ההפעלה. פירוט בשק' 50. בקורס זה לא נשתמש בספריות דינמיות. עם זאת, יש עוד חומר באתר ובאינטרנט על הנושא.

חלק II

C++

22/08/2011

מהם ההבדלים בין C ל־$C++$?

בשנות השמונים, שטרוסטרופ ביקש לפתח את C.
$C++$ מוסיפה לפונקציונליות של C - מתרגמת את $C++$ ל־C. הקומפיילר מקמפל את $C++$, אז מתרגם ל־C ומקמפל ב־C.
מה שטרוסטרופ רצה כאשר פיתח את $C++$? יתרונות:

- יותר בטוח
- התאמה ל־Object Oriented Design.
- נוספו עוד בדיקות ועוד אקספשנים.
- הוספת טמפלטים.

חסרונות של $C++$:

- שפה מאוד מורכבת לעומת C שהיא מאוד מאוד פשוטה - יש הרבה מאוד מקרי קצה, לוקח הרבה מאוד זמן להתמקצע בה.
- הסינטקס הרבה פעמים יכול להתפרש בשתי דרכים, וקומפיילרים במערכות הפעלה שונות יכולות לקמפל את הקוד בצורות שונות.
- הרבה יותר מורכבת ומסובכת (אך מרשה הרבה יותר דברים).

התקדמות השפה - שק' 10 - פיתוח בשנות השמונים, הגרדא ה"נוכחית" - $C++98$ (יש גרסא נוספת מ־2003 הכוללת שינויים קלים), ולאחרונה אושר תקן לגרסא החדשה - $C++0x$.

4.7 תוכנית ראשונה, ובסיס

שק' 11 - השורה הראשונה מגדירה את הספריה הסטנדרטית של ה־I/O.
שורת הקוד בתוך ה־main - מזין לתוך ה־stdcout את השורה: Hello class! - כלומר מדפיס אותה למסך.
קימפול - ע"י $g++$, בצורה דומה מאוד לקימפול של קוד ב־C (שק' 12). בקורס זה נעבוד עם קבצים בעלי סיומת cpp.

4.7.1 סטרינגים

שק' 13 - הוספת הספריה של פעולות על סטרינגים, הצהרה על סטרינג בשורה הראשונה ב־main.
נקודותיים - מבהירה מהיכן מגיעה הפקודה - לדוגמא, cin מגיעה מ־std שהוא סטרינג. נראה בהמשך שימוש נרחב בנקודותיים וויתור עליהם.
נשים לב לכיוון של החצים - השימוש אינטואיטיבי, "שרשור".
כאן בהדפסות ב־cout אין צורך לכתוב את הסוג של המשתנה - C++ מזהה זאת בצורה אוטומטית.

4.7.2 משתנים בוליאנים

שק' 14 - ב-C++ קיימים משתנים מסוג בוליאני - bool.
המילים false ו-true הן מילים שמורות. משתנה זה הוא בגודל בייט אחד, ועל כן לא חוסכים לעומת במקום לעומת שימוש ב-char.

4.7.3 function overloading

ראינו כי ב-C אין אוברלודינג לפונקציות.
לעומת זאת, ב-$C++$ קיים אוברלודינג יותר מתוחכם - יתקמפל - ראה שק' 16.

משתנים בברירת מחדל שק' 17 - אם לא העברנו משתנה, יתקבל הפרמטר הדיפולטיבי - אחרת, יתקבל המשתנה שהעברנו.
אם נקרא לפונקציה ללא פרמטר - האם התוכנה תקרא לפונקציה ללא הפרמטר, או עם הפרמטר הדיפולטיבי?
על מנת לקבוע זאת, קיימת מערכת חוקים - שק' 18 - הטבלה מימין מציינת מהרמה הראשונה בהיררכיה ומטה:

- דרגה ראשונה - התאמה מלאה.
- דרגה שניה - המרה ללא איבוד מידע.
- דרגה שלישית - המרה עם עיבוד מידע.
- דרגה רביעית - המרה שהוגדרה על ידי המשתמש.

בדוגמא הראשונה הקומפילציה נכשלה, כי היו שתי התאמות באותה רמת עדיפות - האופציה הראשונה והאופציה השלישית.

