תכנות מתקדם 2 – הרצאה

מס' קורס : 89-211

מרצה : יריב טל

תוכן עניינים

®

הרצאה 1 (25/02/14)

מה זה תכנות מתקדם? דברים שלא עשינו עד היום. נראה בקורס בעיקר דברים מהתעשייה. הקורס יהיה ב ++C. נעסוק גם בדברים שהם לא תכנות אלא אדמיניסטרציה מכיוון שזה חלק מהתעשייה. בסמסטר הקודם למדנו בעיקר איך דברים עובדים, בסמסטר הזה נלמד בעיקר איך לכתוב קוד יותר טוב. מהו קוד יותר טוב? יעילות, קריאות, קל לתחזוקה, גנרי ועוד... הבעיה היא שאנחנו יכולים לבנות קוד שיהיה טוב מאוד רק באחד מהפרמטרים.

למה יש שפות תכנות שונות?

- השפה שנכתוב בה היא השפה שבה נמדל את הרעיונות שלנו.
- למנוע באגים, לכל שפה יש את היתרונות שלה במניעת באגים.
- שפה נותנת משמעיות שונות.
- בגלל שאנחנו בני אדם – צריך להסביר למחשב למה אנחנו מתכוונים.

נשתמש ביכולות של השפה כדי למנוע מאתנו לעשות באגים ב C למשל זו שגיאה לוגית נפוצה לכתוב if(i=0) ולכן נתרגל לכתוב if(0=i) והקוד לא יתקמפל כך השפה תגן עלינו. נוכל גם להוסיף מידע כדי למנוע טעויות, אם למשל אנחנו יודעים שאנחנו רוצים משתנה למס׳ הסטודנטים נבנה אותו unsinged ואז הוא לא יוכל להגיע למספר שלילי.

<u>עלות האיכות</u>

האם שווה לעשות את כל זה בשביל קוד טוב יותר? כי לכל דבר יש מחיר גבוה. בתוכנה בד״כ נמדד בזמן ואת האיכות משאירים רק אם נשאר זמן.

מהי איכות? התאמה לשימוש, התאמה לדרישות, ללא דפקטים.

מדדי איכות לדוגמא:

- כמה פעמים התקשרו למרכז תמיכה
- יעילות ברמה שלא יהיה לא יעיל ברמה גדולה מאוד
- כמה זמן לוקח ללמוד את התוכנה
- זמן עד לכישלון יש אפליקציות סלחניות יותר ויש כאלה שלא (קוצב לב למשל).

איכות לוקחת זמן? **בגרף שבמצגת** אפשר לראות שלא, מי שכתב מהר יש לו פחות באגים.

מה העלות של חוסר איכות? כמה שהבאג יהיה מוקדם יותר (למשל בדיזיין) העלות שלו גדולה יותר כך שככל שגילינו אותו מאוחר יותר העלות גדולה יותר. גדל בזמן אקספוננציאלי. כמה שנתפוס את הבאג מוקדם יותר זה זול יותר!

אדווארד דמינג – אבי האיכות. תורתו: רוב הזמן ההנהלה אשמה, הם הגורמים לחוסר איכות. להגדיל את האיכות יגדיל את הפרודוקטיביות ולכן משתלם יותר.

פיליפ קרוסבי – אמר שכדאי למדוד חוסר איכות במקום איכות. כדאי להביא להנהלה נתונים על חוסר איכות מכיוון שאז ישקיעו באיכות. הסטנדרט צריך להיות 0 באגים! אין סלחנות גם לבאג של 1 לאלף מכיוון שתמיד עדיף לעשות נכון

מהתחלה ואז זה יהיה זול יותר. הרעיון העיקרי שלו: "איכות זה חינם. זו לא מתנה, העלות של חוסר איכות היא המחיר האמתי, חוסר איכות עולה יותר מאיכות".

איך משפרים איכות תוכנה?

- מגדירים זאת כמטרה.
- צריך תהליך מפורש של איכות תכונה, מה עושים בשביל זה.
- אסטרטגיה של בדיקת תוכנה.
- אסטרטגיה של מציאת באגים ומעבר על הקוד.
- תהליך הפיתוח שכולל כמה שלבים משלו.

תהליכים שונים מורידים את הבאגים בקוד ברמות שונות (הטבלה במצגת).

איכות היא בחינם גם בתוכנה. אם לא נשקיע מראש באיכות נצטרך המון זמן לדיבאג ותיקונים לעומת שאם נשקיע נוריד משמעותית את זמן הדיבאג והתיקונים.

ב IBM גילו שהפרויקטים שהיו עם הכי פחות באגים הם אלה שהסתיימו הכי מהר. כשמחפשים באגים תשתמשו לפחות ב2 שיטות כדי לייעל את התהליך.

שיטות שונות עולות מחירים שונים. יש שיטות למניעת באגים שלא עולות כלום ועד פרוטוטיפ שהוא מאוד יקר.

תכנות אקסטרים XP, לא הצליח אבל ממנו נולדה agile.

עד עכשיו רק הראינו למה כדאי לכתוב קוד יותר טוב. נעבור לשינויים בקוד.

<u>Coding convections:</u>

איך נהוג לכתוב קוד בחברה. המטרה: לחסוך זמן תחזוקה ע"י מניעת באגים ושיפור הקריאות. קריאות היא עניין סובייקטיבי ולכן רוצים שתהיה אחידות בחברה.

קונבנציות נפוצות: לא לסמוך על הסדר של האופרטורים ולשים סוגריים, קבוע משמאל ב if, לבדוק ערכים של פונקציות בולאניות (למשל הכנסה לווקטור), להשתמש בספריות הסטנדרטיות, לאתחל תמיד פויינטרים ל NULL, לדאוג שאין אזהרות, לא להשתמש ב malloc ב C++, דיסטרקטורים תמיד וירטואליים, להשתמש תמיד ב }{ גם בתנאי או לולאה של שורה אחת.

קונבקציות בשמות משתנים:

CONSTANT_VALUE, ClassName, variableInCode, m_memberVariable, s_staticVariable, g_globalVariable

יש גם קונבנציות לא טובות: לשים בתנאי עם או את זה שכמעט תמיד נכון ראשון יכול להיות שזה הורס את הקוד אם הוא אמור להיות השני, לבדוק לפני כל חלוקה אם אנחנו מחלקים בטעות ב 0 גורם לקוד להיות ארוך ולא קריא. בגדול לא להקשיב לקונבנציות בשם היעילות אלא רק כדי למנוע באגים או בשל הקריאות.

גוגל מפרסמים את שלהם עם הסבר.

הרצאה 2 (04/03/14) Defensive Programming

Defensive programming – שיטת תכנות כדי להפחית באגים.

מחקר משנת 1975 מראה ש 3 המתכנתים הטובים עשו 40% פחות באגים מ 3 הגרועים למרות שלשניהם היה את אותו הותק והניסיון. אז מה גורם לזה? ניסיון? ריכוז? תשומת לב לפרטים? כשרון? זה לא באמת משנה, נלמד איך לפתח את זה.

Defensive programming יש 3 משמעויות:

1. להניח שקיבלנו קלט גרוע תמיד ולכן לבדוק אותו תמיד.
2. הפרדה בין מקומות שהקוד בטוח למקומות שלא ושיטת ההגנה שלהם.
3. שיטות למניעת באגים – נדבר על שיטה זו. נקרא לזה תכנות מודע באגים.

כישלון הוא לא אופציה, הוא מובנה בתוכנית. כל תוכנית עלולה להיכשל.

יש מקומות בקוד שמועדים לבאגים. אם נהיה מודעים למקומות הבעייתיים האלו נדע להפחית את הבאגים.

מה כ"כ גרוע בבאגים? הם גוזלים המון זמן. דיבגניג זה רע. בראין קרדיגם: "דיבגינג הוא פי 2 קשה מלכתוב את הקוד מלכתחילה". זה תהליך קשה, איטי ומתסכל.

אנחנו כבר לא מנסים להימנע מבאגים? אבל כדי להספיק הכול בזמן אנחנו עושים קיצורי דרך וכך נוצרים באגים. למשל העתק-הדבק, זה קל ולא צריך לחשוב אבל זה מגדיל את זמן הדיבאג.

