חוברת כיתה בקורס

תכנות מונחה עצמים

חוברת זו הוכנה ונערכה ע"י אוהד מתתיהו.
החוברת מיועדת לתלמידי הקורס **תכנות מונחה עצמים** ולכל המעוניין ללמוד תכנות מונחה עצמים דרך שפת התכנות C++.
אשמח לקבל הערות ותיקונים ל- ohadm@cs.haifa.ac.il

מקרא:

מבוא...

שיעור מספר 1............................ הקדמה

שיעור מספר 2............................ חזרה כללית על C

שיעור מספר 3............................ תוספות בסיסיות ב- C++ לעומת C

שיעור מספר 4............................ המחלקה

שיעור מספר 5............................ בניית עצמים והריסתם

שיעור מספר 6............................ העמסת אופרטורים (operator overloading)

שיעור מספר 7............................ קלט-פלט ועבודה עם קבצים

שיעור מספר 8............................ תבניות (Templates)

שיעור מספר 9............................ STL (Standard Template Library)

שיעור מספר 10.......................... ירושה (inheritance)

שיעור מספר 11.......................... פולימורפיזם (Polymorphism)

שיעור מספר 12.......................... פולימורפיזם – המשך

תכנות מונחה עצמים – מבוא

שפת התכנות וסביבת העבודה
שפת ++C נבחרה כשפת התכנות להמחשת העקרונות העיוניים של הקורס. קיימים מהדרים שונים של שפת ++C עבור פלטפורמות שונות (למשל, עבור DOS, Windows , Unix). בקורס נשתמש בסביבת הפיתוח של מייקרוסופט – Visual C++ 6.0.

ביבליוגרפיה
- ++C מדריך מקצועי, הוצאת *מרכז ההדרכה 2000* (1999-2000).
- The C++ Programming Language, 3rd ed., Addison Wesley, 1997, Bjarne Stroustrup .
- ++C ו- OOP למתכנת המקצועי, שמעון כהן.

מבוא
מטרות
לתת רקע כללי ומושגי יסוד לגבי גישות פיתוח תוכנה שונות.
להבחין בין גישת הפיתוח מונחית העצמים לבין שפות הממשות אותה.
להכיר את שפת ++C כשפת הרחבה ל- C התומכת בפיתוח מונחה עצמים.
פירוט התכנים
גישות פיתוח תוכנה: תכנות מונחה פרוצדורות (תכנות מבני). תכנות מונחה נתונים. תכנות מונחה עצמים
(Object Oriented Programming). מודל העצם. שפות תומכות בתכנות מונחה עצמים. מנגנונים תומכים במודל העצם. היררכיות הכלה. היררכיית ירושה. פולימורפיזם.
שפת ++C. ++C כהרחבה לשפת C. מבנה ומנגנוני השפה. היסטוריה.

++C כשפת C משופרת
מטרות
להכיר את חסרונות שפת C.
להבין כיצד ניתן לפתח תוכנה בגישה הפרוצדורלית ב- ++C כשפת C משופרת .
פירוט התכנים
מנגנונים משופרים ב- ++C : קלט / פלט ע"י אופרטורים (cin, cout). אופרטורים להקצאת/שחרור זיכרון (new ו- delete). פונקציות נפרשות – inline. העמסה של שמות פונקציות. reference. ערכי ברירת מחדל לפרמטרי פונקציות.

מחלקות ועצמים- מושגי יסוד
מטרות
להכיר את מושג המחלקה ואת מושג העצם.
להבין את עקרון הסתרת המידע וכיצד הוא ממומש במחלקה.
לכתוב תכניות מונחות עצמים בסיסיות ב- ++C.
פירוט התכנים
מושג המחלקה ומושג העצם. הסתרת מידע. תכנית בסיסית ב- ++C.
מצביעים לעצמים ומערכי עצמים. המצביע this. פונקציות חבר const.

<u>מחלקות ועצמים – איתחול והריסת עצמים</u>

<u>מטרות</u>

להרחיב בנושא המחלקות והעצמים.
להבין כיצד ומתי נוצרים עצמים ומתי הם נהרסים.
להכיר את מנגנון האיתחול וההריסה האוטומטיים (constructor & destructor).

<u>פירוט התכנים</u>

בניית עצמים והריסתם. סוגי עצמים, מתי והיכן הם נוצרים. מתי והיכן נהרס העצם.
מנגנוני איתחול והריסה אוטומטיים: בניית עצם ע"י constructor. הריסת עצם ע"י destructor .
constructor העתקה.

<u>מחלקות ועצמים - מודל ההכלה</u>

<u>מטרות</u>

להבין את עקרון ההכלה בתכנות מונחה עצמים..
להכיר את מודל ההכלה ההירככי ב- ++C.

<u>פירוט התכנים</u>

מחלקות המכילות עצמים. רשימת איתחול. מחלקת טבלה. מחלקת רשימה.
פונקציות ידידות – friend. חברי מחלקה סטטיים - static members.

<u>העמסת אופרטורים</u>

<u>מטרות</u>

להבין את עקרון העמסת האופרטורים.
להכיר את העמסת האופרטורים ב- ++C.

<u>פירוט התכנים</u>

עקרון העמסת אופרטורים. העמסה ע"י פונקציה חברה לעומת פונקציה גלובלית. העמסת אופרטורים בינריים ואונריים. העמסת אופרטורים לוגיים.
אופרטורים מובנים בשפה. העמסת האופרטורים "=" , "[]". אופרטורי המרה.
העמסת אופרטורי קלט/פלט "<<", ">>" וקלט / פלט לקבצים.

<u>הורשה</u>

<u>מטרות</u>

להכיר את מנגנון ההורשה וכיצד הוא תומך בפיתוח תוכנה מונחה עצמים.
להבין מהו פולימורפיזם וכיצד הוא פועל.

