קורס קיץ - 67322 $C++$

13 בספטמבר 2012

מרצה: אווה קלמן

איני לוקחת אחריות על מה שכתוב כאן, so tread lightly
אין המרצה או המתרגלים קשור לסיכום זה בשום דרך.
הערות יתקבלו בברכה - noga.rotman@gmail.com. אהבתם? יש עוד!
http://bit.ly/integrali

הערה: סיכום זה מכיל את "זרם התודעה" של המרצה, על כן כדאי ומומלץ להעזר ברשימות אלו יחד עם המצגות הרשמיות של הקורס (עד שאספיק להעתיק גם את השקופיות עצמן - אך איני מבטיחה שאגיע לזה).

תוכן עניינים

I שיעורים **4**

1 השוואה ל-C 4
1.1 יתרונות וחסרונות 4
1.2 קימפול 5
1.3 הבדלים בסוגים 5
2 Classes 6
2.1 סינטקס 6
2.2 קונסטרקטורים ודיסטרקטורים 7
2.3 קלאסים והקצאת זיכרון, ועל האופרטור New 8
2.4 רשימת אתחול - Initialization List 8
2.5 רפרנסים 9
2.6 תזכורת - משתנים קונסטיים, אובייקטים ופונקציות 10
2.7 nested class 10
2.8 ירושה 12
2.9 Virtual Classes and Polymorphism 13
3 סוגי פונקציות\קלאסים\משתנים שונים 15
3.1 Friend Functions, Friend Class 15
3.2 Static Class Members 15
3.3 Inline 16
4 העתקה ו-Casting 17
4.1 העתקה 17
4.2 Casting 19
5 Templates 20
5.1 הקדמה, מוטיבציה 20
5.2 סינטקס 21
5.3 פונקציות טמפלייטיות 21
5.4 קלאסים טמפלייטיים 23
5.5 איטרטורים 23
6 כמה דברים קטנים 25
7 ירושה מרובה ו-virtual base class 26
7.1 ירושה מרובה 26
7.2 virtual base class 26
8 אקספשנים 27
8.1 סינטקס 27
8.2 סוגי אקספשנים 28
8.3 כאשר לא תופסים אקספשן שנזרקה 28
9 המרות ו-RTTI 29

9.1 סוגי המרות 29
10 עוד כמה נושאים קטנים 30
11 ניהול זיכרון - פוינטרים חכמים 32
11.1 שתי אסטרטגיות ראשונות שאנחנו כבר מכירים 32
11.2 אסטרטגיה שלישית -מצביעים חכמים 32
11.3 אסטרטגיה רביעית - Reference Counting 33
11.4 במה להשתמש מתי? 33
12 STL 34
12.1 על איטרטורים 34
12.2 אלגוריתמים 35
12.3 אדפטורים 36
13 R-Value Reference 36

II תרגולים **38**

הערה 0.1 התרגילים הם חלק מחומר הקורס!

0.0.1 טיפים לתכנות ודיבוג ב־C++

- ב־{{ ב־cpp רצים הדיסטרקטורים, אם התוכנה "תעוף" רב הסיכויים שהיא תעוף שם.
- אם רוצים לדבג איזה קונסטרקטורים ודיסטרקטורים רצים, אפשר להוסיף להם הדפסות. לעיתים זה ישנה את התנהגות התוכנית, אך נדבר על זה בהמשך, בכל מקרה לרב זה אמור לעזור.
- "לא להסתבך יותר מידי" עם מצביעים ורפרנסים - להשתדל שהקוד יהיה פשוט וקריא.
- לשים לב שלכל new יש delete.[1]

0.0.2 נקודות מהמבחן ב־C

- שקופית 3 מה־28/8:
  - שאלה ראשונה - זה תמיד נכון! arr הוא פשוט להגיד הכתובת של תחילת המערך, והחזרת הכתובת של התא האפס - &arr[0] הוא גם אותו הדבר. נשים לב כי sizeof של שני הביטויים האלה יתן ערכים שונים: הראשון את הגודל של המערך (5 אינטים), והשני גודל של פוינטר.
  - שאלה שניה - גם תמיד נכון! מאותה הסיבה.
  - הקוד באמצע המצגת - כולם אותו דבר, שכן במעבר לפונקציה ב־C, לא משנה מה כותבים, מתעלמים לגמרי למה שכתוב, ומתייחסים רק לסוג עצמו.
  - השורה האחרונה - arr+24 (גודל של אינט כפול 3). כאשר עושים arr+24, מתקדמים 24 אינטים (הגודל שה־arr מצביע עליו), לא 24 ביטים, לכן הערך של arr+sizeof(int)*3=192.

חלק I
שיעורים

1 השוואה ל־C

1.1 יתרונות וחסרונות

9/8/2012

"למה c++ הרבה יותר כיף מ־c?"

- Object Oriented design.
- תכנות גנרי - טמפלטים מאפשרים שימוש חוזר בקוד.
- מערכת סוגים הדוקה יותר (למשל ארגיומנטים לפונקציות).
- בדיקות בעת ריצה ושליטה בזיכרון.

[1] למערך עם סוגריים, לאובייקט רגיל, וכו'.

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).

1.2 קימפול

בשקופית 12 - תוכנית ראשונה ב־C++, קימפולה - בשקופית 13:

```
g++ -Wall hello.cpp -hello
```

1.3 הבדלים בסוגים

1.3.1 סטרינגים

שק' 14 - ב־C++ נוספת ספריה 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 עליהן.

נשים לב לכיוון של החצים - השימוש אינטואיטיבי, "שרשור".

כאן בהדפסות ב־cout אין צורך לכתוב את הסוג של המשתנה - C++ מזהה זאת בצורה אוטומטית (ועוד על overloading של האופרטור הזה - בהמשך).

1.3.2 משתנים בוליאנים

שק' 15 - ב־C++ קיימים משתנים מסוג בוליאני - bool (בניגוד לסטרינג, אין צורך להוסיף דבר ב־ include - מה שמופיע הוא עבור ה־cout).

המילים false ו־true הן מילים שמורות. משתנה זה הוא בגודל בייט אחד, ועל כן לא חוסכים לעומת במקום לעומת שימוש ב־char.

1.3.3 function overloading

ראינו כי ב־C אין אוברלודינג לפונקציות - כלומר, אי אפשר להשתמש בשם של פונקציה פעמיים עם ארגיומנטים שונים.

לעומת זאת, ב־$C++$ קיים אוברלודינג יותר מתוחכם: ב־C שקופית 16 לא תתקמפל, ואילו שקופית 17 ב־C++ תתקמפל.

משתנים בברירת מחדל שק' 18 - 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?

ההחלטה מתבצעת באופן הבא:

- נמצאות כל הפונקציות עם אותו השם.
- מתוך הקבוצה מעלה, מצטמצמים לפונקציות בעלות המספר הנכון של הארגיומנטים.
- מתוך הקבוצה השניה, מצטמצמים לקבוצה בעלת הארגיומנטים המתאימים ביותר.

אם הקבוצה האחרונה מכילה רק פונקציה אחת - משתמשים בה, אחרת תהיה טעות קומפילציה.
על מנת לקבוע את הקבוצות, קיימת מערכת חוקים - שק' 20 - הטבלה מימין מציינת מהרמה הראשונה בהיררכיה ומטה:

- דרגה ראשונה - התאמה מלאה ללא צורך בהמרה, מערך לפוינטר, לא קונסט לקונסט.
- דרגה שניה - המרה ללא איבוד מידע - למשל float לדאבל או צ'אר לאינט.
- דרגה שלישית - המרה עם עיבוד מידע - למשל דאבל ל-float, float לאינט.
- דרגה רביעית - המרה שהוגדרה על ידי המשתמש.

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

1.3.4 אופרטורים

ב-$C++$ יש הרבה אופרטורים, למשל:

$$+ \; - \; \& \; * \; \gg \; \ll \; [] \; ()$$

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.

Classes 2

2.0.5 מוטיבציה

נרצה לממש חבילת גרפיקה, המטפלת בציור של מספר צורות, ולתחזק רשימה של הצורות.
פתרון ראשון - שימוש ב-*struct* עם enum - שק' 24. קל לישם, אך במקרה זה אין גמישות להוספת צורה מסוג חדש.
פתרון שני - שק' 26 - 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.

2.1 סינטקס

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.

אחת הפונקציות - constructor, בונה את האובייקט - עוד עליו בהמשך.

יישום של הקלאס בשק' 30 והלאה. נזכור לכלול את קובץ ה־header, וכן את cstdio, הספריה הכוללת את פונקציית ההדפסה printf. 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), לכן הזיכרון ישתחרר אוטומטית בסוף ה־scope.

שק' 31 - 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' 32) - מבחינת הסינטקס כמו פונקציה, ללא ערך החזרה.

קימפול כמו ב־C בעזרת הפקודה g++ - דוגמא בשק' 33: יוצרים קובץ o. נפרד לכל אחד מקבצי ה־cpp. ולבסוף מקמפלים אותם לתוכנית.

2.1.1 הרשאות

public - נגיש למשתמש.

private - לא נגיש מבחוץ, אך המשתמש יכול לראות.

כאמור, באופן דיפולטיבי רכיבים של קלאס יהיו פרטיים.

דוגמאות נוספות לשימוש - שק' 6־35.

2.1.2 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' 37 - אפשר לרשום פשוט את השם MyClass במקום class MyClass (בניגוד להכרזות של סטראקטים).

אין צורך להשתמש ב־typedef של קלאס וסטראקט בעת קריאה לקונסטרקטור.

2.2 קונסטרקטורים ודיסטרקטורים

התוכנה משתמשת באופן אוטומטי בקונסטרקטור (ודיסטרקטור) דיפולטיבי. אפשר ליצור קונסטרקטורים יותר מתוחכמים שמקבלים משתנים (דוגמא - שק' 39).

ברגע שכתבנו קונסטקטור חדש, נאבד את הקונסטקטור הדיפולטיבי אם לא נכריז עליו - ואם ננסה להשתמש בו נקבל שגיאת קומפילציה - דוגמא בשק' 42.

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 את המקום שהוקצה - על כן משתמשים בדיסטרקטור. כמו כן, משתמשים בו למשל למטרת נוטיפיקציה לאובייקטים אחרים. דיסטרקטור מסומן בתחילה בטילדה - ~. דוגמא בשק' 44. 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.

2.3 קלאסים והקצאת זיכרון, ועל האופרטור New

2.3.1 Interface

דוגמא לשימוש ב־C בשק' 45. בסיפיפי אין צורך להעביר פוינטר בעת הפעלת פונקציה על האובייקט - הוא מסתתר באובייקט עצמו - בעת יצירת אובייקט אנו למעשה גם מעבירים פוינטר. נביט בשק' 46 - 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.

2.3.2 new

השורה הראשונה בשק' 47 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' 49 - השימוש במילה new **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. ניתן לעשות לו overloading. דוגמא לשימוש בשק' 51. אופרטור זה, בניגוד ל־malloc, יכול לזרוק אקספשן במידה ונגמר הזיכרון. דוגמא לשימוש - שק' 2־51. עוד על זריקות אקספשנים בהמשך.

צריך להשתמש גם ב־new לצורך הקצאת מערכים - דוגמא בשקופית 53 והלאה להקצאה ושחרור. אופרטור מיוחד למחיקת מערכים - סוגריים מרובעים. לא חייבים להשתמש בסוגריים המרובעים, אך בלעדיהם עלולה להיווצר דליפת זיכרון!

2.4 רשימת אתחול - Initialization List

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

דוגמא פשוטה ראשונה - שק' 58 - 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' 59, 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' 60:

- 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.

2.4.1 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).

2.5 רפרנסים

רפרנס הוא אליאס - שם אלטרנטיבי לאובייקט קיים.

בדוגמא בשק' 66 - שינוי של הרפרנס (מצויין ע"י &) משנה גם את הרפרנס וגם את הערך המקורי. השורה השניה תשנה גם את i וגם את j. שינוי של i לא ישנה את j. השורה ref=j תגרום לערך ש־ref מסתכל עליו להיות 5, אבל אי אפשר לשנות המיקום אליו הוא מצביע - אי אפשר לשנות את הרפרנס לאחר היצירה. דוגמא לכך: בשורה שלישית יודפס 5,7,5 - j השתנה, אולם ref לא - למרות שה־ref השתנה, הוא השתנה לערך, לא למצביע. i גם כן לא השתנה.

דוגמא להבדלים בין C לסיפיפי בשק' 67:

- 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' 68:

- העברה var0* מעבירה פוינטר.
- העברה &var1 מעבירה by reference.
- העברה var2 מעבירה by value.

lvalue - משתנים, רפרנסים (לא קונסטים!),....

rvalue - רפרנסים למשתנים קבועים, מספרים, ערכים זמניים...

ניתן גם להחזיר רפרנס מפונקציה למשתנה - שק' 70 כדוגמא לסינטקס - כך מונעים העתקה של אובייקטים, ולא מרשים שינוי של הערך שלהם.

2.6 תזכורת - משתנים קונסטיים, אובייקטים ופונקציות

דוגמא בשקופית 73 - הכרזנו על משתנה קונסטי, הקונסט על הימין, אם אין על השמאל - אז אסור לנו לשנות את p1, אבל מותר לשנות את מה שהוא מצביע עליו.

מצד שני, אם הקונסט משמאל הוא מגן על האינט עצמו - כך שה"חוקים" הפוכים: אפשר לשנות את p2, אבל אסור לשנות על מה שהוא מצביע עליו. אפשר להכריח קונסט גם וגם.

אובייקטים ופונקציות קונסטיות - אסור להפעיל על אובייקט קונסטי פונקציה שאיננה קונסטית. כמו כן, אפשר להפעיל על אובייקט שאינו קונסטי פונקציה קונסטית, אבל אם יש שתיים בעלות אותו שם ואותה חתימה - אחת קונסטית ושניה לא, אזי אם נפעיל על אובייקט לא קונסטי פונקציה שכזו, תיקרא הפונקציה שאיננה קונסטית - כפי שניתן לראות בשקופית 75.

טעות קומפילציה - שקופית 77.