מניעת באגים בעזרת שימוש במשתנים:

1. תמיד לאתחל משתנים מיד אחרי ההגדרה - כדי להגן עלינו הקומפיילר נותן אזהרה כשלא מאתחלים משתנה. כדאי להגדיר משתנים כמה שקרוב לשימוש. בשמות משתנים תמיד כדאי שיהיו כמה שיותר שונים והשינוי לא יהיה באמצע אחרת מתבלבלים בקלות. כדי שלא נתבלבל נגדיר סקופ (בלוק) ובו נשתמש במשתנים שאנחנו צריכים רק שם ואז בכל מקום אחר שנרצה להשתמש במשתנה הזה נקבל שגיאת קומפילציה.
2. נגדיר משתנה לבלוק המינימלי האפשרי – לכן נוסיף בלוקים מתאימים והערה אחרת, לא נדע מה מטרת הבלוק. דרך טובה ואלגנטית יותר בעזרת פונקציות.
3. תגדירו משתנים רק כשאתם יכולים לאתחל אותם – נשים אותם בבלוק המתאים, בקומפיילרים לא עתיקים לא חייב להצהיר בתחילת הבלוק ב C. אם מאתחלים משתנה כך ששולחים את הכתובת שלו לפונקציה והוא מאותחל שם זה בסדר אם הפונקציה מגיעה מיד אחרי ההצהרה, אחרת נצהיר עליו יותר מאוחר.
4. להשתמש ב struct – כך נוכל להחזיר כמה ערכים מפונקציות ונחייב לאתחל את המשתנים בעזרת הקונסטרקטור שלו (אפשרי גם ב C). בצורה כזו לא נשתמש בפויינטרים כדי להחזיר כמה ערכים מפונקציה ולא נשכח להחזיר את הערך.
5. להפוך אזהרות לשגיאות – אפשר להפוך את זה ל error ע"י כך שנרשום #pragma warning (num, error) כאשר num מייצג את מס' האזהרה.

מניעת באגים בדגש על הקלט:

החוק הראשון: לא להניח שום דבר!!!

עבור כל קלט נגדיר מהו קלט חוקי, איך נבדוק אותו ומה נעשה במקרה של שגיאה.

לא להניח גודל של טיפוס פרימיטיבי – יכול להשתנות.

הרצאה 3 (11/03/14) Testing

אנשים שונאים שינויים. למה? אחרי השינוי מגיע הרבה בלגן – שלב הכאוס לאחריו אנחנו מבינים שהשינוי היה טוב (או לא) עד שנגיע לרמה שאנחנו שולטים בסטאטוס החדש. גם בתוכנה קשה לנו עם שינויים. שונאים במיוחד לשנות קוד קיים, נתבקש תמיד לעשות מינימום שינויים בקוד מסיבות שונות, בעיקר מכיוון שאנחנו מפחדים להשפיע על חלקים אחרים בקוד שפתאום לא יעבדו.

בדיקות אוטומטיות – במקום שאדם יבצע את הבדיקות המחשב יריץ סדרה של בדיקות, כך לא נפספס דברים. כתוצאה מכך נוכל לפחד פחות משינויים בקוד!

Unit testing – נבדוק את היחידה הקטנה ביותר האפשרית (ברמת המחלקה). כשעושים שינוי בפונקציה אחת נבדוק את כל מה שהגיוני לבדוק איתה או מושפע ממנה. לא צריך לבדוק את כל הפונקציות במחלקה, יש פונקציות שלא משפיעות אחת על השנייה (בד"כ מראה על דיזיין לא נכון, הפונקציות האלו היו צריכות להיות במחלקות שונות). מתי עדיף להריץ את הבדיקה? אחרי כל שינוי, מומלץ בכל ערב! כך נעקוב על הקוד בכל שלב. נעשה ע"י מחשב.

Integration testing – נשלב כמה מחלקות ביחד ונבדוק אותם. נעשה ע"י מחשב.

System testing – בחינה של המערכת כולה בפרמטרים שונים. אדם עושה את כל הבדיקות האלו (מחלקת QA).

דוגמא ל Unit testing מחלקה שמראה את הזמן (שקף 12).

נרצה לבדוק בדוגמא בהתחלה לבדוק שאכן הקונסטרקטור מאתחל כפי שצריך. איך נבדוק? אפשר עם assert – מופעל על ביטוי בולאני, אם true ממשיך אחרת עוצרת את התוכנית. נבנה לכל בדיקה פונקציה בפני עצמה. הבדיקות צריכות להיות רחבות ככל שניתן ולכן ננסה לבנות אפשרויות אוטומטיות כדי לבדוק כמה שיותר (כמו הלולאה בדוגמה). חשוב לבדוק גם ערכים לא חוקיים, במקרים שיש ערכים לא חוקיים נזרוק exception. טסטים יכולים להכיל באגים ולכן במקרה שטסט נכשל חשוב לבדוק גם אותו עצמו אם אכן היה צריך להחזיר שגיאה.

כשנרצה לבדוק את הפונקציה שמחזירה את הזמן עכשיו נוכל להשוות את הפלט שלה מול הפונקציה של מערכת ההפעלה שקובעת מה הזמן הנוכחי, הבעיה היא שעובר זמן בין 2 הפונקציות ולכן יתכן שתעבור שניה. קשה לנו לבדוק בגלל תלות במערכת ההפעלה, לכן נוכל לבנות מחלקה מתווכת ולבדוק מולה ואז לא תהייה תלות במערכת ההפעלה – לטכניקה הזו קוראים dependency injection (חשוב מאוד לדיזיין ועוזר בשינויים עתידיים בקוד). כך נוכל בעצם לשלוט בזמן ש now רואה ולבצע בדיקה ללא כל בעיה. עדיף שיהיה בתור פונקציה ולא משתנה מהמון סיבות, למשל אם נרצה להשתמש בו בעוד מקומות וגם לא מכריח אותנו לשלוח אותו כפרמטר לפונקציה ואז לא נחשוב על מקרי קצה.

קוד צריך להיכתב כך שיהיה אפשר לבדוק אותו – עוזר לכתוב דיזיין יותר גמיש ונכון.

נרצה גמישות רבה בתכנון הבדיקות, יש המון ספריות שעוזרות לנו בביצוע הטסטים, נותנים בעיקר מעטפת אדמיניסטרטיבית, מוריד עבודה שחורה ונותנים להתמקד בבדיקות.

כדאי לעשות טסטים! משפר את האיכות, משפר את הדיזיין והקוד, מוצא את הבעיות מוקדם, זול יחסית.

The Chaos Monkey – תוכנה שמפריעה לתוכנות אחרות. מכניס קלט ופעולות אקראיות כך שנוכל לבדוק המון קלטים שונים. במקינטוש בדקו ככה את המעבד התמלילים. נטפליקס השתמשו בו אחרת, רנדומלית הרג תוכנה וכך בדקו אם גם אחרי שמשהו נסגר עדין המערכת עובדת.

Test Driven Development:

אחרי שראינו שטסטים זה חשוב, לא נבנה אותם בסוף כתיבת הקוד אלא נבנה אותם בשלב הדיזיין וכך נבנה את הקוד שלנו מונחה בדיקות.

טסטים בודקים בעיקר את הדיזיין ולכן קודם נכתוב טסטים ואז את הקוד, כך נהנה מהיתרות של הדיזיין מההתחלה.

כשלוקחים את השיטה לקיצוניות: נכתוב טסט על פונקציה בעלת מימוש ריק, הפונקציה תיכשל נכתוב כמות מינמלית של קוד, אם עבר נכתוב עוד טסט אחרת נתקן ונתחיל מחדש.

אחרי שהבדיקה כתובה הרבה יותר קל לחשוב על איך לממש את הפונקציה (דוגמה לשינוי אזור זמן במצגת).

מתי כדאי לעשות תכנות מונחה בדיקות? בפרויקטים חדשים, אחרת קשה מאוד להפוך קוד ישן לקוד מתאים לבדיקות. להתחיל להתנסות בזה עכשיו!

מעבר על הקוד (בקבוצה) מול בדיקות: מעבר על הקוד (בקבוצה) הוכח כיותר טוב.