<u>פירוט התכנים</u>

עקרון ההורשה.יתרונות ההורשה.
בקרת גישה בהורשה. מצביע ו- reference בהורשה. העמסת פונקציות בהורשה.
פולימורפיזם. פונקציות וירטואליות. פולימורפיזם טהור.

שיעור מספר 1

הקדמה

נושאי השיעור:

- ההיסטוריה של ++C בקצרה.
- תכנות נוהלי, תכנות מובנה ותכנות מוכוון עצמים.
- ++C ותכנות מוכוון עצמים.
- תכנית ראשונה ב- ++C.
- תוספות כלליות לשפה

ההיסטוריה של ++C בקצרה

שפות מחשבים עברו התפתחות דרמטית מאז שנבנו המחשבים האלקטרוניים הראשונים, כדי לסייע בחישוב מסלולים של קליעי תותח במהלך מלחמת העולם השנייה. בתחילה השתמשו המתכנתים בהוראות המחשב הבסיסיות ביותר, בשפת מכונה (language machine). הוראות אלו יוצגו על ידי מחרוזות ארוכות של ספרות 'אחת' ו-'אפס'. במהרה, פותחו שפות סף, או אסמבלי (assemblers) כדי למפות את הוראות המכונה לביטויים שקל יהיה לזכור ולקרוא אותם, וגם יהיו ניתנים לניהול, כמו ADD ו- MOV.

עם הזמן פותחו שפות ברמה יותר גבוהה, כמו BASIC ו- COBOL. שפות אלו אפשרו לכתוב את פקודות המחשב במילים ובמשפטים שיש להם דמיון לשפה מדוברת, או לנוסחאות מתמטיות, כמו LET I=100. הוראות אלו תורגמו לשפת מכונה על ידי מפרשים (interpreters) ומהדרים (compilers). 'מפרש' מתרגם את התוכנית במהלך קריאתה והופך את הוראות התוכנית, או את הקוד שלה, לפעולות שהמחשב מבצע מייד. 'מהדר' מתרגם את הקוד לפקודות במבנה ביניים בתהליך הידור (compiling) שיוצר קובץ יעד (object). המהדר קורא למקשר (linker) והופך את קובץ היעד לתוכנית בת-ביצוע (executable).

מכיון שמפרשים קוראים את הקוד כפי שהוא כתוב ומבצעים אותו מייד, קל למתכנת לעבוד עם מפרשים. מהדרים מציגים את הצעדים הנוספים הלא נוחים של הידור וקישור הקוד. עם זאת, מהדרים מייצרים תוכנית מוכנה שאין צורך לפענח ולפרש אותה מחדש בכל הרצה , ולכן הפעלתה מהירה יותר. תרגום קוד המקור (source code) לשפת מכונה כבר בוצע, ואין צורך לחזור עליו.
יתרון נוסף של שפות מהודרות רבות כמו ++C הוא, שניתן להפיץ את התוכנית בת-הביצוע לאנשים שאין להם מהדר. בשפה של מפרש, חייב המפרש להיות מותקן במחשב, כדי שניתן יהיה להריץ בו את התוכנית.
שפות מסוימות, כמו Visual Basic, קוראות למפרש Run Time Library (ספריית זמן ריצה). Java קוראת למפרש זמן הריצה שלה (VM - Virtual Machine), אבל במקרה זה המפרש הוא חלק מהדפדפן (browser), כמו Internet Explorer או Netscape.

קוד מקור יכול להיות מוסב לתוכנית בת-ביצוע בשתי דרכים: מפרשים (interpreters) מתרגמים את קוד המקור להוראות מחשב, והמחשב פועל בהתאם להוראות אלו מייד. לחילופין, מהדרים (compilers) מתרגמים את קוד המקור לתוכנית, שניתן להריץ אותה במועד מאוחר יותר. למרות שקל יותר לעבוד עם מפרשים, רוב התכנות מתבצע עם מהדרים, מכיון שקוד מהודר רץ הרבה יותר מהר ואפשר להפעיל עליו תהליכי מיטוב (אופטימיזציה). ++C היא שפה שעוברת הידור.

עם ההתפתחות של רשת האינטרנט (Web), המחשבים נכנסו לעידן חדש של חדירה לשוק. יותר אנשים משתמשים במחשבים מאי פעם והציפיות שלהם מאוד גבוהות. התוכניות הפכו גדולות ומורכבות יותר, והצורך בשיטות תכנות מוכוון עצמים (programming object oriented) כדי לפצח מורכבות זו הפך לברור וגלוי.
קורס זה יתמקד בעיקרי השוני בין תכנות נוהלי (procedural programming) לתכנות מוכוון עצמים.

<u>תכנות נוהלי, תכנות מובנה ותכנות מוכוון עצמים</u>

עד לא מזמן, המתכנתים ראו את התוכנית כסדרה של הליכים שמבוצעים על נתונים. הליך (procedure), או פונקציה, הוא סדרה של הוראות מפורטות המבוצעות זו אחר זו. הנתונים היו נפרדים מההליכים, ובמהלך התכנות צריך היה לעקוב אחרי הקריאות של פונקציה אחת לאחרת, על איזה נתונים היא פועלת ואיזה נתונים עודכנו.
כדי להבין את המצב שיכול לבלבל, נוצר תכנות מובנה (structured programming).
הרעיון העיקרי שמאחורי תכנות מובנה פשוט כמו הרעיון של "הפרד ומשול". תוכנית מחשב יכולה להיחשב כמכילה סדרה של משימות. כל משימה אשר מורכבת מדי מכדי להיות מתוארת בפשטות, תפורק לסדרה של רכיבי משימות קטנים יותר, עד שהמשימות יהיו מספיק קטנות ומאורגנות כיחידות נפרדות, ושלמות כך שהם יובנו בקלות.