מקרי קצה - שקופית 78.

אתחול - שקופית 79: איברים קונסטיים מאותחלים פעם אחת ויחידה בתוך רשימת האיתחול.

המילה השמורה this היא הכתובת של האינסטנס עבורו מתבצעת הפונקציה. 21/8/2012

משהו קטן מ־C

לרגע - על משהו שהרבה נפלו בו בבחינה ב־C (שקופית 1 שיעור 2):

פונקציה המאותחלת עם אותו המערך פעמיים. האם תקין? אם לא תקין, מה לא תקין?

טיעון: תקין, כי בכניסה לפונקציה המערכים מועתקים by value. הערך בתא 0 באמת מעודכן להיות 1. צריך לשים לב מה מועתק by value: הכתובת היא מקומית בתוך הפונקציה, אבל הזיכרון עליו הפונקציה מצביעה כן משתנה.

ראינו ש־new מקצה זיכרון על הערימה וקורא לקונסטרקטור, delete משחרר את הזיכרון וקורא לדיסטרקטור.

אם קוראים בלי new - מוקצה זיכרון על הסטאק, אולם לאחר שקטע הקוד מסתיים (ה־scope), נקרא הדיסטרקטור אוטומטית והזיכרון משתחרר.

MIL - רשימת איתחול. ראינו שמשתנים מסוג const ו־*& חייבים לאתחל ברשימת אתחול.

בדר"כ לא קוראים לדיסטרקטור באופן ברור - או שעושים delete, או שהוא נקרא אוטומטית בסוף ה־scope. על כן, הדיסטרקטור גם לא מקבל ערכים - אין להם חשיבות.

רפרנסים - הכי שימושיים להעברת אובייקטים גדולים ממקום למקום. אז לא נרצה לקבל by value, אלא by reference.

2.7 nested class

nested class הוא קלאס פנימי שהוא פרטי. שימוש מבפנים - כאילו הוגדר גלובלי.

דוגמא - שקופית 9/8: גישה ל־node מבחוץ לא תתקמפל.

איך נשתמש בו מבחוץ? הדוגמא בשקופית 10 אינה נכונה, יש לעשות בה שני שינויים:

- הצעה ראשונה: יצירת אובייקט מסוג IntStack ואז לגשת דרכו. הצעה זו אינה נכונה!
- הצעה שניה: להפוך את node לפומבי, ולהוסיף את האופרטור IntStack::Node בפניה אליו.

איך מממשים פונקציות?

בתוך הקובץ cpp של IntStack - למשל:

```
void IntStack::Node::func(void)
```

2.7.1 הקבלה\אי הקבלה לג'אווה?

ב־cpp משתמשים ל:

- החבאת שמות
- יצירת סדר
- אין קשר בזמן ריצה בין המחלקות הפנימיות למחלקה החיצונית
- יותר כמו design pattern

בג'אווה לעומת זאת, יש קשר בזמן ריצה בין המחלקות הפנימיות למחלקות החיצוניות.
ב־cpp ניתן לחשוב על nested classes כמחלקות פנימיות סטטיות בג'אווה.

2.7.2 Namespaces

זהו מכניזם לקיבוץ לוגי של הגדרות והכרזות תחת אותו איזור הגדרתי. ניתן לגשת לתוכן של namespace התחום בעזרת {}. דוגמא ליצירת חדש ־ שקופית 14.
ראינו כי בעת קריאה לפונקציה צריך להשתמש ב־scope resolution operator (כלומר ::), או ב־using namespace.
השיטה השניה כבר מוכרת לנו:

```
using namespace std;
string answer="..";
cout«answer«endl;
```

ללא std::cout וכו'! אפשר לעשות גם לחלק מ־std, למשל std::cout ספציפית. דוגמא לשימוש ־ שקופית17.
מתי לא כדאי להשתמש בזה? נקבל שגיאת קומפילציה אם מימשנו פונקציית cout באופן עצמאי בצורה אחרת ־ הקומפיילר לא ידע במה להשתמש.
מה ההבדל בין Namespaces ל־Nested classes?

- אין הבדל בזמן הריצה.
- אפשר להגדיר את אותו ה־namespace בקבצים רבים (std למשל).
- משתמשים ב־nested classes עבור מידע פרטי, ב־namespaces לקשרים לוגיים, למשל ספריות מתמטיות.

עוד Namespaces:

- Namespaces אנונימיים ־ (שקופית 19) רואים רק מתוך הקובץ עצמו.
- namespaces מקוננים.

2.8 ירושה

הרעיונות הראשיים מוכרים לנו כבר מג'אווה.
ל־cpp אין interfaces, אבל אפשר להשתמש במספר ירושות. למשל - ירושת יהלום.
לעת עתה, לא נדון בבעיות שעלולות להווצר.
דוגמא ראשונה מתחילה בשקופית 22.

2.8.1 הסינטקס של ירושה

protected גורם לאיברים של קלאס האב להיות נגישים מהתת־קלאסים שלו בלבד - לא תהיה גישה מבחוץ. דוגמא בשקופית 24 והלאה:
programmer יורש מ־person: מכיל את כל מה ש־person מכיל, ומשתנים נוספים שהוגדרו עבורו.
אם המשתנים ב־person היו private, לא היינו יכולים לשנות אותם - על כן הצורך בשימוש ב־protected.
שקופית 28 - שימוש ברשימת אתחול, הקונסטרקטור עצמו נותר ריק - מימוש זה נחשב אלגנטי.

2.8.2 סוגי ירושות

לקלאס הבסיס יש גם access modifier, המציין כיצד מתייחסים לאיברים של הבסיס:

class Programmer : public/private/protected Person

- לרב משתמשים ב־public.
- אם הירושה היא private, צריך להביא את המשתנים על ידי פונקציות מתאימות - מתוך programmer יש גישה, משתמשים של programmer **לא** יוכלו לגשת ישירות.
- ירושת protected אומרת שכל מה שהיה private נשאר private, public עובר ל־protected, protected נשאר protected.

נשים לב - הירושה יכולה רק להחמיר את הגישה, לא להקל עליה!
נקודה חשובה - כל הדברים האלו הם ברמת ה־class ולא ברמת ה־instance - דוגמא בשקופית 31־2: person יכול לשנות שדות פרטיים של person בעזרת פונקציה ציבורית מתאימה.

2.8.3 רשימת אתחול לקלאס האב

שקופית 33:
לקלאס A הקונסטרקטור הוא protected (בהמשך נדבר למה כתיבה שכזו משמשת).
B יורש מ־A. למה הקוד ב־main לא יעבוד?

- השורה הראשונה מנסה ליצור את A על ידי קריאה לקונסטרקטור - אבל הוא protected, אז אין גישה, ונקבל שגיאת קומפילציה.
- new הוא אופרטור גלובלי, לכן הוא לא יכול לגשת לקונסטרקטור של A, ולכן גם השורה השניה טעות - אפשר ליצור את B, אבל לא נצליח לעשות new A.

- ה־delete של B בעייתי - גם אם היינו מצליחים ליצור (דרך כתובת ממקום אחר), התוכנה היתה נופלת שם.
- אפשר היה ליצור ב־B את A ע"י ;A a.

איך אפשר לתקן? לשנות protected ל־public. ליצור את A לא בעזרת new אלא על הסטאק - A a וכו', לא ברור אם יעבוד (אווה תבדוק).
הנקודה העקרונית להבין - protected מיועד לקלאס היורש, ויש להבין מה שייך לקלאס היורש, ומי גלובלי.

2.8.4 סדר C-tors ו־D-tors

בשיעור שעבר דיברנו על כך שהקונסטרקטורים ה"קטנים" מבוצעים לפני הקונסטרקטורים ה"גדולים".
בדומה - קודם מבוצע הקונסטרקטור של ה־base, לאחר מכן ה־members נוצרים, ולבסוף הקונסטרקטור של הקלאס עצמו מבוצע.
הסדר של הדיסטרקטורים - הפוך.

הערה 2.1 הסדר של ה־MIL עצמו לא משנה.

בדוגמא, מה יהיה סדר הקריאות? a,b,c,c,b,a: יוצרים קלאס שיורשת משתי קלאסים אחרים, אז קודם כל ה"עמוקה" ביותר נוצרת, אז השניה, וכו', עד זו שביקשנו.
שחרור - בסדר הפוך.

2.8.5 Overloading

נחזור לדוגמאת ה־person (שקופית 39). הפוקנציה המסומנת מקבלת איזה סטרים וכותבת אליו את הפרטים. עושים אוברלאודינג מדויק ל־programmer. כעת, יש שתי אופציות:

- אופציה ראשונה: המתודה של ה־programmer תדרוס את המתודה ב־person.
- אופציה שניה: לקרוא למתודה של person, ולשרשר את הפרט הנוסף שרצינו להדפיס.

2.9 Virtual Classes and Polymorphism

דיברנו על ירושה. נרצה לראות כיצד להשתמש בה ובמושג שכבר אנו מכירים - פולימורפיזם.
נחזור לדוגמא מהשיעור הקודם - נרצה ליישם מערכת לציור צורות. בפתרון הראשון בשקופית 45 יש enum לצורות השונות. דיברנו על כך שמדובר במעטפת אבל זה לא פותר לנו את כל הבעיות.
פתרון שני - ליצור קלאסים שונים. אבל בדרך שיצרנו, אין התחייבות לייצר את הקלאסים האלו עם פונקציות מסויימות, וכן הן מסוגים שונים, אז לא ניתן לשתף קוד ביניהן.
פתרון שלישי - היררכיה: הצורות כולן ירשו מקלאס אב shape.
דוגמא בשקופית 50 - יצירת מערך של צורות, ובתוך כל תא צורה אחרת.
מה יקרא אם נעשה myShapes[0]->Draw();? תופעל פונקציית הציור של Shape - אבל מן הסתם זה לא מה שרצינו!
כאשר עובדים עם הפוינטר, ה־Draw נקרא על ידי הקומפיילר לפי סוג הפוינטר. עוד דוגמא שממחישה את זה - שקופית 52: באובייקטים, עובד כמו שמצפים: אם אין מימוש, תמומש הפונקציה של האבא, אם ממומשת - של הילד המדובר.

2.9.1 Static Resolution, Dynamic Resolution ומתודות וירטואליות

כיצד הקומפיילר מחליט לאיזו מתודה לקרוא?

- רזולוציה סטטית - מתבססת על סוג המשתנה, לא על סוג האובייקט: הקומפיילר מחפש את המימוש הכי ספציפי של המתודה, וקורא לה. למשל, שקופית 55. אם זה היה המצב, הפולימורפיזם היה מאוד לא יעיל. מה כן עושים?
- רזולוציה דינמית - מתבססת על סוג האובייקט, ההחלטה מתבצעת בזמן ריצה. ממומש בעזרת המילה החדשה virtual, שתפקידה לציין כי ניתן לעשות override באופן דינמי - יש לחפש את המתודה הכי רלוונטית.

פתרון רביעי לבעיה אם כך ישתמש ברזולוציה דינמית -
נוסיף virtual ל־Draw (שקופית 59), ואז הפונקציה המבוקשת תפעל כפי שרצינו.
כמה דגשים חשובים:

- אם מוגדרת מתודה וירטואלית, היא תשאר וירטואלית לאורך כל הירושה (ילדים, נכדים וכו'), ועל כן תתקבל ההתנהגות המבוקשת. זוהי החלטה גורלית, המבזבזת קצת מקום וזמן (נדון בכך בהמשך).
  - למען קריאות של הקוד, למרות שלא חייבים, אם ממשים שוב את המתודה, יש לכתוב virtual.
  - אין בעיה לממש בנכד ולא בילד.
- המילה virtual לא מחייבת מימוש בקלאסים נוספים.

איך נראית הירושה בזיכרון? דוגמא בשקופית 62 - לאחר איברי ה־base משורשרים ה־members החדשים המיוחדים לילד.
הבעיה: אפשר להתייחס לילד כאב. איך הקומפיילר ידע אם כך לאיזו פונקציה מתכוונים?
אפשר היה לשמור פוינטר בכל אובייקט "לאיזו פונקציה מתכוונים". זה פתרון אפשרי אחד, וזה קצת דומה למה שעשינו ב־C. אבל זה מכריח לשמור פוינטר לכל instance, וזה מבזבז מקום.
הפתרון - virtual table נוצרת בכל יצירת instance (שכן בעת יצירה יודעים מאיזה סוג האובייקט), המצביעה לפונקציות הוירטואליות.
בדוגמא בשקופית 66, רואים כי מכיוון ואין מימוש של f2 בקלאס הבן, השורה בטבלה עבור אובייקט בן b מצביעה למימוש בקלאס האב. מכיוון ו־f4 לא וירטואלית, אין לה שורה בטבלה כלל. שאר הפונקציות הן וירטואליות, ולכן מצביעות למימוש שונה מזה של טבלת A.
בצורה זו בזבוז המקום הוא קטן יותר - לכל סוג יש טבלה, ולכל instance פשוט מצביעים למקום הרלוונטי. הבעיה: צריך לעבור בין שני פוינטרים כדי להגיע לפונקציה המבוקשת. הקביעה אם כך במקרה הזה היא בזמן ריצה, לא בזמן קומפילציה.
על כן לא שמים virtual סתם, אבל משתמשים בה לביצוע פולימורפיזם אמיתי.

2.9.2 על דיסטרקטורים

בעיה בשקופית 68 - נרצה שהדיסטרקטור של הבן יקרא.
אז, כמו כל פונקציה אחרת - נוסיף virtual לדיסטרקטור גם כן.

2.9.3 ירושה ו־Interfaces

בדוגמא של הצורות למשל, כל המתודות של shape הן וירטואליות - אנו למעשה לא רוצים "להתעסק" לעולם עם אובייקט מסוג shape, רק עם מחלקות היורשים ממנו. איך נכפה זאת?

שימוש ב־assert(false) יעיל אך "אלים מידי".

במקום זאת נרצה דרך ציין כי Shape::Draw() לא קיים. כיצד? הכרזה על מתודה כ־pure virtual:

```
virtual void Draw()=0
```

במקרה ויש לפחות מתודה אחת pure virtual - הקלאס נהיית אבסטרקטית pure virtual class, ואז לא ניתן ליצור אינסטנסים שלה.