בעיות עם טסטים:

1. נוטים לבדוק את המקרים בהם זה יצליח, צריך תמיד לזכור לבדוק גם ערכים לא חוקיים. לבדוק אפילו יותר ערכים לא חוקיים מאשר חוקיים.
2. לוודא שבדקנו באמת את כל האפשריות, ממש כל אופציה אפשרית בכל if.

בדיקות רגילות לא משפרות את איכות הקוד.

הרצאה 4 (18/03/14) Debugging and Asserting

דיבאגינג – תהליך מציאת שורש הבעיה ותיקונה.

לדבג ע"י ניחוש – לא ננסה להבין את הקוד פשוט ננסה כל מיני שינויים שנראה לנו שיעבדו (להוסיף & ב C למשל) או שנשים "פלסטר" תיקון שפשוט מטפל בבאג שמצאנו.

דיבאגינג ע"י העתקה (cargo cult) – נעתיק קוד ממקום אחר שעובד בלי להבין אותו.

דיבאגינג ע״י אמונות טפלות – נקמפל מחדש, אולי זה יעבוד. הבאג לא אצלנו אלא בקומפיילר או הווינדוס. לא עשינו כל מיני תהליכים מוקדמים. הקלט לא טוב. לכל הדברים האלה יש סיבה אמיתית וצריך לפתור אותם...

לא לדבג מכיוון שבטוחים – להיות בטוחים שהבאג דווקא במקום מסוים ומחפשים אותו שם, או ההפך הבאג לא יכול להיות כאן. לא יכול להיות שהבאג כאן כי עד עכשיו זה עבד... וכדומה.

כל הצורות האלה לא עובדות, לכן נשתמש בשיטה המדעית:

1. אספו מידע
2. נסחו תאוריה
3. תכננו ניסוי שיבדוק את התאוריה
4. בצעו ניסוי
5. חזרו להתחלה עד שתצליחו
6. בסיום תקנו את הבעיה

שיטה קשה ועמוסה. עדיף את השיטה של האוס במקום:

יש תאוריה, מתקן מיד את הבאג במקום לחכות לתוצאות של בדיקות. ליצור מידע במקום לאסוף מידע, לנסות לשחזר את הבעיה מכיוונים שונים למשל. אחת השיטות הנפוצות כדי לאסוף מידע היא להוסיף הדפסות. במקום זה ניצור מידע ע״י logging: ביצוע כל ההדפסות בצורה יותר מסודרת. יש הרבה סוגים מוכנים של לוגים שהם בעצם כמו הדפסות רק מסודרות, שולח מידע יותר מסודר ולפי מקרים. אפשר להוציא הרבה יותר מידע כך (ובצבעים...) או לשלוח את המידע ברשת.

נשחזר את הבעיה מכמה כיוונים כדי למצוא יותר שגיאות.

ננסה לבודד את הבעיה תמיד כדי למצוא את הבעיה עד כמה שניתן.

תעשו מה שחייבים לעשות! גם אם יש לזה עלות.

העלמת רעש: באתחול משתנים נאתחל עם מספרים שאין סיכוי שישתמשו בהם בתוכנית כדי שנבדוק את ההתנהגות שלהם. כשמגדירים מערכים להגדיר מערך קצת גדול יותר ולהציב בתאים הנוספים ערכים מיוחדים ונבדוק בסוף הריצה שהערכים האלו לא השתנו, אם השתנו חרגנו מגבולות המערך המקורי.

ללמוד מהשגיאות הקבועות שלנו, כנראה נעשה אותם שוב. לכן כשיש באג נחפש את השגיאות האלו ונלמד איך לכתוב את הקוד טוב יותר בפעם הבאה כך שהשגיאה לא תחזור על עצמה.

ASSERT:

״הכלי הפשוט הכי חשוב והכי חזק כדי להבטיח נכונות של תוכנות״ זה כלי מאוד חשוב.

יש באגים שאנחנו לא יודעים עליהם ולכן נרצה שכל הבאגים יהיו גלויים.

Assert נותן לנו דרך לגרום לבאגים להיות גלויים.

שימוש נכון ב assert: רק כשנרצה להגיד כשדברים חייבים לעבוד בדרך הזו.

Assert טוב הוא כזה שיחזיר false רק במקרה ומשהו לא עבד כמו שצריך.

שימוש שגוי ב assert הוא על פרמטרים שפונקציה מקבלת, עושים את זה רק אם אין שום דרך אחרת לטפל במצב כלומר שהפונקציה אומרת שעבור ערכים מסוימים היא לא מוגדרת למשל מס׳ שלילי לפונקציה שמחשבת שורש ריבועי.

Assert יכול להחליף הערות המון פעמים, יש לו ערך של תיעוד של הערכים שנרצה לקבל עד לשלב הזה.

כש assert נכשל כלומר העולם שגוי – חייבים לטפל בזה. אם assert קורס יש באג נסתר שלא ידענו עליו לפני כן.

נכון מול עמיד: תוכנה שמעדיפה נכונות תעצור את העבודה עבור כל באג קטן לעומת זאת יש תוכנות שמעדיפות עמידות כלומר להמשיך לרוץ בכל מקרה גם במחיר של תוצאות שגויות. Assert הוא כלי של נכונות בלבד.

שיטות לעמידות: להחזיר ערך נטרלי, להחזיר את התשובה האחרונה, להחליף עם הערך החוקי הקרוב ביותר, לקפוץ למקום שעובד טוב, להתעלם מהבעיה...

Assert הוא לא תחליף לטיפול בשגיאות! נטפל בשגיאות בזמן ריצה בדרך שאנחנו יכולים לטפל נשים assert כשזה מצב שבכלל לא אמור לקרות.

Assert ב JAVA – צריך לאפשר assert באקליפס כדי שיפעל. נמצא ב edit configuration.

הרצאה 5 (25/03/14) Const

כשנבנה פונקציה לא נרצה בד״כ לשנות את הערכים שהיא מקבלת וכדי למנוע שינוים הצהרות הערכים יהיו const אחרת אין התחייבות שהערכים לא ישתנו...

ניתן לשנות const ע״י casting עם Template, כך שלא הכול באמת מוגן. אבל בד״כ אם צריך const שכזה זה בד״כ באג בדיזיין. ולכן אם לא רואים const בהצהרה על משתנה של פונקציה צריך להניח שהיא תשנה את הערכים. Const עוזר למצוא באגים וגם לדוקומנטציה, הוא בעצם אומר לנו מה לא אמור להשתנות ואם אין אז מבחינתנו הוא אמור להשתנות.

סוגי const:

- במשתנה גלובלי – הוא בעצם קבוע רגיל.
- במשתנה סטטי – הוא קבוע של המחלקה.
- בפונקציה const – לא משנה אף אחד ממשתני המחלקה.
- משתנה מחלקה – מאותחל פעם אחת במחלקה ולא משתנה יותר.
- Const int* - ה int הוא const
- Int* const – הפויינטר const (הולכים לפי מי שה const קרוב אליו יותר).

לאובייקטים const ניתן לקרוא רק בפונקציות const. כי ברגע שהפונקציה כזאת הוא מתייחס לכל משתני המחלקה כ const ולכן לא ניתן לשנות אותם.

בהעמסת אופרטורים עבור האופרטור [ ] לעיתים נרצה רק לקבל מידע ולהשתמש בו בפונקציות const ולכן החתימה שלו גם חייבת להיות const ולעיתים נרצה לשנות מידע ואז לא יוכל להיות const, ולכן יוצרים 2 פונקציות אחת שהיא const ואחת רגילה ובכל אחת נשתמש לפי הטיפוס אוטומטית.

Const עובד ברמה רדודה כלומר אם עשינו const על פויינטר או רפרנס הוא ישמור שהם לא ישתנו אבל הערך שאליו הם מצביעים יכול להשתנות. ולכן כשמשתמשים בהם לא מומלץ להפוך את הפונקציה ל const כי זה יכול לבלבל כי את הערכים עצמם תוכל לשנות, למשל במקרה של וקטור של *string - נוכל לשנות את המחרוזת עצמם ולא את הכתובות גם כשהפונקציה היא const וכנראה שזה לא מה שחשב מי שמשתמש בפונקציה ורואה שהיא const.