דוגמא:
חישוב של משכורת ממוצעת של כל עובד בחברה מסוימת הוא משימה די מורכבת, אולם, ניתן לפרק אותה לתת המשימות הבאות:
1. מצא כמה מרוויח כל עובד.
2. ספור כמה אנשים ישנם.
3. חשב את סכום כל המשכורות.
4. חלק את הסכום הכללי במספר האנשים שישנם.

סיכום המשכורות לכל העובדים (3) יכול להיות מפורק לשלבים אלה:
1. השג את הרישום על העובד.
2. גש למשכורת.
3. הוסף את המשכורת לסכום הכללי המצטבר.
4. השג את הרישום על העובד הבא.

קבלת נתוני כל עובד (4) יכולה להתפרק לשלבים אלה:
1. פתח את הקובץ של העובדים.
2. גש לרשומה המתאימה.
3. קרא את הנתונים מהדיסק.

תכנות מובנה הינו גישה טובה לטיפול בבעיות מורכבות. אולם, בשנות השמונים המאוחרות התבררו כמה מהחסרונות של שיטות התכנות המובנה.
ראשית, נטייה טבעית היא לחשוב על נתונים (לדוגמה, רישומים על כל עובד) ומה שניתן לעשות אתם (למיין, לערוך וכו') כרעיון אחד. תכנות מובנה נוגד את הנטייה הזו בכך שהוא מפריד את מבני הנתונים מהפונקציות שמטפלות בהם.
שנית, תוכניתנים נוכחו לדעת שהם ממציאים מחדש פתרונות חדשים לבעיות ישנות בקביעות. מה שנקרא "להמציא את הגלגל מחדש", שהוא ההפך ממִחזור עבודות קודמות לשימושים חדשים. הרעיון שמאחורי מִחזור הוא לבנות רכיבים שיש להם תכונות ידועות, שניתן אחר כך

לחבר אותם לתוכנית ככל הנדרש. הרעיון נלקח מעולם החומרה – כאשר מהנדס זקוק לטרנזיסטור חדש, הוא אינו ממציא או מפתח אותו, אלא הולך לתיבה הגדולה של הטרנזיסטורים ומוצא את המתאים לו ביותר לפי הצרכים שלו, או אולי מתאים אותו לצרכיו. עבור מהנדס תוכנה לא היתה קיימת אפשרות כזו.
הדרך בה אנו משתמשים כיום במחשבים – עם תפריטים,לחצנים וחלונות – מאמצת גישה יותר אינטראקטיבית ומונחית אירועים (event-driven) לתכנות מחשבים. "מונחית אירועים", משמע שכאשר מתרחש אירוע, המשתמש לוחץ על לחצן או בוחר מתפריט – והתוכנית חייבת להגיב. תוכניות נעשות אינטראקטיביות יותר ויותר ונעשה חשוב לתכנן עבור סוג כזה של תפקודיות. תוכניות מיושנות אילצו את המשתמש להתקדם צעד אחר צעד דרך סדרה של מסכים. תוכניות מונחות אירועים מציגות את כל האפשרויות בבת אחת ומגיבות לפעולות המשתמש.
תכנות מוכוון עצמים (object oriented programming) מנסה לענות על צרכים אלה. הוא מספק שיטות לניהול, משיג מחזור של רכיבי תוכנה ומשלב בין נתונים לבין משימות שמתפעלות אותם. התמצית של תכנות מוכוון עצמים היא ההתייחסות לנתונים ולהליכים שפועלים על הנתונים כ"עצם", "אובייקט" (object) יחיד – ישות שלמה שיש לה זהות ומאפיינים ייחודיים.

<u>C++ ותכנות מוכוון עצמים</u>

C++ תומכת בצורה מלאה בתכנות מוכוון עצמים (object oriented), כולל שלושת אבני היסוד של פיתוח מוכוון עצמים: כימוס (encapsulation), ירושה (inheritance) ופולימורפיזם (polymorphism).

<u>כימוס</u>
כאשר מהנדס צריך להוסיף נֶגֶד למכשיר שהוא בונה, הוא משתמש בדרך כלל ברכיב קיים שהוא מוצא בתיבה. הוא בודק את הפסים הצבעוניים שמציינים את התכונות ובוחר באחד שדרוש לו. הנגד הוא "קופסה שחורה" ככל שזה נוגע למהנדס – לא אכפת לו כיצד הוא פועל, כל עוד הוא מתאים לדרישותיו. הוא אינו צריך ל'הסתכל' בתוך הרכיב כדי להשתמש בו.
התכונה של היות הרכיב יחידה שלמה ומוכללת מבחינה פונקציונלית נקראת כימוס (encapsulation). הכימוס מאפשר הסתרת נתונים (data hiding). הסתרת נתונים היא מאפיין חשוב ומהותי, המצביע על כך שאובייקט יכול להיות בשימוש בלי שהמשתמש ידע, או שיהיה אכפת לו, כיצד הוא מורכב ופועל בתוכו. כפי שניתן להשתמש במקרר מבלי לדעת כיצד המדחס פועל, כך ניתן להשתמש באובייקט מתוכנן היטב מבלי לדעת על הרכבו ועל הנתונים הפנימיים שלו.

בדומה, כאשר המהנדס משתמש בנגד, הוא אינו צריך לדעת דבר על מבנהו הפנימי. כל תכונות הנגד כמוסות (encapsulated), או מוכללות, באובייקט הקרוי "נגד". הן אינן מפוזרות בין כל המעגלים האלקטרוניים. אין צורך להבין כיצד הנגד פועל כדי להשתמש בו ביעילות. הנתונים שלו חבויים בתוך המארז ואינם גלויים למשתמש.