4.7.4 אופרטורים

ב-$C++$ יש הרבה אופרטורים. ניתן לחשוב עליהם כפונקציות, אותן ניתן לדרוס ולהחליף ב"דברים אחרים".
לכל אופרטור יכולה להיות משמעות שונה, בהתאם לאובייקט עליו הוא מופעל. כלומר - לא בהכרח אופרטור הפועל על שני אובייקטים שונים יפעל באותה הצורה.

4.8 Classes

4.8.1 הקדמה ויצירת קלאסים

פתרון ראשון - שימוש ב-*struct* עם enum - שק' 3-22. במקרה זה אין גמישות להוספת צורה מסוג חדש.
פתרון שני (שק' 5-24) - שימוש בסטראקט ופונקציה. כאן נוכל להוסיף צורות מסוגים חדשים, אך לא נעשות בדיקות, ולכן עלולות להיווצר שגיאות.
ב-C++ יש כלים חדשים ומאגניבים ליצירת קלאסים. הרעיון הראשי - שימוש באובייקטים.
דוגמא פשוטה - שק' 27 - קובץ header שמכריז על המבנה הכללי של הקלאס (הממשק) - המשתנים, הפונקציות, וכו'. בתוך הקלאס נוכל לבחור את הויזיביליות של המשתנים - public או private. דיפולטיבית, כל מה שמוגדר בקלאס הוא private.

ה־header הוא קובץ גלוי לכל משתנה - כלומר, השמות של הרכיבים של הקלאס חשופים, גם אם הם private, אבל היישום שלהם (שנמצא בקובץ אחר) לא יהיה חשוף במקרה והמשתנה הוא private.

יישום של הקלאס בשק' 28 והלאה. נזכור להכליל את קובץ ה־header. שורה ראשונה ב־main - קורא למשתנה ויוצר אותו בעזרת הקונסטרקטור. בהמשך נדבר יותר על קונסטרקטורים. בשיטה הזו של קריאה, **האובייקט נוצר על הסטאק** (לא על הזיכרון).

שק' 29 - הנקודותיים מסמנות כי אנו מממשים את הפונקציה השייכת ל־Counter.

עדיף לעיתים לשמור על ההגדרה ועל המימוש בקבצים נפרדים, גם אם אנחנו לא רואים כעת את היתרון, אם ירצו לשנות את הקוד בהמשך, זה יחסוך זמן.

יצירת קונסטרקטור (שק' 30) - מבחינת הסינטקס, כמו פונקציה, אין ערך החזרה.

קימפול כמו ב־C בעזרת הפקודה g++ - דוגמא בשק' 31.

4.8.2 עוד על הרשאות

public - נגיש למשתמש.

private - לא נגיש מבחוץ, אך המשתמש יכול לראות.

כאמור, באופן דיפולטיבי רכיבים של קלאס יהיו פרטיים.

דוגמאות נוספות לשימוש - שק' 4־33.

4.8.3 מה קרה לסטראקטים?

בסיפיפי קלאסים הם סטראקטים, וההבדל היחיד הוא שבסטראקט הרכיבים האופן דיפולטיבי הם ציבוריים, לעומת קלאס, בה הרכיבים בצורה דיפולטיבית הם פרטיים.

בשימוש - בדוגמא בשק' 35 - אפשר לרשום פשוט את השם MyClass במקום class MyClass (בניגוד להכרזות של סטראקטים).

4.8.4 קונסטרקטורים ודיסטרקטורים

התוכנה משתמשת באופן אוטומטי בקונסטרקטור (ודיסטרקטור) דיפולטיבי. אפשר ליצור קונסטרקטורים יותר מתוחכמים שמקבלים משתנים (דוגמא - שק' 37).

ברגע שכתבנו קונסטקטור חדש - נאבד את הקונסטקטור הדיפולטיבי - ואם ננסה להשתמש בו נקבל שגיאת קומפילציה - דוגמא בשק' 40.

נשים לב להגדרות של אובייקטים המשתמשים באובייקטים, שם לוודא שכל הקונסטרטורים (של אובייקט האב ואובייקט הבן) קיימים - עוד על כך בהמשך.