במקרים שבהם צריך לעשות casting כדי להוריד const עבור משתנה מחלקה נוכל להשתמש במילה mutable בהצהרה של המשתנה, כך הוא לא משתתף במשחק של ה const ואפשר לשנות אותו גם כשהפונקציה מוגדרת const. נכון להשתמש בו רק במקרה שהמשתנה הוא לא ממש חלק מהמחלקה אלא רק במקרים בהם הוא משמש רק לחישובים זמניים או נוחות, אם המשתנה הוא חלק אינטגרלי מהמחלקה שאי אפשר להגיע אליו באמצעות חישוב פנימי לא נכון להפוך אותו ל mutable. משתמשים רק בשינוי שאי אפשר לדעת מבחוץ שמתקיים.

שימוש ב #define הוא רק העתק הדבק לפני תהליך הקומפילציה, ללא טיפוס. שימוש ב const הוא יצירת משתנה ממש בזיכרון שאותו אי אפשר לשנות.

בג׳אווה אין const אלא יש final שהוא קצת שונה. Final מאתחלים רק בקונסטרקטור (אחרת מאותחל ל 0). בדומה ל c++ הוא עובד רק על הכתובת של האובייקט עצמו ולא על מה שניתן ולכן אפשר גם לשנות את האובייקט.

Final על מתודה אומר שמחלקה שיורשת לא יכולה לשנות אותה. (טוב מכיוון שמרשה לקומפיילר בעצם לעשות inline).

אם בכל זאת נרצה ליצור אובייקט לקריאה בלבד בג׳אווה נוכל להשתמש ב unmodifiable ואז במקרה שירצו לשנות לא נקבל שגיאת קומפילציה אבל בזמן ריצה יקפוץ exception.

הרצאה 6 (01/04/14) Macros vs Templates

להשלים

הרצאה 7 (08/04/14) TemplatesMetaProg

Bounded Array:

ניצור בעזרת Templates מערך שאפשר לדאוג שלא יחרוג מגבולות הגזרה שלו.

כדי לא לממש דברים פעמיים אפשר לרשת מ Templates.

CRTP : מכיוון שברוב המקרים בעזרת האופרטור > ניתן להגיע לכל אופרטורי ההשוואה האחרים (<,>=,==,!=,<=) ניתן לייצר Template של כל זה ואז בהעמסת אופרטורים אחת דואגים לכולם.

Templates שקולים למכונת טיורינג ולכן הם שקולים למחשב. בקיצור אפשר לעשות איתם חישובים שונים.

דוגמא ל if בעזרת Template : נבנה 2 Templates שבאחד מהם נגדיר שהערך החזרה הוא התוצאה במקרה שנכון ובשני התוצאה כשהתנאי לא מתקיים כאשר נגדיר לאחד מהם שיכנס לטמפלט רק אם קיבל false/true וכך בזמן אמת יכנס למקום הנכון לפי התנאי.

גם רשימה מקושרת אפשר לבנות בצורה דומה, מגדירים struct של ראש וזנב ובונים Template שאליו שולחים את הפרמטרים המתאימים. כשבונים את הרשימה ממש נשלח כפרמטרים את האבר הראשון ובתור הזנב שלו נעשה Template לאיבר השני וכן הלאה... כלומר אם הסדר ברשימה הוא a,b,c List : <a,<b,<c>>>. חישוב אורך רשימה מקושרת מחשבים באופן כאילו רקורסיבי בעזרת Templates באופן דומה כך שעושים ייחוד לפי הטיפוס.

Tuple : כמו שניתן להגדיר זוג אברים (a,b) נרצה בעזרת Templates ליצור משהו אוטומטי לכל מס' של אברים.

נניח שנבנה Tuple של 5 אברים, כשנקצה Tuple של 2 אברים נגדיר אחד חדש אבל נשתמש בספזילציה וב 3 הפרמטרים שאנחנו לא משתמשים נגדיר מחלקה חדשה (NullType) שלא עושה כלום... פשוט כדי שלא יכנסו לזה אף פעם עם 5 טיפוסים. וכן הלאה לכל גודל של Tuple שנרצה. זה ארוך ומעצבן.

נוכל להשתמש ב Tuple ארוך ולהשתמש בערכי דיפולט (בעזרת = בהצהרה) כאשר הערך הדיפולטיבי הוא NullType. וכך נוכל להגדיר לכל מס' אברים עד לכמות המקסימלית.

הבעיה כרגע היא שאנחנו "מבזבזים" הרבה זיכרון סתם כי אנחנו לא משתמשים במחלקות האלו. ב c++ יש טריק שמחלקה ריקה שיורשים ממנה לא תופסת עוד מקום בזיכרון ולכן נדאג שה Tuple יירש מכל מה ששולחים אליו ואז לא יתפוס סתם מקום. אבל במקרה ויש לנו טיפוס פרימיטיבי שלא ניתן לרשת ממנו למשל int? ניצור מחלקה שתכיל את ה int בעזרת Template ואז נוכל לרשת, בעצם נעטוף את ה int במחלקה. אבל זה בעיה לרשת מ 2 מחלקות שהן מאותו טיפוס מכיוון שהקומפיילר לא יודע למי מהם הכוונה ולכן נצטרך פתרון אחר.

נשתמש ב Templates כדי ליצור בצורה רקורסיבית Tuple מתאים. באופן דומה ליצירת הרשימה המקושרת.

בג'אווה אין Templates, יש generics שמשתמשים בהם בעזרת <>. משתמשים בזה למשל ברשימות, כשמגדירים רשימה מודיעים מראש מי הטיפוס שיכול להיות בה בעזרת <>. מאחורי הקלעים הקומפיילר מקמפל את זה בכל זאת בתור אובייקט

גנרי אבל כשמוציא את זה הוא עושה casting לטיפוס המתאים השיטה הזאת נקראת type erasure.

Template לעומת generics:

- Templates יוצרים קוד בכל פעם שאנחנו מכניסים פרמטרים, לעומת generics שהקוד נוצר פעם אחת.
- ב Templates הפרמטרים יכלו להיות כל דבר, לעומת generics אי אפשר פרימיטיביים.
- ב Templates צריכים את הקוד בכל פעם מחדש לעומת generics שבה אפשר להשתמש במה שכבר קומפל.
- ב Templates יש ספזיליזציות לעומת generics שאין.
- ב Templates ש ערכים דיפולטיבים וב generics אין.
- Templates הם העתק-הדבק לעומת generics שהם מונחה עצמים.

הרצאה 8 (29/04/14) Functors

לכל אובייקט יש את ה state שלו, הערכים של כל המשתנים שלו.

סטרילזציה – לקחת את משתני המחלקה ולרשום אותה במקום כלשהוא.
דה-סטרלזציה – לקרוא את ערכי המחלקה ממקום מסוים. למשל לקרוא ולכתוב מקובץ או מהרשת. יש כאן הרבה דברים דומים שחוזרים על עצמם אם רק שינוי של שם הפונקציה ולכן נרצה פונקציה שתדע גם לקרוא וגם לכתוב.

נוכל לבנות Template ופונקציה שעושה את שניהם ומקבלת כפרמטר אם לכתוב או לקרוא, אבל זה יוצר סרבול של הקוד עם הרבה if וזה בעייתי. נוכל לעשות Template לפי הסוג של ה stream שמתקבל וכך אם נקבל את הפונקציה עם stream של קריאה היא תקרא ועם stream של כתיבה היא תכתוב. אנחנו עדין לא יכולים להתמודד עם ביצוע קריאה/כתיבה עבור גרסאות שונות שבהם נרצה שהפעולה תתבצע אחרת. בעצם היינו רוצים שיהיה דרך להעביר את הפונקציה שנרצה שתקרה ולכן יש לנו:

פויינטרים לפונקציות:

```
int (*pt2Function)(float f, char c1, char c2) = NULL;
```

יש כאן משתנה בשם **pt2Function** שהוא פויינטר לפונקציה שמחזיר int ומקבלת float ו 2 char. ומכיוון שזה פויינטר אנחנו מאתחלים אותו ל NULL.