C++ תומכת בתכונות הכימוס על ידי יצירת טיפוסים המוגדרים על ידי המשתמש (user-defined types), הנקראים מחלקות (classes). בפרקים הבאים נלמד איך ליצור מחלקות. ברגע שנוצרה מחלקה מוגדרת היטב, היא מתפקדת כישות כמוסה (encapsulated) ומשתמשים בה כיחידה שלמה מבלי לחדור לרכיביה הפנימיים. הפעולות הפנימיות של המחלקה צריכות להיות מוסתרות. המשתמשים במחלקה מוגדרת היטב אינם צריכים לדעת כיצד היא פועלת. הם רק צריכים לדעת כיצד להשתמש בה.

ירושה ומחזור
כאשר המהנדסים של "סוסיתא תעשיות רכב" רצו לבנות מכונית חדשה, עמדו בפניהם שתי אפשרויות: הם יכלו להתחיל מאפס, או שיכלו לשנות מודל קיים ולהתאימו לדרישותיהם החדשות. מודל "כרמל" שלהם נראה מתאים, אבל בכל זאת רצו להוסיף רכיבים אחרים, כמו למשל תיבת הילוכים אוטומטית, או הזרקת דלק במקום קרבורטור רגיל. המהנדס הראשי העדיף לא להתחיל מהבסיס, אלא לומר "הבה נבנה מכונית כרמל חדשה אשר נוסיף לה את היכולות החדשות ונכנה אותה בשם כרמל דוכס". כלומר, "כרמל דוכס" היא סוג של "כרמל", אך בעלת ייחוד כלשהו, בעלת מאפיינים חדשים.

C++ תומכת בהורשה (inheritance). ניתן להכריז על טיפוס (type) חדש, שהוא הרחבה של טיפוס קיים. תת-המחלקה החדשה נגזרת מהטיפוס הקיים, ולכן היא קרויה גם טיפוס נגזר (derived type). בדוגמה שלנו, המכונית "כרמל דוכס" נגזרה מ"כרמל", ולכן יורשת את כל תכונותיה, אך יכולה להוסיף או לשנות ככל הנדרש. כך למשל, אם נגדיר מחלקה בשם "דוח" נוכל לגזור ממנה דוחות ייחודיים שונים ורבים, אך לכולם יהיו המאפיינים המבדילים בין דוח לבין מסד נתונים, למשל.

פולימורפיזם
בשעה שמעבירים הילוך, סביר שמכונית כרמל דוכס החדשה תגיב אחרת מאשר מכונית כרמל הקיימת. לצורך מעבר הילוך בכרמל דוכס צריך להפעיל את ידית ההילוכים בלבד, בשעה שבמכונית כרמל גם צריך לשחרר את המצמד שבין המנוע לתיבת ההילוכים. כאן המשתמש מודע להבדלים. אך אם נתבונן בשוני שבין הזרקת דלק לבין שימוש בקרבורטור רגיל, ניווכח שהמשתמשים אינם חייבים לדעת על ההבדל כלל וכלל. הם רק צריכים ללחוץ "פול גז" והדבר הנכון יקרה, בהתאם למכונית שבה הם נוהגים.

C++ תומכת בתפיסה שאובייקטים שונים יבצעו את "הדבר הנכון" על ידי מנגנון הקרוי פולימורפיזם (polymorphism), אשר מתייחס לפונקציות ולמחלקות. פולימורפיזם מתייחס לאותו שם של פונקציה או של מחלקה, אשר מקבל צורות רבות.

תכנית ראשונה ב- C++

כמיטב המסורת לתכנית ראשונה בתכנית ראשונה בשפת תכנות חדשה, נציג את התכנית המדפיסה לערוץ הפלט התקני (מסך, בקיצור) "Hello world!" :

```
#include <iostream.h>

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

פלט:

```
Hello world!
```

נתאר בקצרה את אבני היסוד של השפה – קבצים, משתנים וייצוגם בזכרון, הערות ופלט למסך:

<u>קלט-פלט ב- ++C</u>

כידוע, include היא הוראת קדם-מעבד שאומרת: "הדבר הבא הוא שם של קובץ. מצא את הקובץ הזה, והכנס אותו בנקודה זו". הסוגריים המשולשים מסביב לשם הקובץ אומרים לקדם-המעבד לחפש בכל המקומות הרגילים לקובץ זה. אם המהדר הותקן כראוי, הסוגריים המשולשים יגרמו לקדם המעבד לחפש אחר הקובץ iostream.h בתיקיה שמחזיקה את כל קבצי ".h" של המהדר. הקובץ iostream.h, המשמש לערוצי קלט-פלט נחוץ להפעלת cout. cout נחוץ לכתיבת דברים על המסך. השורה הראשונה גורמת להכנסת הקובץ iostream.h בתחילת התוכנית, כאילו הקלדת אותו שם בעצמך. קדם-המעבד רץ לפני המהדר בכל פעם שהמהדר מופעל. הוא מתרגם כל שורה שמתחילה עם סימן סולמית (#) לפקודה מיוחדת, כדי להכין את הקוד עבור המהדר. האובייקט cout משמש כדי להדפיס הודעה על המסך. אובייקטים יידונו בכלליות בהמשך, והאובייקטים cout ו-cin בפרט, מאוחר יותר. שני האובייקטים, cin ו-cout משמשים בהתאמה לטיפול בקלט (לדוגמה מהמקלדת), ובפלט (למסך).

השימוש ב-cout נעשה באופן הבא: הדפיסו את המלה cout ואחריה אופרטור הפניית הפלט (<<). מה שמופיע מימין לאופרטור ההפניה מודפס על המסך. אם תרצו להוציא לפלט מחרוזת תווים, ודאו שהיא נמצאת בין סימני גרשיים, כמודגם בתכנית לעיל.

<u>מבט קצר על cout</u>

עד שנלמד כיצד להשתמש ב- cout לשם הדפסת המידע על המסך, נוכל להשתמש ב-cout ללא הבנה מקיפה על אופן פעולתו. כדי להדפיס משהו למסך, יש לכתוב את המלה cout, אחריה אופרטור ההכנסה (<<). למרות שאופרטור זה מורכב משני תווים של "קטן מ-" (<), שפת ++C מתייחסת אליהם כתו אחד.

לאחר אופרטור ההכנסה, רישמו את הנתונים הרצויים.

נביא דוגמא לשימוש ב- cout :