אנו מקבלים "ממש interface" - מכריח מימוש של הפונקציות.

אפשר שהבן לא יממש, אך יהיה לו בן (נכד של Shape בדוגמאות שלנו) שכן יממש, ואז אפשר ליצור את הנכד, אך לא את הבן.

דוגמאות למה מותר ולמה אסור - שקופית 74.

2.9.4 טיפים לשימוש במתודות וירטואליות

- אם יש מתודות וירטואליות בקלאס, **תמיד נכריז על הדיסטרקטור כוירטואלי**[2].
- אף פעם לא לקרוא למתודות וירטואליות במהלך יצירה או destruction.
- לעיתים יש צורך ב־factory: קלאס גדול היוצר קלאסים קטנים.
- משתמשים ב־pure vitrual ע"מ להגדיר interfaces.
- יש להזהר בעת שימוש בירושה: לוודא כי זהו הפתרון המתאים (יחס "is a").

3 סוגי פונקציות\קלאסים\משתנים שונים

3.1 Friend Functions, Friend Class

מתוך התרגול השלישי - פונקציה שהיא friend של קלאס היא פונקציה שאינה מתודה של אותה הקלאס, אך יש לה גישה ל־data members הפרטיים וה־protected של קלאס זו.

סינטקס - הצהרה בקלאס בתוך ה־class scope (פונקציה לא יכולה להכריז זאת על עצמה - אחרת לא יכולנו להגן על שום מידע).

קלאס גם יכולה להרשות לקלאסים אחרים לגשת ל־private data members שלה. יחס זה הוא חד צדדי. דוגמא - שקופית 24 תרגול 3.

3.2 Static Class Members

23/8/2012

כמו שאנו מכירים מג'אווה, הם שייכים לקלאס ולא לאינסטנס ספציפי. פונקציות סטטיות יכולות להשתמש רק בממברים סטטיים כי הן לא תלויות אינסטנס (קשורות לקלאס עצמו, אפשר לקרוא להן ללא יצירת אינסטנס מסויים).

[2] אחרת, מנסיון כואב, תקבלו דליפות זיכרון שלא תבינו מאיפה הן באות.

ב־c++, בניגוד לג'אווה, הצהרה והגדרה במקומות שונים: בקובץ h מצהירים, ב־cpp מגדירים ומאתחלים באיזור הגלובלי. בקורס זה נאתחל אותו מיד בהגדרה (למרות שלא חייבים).

בקובץ h אי אפשר לאתחל משתנה (לא יעבור קומפילציה), יש בו רק הצהרות. עם זאת, במקרה של משתנה סטטי וקונסטי, והוא רק משתנה של השפה (טיפוס בסיסי מספרי, בלי שקונסטרקטורים רצים - אי אפשר למשל לאתחל כך מחרוזת), ניתן לאתחל אותו בקובץ h - ההגיון: הוא נוצר פעם אחת (ללא קשר לאינסטנס), והוא קבוע. זה לא רלוונטי לאף מקרה אחר!

משתנים שאינם בסיסיים נגדיר ב־cpp (אבל נכריז עליהם ב־h, כרגיל), וכך נמנע מהגדרה כפולה בקבצי o. שונים.

דוגמא לשימוש בפונקציה סטטית - שקופית 11.

דוגמא לאתחול משתנה קונסטי סטאטי - שקופית 12.

3.3 Inline

הגדרת פונקציה מסוג זה מהווה "רמז" לקומפיילר לשים את הקוד של הפונקציה במקום שבו היא מופיעה. כלומר, אם יש 10 הפניות לפונקציה, לא יהיו הפניות, אלא בכל מקום שקוראים לה היא תופיע בעצמה.

נשים לב כי הקומפיילר לא חייב לעשות זאת, על כן זהו רק רמז.

הרעיון: אם יש פונקציה קטנה, שמקבלת מעט פרמטרים או קטנה שמשתמשים בה הרבה, במקום לעבור במקומות שונים בזיכרון, להעביר דפים מהזיכרון וכו', פשוט מעבירים את הקוד למקום שבו הוא רץ לשם שיפור ביצועים. יש לשים לב מתי משתמשים ביישום זה, כי לעיתים זה עלול לגרום להאטת הביצועים (עוד על כך בהמשך).

פונקציה זו נגישה רק מהקוד שבו היא נמצאת (אי אפשר לעשות לה extern).

3.3.1 מקרו מול אינליין

דוגמא בשקופית 15:

במקרו - לא תמיד מה שציפינו שיקרה: הערך של i לא צפוי בסוף.

באינליין, השימוש הוא "חכם", או שקוף: ההתנהגות כמו שציפינו.

3.3.2 מימוש

אחת הדרכים להגיד שהמתודה היא אינליין, היא מימוש בקובץ ה־h. זהו המימוש הנפוץ ביותר לפונקציות מסוג זה (סביר לעשות זאת למשל ל־getters ו־ setters למשל). אחרת, אפשר לרמוז בעזרת המילה inline בעת הגדרת הפונקציה (שקופית 17).

3.3.3 ההבדל בין אינליין לרגיל

כאמור בעת ביצוע פונקציה רגילה, נעשית "קפיצה" למקום בקוד בו היא מופיעה, נשמרים רגיסטרים מסויימים, לעיתים יש גם החלפה של דפים בזיכרון, וכו'.

בעת ביצוע פונקציית אינליין, מכיוון והקוד של הפונקציה מופיע בכל מקום שצריך אותו, אין צורך לקפוץ ממקום למקום בזיכרון, ואין צורך להחליף דפים בזיכרון. עם זאת, יש צורך בשמירה של כל המשתנים - גם של הפונקציה, גם של ה־main.

3.3.4 ביצועים

לא הכל טוב באינליין - רקורסיה למשל (limited unrolling).
מתי זה טוב?
במקרה שיש פונקציה ארוכה מאוד שמשתמשים בה הרבה פעמים קובץ ההרצה יכולה לתפוח. יתרה מזו, יש יותר משתנים מקומיים כרגע, והעלאת הרבה משתנים לזיכרון יכולה גם להיות בעייתית.
על כן משתמשים באינליין רק למטרות שציינו בהתחלה.

4 העתקה ו־Casting

4.1 העתקה

לדוגמא, נממש את MyString (שקופית 22 והלאה).

4.1.1 אופרטורים

שאלה ראשונה - מה יקרה בקוד בשקופית 23? לא יעבור קומפילציה, שכן לא הגדרנו את האופרטור.
על כן - נוסיף אופרטורים לקלאס. כעת אם נריץ את אותו קוד, הוא יעבור קומפילציה - למרות שלא הגדרנו את האופרטור = הקומפיילר יוסיף את הדיפולטיבי אוטומטית, כפי שראינו בעבר. אבל, מה תהיה ההתנהגות שלו (שקופית 27)? כדי לעשות זאת, צריך להבין כיצד עובד האופרטור הדיפולטיבי.
בדוגמא רואים את ההתנהגות - מימין לשמאל, בסוף מחזיר רפרנס לימיני.
אם לא היה מוחזר רפרנס אלא int, עבור הקוד הימני בשקופית 28 הקוד לא היה מתקמפל (ההשמה לא חוקית).

קלאסים חתימה:

$$\text{X\& X::operator=(const X\&)}$$

מעבירים קונסט רפרנס כדי לוודא שהצד הימני של העותק לא ישתנה.
ערך ההחזרה הוא רפרנס כדי להרשות x=y=z כמו באופרטור הדיפולטיבי.

המשך הדוגמא ללא הגדרת האופרטור, הקומפיילר ביצע member wise copy - העתקה member member , משמע: מעתיק ממש כל ערך של הממברים של האובייקט. למה זה בעייתי? מכיוון ובמקרה של פוינטרים, הכתובת מועתקת ביט ביט.
נביט שוב באותו קטע הקוד - תיווצר זליגת זיכרון! מדוע?

- רץ הקונסטרקטור של Foo.
- רץ הקונסטרקטור של Bar.
- באופרטור שווה מתבצעת העתקת member wise, אז הפוינטר של Foo מצביע ל־Bar.
- בסיום רץ הדיסטרקטור של str1 וימחוק את הפוינטר, לאחר מכן הדיסטרקטור של str2 ירוץ, ואף אחד לא דואג להצבעה הראשונית של Foo.

תיקון - הגדרת האופרטור כך שימחק את הסטרינג הנוכחי, יצור סטרינג חדש ויעתיק את הזיכרון.

לכאורה, יצרנו העתק נוסף וזהה של המחרוזת, וזה יעבוד. אבל, מה יקרה אם נעשה str=str? השורה עצמה חוקית, אבל כאשר היא תתבצע, מתבצעת מחיקה ולאחריה נסיון העתקה, אבל המקור כבר נמחק... בעיה!

כדי לתקן נוסיף בדיקת השמה עצמית: אם זו באמת השמה שכזו, נחזיר פוינטר ל־this, ולא נעשה כלום.

האם עכשיו פתרנו את כל הבעיות? לא! דוגמא בשקופית 36:

האובייקט str נוצר, S מאותחל להצביע לאותו האובייקט, בסופה נקרא dtor עבורו, ובסוף הפונקציה המקורית (השניה) נקרא שוב dtor עבורו.

4.1.2 copy constructor

כאן אנו מסיימים את הרביעיה ש־cpp עושה עבורנו באופן דיפולטיבי - האובייקט הרביעי הוא copy constructor: נקרא בעת העברה לפונקציה, וגם בעת אתחול אובייקט חדש (לא בעזרת new - כפי שמופיע בשקופית 39).

כמו אופרטור =, הקופי קונסטרקטור הדיפולטיבי מבצע member wise copy.

כדי לתקן את הבעיה, ניצור קופי קונסטרקטור משלנו (דוגמא - שקופית 42):

- נקבל משתנה כקבוע, כדי להגן על המקור. קבלה כקבוע היא הכרחית, שכן אם לא היה const, היינו מקבלים פעולה מעגלית (קריאות אינסופיות לקופי קונסטרקטור).
- מקצים זיכרון בגודל של הזיכרון הישן.
- העתקה של הזיכרון.
- כאן העתקה עצמית לא מתרחשת - זהו קונסטרקטור!

קופי קונסטרקטור ואופרטור = עד כה ראינו כי קופי קונסטרקטור והאופרטור הנ"ל מאוד דומים. כמתכנתים נרצה ליעל העבודה, כלומר לא לכתוב את אותו הקוד או קוד דומה פעמיים. יש לכך שלוש אופציות:

- (לא רשומה במצגת) ליצור עוד פונקציה בקלאס, לקרוא לה פעם אחת מהאופרטור, פעם אחרת מהקופי קונסטרקטור.
- לקרוא לקופי קונסטרקטור מתוך האופרטור.
  - שימוש ב־swap.
  - חסרונות:
    - סתם יוצרים אובייקטים והורסים אותם.
    - צריך swap לביצוע ההחלפה.
- לקרוא לאופרטור מתוך הקופי קונסטרקטור:
  - מעדיפים להשתמש ב־MIL: יותר קריא ויותר בטיחותי.
  - באופרטור = עושים בדיקת השמה, אז כאן נעשה אותה "סתם".

שלוש האופציות אפשריות, לרב בקורס זה (וגם בחיים האמיתיים) נממש את שתי הפונקציות.

4.1.3 פיצ'רים של השפה

לא לאפשר העתקה לעיתים נרצה לא לאפשר העתקה (ע"י אופרטור = וקופי קונסטרקטור) ־ למשל אם מדובר באובייקטים גדולים.

פתרון 1: לא נגדיר אותם. בעיה: הדיפולטיביים יעבדו, אז זה לא באמת פתרון.

פתרון 2: יצירת טעות בעזרת אסרט בזמן ריצה. אבל זה לא יפה, והורס למשתמש את הריצה.

כלל אצבע: כל מה שאפשר לעשות בזמן קומפילציה ־ לעשות שם (ולא אחרי).

הפתרון האמיתי: מגדירים את האופרטור והקופי קונסטרקטור כ־private, אבל לא נכתוב מימוש ־ במקרה זה נקבל טעות בזמן קומפילציה.

אם לא נגדיר כ־private, התוכנית כן תעבור קומפילציה, אבל תכשל בשלב ה־linkage, וזה קשה יותר לדבג (ונוגד את כלל האצבע הנ"ל). בכל אופן, אם נעשה פרייבט ונקרא מתוך האובייקט, התוכנה תיפול בשלב ה־linkage.

העתקה וירושה מה יקרה בשקופית 49? התשובה לא טריוויאלית. אם הטיפוסים שונים זה מזה, נקבל שגיאת קומפילציה. אולם בדוגמא זו לא נקבל קומפילציה, בשל הירושה, שכן b הוא סוג של a. במקרה זה יתבצע upcasting, ותתבצע העתקה איבר־איבר (שכן לא הוגדר האופרטור =, ועל כן יופעל הדיפולטיבי) ־ נשים לב כי האיבר המיוחד ל־Derived לא יועתק.

למה זה עובד? בשל overload resolution: ראינו בשיעור הראשון סוגי resolution שונים.

מה קורה בשקופית 51? טבלת הוירטואל לא מועתקת (הגדרה של השפה), ועל כן מופעלת f1 של A, לא של B.

בשקופית 52 הפלט יהיה 2 ־ מתבצעת העתקה איבר־איבר ע"י הקופי קונסטרקטור.

בשקופית 53 הפלט יהיה 5 ־ ההבדלים בקוד בין שקופית זו לקודמת ־ ממומש קופי קונסטרקטור של B שלא קרא לקונסטרקטור של A, ולכן הדיפולטיבי נקרא. כדי לתקן ־ מוסיפים ל־MIL גם את הקופי קונסטרקטור של האב.

4.2 Casting

באופן כללי, כאשר מפעילים פונקציה foo(x) הקומפיילר יחפש את הפונקציה foo המקבלת את הסוג של הטיפוס המתאים ל־x. דיברנו על כך שלעיתים יש casting, היום נראה עוד סוג. למשל ־ upcasting כמו בשקופית 67 שאנחנו כבר מכירים.