כדי שיהיה נוח לעבוד עם הפויינטרים האלו עושים typedef שנראה כך:

```
typedef int (*myfuncType)(float f, char c1, char c2);
myfuncType pt2Function = NULL;
```

חשוב לזכור ש pt2Function הוא פויינטר למרות שאין שם *.

אפשר לעשות פויינטר למתודה בתוך מחלקה ע"י שימוש ב :: לפני שם הפונקציה. פויינטר לפונקציה במחלקה טוען משהו נוסף לפונקציה גלובלית את המצביע this ולכן חייב להוסיף גם את השם המחלקה בהצהרה. כשנרצה להשתמש בפונקציה נצטרך נעשה כך : עבור מחלקת obj והפויינטר f : (obj.*f).

טבלת המרה – מערך של structs שמכילים מפתח ופויינטר לפונקציה וכך נוכל להתאים את הפונקציה המתאימה לפי המפתח בכך שנעבור על המערך ונחפש השוואה. (הדוגמה במצגת עם המחשבון)

אז נרצה לשלוח מצביע לפונקציה שתבצע את הקריאה/כתיבה אבל לא ניתן לשלוח מצביע לפונקציה Template ולכן בינתיים נראה רק על int. אבל זה מביא אותנו למצב של casting שאת זה אנחנו לא רוצים. ולכן נמצא פתרון חדש :

נשתמש באובייקטים במקום פונקציות לפויינטרים. נוכל לבנות מחלקות שכל תפקידם הוא לעשות את הפעולה האחת הזו ויהיה לכולם מחלקת אב שבה יש פונקציה וירטואלית וכך נוכל לשלוח ולהפעיל בכולם את הפעולה. כדי שיהיה קל יותר במקום לבנות פונקציה כזו נוכל לדרוס את האופרטור סוגריים ואז להפעיל כמו פונקציה – זה נקרא functor.

את הfunctor נוכל לשפר בכך שנכניס אלין כבר את ה stream שנרצה (קלט או פלט) כדי למנוע שכפול של העברת הפרמטר בכל מקום בקוד. מכיוון שאפשר לשמור במחלקת הfunctor משתנים אפשר לעשות כל מיני שינויים והגדרות בקוד בהתאם לאתחול של המשתנים האלה. בנוסף כדי להימנע מעבודה עם טיפוס אחד נגדיר את הפונקציה שמקבלת functor שהוא Template.

functor עם Template נותן ביצועים טובים יותר מאשר פויינטרים לפונקציות וfunctor שיורש ממחלקת אב משותפת.

	Function Ptr	Functor with Base Class	Functor as template argument
Flexible signature	☒	☒	☑
Template function	☒	☒	☑
Additional parameters	☒ (passed in call)	☑ (in object state)	☑ (in object state)
Has state	☒	☑	☑
Customizable	☒	☑	☑
Direct call	☒	☒	☑
Inline	☒	☒	☑

functorים ב JAVA – טכנית אין, מכיוון שאין העמסת אופרטורים. במקום זה יש אינטרפייסים שמכילים מתודה אחת בלבד.

functorים ב C# - יש את המילה delegate שזה סוג של מצביע משופר לפונקציה. שההצהרה עליו נראית כך : delegate void **MyDelegate**(int i);

סטרילזציה ב JAVA :

כדי ליצור סטרלזציה נצטרך רק לממש אינטרפייס בשם sterializable ואז להשתמש באובייקטים מוכנים של קריאה וכתיבה שגם את הקריאה והכתיבה בהם ניתן לדרוס למה שנרצה כדי שיתאים לנו.

אלגוריתמים : (לא למבחן)

אלגוריתמים שקיימים ב STL. כדי שנוכל לעבור על מס' סוגים של קונטיינרים עם אותו אלגוריתם עוברים עם איטרטור. איטרטורים עובדים על העיקרון של אריתמטיקה של פויינטרים.

הרצאה 9 (13/05/14) Exceptions

קשה לכתוב קוד שהוא exception safe, כלומר שבמקרה של exception אין באגים. כל פונקציה שכותבים צריכה להיות מוכנה לexception כי אי אפשר לדעת מאיפה זה יגיע. כשנזרק exception ועולים רמה למעלה בשלבים בכל שלב יופעל Dtor של כל מחלקה. ברגע שהופעל Dtor בעקבות exception אנחנו לא יודעים באיזה שלב זה יקרה ולכן יתכן ונשחרר מצביעים שאנחנו עוד לא יודעים מה קורה איתם וכך נשחרר כתובת אקראית וניצור בלאגן בקוד. הפתרון הוא לאתחל את כל המצביעים ל NULL וכך גם אם נשחרר אותם לא יקרה נזק. במקרים אחרים יתכן שבעקבות הexception אף אחד לא ישחרר זיכרון שהקצנו והפתרון הוא ב catch לשחרר הכול ולזרוק הלאה – זה קוד ארוך ומכוער. במימוש של אופרטור השמה = בודקים תחילה אם מנסים להשים את אותו המשתנה לעצמו כדי שלא למחוק בטעות את הזיכרון העצמי, אחרי שבדקנו מוחקים את הזיכרון הקיים ומציבים ההעתק של האובייקט החדש. (זה בעייתי כי כל if מאט את הקוד ולכן נרצה לעשות כמה שפחות.) נניח ובמהלך האופרטור השמה קפץ exception יוצא שיש לנו אובייקט חצי אפוי, חלק מהאובייקטים הוקצו וחלק לא ואנחנו מגיעים לבעיה מכיוון שכש Dtor יופעל התוכנית תקרוס. לא זורקים exception ב Dtor מכיוון שאז אי אפשר לשחרר את האובייקט, זה חוקי בשפה אבל הסטנדרט ממליץ שלא לעשות את זה. RAII – ברגע שמקצים משאב מציבים אותו מיד בתוך אובייקט שאחראי לשחרור שלו (נקרא גם Guard). אי אפשר לכתוב לכל דבר RAII ולכן יש מובנה ב stl : auto_ptr ואז יש אובייקט לכל פויינטר ויש לו דיסטרקטור שמשחרר אותו. כעת אם נשתמש בזה וקפץ exception יופעל Dtor וימחק את הפויינטר שהוא מחזיק, אבל כדי שזה לא יקרה כשאנחנו לא רוצים (כשיוצאים מהפונקציה למשל) יש ל auto_ptr פונקציה בשם release שמחזיקה את המצביע הפנימי ומאפסת את המצביע שהיא מחזירה ל NULL וכך כשיופעל Dtor זה לא יפריע וניתן להחזיק עכשיו את הערך החזרה של הפונקציה בתור הפויינטר שאנחנו רוצים. כעת גם אין בעיה של השמה עצמית כשמשתמשים ב auto_ptr ואפשר לוותר על ה if. כדי להימנע משימוש באופרטור השמה = מכיוון שלא יודעים בכל אובייקט אם יזרוק exception משתמשים ב swap לכל האובייקטים ב stl, כל ה swap ב stl לא זורק exceptions. במבחן יהיה לכתוב swap. מימוש חדש לאופרטור השמה = :

ניצור אובייקט חדש שמכיל את האובייקט שנרצה אצלנו ואז נעשה איתו swap, כך באובייקט הזמני יש הפויינטרים הקיימים שימחקו בדיסטרקטור כשנצא מהפונקציה ואצלנו יש את האובייקט החדש.