בסוף השימוש באובייקט יש לפנות המקום - על כן משתמשים בדיסטרקטור. דיסטרקטור מסומן בתחילה בטילדה - ~. דוגמא בשק' 42. אם לא צריך לשחרר שום משאבי זיכרון, התוכנה תשתמש באופן אוטומטי בדיסטרקטור דיפולטיבי.

נשים לב כי מכיוון (כמו בדוגמא במצגת), במידה והקלאסים על הסטאק - ברגע שנגמר קטע הקוד, יקרא הדיסטרקטור בצורה אוטומטית, בין אם מדובר בדיסטרקטור הדיפולטיבי, ובין אם הגדרנו דיסטרקטור חדש.

4.8.5 Interface

דוגמא לשימוש ב־C בשק' 43. בסיפיפי אין צורך להעביר פוינטר - הוא מסתתר באובייקט עצמו - בעת יצירת אובייקט אנו למעשה גם מעבירים פוינטר. נביט בשק' 45 - זוהי דוגמא ראשונה וחביבה ליתרונות של שימוש בסיפיפי לעומת C - הכתיב הרבה יותר פשוט וקצר.

נשים לב כי בכתיב של סיפיפי האובייקט יושב על סטאק, בעוד בכתיב של C האובייקט ישב בזיכרון.

4.8.6 קלאסים והקציית זיכרון, ועל האופרטור New

השורה הראשונה בשק' 46 אינה קוראת לקונסטרקטור - כלומר, הרכיבים אינם מאותחלים. כמו כן, שורת הקוד השניה אינה קוראת לדיסטרקטור!

כאן אנחנו רואים שיטה נוספת לקריאה - על ידי פויינטרים.

בשק' 47 - הקריאה **מקצה זיכרון על הדיסק** (לא על הסטאק!), כלומר האובייקט יהיה זמין גם לאחר סיום הפונקציה, קוראת לקונסטרקטור, ומחזירה פוינטר לאובייקט. המחיקה קוראת לדיסטרקטור ומשחררת הזיכרון. בדומה לקריאה על הסטאק, ניתן גם כאן להשתמש בקונסטרקטורים מסובכים יותר (המקבלים משתנים).

אפשר להשתמש ב-new גם לגבי משתנים פרימיטיביים - int, char וכו' - זהו אופרטור גלובלי. דוגמא לשימוש בשק' 48. אופרטור זה, בניגוד ל-malloc, יכול לזרוק אקספשן במידה ונגמר הזיכרון. דוגמא לשימוש - שק' 2-51.

אופרטור מיוחד למחיקת מערכים - סוגריים מרובעים - דוג' בשק' 53. לא חייבים להשתמש בסוגריים המרובעים, אך עלול לגרום לדליפות זכרון!

4.9 רשימת אתחול

לקונסטרקטורים יש משהו מיוחד בשם "רשימת אתחול". מסומן ע"י ":" יחיד! יכול להופיע רק בקונסטרקטור.

דוגמא פשוטה ראשונה - שק' 56 - האובייקט B מכיל שני אובייקטים מסוג A. בעת יצירת B, יווצרו שני האובייקטים ה"בנים", ורק לאחר מכן ה"אב" - B.

בשלב הכניסה לקונסטקטור של B מניחים שכל הרכיבים של B שאנחנו יוצרים - במקרה זה, שני אובייקטי A, קיימים בזכרון. אבל אם, כמו בשק' 57, יש קונסטרקטור חדש ומורכב המכיל משתנים (שכאמור דורס את הקונסטרקטור הדיפולטיבי), אזי בקריאה ל-B תהיה שגיאת קומפילציה - כי אנחנו לא יודעים איך ליצור את האובייקטים מסוג A, שהרי לא הזנו פרמטרים (בסתירה להנחה שהכל כבר קיים). בעיה זו מובילה אותנו לרשימת אתחול, הפותרת אותה.

כלומר, נוכל להשתמש ברשימת אתחול כאשר שלא קיים קונסטרקטור דיפולטיבי ל-A, או במקרה בו פשוט נרצה להשתמש בקוסנטרקטור שאינו הדיפולטיבי.