4.2.1 Tailored casts

דוגמא בשקופית 58 ־ אפשר להסתכל על הקונסטרקטור כפונקציית המרה. מה יקרה בשקופית 59? אם לא היתה פונקציית ההמרה, היינו מקבלים שגיאת קומפילציה. אולם, מכיוון ויש קונסטרקטור הבונה מאינטג'ר אובייקט מסוג MyClass, האינטג'ר ישלח לקונסטרקטור, והאובייקט שישלח הוא זה שנוצר.

4.2.2 Implicit casts

בשביל קריאה לפונקציות המצפות, למשל, ל־string: בשקופית 60 למשל, המחרוזת מגיעה לקונסטרקטור שבונה ממנה str, וה־str הזמני הזה נשלח לאחר מכן לאופרטור. כאן יש שתי אפשרויות: או שתהיה שגיאת קומפילציה אם לא הוגדר == (אין לו הגדרה דיפולטיבית), או אם הוגדר, ה־str שנוצר נשלח אליו.

צורת קאסטינג כזו היא בעייתית - דוגמא בשקופית 61: היינו רוצים לקבל שגיאת קומפילציה בשורה האחרונה, שכן לא נרצה להגדיר השוואה בין המערך למיקום מסויים במערך. אולם במקרה זה לא תתקבל שגיאת קומפילציה, שכן הקומפיילר יבנה מערך זמני בגודל B[i] בשל "פונקציית ההמרה" = הקונסטרקטור וישווה אותו ל-A, וזו לא היתה הכוונה שלנו.

הפתרון - שימוש במילה explicit לקונסטרקטור, המהווה הוראה לקומפיילר כי רק במקרה של קריאה חד-משמעית (explicit) לקונסטרקטור, יש להשתמש בו. לאחר הוספת מילה זו, באמת נקבל שגיאת קומפילציה. זה לא פוגע לנו ביכולות הביצוע (אפשר לבצע את אותו הדבר אם מגדירים מה שרוצים בצורה חד-משמעית), אך מונע בעיות.

לסיכום!

ארבע פונקציות מוגדרות דיפולטיבית:

- קונסטרקטור דיפולטיבי.
- Copy constructor.
- האופרטור =.
- דיסטרקטור.

אם מגדירים אחד מהם ידנית, לוקחים אחריות לפעולות של פונקציות אלו:

- עבור הקונסטרקטור - יש לקרוא ל-bast c-tor הנכון.
- עבור אופרטור =, יש לבדוק הצבה עצמית, ולהעתקה נכונה של הממברים.

באופן דיפולטיבי, לכל המחלקות יש עותק "רדוד". אם עותק זה לא יוצר עותק עמיתי, עושים אחד מהשניים:

- אוסרים על העתקה - מכריזים כ-private ולא כותבים יישום של הקופי קונסטרקטור ושל האופרטור =.
- מיישמים קופי קונסטרקטור ואופרטור = פומביים משלנו עם העתקה עמוקה.

הקופי קונסטרקטור והאופרטור = מצייתים לחוקי הרזולוציה, כולל קאסטים, ועל כן יש לשים לב לבעיות שיכולות להיווצר. יש להשתמש ב-explicit לקונסטרקטורים לא טריוויאלים.

על מנת להמנע מבעיות - העבר אובייקט by reference! 28/8/2012

נחזור לדוגמא מהשיעור הקודם (שקופית 1). הוכנס כאן תיקון קטן. דיברנו על זה כשיש צורך ב-deep copy נממש copy constructor באופרטור =. ראינו שהבעיה היא ב"=" שהפוינטר פשוט עובר לאיבר השני, ואז בדיסטרקטור הוא מנסה למחוק פעמיים.

נשים לב לשינוי בסדר של הדיסטרקטורים - הסדר שלהם הפוך לסדר של הקונסטרקטורים (שכן הם נוצרים על המחסנית, ועל כן יש לשחרר לפי הסדר של המחסנית).

Templates 5

5.1 הקדמה, מוטיבציה

תחילת החומר המתקדם של C++!

לפני שנתחיל, נזכר בשלבים של הקימפול:

- Preprocessing - המקרואים וה־define מועתקים לקוד, ההערות עפות, מנסים להשאר עם קוד נטו.
- Compilation - מתבצעת על כל קובץ בנפרד (מודול), בשלב זה הכל צריך להיות נגיש וידוע שקיים, אבל אין צורך שהמימוש יהיה נגיש. משלב זה מקבלים קבצי o. כמספר קבצי ה־cpp שהיו.
- Linkage - o. הופכים לתוכנה המוכנה להרצה - קובעים את הכתובות של המימושים והייחוסים.

בטמפלייטים זה עובד אותו דבר, אך דברים יקרו בשלב שונה ממה שהתרגלנו עד כה.

5.1.1 מוטיבציה

דוגמא בשקופית 7 - מה אם נרצה להחליף דאבל או סטרינג? לכל אחד, נצטרך פונקציה שונה. המימוש דומה, ולכן נרצה פונקציה אחת שתממש את שניהם (או סוגים נוספים).
הגישה של C - קבלת משתנה כ־* void, ואז מעתיקים בייט בייט.
החסרונות - העתקת זכרונות עם כל החסרונות שנובעים מכך, אין שימוש באופרטור שווה "חכם".
ב־C++ נרצה איזה swap שיקבל "משהו" ויעתיק".
זה בדיוק הרעיון של טמפלייטים.
נשים לב כי ההערות של הקומפיילר לטמפלייט לא ברורות, לכן יש לשים לב!

5.2 סינטקס

```
template<class T>
```

ה־T& "שומר מקום" למה שנרצה שיכנס למתודה.
המילים typename ו־class שקולות לגמרי - אם נכתוב את שני קטעי הקוד בשקופית 11, נקבל טעות קומפילציה - אין חשיבות לשם עצמו T,P למשל במצגת.
כיצד נשתמש? שתי דרכים:

1. swap<double>(d1,d2) - explicit
2. swap(d1,d2) - implicit.

בשקופית 13 - כאשר הקומפיילר מגיע לשורה השלישית, הוא יוצר אינסטנס של של הטמפלייט עם $T = int$, ואז מבצע קומפילציה של הקוד המוגדר על ידי הטמפלייט הזו.

5.3 פונקציות טמפלייטיות

נשים לב כי בשקופית 14 מדובר בשתי פונקציות שונות! כשרוצים להפעיל את הפונקציה עבור דאבלים למשל במקום אינטים, כביכול היה אמור לעבוד לפי קאסטינג (אין צורך בפונקציה חדשה), אבל זה לא קורה - הפונקציה המתאימה לסוג תמיד מנסה להיווצר בזמן קומפילציה, אם לא מצליחה - תהיה טעות קומפילציה. למה? בדוגמא שלנו למשל הקומפיילר לא יכול לדעת מהו המימוש של סוג מסויים עבור האופרטור שווה. על כן הקומפיילר צריך לבדוק כל פעם.

בשקופית 15 יכולה להיות בעיה משני דברים - אופרטור שווה וקופי קונסטרקטור. כל אחד מהם יצור שגיאת קומפילציה בלתי תלויה - יש כאן שימוש בקופי קונסטרקטור (לא העברה by reference) כי ראינו כי נעשה שימוש בו אם מצהירים על אובייקט ומשתמשים באופרטור שווה.

הטמפלייט הוא בעצם הצהרה על מה שהולך לקרות - הבדיקה עצמה (מתודות, דברים שמשתמשים בפנים, וכו') נעשית כאשר הקוד מתקמפל וכשיש את האינסטנס הספציפי. לכן הקוד של הטמפלייט צריך להיות נגיש בזמן קומפילציה - על כן כולו, כולל המימוש, יהיה ב-h. על כן למשל יכולה להיות שגיאת קומפילציית עבור אינסטנס מסויים, ולא תהיה שגיאת קומפילציית עבור אינסטנס אחר.

מה קורה בשקופית 17 - מהן ההנחות? קיום "<", וזה היחיד (והדברים ש-swap) שהפונקציה דורשת. אין בעיה בהשוואות, שכן אלו השוואות של פוינטרים.

בשקופית 18 - אם =! לא קיים - שגיאת קומפילציה. מתי נשתמש בזה? כשיש אתחול ספציפי של הטמפלייט - דוגמא בשקופית 19.

איך הקומפיילר יודע מתי להשתמש באופרטור הרגיל עבור סוג בסיסי, ומתי להשתמש עבור האופרטור הטמפלייטי?

5.3.1 כיצד בוחרים באיזה פונקציה להשתמש?

השלבים מופיעים בשקופית 20:

- יצירת סט של פונקציות אפשריות\מועמדות:
  - פונקציות שאפשר לעשות להן overloading.
  - פונקציות טמפלייטיות שיכולות לעבור אתחול לסוג הזה - specialization.
- לפונקציות לא טמפלייטיות יש קדימות על פונקציות טמפלייטיות.

דוגמא בשקופית 21:

- בשורה הראשונה של המיין, הפונקציה הטמפלייטית בכלל אינה אופציונלית, שכן הקומפיילר לא יכול להסיק מיהו T, על כן שתי הפונקציות השניות אופציונליות, ונבחרת זו המתאימה ביותר.
- בשורה השניה אותו דבר, ונבחרת המתאימה ביותר - דאבל.
- בשורה השלישית הקומפיילר "מתאים" משהו עבור T - מה שמתאים לו הוא char.

וריאציה שניה בשקופית 22: הפונקציה עבור האינט בהערה:

- בשורה הראשונה, יש רק מועמד אחד - דאבל.
- בשורה השניה - כנ"ל.
- בשלישית כמו מקודם.

וריאציה שלישית בשקופית 23: הפונקציה של דאבל בהערה: תלוי קומפיילר (אולי תהיה הערה או טעות):

- שורה ראשונה - אינט.
- שורה שניה - כנ"ל.
- בשלישית כמו מקודם.

5.3.2 על אינסטנס יחיד

שקופית 24 - נשים לב כי T לא יכול להיות בו"ז שני טיפוסים בעת הקימפול.

- שורה ראשונה - שתי הפונקציות מועמדות, יש עדיפות ללא טמפלייטית, על כן היא נבחרת.
- שורה שניה - שתי הפונקציות מועמדות, עבור הקריאה הזו הטמפלייט נבנה עבור char, והטמפלייטית הכי מתאימה (השניה מצריכה המרה לאינטים), על כן היא נבחרת.
- שורה שלישית - הקומפיילר רואה שיש לו אינט ו־char, הקומפיילר ינסה לייצר טמפלייט עם אינט ו־char, וזה לא יכול להיות, על כן תקרא הפונקציה הלא טמפלייטית. אם פונקציה זו לא היתה קיימת, היתה מתקבלת שגיאת קומפילציה - הראשונה כלל לא מועמדת!

שקופית 25:

- שורה ראשונה - הטיפוסים הטבעיים הם אינט ודאבל, על כן רק הפונקציה השניה מועמדת.
- שורה שניה - שתי הפונקציות מועמדות, הטמפלייטית מתאימה בדיוק בעוד שעבור שימוש בשניה צריך לבצע המרה, על כן תבחר הטמפלייטית.

5.4 קלאסים טמפלייטיים

בשקופית 27 - מה בקוד נסמך על כך שעובדים על סטרינג?

הקוד די גנרי - הטיפוס סטרינג אכן מופיע בכל מיני מקומות בהחזרה וכו', אבל לא נעשות כאן פעולות ספציפיות על סטרינגים, על כן זהו מועמד מאוד מאוד טבעי להפוך לקלאס טמפלייטי.

הסינטקס של קלאס טמפלייטי מאוד מאוד דומה:

```
template <class T> Stk
```

אתחול:

```
Stk<int> intList
Stk<string>stringList
```

נשים לב כי אלו שני טיפוסים שונים!

כמו במתודות שראינו, הכל מתקמפל מחדש בעת יצירת אינסטנס.

נשים לב כי לא לכל סוג נוכל לעשות טמפלייט - רק לאלו המיישמים אופרטורים מסויימים. עוד על כך - בהמשך.

5.5 איטרטורים

איטרטור הוא אובייקט המתנהג כמו פוינטר, דרכו מקבלים גישה מבוקרת לאלמנטים המוכלים באיזהשהוא מבנה נתונים.

בשקופית 33 - הצורה השניה של הגישה עדיפה, כי היא מאפשרת איזושהיא גמישות.

בקוד המופיע, אילו אופרטורים מצפים שיהיו בעלי מימוש עבור האיטרטור? ++, !=, =, אבל לא », כי מופיע i* ולא i - עם זאת, האופרטור * צריך להתקיים.

רשימה מלאה של המימושים שצריך שיהיה לאיטרטור - שקופית 37.

איך יוצרים איטרטור? הוא יהיה inner class, המאפשר עוד שכבה של הפשטה - אין צורך לדעת מה בדיוק התכולה, והוא יכיל פוינטר.

כמו כן, נרצה לוודא שהאיטרטור לא משנה את המבנה של ה-container. דבר זה יספק לנו עוד שכבה של הגנה.

קוד של איטרטור - שקופית 39. נשים לב למספר נקודות:

- הקלאס של האיטרטור נמצא בתוך קלאס אחר.
- לאופרטור ++ יש כפי שהכרנו שתי חתימות, בשקופית זו מופיע מימוש של שתי האפשרויות, הדומה מאוד לזה שראינו בקלאסים רגילים.
- מימוש == על ידי השוואה לפוינטר.
- המתודות begin, end הן של Stk, והן מחזירות איטרטור להתחלה או לסוף בהתאמה. הן אלו שנותנות לנו את היכולת להשתמש באיטרטור.

כעת נוסיף שכבה נוספת של טמפלטיזציה והיא מופיעה בשקופית 42:

template<class Iterator>

המתודה copy תהיה גם טמפלייטית - נרצה להכליל את ההתנהגות של האיטרטורים כפי שראינו קודם, גם עבור פוינטרים. במימוש זה, Iterator יכול למשל להיות *int. עכשיו אפשר לאתחל עם איטרטור או עם פוינטר.