סיכום של רמות exception:

a. No-throw guarantee – התחייבות לא לזרוק אקספשן ולטפל גם ב"אירועים חריגים" לדוגמה יש אפשרות לקרוא לפונקציה new שמקצה זיכרון דינמי באופן הבא new(std::nothrow) וכך במידה ויהיה "אירוע חריג" והמערכת לא תוכל לספק לנו זיכרון במקום לזרוק אקספשן, נקבל פשוט NULL.

b. Strong exception safety – התחייבות שבמידה ויש "אירוע חריג" אז יהיה גיבוי, כלומר דברים אחרים לא ייפגעו. למשל הקצאת זיכרון נוסף, נקצה זיכרון ל buffer נוסף ובמידה ולא יהיה שגיאות נבצע איחוד או נחזיר אותו ולא נעבוד ישירות על הזיכרון המקורי וכך לא נסתכל בפגיעה בו.

c. Basic exception safety – התחייבות לטפל ב"אירועים חריגים" כלומר ייתכן ויהיה "אירוע חריג" שיגרום לבעיות. אך התוכנית תוכל להמשיך לפעול, לא יהיו דליפות, לא מובטח שכל המידע במשתנים זהו אכן המידע הנכון (שהיה לפני האקספשן). בעצם מובטח שהאינוריאנטה של המחלקה תשמר, כלומר יהיו משתני מחלקה אבל לא נוכל לדעת אם הם החדשים או הישנים.

d. No –leak safety – התחייבות שלא יקרוס ושאין דליפת משאבים וזיכרון. אין התחייבות לתוכן המצביעים או שהאינוראנטות של המחלקה ישמרו.

e. No exception safety - אין התחייבות לתקינות אחרי טיפול ב"אירוע חריג" ייתכן ותהיה דליפת זיכרון, ערכים מזובלים ולא נכונים במשתנים וכו'..

יש מחיר בביצועים כשרוצים לכתוב exception סייף.

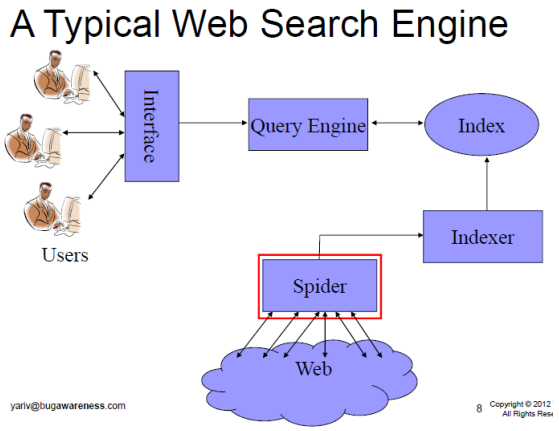
כל דבר ב stl שואף להיות בטוח ל exceptions כל עוד Dtor לא זורק exceptions.

בג'אווה החיים יותר קלים... רוב הדרישות ב ++c נגעו לדיסטרקטורים ולכן לא רלוונטי בגאווה. כדי להבטיח שתמיד יתקיים הקוד גם אם היה exception משתמשים ב finally.

יש משהו חדש בג'אווה שלקחו מ #c ניתן אחרי try לעשות () והאובייקטים ששם חייבים לממש אינטרפייס עם מתודה close שאם היה exception בעקבות מה ששם הם מפעילים את הפונקציה הזו.

הרצאה 10 (20/05/14) IO Models

מנוע חיפוש באינטרנט משתמש ב index כדי לחפש, ה index נוצר ע"י ה indexer שהוא מקבל את כל המידע מה spider שעובר באופן רקורסיבי על כל האינטרנט.



כשהספיידר מסיים לעבור על האינטרנט הוא מתחיל מחדש. איך עובד הספיידר: יש דפים שכבר עברנו עליהם, לינקים שגילינו ועוד לא עברנו עליהם ואזור שעוד לא ביקרנו בו. בכל דף באינטרנט יש מישהו שמצביע עליו ולכן מכל אתר נגיע גם לכל מי שהוא מצביע עליו עד שנעבור על הכול.

תהליך הבאת דפים מהאינטרנט הוא איטי מאוד ממש זחילה ולכן נקרא crawl.

איך ה crawl עובד? מביא דפים, מעבד את הטקסט ושומר קישורים בנפרד וטקסט בנפרד וממשיך לעבור לקישורים הבאים. ה fetch מביא את הדפים מהאינטרנט.

מה יש ב fetch ? מקבל מחרוזת וממיר אותה ל URL שממנו מייצרים קישור, מהקישור מביאים את הדף עצמו בצורה באה: מידע גולמי והקישור לדף עצמו. מעבד את הדף ומוציא ממנו את הקישורים הבאים וקיצור של הדף שאותם שומרים (כי האינטרנט גדול וצריך לשמור רק מה שחשוב).

איך עובדת ההבאה של הדף? אנחנו כבר יודעים את הכתובת של הדף, נתחבר לשרת (לוקח זמן) וכשהוא יגיב נקבל ממנו מס' חבילות (פאקטים) שיכילו את הדף ושאותם נחבר לדף שלם. יש כמה המתנות בתהליך ולכן נעבוד במצב של multiprocessing בזמן שיש המתנה לעבור לתהליך אחר שמביא עוד דף. כדי למהר יותר את התהליך ולחסוך זמן תקורה של יצירת וסגירת התהליך ניצור מקום בזיכרון שמכיל טבלה של כל הלינקים ויהיו כמה תהליכים שידעו לרוץ על הטבלה הזו ואז ניצור רק פעם אחת תהליכים ונחסוך בזמן של לפתוח תהליך ולהרוג אותו בכל פעם.

כעת יש לנו תקורה של מעבר בין תהליכים ולכן נשתמש ב multithreading בתוך תהליך אחד יש כמה thread ולהם היפ משותף, זמן העברה בין thread זול משמעותית יחסית לתהליכים. צריך רק להיזהר בגלל שההיפ משותף. כעת אפשר להוריד משמעותית את זמן ההמתנה (תלוי בכמות ה thread שנפתח) מכיוון שאפשר לעבוד כמעט במקביל. גם ב thread יש עלות של להחליף בין התהליכים.

כאשר נרצה לעשות i/o נפתח ערוץ handle מול מערכת ההפעלה, וישנן מספר אפשרויות לקבל את הhandle ב i/o שלנו:

- blocking i/o - התהליך במצב blocked עד שמקבל שהhandler של הi/o יהיה מוכן {לקוח סבלני}. חסרונות:

1. פשוט ממתינים עד שיתפנה לנו handler ל i/o אחד, והכל נעצר בינתיים.

- nonblocking - התהליך בודק שוב ושוב האם הhandler עבור ה-i/o מוכן{לקוח נודניק}, עד שמקבל אוקיי ואז ממשיך. ניתן גם לרוץ על רשימה של i/o כל פעם מחדש בלופ. חסרונות:

1. מבזבז cpu time, שכן רוב הזמן לא יהיה שום דבר לקרוא, אז פעולה ה"נדנוד"{ניסיון לבצע read}, סתם יבזבז כח עיבוד יקר.
2. כמו כן, עלינו לנחש את זמן ההמתנה האידאלי בין "נדנוד" ל"נדנוד". {לא כתוב במצגות של יריב}.

- Multiplexing - בונים רשימה של כל הפעולות i/o שאנו רוצים לבצע. התהליך, באמצעות הפקודה select , מחכה שלפחות אחד מהhandlers עבור i/o כלשהו יהיה מוכן, וברגע שלפחותאחד מהם מוכן הוא מטפל בהם{מחלץ את המידע הדרוש עבורם}. אנחנו בבלוק לא רק עבור פעולה אחת, אלא עד שאחת מהפעולות שאנחנו מחכים לבצע יכולה להתבצע{ומובטח לנו בשלב זה שזה יהיה ללא בלוק.}
חסרונות:

1. טיפול בשני שלבים

- async i/o – התהליך מבקש ממערכת ההפעלה להודיע לו כשיתפנה handler, ובינתיים הולך לעשות דברים אחרים. התהליך מטפל גם בהעתקה

חסרונות:

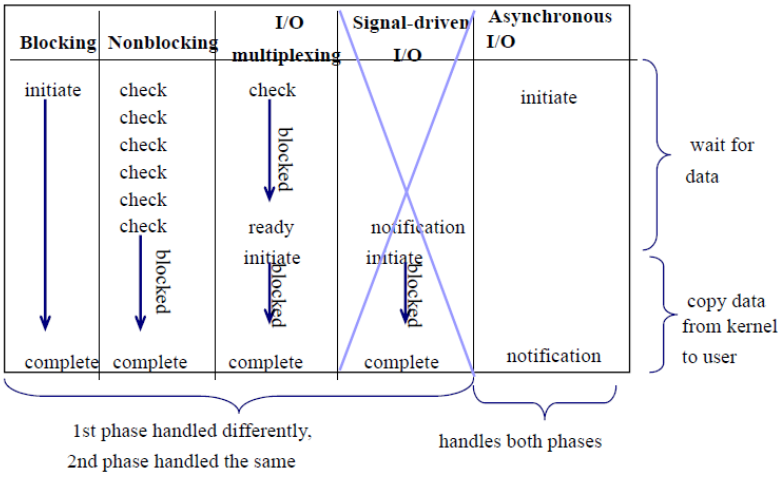
1. קשה מאוד למימוש
2. לא נתמך ע"י כל המערכות{לא כתוב במצגות של יריב}.
3. בעייתי להשתמש ביותר מסיגנל אחד בכל תהליך, שכן בעייתי לקבוע עבור מי הגיע הסיגנל{ואז שוב צריך לשלב blocking}.{לא כתוב במצגות של יריב}.