```
// using cout
#include <iostream.h>
int main()
{
    cout << "Hello there.\n";
    cout << "Here is 5: " << 5 << "\n";
    cout << "The manipulator endl writes a new line to the screen.";
    cout << endl;
    cout << "Here is a very big number:\t" << 70000 << endl;
    cout << "Here is the sum of 8 and 5:\t" << 8+5 << endl;
    cout << "Here's a fraction:\t\t" << (float)5/8 << endl;
    cout << "And a very very big number:\t";
    cout << (double) 7000*7000 << endl;
    cout << "I am a C++ programmer!\n";
    return 0;
}
```

פלט:

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:      70000
Here's the sum of 8 and 5:      13
Here's a fraction:           0.625
And a very very big number:     4.9e+07
I am a C++ programmer!
```

סוגי הערות
ישנם שני סוגים של הערות בשפת ++C. ניתן ליצור הערות עם סימן קו אלכסוני כפול (//), או בעזרת קו אלכסוני-כוכבית (*/). הערה המתחילה עם קו אלכסוני כפול (//) אומרת למהדר להתעלם מכל מה שכתוב בין הסימן לבין סוף השורה.
הערת קו אלכסוני-כוכבית אומרת למהדר להתעלם מכל מה שעוקב אחריה, עד לסימן כוכבית-קו אלכסוני (/*). לכל */ צריך להיות /* סוגר.
המהדר מתעלם מכל מה שנמצא בין סימני ההערה (/*...*/) כולל הערות מסוג // הכתובות בין הסימנים /* לבין */.
בסך הכל, כאשר כותבים הערות, יש לזכור כי ההערות אינן צריכות לומר מה קורה, אלא למה.

ייצוג סכמטי של הזיכרון
RAM (Random Access Memory) הוא זיכרון גישה אקראית. כאשר אתה מריץ תוכנית, היא נטענת אל תוך זיכרון RAM מקובץ התוכנית. כל המשתנים נוצרים ב-RAM. כאשר מתכנתים מדברים על זיכרון, הם מתייחסים בדרך כלל ל- RAM.

הקצאת זיכרון
כשתגדירו משתנה בשפת ++C, יהיה עליכם לומר למהדר איזה סוג של משתנה ברצונך ליצור: מספר שלם, תו וכו'. נתון זה אומר למהדר כמה מקום יש להקצות בזיכרון עבור המשתנה ואיזה סוג של ערך יישמר בו.
כל תא הוא בגודל בית (byte) אחד. אם תרצו ליצור משתנה בגודל ארבעה בתים, הוא יתפרס על פני ארבעה תאים בזיכרון. טיפוס המשתנה (לדוגמה מספר שלם, int) אומר למהדר כמה מקום (תאים) יש להקצות למשתנה.

גודל משתנים שלמים
בכל מחשב, תופס טיפוס מסוים של משתנה כמות קבועה של זיכרון. כלומר, מספר שלם עשוי להיות בגודל שני בתים (byte) במחשב אחד, ובגודל ארבעה בתים במחשב אחר, אך בכל אחד מהם יישאר המספר הזה קבוע לגבי כל משתנה מאותו הטיפוס.
משתנה מטיפוס char (המשמש בדרך כלל לאחסון תווים), תופס ברוב המקרים בית (byte) אחד. משתנה שלם "קצר" (short) הוא בן שני בתים ברוב המחשבים, ומשתנה שלם "ארוך" (long) הוא בן ארבעה בתים. משתנה שלם (ללא מילות המפתח short או long) עשוי לתפוס שני בתים או ארבעה. גודל המשתנה השלם תלוי בסוג המעבד (16 או 32 סיביות) והמהדר. במחשבים מודרניים בני 32 סיביות (Pentium) ובמהדר מודרני (לדוגמה, Visual C++ 6.0 ומעלה), המשתנים השלמים הם בני ארבעה בתים.

סוגי משתנים בסיסיים
קיימים מספר טיפוסים מובנים בשפת ++C. ניתן לחלקם לשם הנוחות למשתנים שלמים (עליהם דיברנו עד כה), משתנים בעלי נקודה צפה, ומשתני תווים.
טיפוסי המשתנים המשמשים בשפת ++C מוצגים בטבלה בעמוד הבא:

טיפוסי משתנים בסיסיים

טיפוס	גודל	ערכים
unsigned short int	2 bytes	0 to 65,535
short int	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int	4 bytes	-2,147,483,648 to 2,147,483,647
int(16 bit)	4 bytes	-32,768 to 32,767
int(32 bit)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int(16 bit)	2 bytes	0 to 65,535
unsigned int(32 bit)	4 bytes	0 to 4,294,967,295
Char	1 byte	256 character value
float	4 bytes	1.2e-38 to 3.4e38
Double	8 bytes	2.2e-308 to 1.8e308

סוגי קבצים

הקבצים שנוצרים על ידי עורך נקראים קבצי מקור (source files), ועבור C++ בדרך כלל מקבלים את הסיומות ".cpp", ".c" .אנו נשתמש בקבצים בעלי סיומות ".cpp" עבור קבצי מקור ב- C++.
יש להיזהר, כי יש מהדרים המתייחסים לקבצי ".c" כקוד C ולקבצי ".cpp" – כקוד C++. חשוב לבדוק את התיעוד של המהדר שמשתמשים בו.