כדי להגדיר טמפלייט עם שני סוגים אובייקטים (או יותר) - פשוט להגדיר בהתאם:

template <class T1, classT2>

אפשר לקבל גם ארגומנטים קונסטנטים - דוגמא בשקופית 45. נראה עוד דוגמאות לשימוש בזה בהמשך.

אפשר גם לתת ערך דיפולטיבי - שקופית 46. זה מאפשר איתחול של קלאס בצורה מאוד מפורשת, או לאתחל רק עם הטיפוס. עם זאת, זה קצת פחות קריא.

עוד פעם נשים לב - הטיפוס הוא לא הטמפלייט, אלא הטמפלייט לאחר שעבר איתחול. כל איתחול הוא טיפוס שונה.

נשים לב לשקופית 48 - אתחול כ-char וכדאבל.

דוגמא מסובכת יותר בשקופית 50:

- בשורה השניה, כל מה שבתוך ב-<> הוא הסוג - זוהי רשימה של רשימות, זה לא זהה לשורה הראשונה.
- בשורה השלישית - הפרשנות היא T=int.
- בשורה הרביעית - הפרשנות היא T=LinkedList<int>.
- בשורה החמישית - הפרשנות היא T=int.

שקופית 51-52:

- מימוש יעבוד רק עבור סוג שיממש <.
- המיין יתקמפל, כי יודעים להשוות בין צ'ארים, אבל זה לא ממש מה שאנחנו רוצים מההשוואה.
- משנים בשביל לקבל את "מה שאנחנו רוצים" - כאשר מאותחל עם צ'ארים, נעשית הבדיקה הספציפית שאנו רוצים, אחרת הפונקציה שתפעל היא הטמפלייטית הכללית.

דוגמא "קצת קריטית" בשקופית 53 והלאה:

- הגדרות של 4 טמפלייטים:
  - השניה - ספציאליזציה חלקית.
  - השלישית - כנ"ל.
  - הרביעית - ספציאליזציה מלאה של הראשונה.
- סטטיק אסרט - אסרט שמתרחש בזמן קומפילציה.
- לשורה הכמעט לפני אחרונה - יש ספציאליזציה לשני דאבלים, לכן האסרט עובר והכל בסדר.
- לשורה האחרונה - יש ספציאליזציה חלקית לאינט וסוג מסויים, ואז ההגדרה השניה נבחרת, והאסרט יפול (לפי ההגדרה בתוך טמפלייט זה).

5.5.1 המילה typename

נשים לב לכך שבשקופית 55 אין הבדל בין typename ל־class, אבל למילה זו יש שימוש נוסף.
מהו? שקופית 57: מניעת שגיאות קומפילציה. דוגמא נוספת לשימוש - שקופית 58: מכפלה סקלרית בין שני וקטורים.

5.5.2 טמפלייטים מול פולימורפיזם

טמפלייטים מגדילים את כמות הקוד, אך זמן הריצה מהיר יותר משימוש בירושה. עם זאת, זמן הקומפילציה של טמפלייטים ארוך יותר מזה של אינליין.
בשילוב עם inline, התקורה יכולה לרדת לאפס.

חוק אצבע: טמפלייטים יותר טובים לקונטיינרים ואלגוריתמים.

6 כמה דברים קטנים

6.0.3 mutable

30/08/2012

"רעיון קצת מוזר" - אם שמים mutable על משתנה של מחלקה, המשמעות - גם אם נגדיר את המחלקה כקונסט במהלך הקוד, עדיין נוכל לשנות את המשתנה הזה. ניתן לשימוש רק על data members שאינם סטטים ואינם קונסטיים.
דוגמא: בשקופית 3 עובר קומפילציה כי מאפשרים לשנות את המשתנה, למרות שהפונקציה היא קונסט.

6.0.4 פוינטר ל־member function

שקופית 4 - השורה עם val= מניחה שמחזירים רפרנס (אחרת יש בעיה). כמו פוינטר רגיל לפונקציה, עם סינטקס קצת אחר.

6.0.5 עוד על nested class

ראינו את שקופית 5 כבר בהרצאה קודמת. בסטנדרט לא אמור להיות גישה, אבל לקלאס הפנימי בהרבה מהקומפיילרים יש גישה לקלאס האב. אנחנו ניתן friend למי שצריך, כדי להתאים לסטנדרט ולכמה שיותר קומפיילרים - כלומר נוסף friend node בתוך הקלאס הפנימי.

בדוגמא בשקופית 6 תהיה שגיאת קומפילציה, כי func פונקציה של הפנימי, var אובייקט של החיצוני: אין var בתוך inner!

7 ירושה מרובה ו־virtual base class

7.1 ירושה מרובה

ראינו כי קלאס אחד יכול לרשת מכמה קלאסים בו זמנית. בדר"כ מצב כזה מצביע על עיצוב גרוע, לכן יש לשים לב לשימוש.

במידה ובכל זאת יש צורך - בשקופית 8 למשל נקבל שגיאת קומפילציה, כי הקומפיילר לא יודע לאיזה b ללכת. לכן יש לשים לב להגיד בפירוש לאן רוצים לגשת.

כמו כן, סדר ההריסה תמיד הפוך לסדר היצירה, כמו כן קודם יוצרים את האב, ואח"כ את הילדים. נחזור לכך עוד בהמשך.

מדוע באותה דוגמא אין בעיה לגשת למשתנים? בסטראקט הדיפולט הוא public - אם היה כתוב קלאס זה לא היה עובד, שכן בקלאס הדיפולט הוא private.

ננסה להמנע באופן כללי מירושה מרובה.

7.1.1 ירושת יהלום

ירושת יהלום - קלאס יורש משני קלאסים שונים, שניהם יורשים מאותו הקלאס.

בשקופית 9 - נשים לב כי המשתנים בשתי השורות האחרונות הם משתנים שונים!

בשקופית 10, בניגוד לשקופית הקודמת, עם השימוש במילה virtual, המשתנה a הוא אותו המשתנה - נשים לב כי המילה virtual חייבת להיות על הירושה של A, לא יעזור אם היא תהיה על B1 ו־B2

הבעיה: למה שהמתכנת שכתב את B1,B2 יחשוב בכלל שהוא צריך להגדיר אותן כ־virtual? לא עושים את זה תמיד, כי זה יוצר overhead שאפשר לחסוך.

אצלנו, אם אין סכנת ירושת יהלום, אין צורך ב־virtual. צריך להיות מודעים לצורה ה"משונה" בה הירושה הזו עובדת.

cross cast היא המרה בין שני "אחים" - צריך לשים לב לבעיות במקרה זה.

7.2 virtual base class

שקופית 12 - מדוע i נשאר עם 28, כלומר - מדוע הופעל הקונסטרקטור הדיפולטיבי? במקרה של ירושה עם רק העתק אחד, הצאצא הכי רחוק שנוצר, במקרה הזה C, מגדיר את A, ובגלל ואין שם קריאה לקונסטרקטור של A, נקרא הדיפולטיבי - אם ירושת A של B לא היתה

וירטואלית, התוצאה היתה כפי שהיינו מצפים. וירטואלי אומר בעצם שקיים A אחד, לכן הקומפיילר מתעלם מהקריאה מתוך הקונסטרקטור של B.

פתרון - קריאה לקונסטרקטור של A בתוך C. נשים לב כי לא נוריד את הקונסטרקטור של A מהקונסטרקטור של B, כי זוהי גם מחלקה שניתן ליצור אינסטנס שלה.

אם היינו יוצרים למשל D לפי טבלת הירושה בשקופית 14, היינו צריכים ש-D יאתחל את A, ולא C! מספיק שתהיה ירושה וירטואלית של ה-base class.

7.2.1 סדר קריאת הקונסטרקטורים

שקופית 15: לפי סדר הירושה - משמאל לימין. הסדר לא נקבע ע"י רשימת האתחול! חשוב לזכור בשביל עבודה נכונה, וגם כי היתה על זה שאלה במבחן בשנים האחרונות.

למשל, בשקופית 16, הקונסטרקטור הראשון שנקרא הוא של B2, ולא B1. על כן יש לשמור שרשימת האתחול תהיה באותו סדר של סדר הירושה, בשביל למנוע בלבול[3] - נשים לב כי שקופית זו מתקמפלת, לכן חשוב להיות מודעים לסדר הזה.

8 אקספשנים

מג'אווה אנו יודעים כי לא הכל קורה כמו שאנחנו רוצים - שגיאות יכולות לקרוא ממספר סיבות. היינו רוצים לתפוס את השגיאות האלו ולהתמודד איתן.

עד עתה התמודדנו בשיטה של C:

- שימוש ב-assert.
- return codes.
- משתנה טעות גלובלי - errno, perror.

אבל, הטיפול הזה מעיק. היינו רוצים להפריד את הקוד הלוגי של הפונקציה מהטיפול בשגיאות. אקספשנים - חריגות, מאפשרות לעשות את זה.

ההבדלים בין אסרט לאקספשן:

באסרט בדר"כ משתמשים בדיבגינג, אקספשנים אפשר להשאיר בתוך הקוד, אם כי "בחיים האמיתיים" לא משתמשים תמיד בהם בגלל עלויות. עם זאת, שימוש בהם נותן קוד יפה. כמו כן, אסרט תופס באג בתוכנית, אקספשן היא בעיה שעלולה להתרחש.

8.1 סינטקס

8.1.1 זריקת אקספשן

מה "זורקים"? שקופית 23 - ב-cpp יכול להיות הכל: טיפוס בסיסי, אובייקטים, רפרנסים...

דוגמא ראשונה של זריקת אקספשן - שקופית 24: מקיפים את בלוק הקוד בו יתכן ויזרק אקספשן ב-try, ואת התפיסה והטיפול ב-catch. נשים לב כי אם נזרקה אקספשן, היא מיד תתפס - דוגמא בשקופית 25. לאחר הטיפול הקוד ממשיך הלאה אל מחוץ לבלוק ה-catch.

ישנן מחלקות היורשות מאקספשן בהן ניתן להשתמש.

כמו כן, אפשר לתפוס סוג מסויים של אקספשן בבלוקי catch שונים, ואז לאקספשנים שונים להגדיר טיפול שונה.

נכון לעכשיו לא צריך להכריז בחתימה של הפונקציה כי יתכן ותזרק אקספשן. זה גם לא יעיל במיוחד. אפשר להכריז כי פונקציה אינה יכולה לזרוק אקספשנים - סינטקס בשקופית

[3] מעתה, השתמשו בדיונים במשפט של אווה: "המשפט שלך לא מתקמפל".

30. מדוע בזה כן משתמשים? בשביל יעול עבודת הקומפיילר. במידה וכן יזרק אקספשן, נקבל התנהגות אותה נראה בהמשך.

8.1.2 תפיסת אקספשן

שקופית 27 - דוגמא לשימוש בפולימורפיזם. בסוף אפשר לבחור מה לעשות - הזריקה throw מעבירה את האחריות הלאה ל"תופס" הבא.

סדר התופסים חשוב! אם הסדר בשקופית 28 היה שונה, היה נתפס בראשון שמופיע. האקספשן יתפס בכל מקום אליו הוא יכול לעשות המרה, וכמובן בבלוק המתאים לו בדיוק. catch(...) תופס את כל האקספשנים.

8.1.3 רצף הקוד

שקופית 31: כאשר אקספשן נזרקת, הסקופ הנוכחי מסתיים מיידית, ומעביר את השליטה לבלוק ה־catch, או לפונקציה מתחתיו ב־stack. פונקציה זו יכולה "להעביר הלאה", או לטפל בה.

אין finally בסינטקס של C++, כלומר אין טיפול שקורה תמיד. השימוש ב־catch(...) עוקף את זה - שמים בלוק זה בסוף הבלוקים של התפיסה, אם האקספשן לא נתפסה באף בלוק, תיתפס שם. כך למשל אפשר לוודא סגירת קובץ, וכו' (שקופית 32).

8.1.4 נקודות קצה

עבור אקספשנים בקונסטרקטור, הדיסטרקטורים של האובייקטים שכבר אותחלו נקראים. עם זאת, זה קורא אם האובייקט בבלוק ה־try: אחרת, עלול להקרא terminate לפני הדיסטרקטור.

עבור אקספשנים בדיסטרקטור, אין התנהגות מוגדרת, עלול לגרום לתוכנית ליפול. על כן לא זורקים בדיסטרקטור.

8.2 סוגי אקספשנים

שקופית 6־35: שימוש בפונקצייה what כדי לאסוף יותר פרטים על מה שהתרחש.

8.2.1 ירושה מקלאס אקספשן

שקופית 37 - קלאס היורש מאקספשן. מממשים בו את הפונקציה what() (כדי שנראה שאכן נזרק, נגיד). משפחת האקספשנים הסטנדרטית - שקופית 38.

8.3 כאשר לא תופסים אקספשן שנזרקה

כאשר אין תפיסה של אקספשן, או אקספשן מדיסטרקטור, וכו', נקראת הפונקציה terminate.

הפונקציה terminate בדיפולט קוראת ל־abort, אך אפשר לשנות אותה! סינטקס בשקופית 39: אפשר להשתמש בה בשביל העברת מידע לגבי השגיאה, לצאת בצורה מסודרת, וכו'.

9 המרות ו־RTTI

9.1 סוגי המרות

דיברנו קצת על סוגי המרות, ראינו שכל סוגי ההמרות מ־C רלוונטיות ל־CPP. נראה כעת ארבעה סוגי המרות מיוחדת ל־CPP. יש לציין כי בעוד שחשוב להכיר, באופן כללי נמנעים משימוש בהמרות.

- static_cast - עובד כאשר קיימת המרה אקספליסיטית: upcast או המרה סטנדרטית או שהוגדרה ע"י המשתמש. בטוחה יותר מקאסטים רגילים: למשל, לא עושה קאסט של int* ל־float*. כישלון גורם לטעות קומפילציה.
- const_cast - מרגיל לקונסט כבר ראינו. המרה זו מאפשרת גם המרה של קונסט ללא קונסט. יכול לגרום להמון בעיות.
- reinterpret_cast - ההמרה הכי מסוכנת - מסוג אחד לאחר. אין כאן שמירת טיפוסים (כמו הקאסט המסוכן של C).