יתרונות:

1. מטפל בשני השלבים – ההמתנה למידע וחילוץ המידע.

יתרונות וחסרונות באופן כללי: ככל שנרד למטה ברשימה דלעיל, כך זה יהיה קשה יותר לממש, אך טוב יותר{טוב יותר אליבא דיריב, הרשת אומרת שדווקא multiplexing הכי טוב}.

סיכום של השיטות: (להתעלם מהשיטה עם ה x)



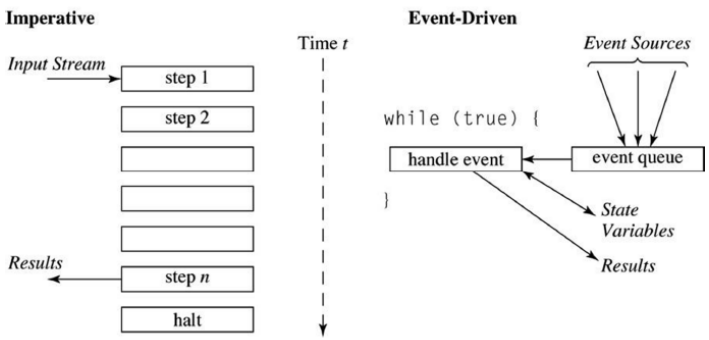
בשלב בקשת הכתובת מה DNS חייבים להשתמש ב Async blocking IO.

ריבוי תהליכים ות׳רדים עוזר רק אם המעבד ממתין אחרת זה לא עוזר.. אלא רק מחמיר את המצב כי הוספנו לחישוב גם את החלפת התהליכים.

הרצאה 11 (27/05/14) Event Driven

מערכות הפעלה באופן כללי כתובות ב C ולכן האינטרפייס לתכנות (API) היא ב C. אבל כדי שיהיה נוח יותר לעבוד עטפו אותם בכל מיני אינטרפייסים שונים. כשיש GUI האפליקציות רוב הזמן פתוחות ומחכות בניגוד לקומנד ליין שהתוכנה יודעת מיד מה היא צריכה לעשות. בנוסף בתכנות פרוצדורלי עובדים לפי הסדר ואילו כשיש GUI סדר הפעולות הוא בהתאם למה שהמשתמש בוחר ללחוץ עליו.

כדי להתמודד עם ה GUI נבנה תור של אירועים לכל אפליקציה, התוכנית שלנו תרוץ בלולאה וכל עוד אין משהו בתור נחכה וברגע שיש משהו בתור נעשה עליו פעולה. לכל אירוע יש מספור והפעולה מתבצעת בהתאם אליו.

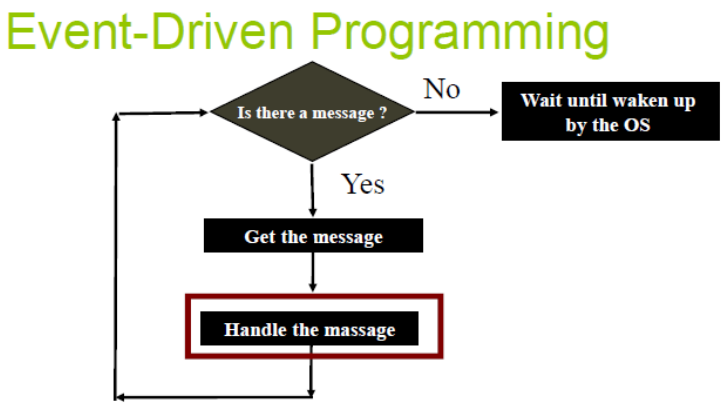


ניתן לשלוח הודעה מאירוע לאירוע, למשל כפתור הסגירה יכול לשלוח הודעת סגירה לחלון ואז לצאת מהתוכנית.

נניח שלחצנו על מקש ולוקח כמה שניות לצורך ביצוע הפעולה, בזמן הזה המערכת תתקע ואירועים לא יטופלו מכיוון שלא ניקח את האירוע הבא מהתור עד שלא נסיים את התור הנוכחי. ולכן דברים ארוכים יש לשים בת׳רד נפרד.

אחרי שהפרדנו לת'רד נפרד נרצה להודיע כשהאירוע הסתיים ולכן משתמשים בפקודה שלמערכת ההפעלה שמוסיפה לתור את האירוע של הסיום. זה חייב להגיע דרך מערכת ההפעלה מכיוון שרק ת'רד יחיד יכול להוסיף לתור ההודעות.

סיכום של איך עובד אירועים:



Time driven – ישנם מערכות שמגיבות לתזמונים, כל X שניות קורה משהו. פועל כמו תכנות אירועים.

דיברנו על תכנות אנדרואיד – לא למבחן.

הרצאה 12 (10/06/14) Version Control & DbC

נדבר היום בעיקר על איך שומרים את הקבצים נכון בדיסק.

ארגון נכון של הקבצים: בתהליך יצירת התוכנית נוצרים המון קבצים מסיומות שונות וזה יכול לגרום לבלגן. הקבצים שבאמת מעניינים אותנו הם קבצי המקור וקובץ הפרויקט. שיטה אחת לסדר את זה היא את כל הקבצים שקשורים לפרויקט והם לא קבצי המקור נשים בתיקייה נפרדת (קורה כבר ב VS). בכל מקום יש את השיטה שלו לארגון קבצי הפרויקט.

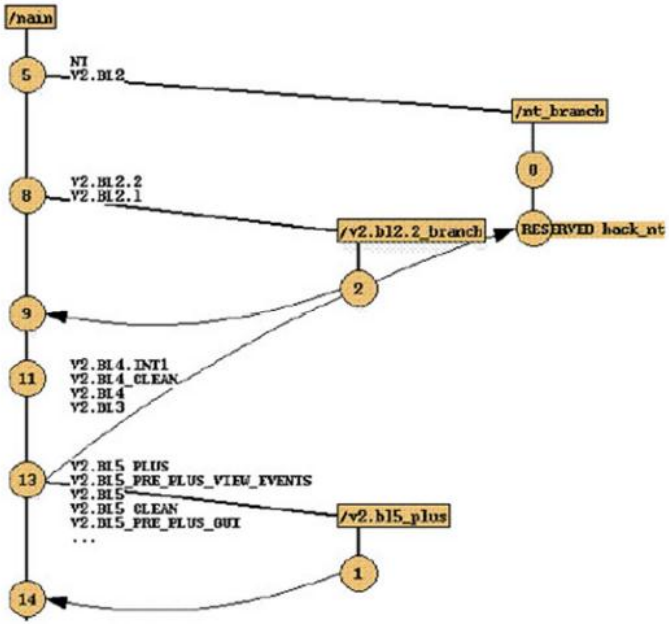
למה צריך version control:

1. יש קוד שעובד אבל אחרי שינוי הקוד לא עובד, גם אם נבטל את השינוי זה עדין לא עובד. לכן נרצה לשמור את הקבצים לפני השינוי.
2. התוכנית עבדה ואז עשינו שינויים שאנחנו עוד באמצע אבל נרצה לתת ללקוח גרסה שעובדת עכשיו גם בלי התוספות. לכן נרצה לשמור גרסאות כל הזמן.
3. שינינו חלק מהתוכנית וזה עובד, שותף נוסף שינה חלק בקובץ אחר וזה עובד לו אבל שממזגים את הקוד זה לא עובד... ולכן נרצה השוואות למה שהיה קודם.
4. כמו 3 באותו הקובץ.
5. שינוי שעשינו ודרסו אותו.