הידור קובץ המקור

למרות שקוד המקור בקובץ עשוי להיראות מוצפן, ולמי שאינו יודע C++ יהיה קשה להבין למה הוא משמש, הוא עדיין נחשב כצורה שניתנת לקריאה על ידי אדם. קובץ קוד המקור אינו תוכנית בת-ביצוע, ולא ניתן להפעיל אותו או להריץ אותו כפי שניתן לעשות עם תוכנית.
כדי להפוך קוד המקור לתוכנית יש להשתמש במהדר. כיצד קוראים למהדר וכיצד אומרים לו היכן למצוא את קוד המקור? הדבר תלוי במהדר שבו משתמשים. זאת יש ללמוד מתוך התיעוד.
לאחר שקוד המקור עבר הידור (compilation), נוצר קובץ יעד (object file). קובץ זה נושא בדרך כלל את הסיומת ".obj", אולם זו עדיין אינה תוכנית שניתנת להפעלה. כדי להפוך את הקובץ לתוכנית שניתנת להפעלה, חייבים להפעיל מקשר (linker).

יצירת קובץ שניתן להפעלה עם מקשר

תוכניות C++ נוצרות על ידי קישור של קובץ ".obj" אחד או יותר עם ספריה אחת או יותר. ספריה (library) היא אוסף של קבצים הניתנים לקישור, ואשר סופקו עם המהדר, שנרכשו בנפרד, או שנכתבו וצורפו לספריה. לכל מהדרי C++ מצורפת ספריה של פונקציות שימושיות (או הליכים, procedures) ומחלקות שניתן לכלול אותם בתוכנית.
השלבים ליצירת קובץ שניתן להפעלה (excecutable file), או קובץ הרצה:

1. צור קובץ קוד מקור עם סיומת ".cpp".
2. הדר את קוד המקור לקובץ עם סיומת ".obj" .
3. קשר את קובץ ".obj" עם הספריות הנחוצות ליצירת תוכנית שניתנת להרצה.

שיעור מספר 2

חזרה כללית על C

נושאי השיעור:

- התניות ולולאות.
- פונקציות.
- מצביעים (כולל מצביעים לפונקציות).
- קלט-פלט ב- C.
- עבודה עם קבצים.
- רשומות.
- רשימות מקושרות (Linked list).

התניות ולולאות

משפטי תנאי:

- מבנה משפט תנאי if :

if (expression) { statement(s) }

במידה ויש ביטוי (statement) יחיד, אין צורך בסוגריים מסולסלים.

דוגמאות:

```
int a=2, b=5;
if(a == b)    // only one statment
    a++;
if(a < b)     // more then one statment
{   a++;  b += a;   }
```

- מבנה משפט תנאי if-else :

if (expression) { statement(s) } **else** { statement(s) }

דוגמא:

```
int a=2, b=5;
if(a == b)
{   a++;  b *= 2;   }
else
{   b++;  a *= 2;   }
```

- מבנה סולם תנאים if-else-if :

if (expression) { statement(s) }
else if (expression) { statement(s) }
else if (expression) { statement(s) }
...
else { statement(s) }

דוגמא:

```
int a=2, b=5;
if (a == b && a>2)
{ a++; }
else if (a == b && a==2)
{ a += 2; }
else if (a != b)
{ a += 3; }
else
{ a += 4; }
```

- משפט הצבה מותנה:
במשפט זה ישנה בחירה בין שני ביטויים בהתאם לערכו של תנאי מסויים.
(<תנאי>) **?** <ביטוי 1> **:** <ביטוי2 >
במשפט זה יבוצע <ביטוי 1> אם התנאי מתקיים, ואחרת יבוצע <ביטוי 2 >.
דוגמא:

```
int a=2, b=5;
a = b >2 ? b : b+2 ;
(a == b) ? printf("equal") : printf("different") ;
```

- מבנה switch :

```
switch ( <משתנה> ) {
    case (<קבוע 1>) : { statement(s); break; }
    case (<קבוע 2>) : { statement(s); break; }
    ...
    default : { statement(s); break; }
}
```

לולאות:

- מבנה לולאת for :

```
for(<אתחול> ; <תנאי עצירה> ; <קידום> )
{ statement(s) }
```

- מבנה לולאת while :

```
while(<תנאי קיום הלולאה>)
{ statement(s) }
```

יציאה מלולאה לפני הגעה לתנאי עצירה יתבצע ע"י הפקודה **break**.
מעבר לאיטרציה הבאה של הלולאה יתבצע ע"י הפקודה **continue**.
דוגמא:

```
int a, b=10, c=12;
for(a=2 ; a<b ; a++)
{
    while ( b < c)
    {
        if(b == 10)
            continue;
        c -= 1;
    }
    if(a == 8)
        break;
    b -= 2;
}
```

לולאות אינסופיות:

```
int a=2, b=10;
for( ; a<5 ; )
{ ... }

while (b == b)
{ ... }
```

פונקציות

השימוש בפונקציות מאפשר חלוקה של משימות תכנות למשימות קטנות יותר ועצמאיות. ע"י חלוקה לפונקציות הופכים את תהליך הפיתוח והכתיבה של תכנית מחשב גדולה לדבר פשוט והדרגתי יותר. תכנית בשפת C היא אוסף של פונקציות , כאשר אחת מהן חייבת להיות פונקצית ה - main( ) אשר ממנה מתחיל ביצוע התכנית. קריאה לפונקציה מפונקצית ה - main( ) או מכל פונקציה אחרת גורמת להפסקה זמנית בביצוע הפונקציה הקוראת ומעבר לביצוע הפונקציה הנקראת. פונקציה אחת יכולה לקרוא לפונקציה נוספת וליצור שרשרת של קריאות לפונקציות. אין כל מניעה לכך שפונקציה תקרא לעצמה ולמצב כזה קוראים רקורסיה .

<u>הגדרת פונקציה</u>

הצורה הכללית של הגדרת פונקציה היא:

```
<טיפוס מוחזר> <שם הפונקציה> (רשימת פרמטרים)
{
   קוד
   return <ביטוי>
}
```

<u>טיפוס מוחזר</u>- הטיפוס של הערך המוחזר.