יש לשים לב להקפיד כי המשתנים ברשימת האתחול יופיעו לפי הסדר של השימוש באובייקטים. דוגמא לשימוש - שק' 58:

- מאתחלים את רכיבי האובייקט.
- מאתחלים את הקבועים והרפרנסים (נלמד עליהם בהמשך).
- בעתיד: מאתחלים גם את הקלאס האב.

גם אם יש דרך אחרת לאתחל המשתנה ללא רשימת אתחול, מומלץ להשתמש ברשימת אתחול.

בשק' 60 נקבל טעות קומפילציה.

4.9.1 אתחול בעזרת פוינטרים

כיצד? כחלק מהיצירה של B יוצרים פוינטר ל־A. את היצירה של האובייקט עצמו אפשר לבצע מאוחר יותר. אין בעיה עם סדר הפעולות, כי אין בעיית זיכרון - זיכרון של כתובת הוא קבוע. בצורה זו אין צורך לאתחל אל A ברשימת אתחול.

כלומר יש שתי אופציות לאתחול - עדיף לקבוע את הערך של הפונקציה ברשימת אתחול - גם יותר יעיל, ובמקרה של יצירת זיכרון שיטה זו גם יותר בטוחה (אפשרות 2).

4.10 רפרנסים

רפרנס הוא אליאס - שם אלטרנטיבי לאובייקט קיים. בדוגמא בשק' 64 - שינוי של הרפרנס (מצויין ע"י &) משנה גם את הרפרנס וגם את הערך המקורי. גם השורה האחרונה תשנה גם את i וגם את j. שינוי של i לא ישנה את j.

דוגמא להבדלים בין C לסיפיפי בשק' 65:

- סינטקס אינטואיטיבי יותר.
- אין שגיאות אריתמטיקה של פוינטרים.
- משתנים רפרנס הם למעשה פוינטרים קבועים.
- חייבים לאתחל את הרפרנסים בעת ההכרז עליהם (רשימת אתחול), כמו משתנים קבועים.

החסרון של הכתיב מימין - לא ברור שאנחנו משנים משהו חיצוני ללא ההאדר של הפונקציה.

השימוש ברפרנסים גם יותר יעיל - כשאנחנו מעתיקים קלאס יועתק כל הקלאס. במקרה של שימוש ברפרנס אין צורך להעתיק, אנחנו רק יוצרים שם חדש.

מסיבות אלו נוהגים להשתמש הרבה ברפרנסים.

שימוש נחמד ברפרנס - העברה של משתנים לפונקציה (שלא נרצה לשנות). דוגמא - שק' 67.

ניתן גם להחזיר רפרנס מפונקציה למשתנה - שק' 68 כדוגמא לסינטקס. דוגמא נוספת - הרצאה 2 שק' 11. צריך לזכור לוודא כי הכתובת בזיכרון קיימת גם לאחר סיום הקריאה לפונקציה. בסינטקס בשקופית זו, המיין משנה את האובייקט הפנימי "באפר" למרות שהוא פרטי - זוהי דרך לעשות get ו־set באותה הפעולה.

האם זה סותר את העקרון של האינקפסולציה? לא! אם לא היינו רוצים שהמשתמש ישנה את הערכים, פשוט לא היינו מספקים את הפונקציה get.

4.10.1 על קצת על פונקציות

24/8/2011

(שק' 1) ב־C, אם כותבים פונקציה ללא פרמטרים, המשמעות היא שהפונקציה יכולה לקבל כל מספר של פרמטרים לא ידועים. שיטה זו לא עובדת בכתיבה ב־C++!

אם נרצה להדגיש שהפונקציה לא מקבלת פרמטרים, נרשום למשל:

$$\text{int foo}(void)$$

4.10.2 Lvalue, Rvalue

(שק' 13) Lvalue - left value - ערכים מצד שמאל.
Rvalue - ערכים מצד ימין.
באופן כללי - Lvalue הם משתנים ורפרנסים (לא קבועים!), Rvalue - מספרים, דברים זמניים, וכו'.
רפרנס יכול לקבל רק Lvalue. רפרנס קבוע - יכול לקבל גם Lvalue וגם Rvalue. יש גם Rvalue reference בגרסא החדשה של סיפיפי, אבל לא נרחיב על זה בקורס זה.