RTTI - Run Time Type Information - מכניזם ב־c++ החושף מידע על הסוג של אובייקט בזמן ריצה. הוא מכיל את הסוג הרביעי של הקאסט (וכן את האופרטור typeid). מדוע יש צורך? ראינו ש־up-casting עובד בלי בעיה. down-casting לעומת זאת מסוכן. לשם כך משתמשים בסוג הרביעי:

- dynamic_cast - מאפשר down-casting באופן הבא:

  dynamic_cast<T>(expression)

  - מחזיר פוינטר תקף אם הביטוי הוא מהסוג T, אחרת מחזיר NULL.
  - קאסט לסוג רפרנס זורק אקספשן אם נכשל. לא נזרק תמיד, כי זוהי פעולה "יקרה".

  איך משתמשים? צריך לבדוק באמת שניתן להשתמש כפי שנרצה - דוגמא בשקופית 48. חשוב לשים לב:

- אפשר להשתמש רק על פוינטרים או רפרנסים - אחרת לא מתקמפל.
- אפשר להשתמש לצורך up-cast, down-cast, cross-cast.
- אפשר להשתמש רק בטיפוסים עם פונקציות וירטואליות - נקראים גם טיפוסים פולימורפיים. נגלה בעיה זו בזמן קומפילציה.

דוגמא - שקופית 50: טעות קומפילצייה כי אין פונקציות וירטואליות.
האופרטור typeid משיג מידע על אובייקט\ביטוי - דוגמא לשימוש בשקופית 51.
דוגמא לשימוש לא נכון ב־RTTI - שקופית 52.

10 עוד כמה נושאים קטנים

4/9/2012

10.0.1 קריאות לקופי קונסטרקטור - מתי?

- קריאה מפורשת - כן.
- פרמטר שמועבר לפונקציה by value - כן.
- פרמטר המועבר לפונקציה by reference - לא.
- ערך החזרה מפונקציה by value - כן.
- ערך החזרה מפונקציה by reference - לא.
- בפקודה MyClass a=b - כן (כבר ראינו).

10.0.2 אופטימיזציה

חוק ראשון - לא לעשות אופטימיזציה!

רב הנסיונות לבצע אופטימיזציה מביאים לבאגיים ושלל התנהגויות מהנות אחרות.

חוק שני - חשוב ובחן (profile)

עוד - בשקופיות שיש ללמוד לבד.

10.0.3 עוד כמה נושאים קטנים

10.0.4 RVO - Return Value Optimization

הסטנדרט של c++ מרשה לקופייל לבצע אופטימיזציה, כל עוד התוצאה שתתבצע מראה את אותה התנהגות כאילו כל הדרישות של הסטנדרט התמלאו. עם זאת, ל-RVO יש רשות לשנות התנהגות זו! פיצ'ר זה נתמך ע"י רב הקומפיילרים. כדי לדעת להשתמש, יש להכיר את הסטנדרט.

שקופית 9-6 - אילו קונסטרקטורים יקראו?
בתוך הפונקציה foo יש החזרה by value והעברת פרמטר by value.

- הפלט הראשון - עבור העברת הפרמטר.
- הפלט השני - x צריך להיות מועתק כדי לחזור לפונקציה הראשית. לאחר מכן היינו מצפים עוד קופי קונטרקטור עבור b - כלומר היינו מצפים לשתי הדפסות, ולא רק אחת. למעשה הערך נשמר ישירות ל-b. זוהי אופטימיזציה שאמורה להפתיע אותנו, כי כל אופטימיזציה אמורה להיות שקופה. אופטימיזציה כזו היא מיוחד ונקראת RVO - return value optimization. אופטימיזציה זו היא תלות קומפיילר - לעיתים תהיה קריאה אחת, לעיתים שתיים, ולפעמים אף לא אחת.

10.0.5 Method hiding

ברב המקרים לא נקבל warning על השורה האחרונה.

הנקודה המפתיעה היא בשורה השניה של הקוד - היינו מצפים (אולי) שיהיה overloading - כאשר קוראים ל־f שמקבל ערך בוליאני, תופעל הפונקציה של B. עם זאת, נקראת הפונקציה של D: המתודה של B מוסתרת ממש על ידי המתודה של D. בין אם יש virtual ובין אם אין, נקבל את אותה ההתנהגות.

מדוע אין העמסה הרגילה? אם d לא היה D אלא פוינטר או רפרנס, היה overloading רגיל. מכיוון והוא D, אנו מקבלים את ההתנהגות הזו - אנו לא נגישים לפונקציות האב דרך האובייקט עצמו (אלא רק דרך פוינטר או רפרנס).

אם לא היתה פונקציה ל־D בשם f, היתה פשוט מתבצעת הפונקציה של B כמה פעמים (לא היה אף אחד להסתיר אותה).

10.0.6 Argument dependant lookup

נשים לב כי הדוגמא היתה עובדת גם ללא טמפלייטים (זה רק בשביל אימון שלנו).

הערה 10.1 זהו קוד קצר. צריך להתרגל לקרוא קוד ולהבין מה הוא עושה (גם למבחן וגם לחיים האמיתיים).

מהי ההתלבטות? C יורש מ־A, ויש גם מתודה גלובלית. אם לא היתה המתודה הפנימית, היתה הדפסה של 0. אבל ההדפסה של 1 במצב הנוכחי היא ה"הגיונית" - הכנסו סוג, אנחנו רוצים לרשת ממנו.

מה קורה בשקופית הבאה? כאן יש כבר שתי חתימות שונות - במקרה זה נקבל טעות קומפילציה, כי הקומפיילר "מניח" שטעינו והתכוונו לקרוא למתודה הפנימית, במידה ויש שתי מתודות באותו השם.

10.0.7 פולימורפיזם עם רפרנסים

משמאל - יודפס B: השורה השניה מפעילה את הקופי קונסטרקטור רק על ה"דברים" של B, כל ה"דברים" הנוספים נשארים בחוץ. פעולה שכזו נקראת trimming.

מימין עם זאת - יודפס D: ציינו קודם כי פעולות על רפרנס מבצעות פולימורפיזם אמיתי, כלומר ל־b מועתקים כל המאפיינים של d, שהוא מסוג D. אין כאן הפעלה של קופי קונסטרקטור.

10.0.8 נקודה חשובה בנוגע לאקספשנים

כאשר עובדים עם A כאובייקט - סדר קריאת הדיסטרקטורים הוא כפי שציפינו.

כאשר עובדים עם A כפוינטר - אין הרצה של דיסטרקטורים של האובייקטים, אלא רק של הפוינטר עצמו.

אפשרויות לפתרון:

- set_terminate - לנסות לשנות אותה כך שתשחרר את הזיכרון.
- catch - לטפל של בשחרור הזיכרון.
- auto_ptr - הנושא הבא שלנו:

11 ניהול זיכרון - פוינטרים חכמים

אחד הנושאים החשובים ביותר בכתיבת קוד ב־c/c++ הוא ניהול של הזיכרון המוקצה דינמית. נשאלת השאלה, כיצד לוודא כי הזיכרון שהוקצה ישוחרר כאשר אין בו יותר שימוש. כמו כן, חשוב לבצע הקצאה ושחרור באופן יעיל במרחב (פרגמנטציה - עוד על זה ב־OS) ובזמן (שימוש ב־new הוא יקר).

11.1 שתי אסטרטגיות ראשונות שאנחנו כבר מכירים

- Fixed ownership - הישות (פונקציה או אובייקט) שמקצה היא זו שמשחררת. דוגמא - מחלקת סטרינג, שמקצה זיכרון בקונסטרקטור, ומוחקת אותו בדיסטרקטור.
- Dynamic ownership - לאובייקט יש בעלים. עם זאת, הבעלים משתנה דרך קריאות לפונקציות ספציפיות. דוגמא בשקופית 22.

ראינו שהאופציה הטובה ביותר היא הראשונה: כל פונקציה שלא משחררת את הזיכרון שהקצתה עלולה ליצור בעיות. עם זאת, במקרה זה אין לפונקציות המקבלות את המידע שליטה על אותו המידע - למשל על גודל של מערך.

11.2 אסטרטגיה שלישית -מצביעים חכמים

אסטרטגיה נוספת:

- Dynamic ownership (runtime) - היינו רוצים אובייקט חיצוני שיזכור מי הבעלים של כל זיכרון שהוקצה.

מה היינו מצפים מאובייקט כזה? קצת דומה למה שראינו על איטרטורים.
מצביע "טיפש" המסופק על ידי הקומפיילר זוכר את הכתובת של האובייקט ונותן לנו דרך לגשת אליו, אך זה הכל. STL מציעה מספר פתרונות בצורה של מצביעים חכמים, נכיר שלושה (הראשון כבר לא יהיה זמין בגרסא הבאה):

- auto ptr - עוטף פוינטר, וכאשר הפוינטר מושמד, משחרר את הזיכרון. העתקה היא העברת בעלות - רק auto_ptr יחיד מצביע לכל זיכרון שהוקצה.
הוא טמפלייטי, לכן הוא יכול להצביע (כמעט) לכל אובייקט כלשהוא. בשקופית 29, בשורת הקוד של foo מופעל קונסטרקטור המקבל פויינטר. myBar מתנהג כמו פוינטר רגיל, אולם היתרון שלו הוא שהוא נמחק אוטומטית: זהו אובייקט, לכן בסוף הסקופ נקרא הדיסטרקטור שלו.
למה הוא לא בדיוק כמו פוינטר? בשקופית הבאה ב־if רץ הקופי קונסטרקטור. בפוינטרים רגילים, הכתובת עצמה היתה מועתקת. עבור קלאסים, בדר"כ מועתק המימוש של קופי קונסטרקטור (או אופרטור שווה). מה הפיתרון?
ההצהרה לוקחת את הבעלות מ־myBar ומעבירה אותה ל־myOtherBar. החתימה של הקופי קונסטרקטור (שקופית 33) אם כך היא לא const, בניגוד למה שהתרגלנו אליו עד כה. יתרה מזו, כל הקונטיינרים ב־STL מצפים להתנהגות ה"רגילה", וכאן כאמור יש הפרה של הכללים הללו. זו הסיבה העיקרית לכך שפוינטר זה אינו מופיע יותר בקודים חדשים.
בעיה בשקופית 33 - מקבל by reference ומבצע release. הסיבה לכך - לוודא כי הבעלות עוברת הלאה ונמצאת אצל אובייקט יחיד בכל שלב. השורה המסומנת מוודאת שלא מדובר בהשמה עצמית (כבר ראינו זאת בקלאסים רגילים).

איך זה עוזר? במקרה ונזרקת אקספשן בפונקציה בה נוצר auto_ptr, נקרא הדיסטרקטור שלו, המשחרר את הזיכרון, בניגוד לפוינטר רגיל, שפשוט היה נמחק (שקופית 34).
סיכום מהיר:

- מנגנון לניהול בעלות.
- לא מחליף את הצורך בפוינטרים - לכל משאב מצביע רק אחד.
- מסוכן כי הקופי שלו בעייתי, לכן לא משתמשים בו בקונטיינרים של STL.
- בסטנדרט החדש הפויינטר הזה "הולך".

- unique_ptr: דומה לauto_ptr ומחליף אותו, עם סינטקס יותר ברור - מבוצע move במקום מחיקה, הקוד שהתקמפל ב־auto_ptr לא יתקמפל הפעם. לא נלמד עליו ממש בקורס זה.

11.3 אסטרטגיה רביעית - Reference Counting

לעיתים לא נרצה להכריז על בעלות - ניהול הבעלות מכריח אותנו לבצע עותקים שאין הם צורך. האם אפשר לנהל את הזיכרון ללא בעלות? Reference Counting נותן לנו פיתרון שכזה.

נניח והקצאנו שלושה אובייקטים, ואז השתמשנו באופרטור שווה למשל - ראינו שאם לא מיישמים אופרטור שווה בצורה טובה, כולם פשוט יצביעו לאותו האובייקט, וזה יכול לגרום להתנהגות לא רצויה. עבור סטרינג, שימוש באופרטור שווה יגרום ליצירת שני עותקים זהים, שאם לא ישונו בהמשך הריצה, אין בהם צורך.

במקום זאת, משתמשים במנגנון נקרא COW - Copy On Write: כל האובייקטים יצביעו לאותו אחד, כאשר ישנו counter הסופר כמה אובייקטים מצביעים אליו. כאשר משנים אותו מבצעים העתקה של האובייקט - "lazy evaluation".

שיטה זו היא למעשה טכניקת אופטימיזציה. למשל - עבור string, כפי שמופיע בשקופית 42, פעולה זו היא אוטומטית.

גם STL מציע משהו דומה -

- shared_ptr: משתמש ב־"shared memory semantics" (לרב - reference counting): מלבד הצבעה לאובייקט, כל האינסטנסים שומרים פוינטרים ל־counter, שעולה בעת יצירה והעתקה, ויורד בעת מחיקה. כאשר ה־counter מגיע לאפס, מופעל הדיסטרקטור.

"היה במבחן של שנה שעברה... לכן לא יהיה במבחן של השנה".

דוגמא לא שלמה - בשקופית 44. חסר למשל מחיקה של ערכים ב־o._cnt בהתחלת האופרטור =, ואם הגענו לאפס אז צריך לעשות delete. אי אפשר למשל לממש עם static, כי כאמרו המטרה שלנו היא לא counter אחד ויחיד לכולם, אלא קאונטרים שונים.

11.4 במה להשתמש מתי?

- auto_ptr/unique_ptr פשוט אך בעייתי. טוב עבור משתנים מקומיים, אך אסור לשימוש בקונטיינרים של STL!
- shared_ptr פשוט, חוסך בזיכרון, ויעיל+־. מאוד טוב עבור אובייקטים גדולים וקונטיינרים של STL.

12 STL

12.1 על איטרטורים

ראינו כי איטרטורים הם דרך לעבור על רצפים. ישנם סוגים שונים של איטרטורים: לתמיכה בקריאה, בכתיבה ובגישה אקראית.