<u>שם הפונקציה</u> - כל פונקציה מזוהה בתכנית ע"י שם יחודי. נהוג ורצוי לתת לפונקציה שם שיעיד על אופייה. למשל , לפונקציה שמדפיסה משולש יהיה נחמד לקרוא: PrintTriangle , ולפונקציה המחשבת את המקדם הבינומי ומחזירה את ערכו נקרא למשל : BinomCoefficient.

<u>רשימת פרמטרים</u>- (<טיפוס> arg1,<טיפוס> arg2,...) רשימה של שמות המשתנים וטיפוסיהם המופרדים ביניהם ע"י פסיק. רשימה זו מגדירה את הערכים (פרמטרים) שהפונקציה מקבלת. הערכים מועברים אל הפונקציה מן הפונקציה הקוראת, הם מועתקים אל תוך הפרמטרים על פי סדר הופעתם ברשימת הפרמטרים. הפרמטרים משמשים כמשתנים מקומיים של הפונקציה.

ניתן להעביר לפונקציה קבועים, משתנים או ביטויים מורכבים.
כשאנו מעבירים לפונקציה ערכים המתאימים לפרמטרים, ניתק הקשר בין התוכנית הקוראת לפונקציה, כלומר- לפעולות הנעשות בתוך הפונקציה לא תהיה שום השפעה על התוכנית הראשית. העברת פרמטרים בדרך זו נקראת <u>CALL BY VALUE</u>.
אם כן נרצה להשפיע דרך הפונקציה על משתני התוכנית הראשית, נצטרך להשתמש בדרך העברה אחרת, שנקראת : <u>CALL BY REFERENCE</u>, אותה נלמד בהמשך.

כאמור, כאשר הפונקציה מחזירה ערך, הוא מוחזר ע"י הפקודה return בצורת קבוע, משתנה או ביטוי, וטיפוסו של הערך ייכתב במקום המתאים (בתחילת ההגדרה). אם הפונקציה **אינה** מחזירה ערך נכתוב במקום הטיפוס המוחזר את הטיפוס void. ואת פקודת ה- return נכתוב כך: return; (כלומר- בלי שום ערך להחזרה).

דוגמא: פונקציה לחישוב מכסימום בין שני מספרים שלמים:

```
int max ( int, int);        // function declaration

int max ( int a, int b)   // function implementation
{
    if (a>b)
        return a;
    return b;               // alternative way: {    return (a>b) ? a : b ;   }
}
```

<u>מצביעים</u>

המצביע הוא משתנה המיועד להכיל כתובת בזיכרון.
בשפת C חובה לתת למהדר דין וחשבון על מה אנו רוצים להצביע. הסיבה העיקרית היא, שיש אפשרות (ע"י הפעלת האופרטור *) לגשת לעצם המוצבע ע"י המצביע, ורק כך יוכל המהדר לדעת מאיזה טיפוס הוא העצם המוצבע.
ניתן להצביע על קובץ, פונקציה, מערך, מחרוזת ועל כל סוגי המשתנים שהמהדר מכיר.
הגדרת מצביע למשתנים אלמנטריים תעשה ע"י הסימן * הנרשם ליד המשתנה בשורת ההגדרה.

אופן ההגדרה:

```
type* name;
```

לדוגמא:
הגדרת מצביע על int :

```
int* p_int;
```

<u>האופרטור & המופעל על משתנה אלמנטרי.</u>
אם k הוא משתנה מסוג int אזי: &k הוא גם משתנה, ולכן יש לו ערך וטיפוס.
ערכו: כתובת המשתנה k.
טיפוס: מצביע המצביע על int.
<u>האופרטור * המופעל על מצביע</u>
להבדיל מהסימן * שמוצמד למשתנה אלמנטרי בשורת ההגדרה, האופרטור * המופעל על מצביע הוא העצם המוצבע ע"י המצביע.

```
int n = 9,k = 5;
int *p = &k;      // p point to k
*p = 8;           // now k=8
p = &n;           // p point to n
*p = 10;          // now n=10
```

<u>מצביע לפונקציה:</u>
כמו שצויין קודם, ניתן להגדיר מצביע גם לפונקציה.
הדרך לעשות זאת היא :

```
<ערך מוחזר> (* name) (<רשימת טיפוסי פרמטרים>);
```

דוגמא :

```
#include <iostream.h>

int sub (int i, int j)
{   return i-j ;    }

int add (int i, int j)
{   return i+j ;   }

void main(int argc, char **argv)
{
    int (*pf)(int,int);         // declare pf as a pointer to function of type 'int f(int, int)'
    pf = &add;                  // initialize pf  to point to 'int add(int, int)'
    cout << "add(2,1) returned " << pf (2,1) << endl;       //call add(2,1)
    pf = &sub;                  // modify pf  to point to 'int sub(int, int)'
    cout << "sub(3,4) returned " << pf (3,4) << endl;       // call sub(3,4)
}
```

<u>קלט-פלט ב- C</u>

התקשורת עם המשתמש בתוכניות נעשית באמצעות פעולות קלט-פלט (I/O). ניתן לקלוט נתונים מאמצעים שונים ומגוונים, וגם לפלוט נתונים לאמצעים שונים.
בשפת C הקלט והפלט מבוצעים בעזרת פונקציות ספריה.
על מנת שנוכל להשתמש בפונקציות קלט-פלט בשפת C נצטרך להכליל בתחילת התוכנית את הקובץ stdio.h :

```
#include <stdio.h>
```

קובץ stdio.h מכיל את הגדרת פונקציות הקלט-פלט בשפת C .
ברירת המחדל לקלט היא stdin ולפלט- stdout. כלומר אם לא נאמר במפורש לאילו קבצים אנו רוצים להדפיס, או מאילו קבצים נרצה לקלוט נתונים, המחשב יקלוט מהמקלדת וידפיס למסך.