4.11 קבועים

תזכורת מ-C - אפשר ליצר פוינטר קבוע לאובייקט לא קבוע (דוג' - שק' 16). אפשר גם לייצר פוינטר לא קבוע למשתנה קבוע, וכן פוינטר קבוע למשתנה קבוע.
בסיפיפי יש לנו אפשרות נוספת לקבוע! שק' 17 - אובייקטים ופונקציות קבועות.
הפונקציה foo1 היא קבועה, כלומר, היא לא תשנה את הרכיבים של האובייקט (data members). כמו כן, היא יכולה לפעול על אובייקט קבוע, וגם על אובייקט שאינו קבוע. הפונקציה השניה foo2 לא יכולה לפעול על האובייקט הקבוע A שכן היא עלולה לשנות את הרכיבים של האובייקט.
נשים לב גם לדוגמא בשק' 18 ובחירת הפונקציות שיפעלו - תמיד יבחרו הפונקציות המתאימות ביותר (אם האובייקט קבוע תבחר הפונקציה הקבועה).
העברה בתור רפרנס (שק' 19) חוסכת לנו העתקה - אם נרצה שלא נוכל לשנות אותו מבחוץ, נשתמש במילה const.
עם פוינטרים (שק' 20) המצב קצת שונה - אנחנו יכולים ליצור למשל פונקציה קבועה foo של A שלא יכולה לשנות את הרכיבים של A, אבל יכולה לשנות את הרכיבים של האובייקט שמצביע עליה! ההגנה היא רק ברובד הראשוני, ואינה מחלחלת - ולכן בקריאה לפונקציה foo הערך משתנה.
אם יש לנו רכיב שהוא פוינטר, והפוינטר הזה מכוון למשתנה שהוא רכיב אחר, אז זהו מנגנון על ידו ניתן לעקוף את ה-const. ישנם מקרים בהם ה-const משמש כהגנה נגד עצמנו, ובמקרים אחרים בהם התוכנה תיפול בריצה.
בדוגמא הבאה - שק' 21, לכאורה הפונקציה foo אינה יכולה לשנות את הרכיבים של A. אבל הסינטקס הזה חוקי, כי i הוא רק רפרנס (מצביע למקום אחר ולא הערך עצמו)! כלומר, אי אפשר לשנות את i עצמו, אבל אפשר לשנות באופן ישיר את הערך שאליו מצביע i.

4.12 ירושה

הרעיון העיקרי כבר מוכר לנו מג'אווה -
ראינו שלושה סוגי קשרים בין אובייקטים - אנחנו יכולים לשים אובייקט מסוג A בתוך אובייקט מסוג B - כלומר, B מכיל את A - זהו קשר הכלה. יכולה להיות גם "היכרות" - פוינטרים - אם ל-A יש פוינטר ל-B, אומרים כי ל-A יש היכרות עם B. לעיתים היכרות זו יכולה להיות גם הכלה.
יש קשר נוסף, שנקרא ירושה - נשתמש במושג זה אם A סוג פרטי של B. בסיפיפי אפשר לעשות ירושה מרובה, דבר שאי אפשר לעשות בג'אווה, נראה זאת בהמשך.
בשק' 28 - מתכנת הוא סוג של אדם, עם תכונות ספציפיות. איך אומרים "סוג של"?

```
class Programmer : public Person
```

כלומר, מתכנת הוא גם אדם. מבחינת הזיכרון, כשניצור מתכנת, החלק הראשון בזיכרון יהיה החלק של התכונות של אדם.

למתכנת יש פונקציית outputDetails אשר שונה מזו שהוגדרה לאדם.

4.12.1 כיצד ניצור ונמחוק?

דבר ראשון צריך להיווצר החלק של אדם (שק' 29). ניתן להשתמש בקונסטרקטור דיפולטיבי או בקונסטרקטור אחר - יש להקפיד שהחלק הראשון בקוד יהיה זה!
במחיקה - גם הסדר חשוב - כלומר (שק' 30) בסדר ההפוך.