ב־STL לכל קונטיינר יש איטרטורים שונים, כל אחד מממש דברים קצת שונים. טבלת סוגי איטרטורים - שקופית 52. לכל קונטיינר יש את האיטרטורים המתאימים לו - למשל וקטור מממש את האיטרטור הכי חזק.

12.1.1 ואלידיות של איטרטורים

שינוי הקונטיינר (ופעולות נוספות) עלול להביא לאינולידציה של האיטרטור.

שקופית 54 - על פניו אין בעיה עם המחיקה. אבל, אם מוחקים, ה־i עובר אינוולידציה, ולכן אין משמעות ל־i++ - אין אפשר להשתמש ב־i יותר.

ב־list, set, map, האינוולידציה תגרום לכך ש־i לא יהיה איטרטור חוקי. ב־vector, i יכול להצביע על האלמנט הבא במקום על זה שאנו רוצים.

נסיון שני בשקופית 56 - מקדמים את הפויינטרים בתוך לולאת while, יוצרים כל פעם איטרטור חדש ובלתי תלוי. עבור חלק מהקונטיינרים פתרון זה באמת עובד, אבל בוקטור למשל נדלג על איבר - i יצביע על <u>אותו המיקום, אבל לא על אותו הערך</u>, הוקטור נמחק וקטן.

איך יודעים מה לעשות? קוראים את התיעוד. 6/9/2012

דיברנו על כך שלעיתים אנו רוצים שפונקציה מסויימת גם תמומש כקונסטית - זה מתרחש גם באיטרטורים: קיים גם איטרטור קונסטי. שני הסוגים האלו עם typedef בתוך הקונטיינר. נשים לב כי הפונקציות begin(), end() מחזירות איטרטורים רגילים ואינן מתודות קונסטיות, על כן יש להשתמש במתודות המחזירות const_iterator.

12.1.2 שלושה סוגי איטרטורים עיקריים

כל סוג מרחיב את הקודם:

- forward: אפשר לקרוא מה שכרגע כתבנו. אפשר גם לייצר מספר עותקים של איטרטור זה, ולהגדיל ולהסתכל על התוכן באופן נפרד.
- דו כיווני: אפשר גם לחזור אחורה. לקונטיינרים ב־STL יש לכל הפחות את הסוג הזה.
- רנדומלי: עם אריתמטיקת פוינטרים.

12.1.3 אלגוריתמים ואיטרטורים

אם אלגוריתם מצפה לקבל איטרטור מסוג מסויים, למשל forward, האיטרטור שהוא חייב לקבל יהיה מסוג זה או נמוך יותר בטבלת האיטרטורים - כלומר את האיטרטור הזה, או כל איטרטור המרחיב אותו. דוגמא בשקופית 8.

12.1.4 איטרטורים וקונטיינרים רציפים

על הפעולות insert(), erase()- שקופית 9. דוגמא לשימוש - בשקופית 10.

12.1.5 איטרטורים וקונטיינרים אחרים

ה־insert אמור לשמור על הקלאס בצורה תקינה. כלומר, בקונטיינרים ממויינים, ה־insert אמור לשמור על המיון.

למה נרצה להעביר ל־insert מיקום? כדי לשפר את היעילות (לא יהיה צורך להשוות ע"מ למצוא את המקום של האובייקט).

12.1.6 איטרטורים וקונטיינרים אסוציאטיביים, מפה

שקופית 13 - אפשר למצוא מפתח, חסם תחתון על המפתח, וחסם עליון על המפתח.

בשקופית 14, מה נקבל במקרה של איטרטור של מפה ב־i*? זוג של key,value - נקרא בשם pair: מבנה המכיל שני איברים (שקופית 16).

הפונקציות שיש למפה - שקופית 17. שימוש באיטרטור של מפה - שקופית 18.

12.2 אלגוריתמים

רובם עובדים על רצפים, המועברים כשני איטרטורים - איטרטור להתחלה, ואיטרטור לאחד אחרי הסוף. הם מסתמכים על סוג האיטרטור, ולא עם סוג הקונטיינר.

יש המוווון אלגוריתמים, פשוטים ומסובכים, למשל שקופית 21. נכיר כמה מהם.

12.2.1 אלגוריתמים שאינם משנים רצפים

- find - מוצא את ההופעה הראשונה של ערך ברצף.
- find_if - מחזיר את המיקום של פרדיקט שמכניסים אליו כפרמטר.

בדוגמא בשקופית 23 - pred הוא מסוג פונקציה - functor המחזירה ערך בוליאני. דוגמא נוספת - שקופית 24.

12.2.2 אלגוריתמים של רצפים מסודרים

- lower_bound
- binary_search
- merge
- sort - מקבלת איטרטורים לא קונסטיים, כי משנה את הקונטיינר (מסדרת אותו).
- copy - האם המימוש בשקופית 27 נראה סביר\אינטואיטיבי\מה שהיינו מצפים לראות?
  - ה־first ו־res לא חייבים להיות מאותו סוג, והמימוש מצפה לכך כביכול - אבל אין בעיה, כי מקדמים איטרטורים שיודעים איך לעבוד עם הקונטיינר.
  - הוא מניח שאפשר לקדם את האיטרטור res, וזו הבעיה - אולי הוא כבר הגיע לסוף הקונטיינר?

  פתרון?
  - דרך ראשונה - להשתמש אקספליציטית ב־insert (להגדלת הוקטור).

- דרך שניה - insert_iterator. זהו לא פיתרון אינטואיטיבי, אך היתרון: עושה הכללה, ואפשר לעבוד עם ה־copy "כמו שצריך". דוגמא - שקופית 31: אופרטור = מכניס אלמנט חדש לקונטיינר, ומאתחל את ערכו לעותק של הארגומנט, אופרטור * מחזיר רפרנס לעצמו, ואופרטור ++ לא עושה כלום - מחזיר this*.

עוד sort:

בשקופית 34, מצפה לקבל איטרטור של random access, אך מקבל איטרטור ללא היכולת הדרושה, לכן תהיה טעות קומפילציה (בשלב מאוד מאוד מאוחר - אחרי יצירת הליסט, וכו', רק בעת הנסיון לביצוע משהו שאי אפשר לעשות). איך נוכל לדעת מראש? כרגע אין דרך אחרת מאשר קריאת התיעוד.

12.3 אדפטורים

אדפטור הוא מבנה נתונים המשתמש ביכולות של קונטיינרים של STL שכבר הכרנו, ומרחיב אותן. הוא לא בדיוק קונטיינר, אבל הוא קלאס המספק אינטרפייס מסויים המסתמך על אובייקט של אחד מהקלאסים של הקונטיינרים לטיפול באלמנטים.

הקונטיינר ה"נרמז" מוחבא כך שניתן לגשת לאלמנטים שלו באופן עצמאי ללא קשר לקונטיינר בו יש שימוש.

stack, queue, priority_queue ממומשים כאדפטורים.

(שקופית 37) סטאק - על הקונטיינר לתמוך בפעולות:

- back()
- push_back()
- pop_back()

הסטאק מספקת את הפעולות push, pop, top, size, empty. בניגוד לג'אווה, pop לא מחזיר ערך - זוהי פונקציה void.

בשקופית 39 - השימוש מביא להתנהגות כמו במחסנית רגילה. האם להחזיר קונסט רפרנס מספיק טוב ב־pop? לא! אי אפשר להחזיר קונסט רפרנס לאובייקט מקומי.

13 R-Value Reference

המטרה - להציג טיפוס חדש. מופיע ב־C++11. הסינטקס החדש: T&&.

rvalues שהכרנו לא היו אמורים להיות יכולים לשינוי - בדר"כ משתמשים מקומיים, ונחשבו לבלתי ניתנות לאבחנה מ־const T&.

T&& הוא סוג חדש של non-const reference - הרפרנס החדש יהיה rvalue, בניגוד למה שהכרנו עד כה - lvalue, ויהיה ניתן לשנות אותם לאחר שהם נוצרים, לשם שימוש ב־move semantics.

מה זה אמור לפתור? כאשר אובייקטים מועברים by value, יתכן ויתרחשו הרבה העתקים "עמוקים" שאין בהם צורך. פתרון זה אמור לפתור כמעט לגמרי את כל ההעתקות הכפולות. כיצד? לדוגמא נדון בוקטור.

- העתקה של הפוינטר ל־c-style array לתוך הוקטור החדש.
- לשנות את הפוינטר בתוך ה־rvalue ל־NULL - הוא זמני ולכן לא יקרא שוב, אז אף אחד לא ינסה לגשת ל־NULL, וכמו כן, מכיוון והוא NULL, הזיכרון שלו לא נמחק כאשר הוא יוצא מהסקופ.

דוגמא - שקופית 45: הקונסטרקטור יקרא ע"י הקומפיילר רק אם לא אכפת לנו שהאובייקט מימין ימות.
יתרונות:

- שיפור ביצועים לקוד קיים מבלי לשנות דבר מחוץ לספריה הסטנדרטית.
- הקומפיילר מפעיל את זה בשבילנו.

דוגמא יפה - במבחן חורף 2011 או 2012.

חלק II
תרגולים

תרגול 2

Operators overloading

21/8/2012

השתמשנו כבר באופרטורים: -, +. אפשר להסתכל עליהם כפונקציות. cpp יש syntactic sugar - מרשה שימוש באופרטורים בצורה "מאגניבה" ואינטואיטיבית: דוגמא בשקופית 3.

חוקיות

ננסה לתת לאופרטורים התנהגות צפויה - + לא יעשה חילוק, וכד'.
כיצד? נבדוק כיצד האופרטורים פועלים על איברים פרימיטיביים, וננסה לחקות את ההתנהגות.

דוגמאות לשימוש - שקופית ?.

אופרטור =

דורס את אופרטור = הדיפולטיבי, אשר פעולתו: ביצוע העתקה "עמוקה" של אובייקט.
אופרטור זה יבוא בדר"כ עם copy constructor (נדבר על כך בהמשך) ו־destructor.
איך מגדירים? דוגמאת סינטקס בשקופית ?:
השורה הראשונה של האופרטור מונעת בעיות במידה ו־a=a.

סינטקס

כיצד מפעילים את האופרטור? שני צורות:

- הצורה הלא אינטואיטיבית - לשלוח לאופרטור את האובייקטים, למשל object1.operator=.object2;.
- הצורה האינטואיטיבית - שימוש בסינטקס טבעי, למשל object1=object2;.

דוגמא - מספר מרוכב, הבנוי משני חלקים: הממשי והמדומה.
בדוגמא ממומשת פעולת מינוס (אונרית - פועלת על אובייקט יחיד) ופעולת חיבור (בינארית - פועלת על שני אובייקטים). השינוי בא לביטוי בחתימה באופן הבא: לפעולת המינוס לא מעבירים אובייקט, לכן יכולה לפעול רק על האובייקט הקורא לפעולה, ופעולת החיבור מקבל אובייקט נוסף (מלבד זה שקורא לפעולה).

- האם יש חוקים קבועים מי האופרטור הימני ומי השמאלי?
  - באופרטורים בינאריים, השמאלי - זה שמפעיל את הפעולה, הימני - זה שפועלים עליו.
- האם אופרטור אונרי תמיד מופיע בצד שמאל של האיבר?
  - לא, נראה בהמשך דוגמא נגדית (++).

השימוש באופרטורים באמת אינטואיטיבי, כפי שמופיע בשקופית. יש לשים לב כי יש לעשות overloading לאופרטור << במקרה הזה, שכן ה־cout לא יודע כיצד להתייחס לאובייקט Complex.

אופרטור ++

[4]אם עושים $i++$ ו־$j++$, נראה דברים שונים:

- ב־$i++$ קודם מוחזר הערך הנוכחי, ורק לאחר מכן עולה באחד.
- ב־$j++$ קודם כל מעלה באחד, אח"כ מוחזר הערך - לכן שתי ההדפסות זהות.

צריך להבדיל בין שני האופרטורים הללו - החתימות שלהם שונות (הראשון מחזיר ערך זמני - נוצר בסטאק ו"חי רק את השורה", השני מחזיר רפרנס לאובייקט הנוכחי), והמימוש יהיה שונה - יש לשים לב לדקויות אלו!
אפילו שאלה נוספת:

- $i++=9$: לא יתקפל!
- $j++=30$: יתקפל, וישכתב את j להיות 30.

תרגיל 3

Operator Overload

23/8/2012

תחילה, נמשיך קצת את הנושא של התרגול שעבר.
בדוגמא בשקופית 1 נביט בחתימה של אופרטור מינוס: לכאורה אם לא מחזירים כאן רפרנס אנו אמורים לקבל כאן שגיאת קומפילציה (כמו שראינו בשיעור). אבל זוהי דוגמא להתייחסות שונה לטיפוס בסיסי. המטרה של קונסט כאן היא למנוע להעביר קומפילציה - להתחקות אחרי ההתנהגות של המינוס, לוודא שמדובר ב־right value ולא left value.

overloading the bracket operator

בדוגמא בשקופית לא חייבים קונסט ב־overloading של הסוגריים (זה רק בשביל אופטימיזציה). מחזיקים by reference אחרת השקופית הבאה לא היתה עוברת קומפילציה.
מתי זה לא יתאים? אם ננסה לגשת ל־constArr[i] כלשהוא נקבל שגיאת קומפילציה, כי משתמשים באופרטור [] שאינו אופרטור קונסטי, לכן אי אפשר להפעיל את האופרטור הזה על המערך הקבוע.
פתרון אפשרי: שינוי לאופרטור קונסטי, אבל זה לא יעבוד עבור מערך רגיל.
במקום זאת נעשה overloading לאופרטור - נממש פעם אחד כקבוע, ופעם שניה כלא קבוע, וכאשר נפעיל את האופרטור, האופציה הנכונה תבחר.
נביט בדוגמא של מטריצה. היינו רוצים לגשת עם האופרטור [][]. אבל, אין operator overloading לאופרטור [][]. על כן, שתי אופציות:

- נעשה overloading לאופרטור () שיכול לקבל שני משתנים.
- פתרון נוסף מתקדם (פירוט במסמך חיצוני): שימוש ב"שני" אופרטורים של סוגריים מרובעים, שיפעל על אובייקטים שונים ואז יוכל לממש את מה שאנו מבקשים. אין חובה להשתמש בפתרון זה במסגרת הקורס, אך הוא אהוב על מתכנתי cpp "אמיתיים".