צריך כלי עזר שמאפשר לבדוק את ההבדלים בין 2 קבצים. זה נקרא diff Tools.

מהם מערכות version control? מאגר של כל קבצי הפרויקט וכל השינויים שבוצעו מתחילת הפרויקט. לכל שינוי שקרה לקובץ ניתן לשים הערה.

דוגמה לעץ שינויים ב version control : (דומה ל TFS של VS)

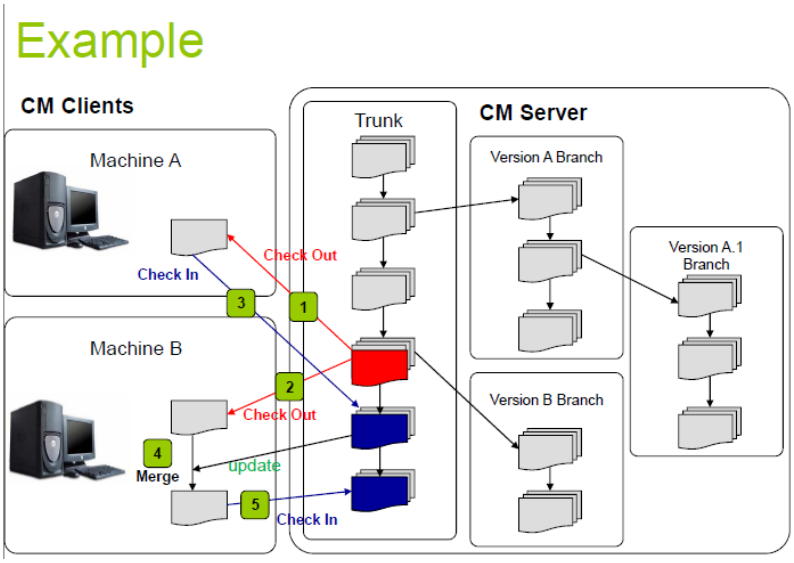


מהגזע הראשי יוצאים המון ענפים שהם גרסאות לא מעודכנות של הגזע, עליהם עובדים ואז ממזגים חזרה לגזע.

מושגים :

- Repository – המאגר של כל הקבצים והשינויים. שרת שמכיל את הכול, מקום אחד מרוכז.
- Check-Out – למשוך את הקבצים להעתק מקומי שלי.
- Change – שינוי שנשמר בחזרה לשרת המשותף.
- Change List – כל הקבצים שמייצגים תיקון אחד.
- Commit or Check-in – קבוצת השינויים שמייצגים פעולת אחת, פרסום שלהם לשרת המשותף.
- Update – לבקש מהשרת את הגרסה המעודכנת שכוללת את כל השינויים שכולם עשו.
- Merge/Integration – כולם עושים שינויים, יתכן גם ששני אנשים שינו את אותו הקובץ. במקרה הפשוט עושים מיזוג פשוט והמערכת יודעת לאחד בתקווה שאין התנגשויות ביניהם.
- Revision or Version – גרסת שינויים ספציפית של קובץ.
- Conflict – אם שני אנשים שינו מקומות שיש בהם חפיפה.
- Resolve – לפתור את ה conflict.

דוגמה לשימוש :



דילגנו על המון שקפים... ועברנו למצגת השנייה.

Design by Contract:

דיזיין לפי assert ולא לפי טסטים (ב C\C++ לא עובד טוב). כשכותבים או משתמשים בפונקציה נכנסים לחוזה בין הקורא לפונקציה לכותב אותה.

בעצם נעשה assert בתוכנית בתור התנאים בחוזה. כל פונקציה עומדת בפני עצמה וב assert מפורטים התנאים של החוזה של כל פונקציה.

סוגים של תנאים בחוזה:

1. תנאים שצריכים להתקיים כדי שהפונקציה תעבוד.
2. בהנחה שהתנאים המקדימים מתקיימים מה מובטח לי שהפונקציה תעשה.
(להפוך את הפונקציה ל const זה תנאי כזה שמצהיר שהפונקציה לא שינתה את המחלקה)
3. תנאים שתמיד מתקיימים - מה שתמיד צריך להתקיים למחלקה לא משנה מה נקראה אינוריאנטה. אם אחת מהם מופרת יש בעיה חמורה במחלקה. ולכן אפשר לבנות פונקציה שבודקת את האינוריינטיות ונקרא לה בכל תחילה וסוף של פונקציה כדי לוודא שלא יצרנו בעיות כלליות שלא קשורות רק לפונקציה.

יתרונות:

1. תיעוד
2. בודקים אותנו תמיד שהקוד עובד
3. מכריח אותנו לחשוב על המימוש
4. מכריח אותנו לחשוב מה המצב הנכון במחלקה

חסרונות:

1. המון עבודה
2. המון תקורה בגלל כל הבדיקות

3. הקוד נראה מכוער
4. יש הגבלה למה ניתן לבדוק בקוד
5. צריך לדעת המון פעמים את המצב הקודם

המבחן: עם חומר סגור, יהיה במבחן קטע קוד שנצטרך לשפר בעזרת כל מה שלמדנו בקורס, צריך לעשות assert במבחן. יהיו 2 שאלות של איגור.

הרצאה 13 (17/06/14) חזרה למבחן

דוגמה לשאלה במבחן:

מצאו דברים לשנות בקוד כדי לעשות אותו יותר טוב:

1. לשים לב לעקביות של השם, אם כולם מתחילים ב m או באותיות קטנות או גדולות.
2. לא להשתמש ב define אלא enum או const.
3. משתנים שצריכים להיות באותיות קטנות או גדולות.
4. אם לא יכול להיות שלילי לשים uint.
5. להוסיף assert איפה שאפשר.
6. להוסיף const איפה שאפשר.
7. סוגריים מסולסלים, להיות עקביים. בשורת ההצהרה או זאת שאחריה.
8. סוגריים מסולסלים ב if של שורה אחת.
9. משתנים שמוגדרים לפני הזמן ולא מאותחלים.
10. אתחול עם ההגדרה.
11. פונקציות שיכולות להיות const.
12. Mutable – גם פונקציות const יכולות לשנות אותו.

Unit testing:

לשנות את הפונקציות הנחוצות כדי שנוכל לבדוק אותם. אחרי כל שינוי מרצים מחדש את הטסטים. לעשות בדיקה ממש על כל שטות.

שאלה: תשתמשו ב assert כדי לכתוב טסטים ל 5 מקרים עבור מתודה מסוימת.
צריך לכתוב לפחות 3 טסטים של מקרי קצה ולפחות טסט אחד שמכיל 4 assert.

דוגמאות:

1. לבדוק שאם נותנים מחרוזת ריקה מקבלים וקטור ריק.
2. אם מתקבל פויינטר לבדוק שהוא לא NULL.
3. אם מקבלים רק מספרים כשרוצים לקבל אותיות.
4. אם יש מילה אחת לוודא שמקבלים את המס' 1
5. לוודא שמקבלים את המילה הנכונה בוקטור.
6. לוודא שמתעלם מרווחים.

7. לשים סטרינג הזוי ולוודא שמקבלים וקטור בגודל הנכון ועם המילים הנכונות.

Templates :

מה עושה הקוד הבא ויינתן דוגמה של קוד עם Templates.

לדעת לכתוב פונקציית Template במקום פונקציה קיימת.

פויינטרים לפונקציות וfunctorים :

לבנות פונקציה ולממש אותה כfunctor.

Assert :

לא שיטה לטפל בשגיאות אלא לחפש באגים. לא אמור לקרות לעולם. לא להשתמש איפה שלא צריך, למשל להעביר דברים ב & ואז לא צריך לבדוק NULL. לזכור שאין assert בגרסת הרצה ולכן צריך לדאוג שהקוד עושה אותו דבר עם ה assert ובלעדיו.

Exception :

ללמוד את 5 הרמות שיש לתוכניות שיזרקו exceptions.

RAII – אובייקטים שאחראים לשחרר אובייקט גם אם היה exception בדרך.