4.12.2 override

המתכנת יכול להשתמש בכל המתודות של אדם (שק' 31).
בשק' 33 - את השם ואת ה-id הגדרנו בתור ערכים פרטיים. אנחנו רוצים לגשת לערכים אלו. כיצד נעשה זאת?
בעזרת protected - כל הערכים הללו לא ניגישים מבחוץ, אבל נגישים עבור האובייקטים היורשים. אזי, אם נגדיר את השם וה-id בתור protected, אזי נוכל להדפיס אותם בפונקציה outputDetails. דברים שהם private כאמור לא נגישים גם על ידי האובייקטים היורשים. אולם, מתוך מתכנת ניתן לגשת לרכיבי אדם.
הדרך המקובלת יותר לבצע ירושה היא על ידי public.
דוגמא לשימוש - שק' 35 - private/protected הם ברמת הקלאס ולא ברמת האינסטנס!
המקרה של הקונסטרקטור הוא קצת מיוחד - בשק' 36 נקבל שגיאה - הקונסטרקטור הוא פרוטקטד, ולכן לא ניתן ליצור את A מבחוץ. עם זאת, מתוך B אפשר להשתמש בקונסטרקטור של A (כי הוא יורש אותו), ולכן הקוד יתקמפל. הבעיה נוצרת כאשר נרצה ליצור אובייקט מסוג B - new הוא אופרטור מיוחד - הוא ברמה הגלובלית - לכן כאשר נעשה new זה "כאילו" אנו רוצים ליצור את A מבחוץ (מתוך המיין), ולכן נקבל שגיאה ולא נוכל ליצור את B.

4.12.3 Virtual Classes & Polymorphism

נרצה ליישם מערכת גרפית - נרצה שיהיו לנו רשימות של צורת, לכל צורה צריכה להיות היכולת לצייר את עצמה, לחשב את גודלה, וכו'.
הבעיה הראשית בפתרון הראשון (שק' 39) - אם נרצה להוסיף צורה נוספת זה יהיה "לא טבעי".
אפשרות שניה (שק' 41) - שימוש בשני קלאסים שונים. החסרון - אלו טיפוסים נפרדים, ואי אפשר להתייחס אליהם באופן כללי, אין להם דבר במשותף. כמו כן, יכול להיווצר לנו שיכפול קוד.
אפשרות שלישית (שק' 43) - ירושה. קלאס כללי לצורה, וסוגים שונים של צורות בקלאסים יורשים. כעת, נוכל לעשות מערך של צורות (שק' 44), ולשים במערך גם מעגלים וגם ריבועים. אבל במקרה זה, אם נרצה להשתמש בפונקציה Draw מתוך המערך, הפונקציה שתתבצע היא זו של shape, לא של הצורה הספציפית!
איך נפתור זאת? לפני הפתרון - דוגמא נוספת (שק' 46) - הפונקציה שתקרא של bar היא זו שיותר ספציפית - במקרה זה, מבין ההמימוש המקורי של Base וזה של ה-Derived תבחר האופציה השניה.
כיצד הקומפיילר בוחר לאיזו מתודה לקרוא? **פתרון סטטי**:
ההחלטה מתבססת על הסוג של המשתנה - לא הסוג של האובייקט! הקומפיילר מוצא את המימוש הספציפי ביותר וקורא לו.
האופציה השלישית - פתרון סטטי עבור הדוגמא הראשונה (צורות) מופיע בשק' 49.

עם זאת, נרצה **פתרון דינמי** (האופציה הרביעית) - כלומר, לא נרצה לקבוע איזו פונקציה תבחר מראש, אלא נראה שכל פעם יבדקו האופציות ותבחר האופציה הנכונה תבחר.

ההחלטה הדינמית מבוססת על הסוג של האובייקט, ומתבצעת בזמן ריצה (כמו ג'אווה). כיצד נממש בסיפיפי? שק' 51 - בעזרת המילה השמורה virtual, המסמנת כי המתודה יכולה "להדרס" באופן דינמי. נדבר בהמשך על התהליך שמתרחש בזמן הקומפילציה.