[4]שאלה אהובה בראיונות עבודה, מסתבר.

המרות

ראינו המרות exlicit והמרות implicit. נראה מספר דוגמאות.

דוגמא ראשונה

דוגמא ראשונה - מה יקרה בפועל בכל שורה ב-g()?

1. יקרא מיד הקונסטרקטור שמקבל integer (אופטימיזציה של הקומפיילר), זאת מכיוון וכאן יוצרים את a1.
2. אותו דבר.
3. באמצע התוכנית, לכן יקרא אופרטור = המצפה לקבל מצד ימין טיפוס מאותו סוג, על כן תקרא קודם כל ההמרה ולאחר מכן האופרטור.
4. יקרא הקונסטרקטור שמקבל integer - המרה, ולאחר מכן ישלח את האובייקט הזמני ל-f, ולבסוף האובייקט הזה ימות.

דוגמא שניה

בדוגמא לדוגמא מהשיעור - עלול להיווצר וקטור זמני!

דוגמא שלישית

כמו הדוגמא הראשונה, אבל הקונסטרקטור עם ה-integer הוא explicit.
מה יקרה עכשיו בכל שורה?

1. לא תתקמפל כי לא קריאה explicit לקונסטרקטור.
2. תתקמפל - יוצרים אובייקט בצורה explicit.
3. לא תתקמפל - אין A בצד ימין, וההמרה היא לא explicit.
4. כנ"ל.
5. יתקמפל - שכן יש קריאה לאופרטור המרה ל-A (שקורא לקונסטרקטור מ-integer).

המרה המוגדרת על ידי המשתמש

דוגמא - מימוש בקלאס A של המרה מ-double ל- A על ידי קונסטרקטור ומ- A ל-double על ידי אופרטור המרה ()double.
דוגמא שניה - מספר רציונלי המממש המרה מ-double ל-int ואופרטור חיבור בינארי.
מה יקרה בקוד בשקופית ??
הצעה: y יהפוך לרציונלי, יעשה חיבור רציונלים, ולבסוף תעשה השמה על ידי downcasting לאינטג'ר.
אבל - זה לא ירוץ!
יש שני מסלולים אפשריים: המסלול הנ"ל, ומסלול נוסף שיהפוך את x לאינטג'ר ויבצע סכימה של שני אינטג'רים. מכיוון והקומפיילר לא יכול לקבוע אילו משני המסלולים להריץ, תהיה שגיאת קומפילציה.
מה היה קורה אם היינו משתמשים ב-operator+, אז כן היה מתקמפל.
פתרון אפשרי נוסף - להגיד ספציפית מה לעשות.

פונקציות חברות וקלאסים חברים

פונקציות לא של הקלאס, אך יכולות לגשת לחלקים הפרטיים של הקלאס. עקרון זה מפר עקרונות שאנו מכירים, אבל אנו נראה כי זה שימושי ולעיתים אף הכרחי.

מימוש בעזרת המילה השמורה friend.

דוגמא - המשך של מימוש מספר מרוכב. ראינו כבר פונקציות חברות - למשל האופרטור << ו־>>. זה מאוד שימושי להדפסת נתונים. באופרטורים הנ"ל מוחזר רפרנס כדי שנוכל שרשר הדפסות.

נשים לב שכאופרטורים בינאריים אפשר להגדיר כ־friend, או להגדירם באופרטורים אונריים בתוך הקלאס עצמו - אין חוקים מוגדרים לגבי איזה אופציה לבחור. המופיע בשקופית יופיע בקובץ ה־h של הקלאס. במימוש עצמו אין צורך לכתוב friend.

חברות יכולה להיות גם בין קלאסים - למשל בדוגמא ל־binary tree יש גישה לכל האיברים הפרטיים של ה־Node.

תרגיל 4

STL - Standart Template Library

28/8/2012

STL הוא בעצם אוסף של קבצי h שנוסיף אותם כשנצטרך, ומאוד יעיל להשתמש בהם. הוא כל כך חשוב לשפה שהוא אפילו מוזכר ב־C++ Standart. השתמשנו כבר בהרבה דברים שבו - מפות, וקטורים, וכו'.

Stream

סטרים הוא צורה אבסטרקטית לביטוי של אינפוט מסויים. לרב הוא יהיה מקושר למשהו פיזי, ויהיה קשור למקלדת, עכבר וכו'.

בעבר עבדנו עם הספריה <iostream>. היא מכילה הרבה קלאסים טמפלייטים ואתחולים שלהם. כמו כן מכילה "מניפולטורים" - למשל, אמרנו ש־cout הוא בעצם באפר - כשנרצה להכריח לכתוב למשל משתמשים במניפולטור flush או endl להדפסה ומעבר שורה.

המשפחה של IOstream מופיעה בשקופית 6. נשים לב - למשל cout הוא אינסטנס של קלאס מסוג Ostream.

>> הוא פשוט אופרטור של Ostream, ולכן הוא עובד על cout. יש לו מימוש לטיפוסים בסיסיים, והוא מחזיר ... כדי שנוכל לשרשר הדפסות.

cerr, clog לא מומלצים לשימוש, הם גם מדפיסים למשך, הראשון כותב מיידית, והשניה אוסף וכותב בשלב מסוים (לאחר אחד מהמניפולטורים).

istream הוא סטרים אינפוט, cin הוא מסוג זה.

בשקופית 11 קוראים עד שאין מה יותר לקרוא. מה בעצם קורה שם? ערך ההחזרה של « הוא istream כלשהוא, אבל while צריך לרוץ על משהו בוליאני. אז מה שמוגדר פה אמפליסיטית הוא המרה - קיים ל־istream אופרטור וויד כוכב, שמבצע את ההמרה בצורה אמפליסיטית. אז כאשר מתרחשת טעות ההערכה מחזירה NULL, ועל כן הערך בתוך ה־while נהיה false.

לאובייקט הזה יש כל מיני מתודות - פירוט בשקופית 12:

- אופרטור bad() שדרכה ניתן לגשת לביט מסויים, שנדלק בעת טעויות מסויימות.
- אופרטור fail() - דומה לקודם אבל עם הרחבה.
- eof() מחזיר true אם הקובץ הפתוח לקריאה הגיעה לסופו.

- אופרטור good().
- clear() מאפסת את כל אלו שרשמנו מעלה. ממבט ראשון זה מוזר - למה צריך בכלל דבר כזה? שקופית 13 - מכיוון והריצה מתרחשת גם כאשר מגיעים לסוף הקובץ, בלי clear נקבל טעות בשורה לאחר מכן, שכן ה־good() יהפך ל־true כשמגיעים לסוף הקובץ, ואז לא תיתכן כתיבה.
  - נשים לב בקוד זה לפתיחת הקובץ - לא אינטואיטיבית, זו דרך לפתוח את הקובץ וגם לעשות append בסוף.
  - יש לסגור את הקובץ בסוף.

מתודות נוספות לעבודה עם קבצים - שקופית 14. יש לשים לב כי יש מתודות השונות ל־ostream ול־istream.

שימוש בסמן

יש פקודות המחזירות את מיקום הסמן בקובץ, ויש פקודות המאפשרות לקפוץ קדימה ואחורה בקבצים.

למשל הקוד בשקופית 17 סופרת כמה בתים יש בקובץ: מחסירים בין מיקום הקובץ בהתחלה לבין מיקום הקובץ בסופו. זה מאוד low level, אבל יכול להיות מאוד שימושי.

אאופפוט של קונטיינרים

ראינו קצת בשיעור - נביט בדוגמא בשקופית 19:

- האופרטור >> מיועד להדפיס וקטור של אובייקט כלשהוא.
- נשים לב - begin() של הוקטור, לא האיטרטור.
- it* הוא מסוג T.
- נשים לב לשימוש ב־out ולא ב־cout (שהוא אתחול מסויים כך שהסטרים יהיה מכוון למסך).

בשקופית 20 - עבור ליסט, מאוד דומה.

מה מסובך ושונה? מה שעושים בשקופית 21: איזה אופרטור >> מופעל? זה של list! האובייקט שמתקבל בו הוא וקטור, ואז האופרטור של הוקטור. זוהי הדגמה של השימושיות הרבה של operator overloading.

תרגול 5

עוד על STL

קונספטים

30/08/2012

רב הטעויות בשימו שמגיע מחוסר הבנה של מחבר הטמפלייט למשתמש. נרצה מנגנון שידאג לשימוש נכון. ספויילר: לא קיים כזה, אבל יש דרכים לעקוף, נראה בשיעור מתקדם.

הרעיון של קונספט: קונספט הוא הסכם בין מחבר הטמפלייט למשתמש. שימוש כזה אמור להסביר למשתמש איפה הוא טעה ולמה.

בשקופית 3 - אם אין תמיכה באופרטור $<$, אם נשתמש נקבל שגיאת קומפילציה.
החוזה מופיע בתיעוד, לכן צריך לקרוא "כמו שצריך".
דוגמאות לקונספטים:

- equality comparable - סוגים עם אופרטור $==$.
- less than comparable - סוגים עם האופרטור $<$.
- assignable - סוגים עם אופרטור $=$ וקופי קונסטרקטור.

Boost

ספריה שמשתמשים בה הרבה לבדיקת קונספטים. עוד על הקונספטים - בתיעוד של הספריה.
דוגמא לשימוש - שקופית 8.

קונספטים של STL

האובייקט המשתמש בו אמור להיות בנוי בצורה גמישה לעבודה עם איטרטורים.

קונטיינרים

מחלקות שאמורות להכיל אובייקטים אחרים - כגון וקטור, רשימה, סט, וכו'.
אלו קלאסים טמפלייטים שיכולים להחזיק את האובייקט המבוקש. מניח כי האלמנטים הם assignable.
שני סוגים עיקריים:

- קונטיינרים סדרתיים. - כגון רשימה, וקטור, ו־deque.
- קונטיינרים אסוציאטיבים - כגון map ו־set.

פירוט על כל קלאס בשקופית 11. הרעיון: אם צריך בתוכנית תור, מחסנית, מערך, וכו', לפני שכותבים לבד משתמשים ב־stl לשם כך.

אלגוריתמים

אלגוריתמים נפרדים מהקונטיינרים. משתמשים בהם כדי לבצע מניפולציות על המידע המוכל בקונטיינרים. הם נפרדים, כלומר ניתן להשתמש באלגוריתמים הללו על כל קונטיינר. למשל, הפונקציה reverse, בה אפשר להשתמש על (כמעט) כל סוג. ממנה מוחזר מבנה נתונים מסוג איטרטור, לא רשימה, וקטור וכו'.

איטרטורים

הוא בעצם קונספט שיש לנו ציפיות ממנו -הכללה של פוינטרים. פוינטרים בעצמם הם איטרטורים. דוגמא - בשקופית 13. צריך גם איטרטור קונסטי וגם לא, כדי שנוכל לגשת גם לאובייקטים קונסטיים.
יש משפחה שלמה של איטרטורים. הם עוזרים לנו להפריד בצורה "יותר טובה" בין אלגוריתמים לבין קונטיינרים.

וקטור

נכיר את Vector<T> - מערך סדרתי של אלמנטים מסוג T. מאפשר גישה רנדומית, ויכול לגדול באופן דינמי. יש הבדל בין size לבין capacity - יכול להיות כי המימוש עצמו בוחר להקצות מלכתחילה יותר מקום ממה שהתבקש.

דוגמא לסינטקס של יצירת וקטור - שקופית 16 והלאה. אפשר להפעיל וקטור שיהיו מוכלים בו אובייקטים, במקרה כזה משתמשים באותו הקונסטרקטור לכל האובייקטים הללו.

סיבוכיות זמן

- הכנסה והוצאה של איברים בוקטור בסוף - amortized = ממוצע. כאן הממוצע הוא זמן קבוע.
- הכנסה והוצאה של איברים בהתחלה או באמצע - זמן לינארי.

גישה לאלמנטים

לוקטור יש סט נוסף של גישות - אופרטור at המקבל אינדקס. למה צריך גם אותו וגם את []? עם סוגריים מרובעים, גישה מחוץ למערך יתן משהו, עם אופרטור at אפשר לתפוס את האקספשן.

אפשר לבדוק את הגבולות בעזרת DEBUG mode במהלך העבודה - סינטקס בשקופית 21, וכן יש אתרים נוספים עם מידע בשקופית 22.

קונטיינרים אסוציאטיביים

תומכים בהבאה יעילה של אלמנטים בהתבסס על מפתחות. מימוש טיפוסי - פונקציות האש ו-... (שקופית 23).

Function Object

בסט - סט של מפתחות ייחודיים.

הקונטיינרים משתמשים באופרטור קטן כסדר הדיפולטיבי. אפשר להגדיר אופרטור אחד לפיו יוגדר הסדר.

Function Object הוא אובייקט המתנהג כמו פונקציה, בו משמשים כדי להגדיר סדר בקונטיינרים מסוג זה. סינטקס: מגדירים קלאס, ואז מכריזים עליה כאובייקט, ומשתמשים באובייקט זה.

דוגמא בשקופית 27: אם יש פונקציה כזו - תהיה שגיאת קומפילציה, לא ברור אם מדובר באובייקט או בפונקציה. שתי דוגמאות להפעלה למטה. השורה האחרונה מגדירה את הסדר החדש.

שקופית 28: צריך לשים לב למה שהקומפיילר חושב שעשינו במידה ויש טעות.

שימוש בקומפרטורים - שקופית 29: סט מקבל קומפרטור, הקומפרטור הדיפולטיבי משתמש ב-less (יודעים זאת כי שני האתחולים למעלה אותו דבר).

מדוע משתמשים בקלאסים כ-function objects?

מקבלים את הכוח של קלאסים - ירושה, לבצע פרמטריזציה,...

באתר יש מצגת לקריאה עצמית!!!! תהיה עליה שאלה בבחינה..