<u>קליטת והדפסת תו בודד:</u>

<u>int getchar(void)</u> – פונקציה זו קוראת תו אחד מהקלט הסטנדרטי (בד"כ המקלדת) ומחזירה את ערכו. אם הפונקציה נכשלת (למשל סוף קובץ) היא תחזיר 1–.
<u>int putchar(int c)</u> – פונקציה זו כותבת לפלט הסטנדרטי (בד"כ המסך) תו אחד.

<u>קליטת והדפסת מספר תווים:</u>

<u>int printf(const char*, ...)</u> – פונקצית פלט מורכב המטפלת בפלט ערוך (Formatted Output)
מבנה הפונקציה:
printf(" <מבנה הפלט> ",<ערך 1>, <ערך 2> ...)
<ערך> - יכול להיות קבוע, משתנה או ביטוי מורכב. מספר הערכים אינו מוגבל.
<מבנה הפלט> – מחרוזת תווים אשר מכילה תווי בקרה שתפקידם פענוח של רשימת הארגומנטים, על מנת להכניסם למחרוזת הפלט.
<u>int scanf(const char*, ...)</u> – פונקצית קלט מורכב המטפלת בקלט. מבנה הפונקציה:
scanf(" <מבנה הקלט> ",<כתובת משתנה 1>, <כתובת משתנה 2> ...)
<כתובת משתנה> - כתובת של משתנה, לאחסון הקלט.
<מבנה הקלט> – מחרוזת תווים אשר מכילה תווי בקרה שתפקידם צורת פענוח של רשימת הארגומנטים מהקלט, על מנת להכניסם למשתנים שאת כתובתם מעבירים לפונקציה.

בדומה ל- printf ו- scanf ישנן גרסאות שונות המאפשרות את אותה פונקציונליות של קריאת קלט וכתיבת פלט מ\להתקנים שונים – קבצים ומקום בזכרון, למשל.

<u>int sscanf(char* **input**, const char*, ...)</u> – קריאת הנתונים מאזור הזכרון המוצבע ע"י input.
<u>int fscanf(FILE* **fd**, const char*, ...)</u> – קריאת הנתונים מקובץ.
<u>int sprintf(char* **buf**, const char*, ...)</u> – כתיבת נתונים לאזור בזכרון המוצבע ע"י buf.
<u>int fprintf(FILE* **fd**, const char*, ...)</u> – כתיבת נתונים לקובץ.

עבודה עם קבצים

בשפת C קימת האפשרות לכתוב לקבצים ולקרוא מקבצים ישירות מהתוכנית עצמה. לשם כך מוגדר הטיפוס FILE אשר מוגדר ב – stdio.h .

בקובץ זה גם מוגדרים שלושה מצביעים לטיפוס FILE והם:

stdin – מצביע לקובץ הקלט הסטנדרטי.

stdout – מצביע לקובץ הפלט הסטנדרטי.

stderr – מצביע לקובץ השגיאות הסטנדרטי.

הקבצים הנ"ל נפתחים ע"י מערכת ההפעלה עם התחלת ביצוע התוכנית.

<u>פתיחת קובץ ()fopen</u>

לפני שנוכל לקרוא מקובץ או לכתוב לקובץ, חובת התוכנית לפתוח אותו. פתיחת הקובץ מתבצעת בעזרת הפונקציה fopen ,מבנה הפונקציה:

```
FILE *fopen(char *filename, char *mode)
```

filename – מחרוזת תווים המכילה את שם הקובץ אשר מעונינים לפתוח.

mode - מחרוזת תווים אשר מציינת את מטרת פתיחת הקובץ.

הפונקציה fopen מחזירה מצביע לקובץ במידה ופתיחת הקובץ הצליחה, ואם פתיחת הקובץ לא הצליחה הפונקציה מחזירה NULL .

דוגמא:

```
fopen(“a:\input.txt”, “r”);
```

<u>סגירת קובץ ()fclose</u>

בסיום השימוש בקובץ חובה לסגור את הקובץ בעזרת הפונקציה :

```
int fclose(FILE *);
```

הפונקציה מחזירה 0 במידה והסגירה הצליחה (ו - EOF אם לא).

<u>קריאה וכתיבה של תווים בודדים מקובץ-</u>

על מנת לקרוא תו מקובץ נשתמש בפונקציה :

```
int fgetc(FILE *fp);
```

פונקציה זו פועלת כמו הפונקציה ()getchar רק שהקלט הוא מהקובץ שמצביעו fp במקום מקובץ הקלט הסטנדרטי.

על מנת לכתוב תו לקובץ נשתמש בפונקציה :

```
int fputc(int c, FILE *fp);
```

פונקציה זו פועלת כמו הפונקציה ()putchar רק שהפלט הוא לקובץ שמצביעו fp במקום לקובץ הפלט הסטנדרטי. במידה והכתיבה לא מצליחה הפונקציה תחזיר EOF.

<u>קריאה וכתיבה של שורות ( )fgets( ), fputs</u>

קבצי הטקסט מכילים שורות אשר מסתימות בתו 'סוף שורה'.

בעזרת הפונקציות ()fputs, ()fgets ניתן לכתוב ולקרוא שורה שלמה.

<u>מבנה הפונקציה ()fputs:</u>

```
char *fputs(char *line, FILE *fp);
```

הפונקציה כותבת את המחרוזת המוצבעת ע"י line לקובץ אשר מוצבע ע"י fp.

התו ’\0‘ מוחלף ב- ‘\n’.

<u>הפונקציה ()fgets:</u>

char *fgets(char *line, int Max,FILE *fp);

הפונקציה קוראת את השורה הבאה בקובץ אשר מוצבע ע"י fp, ומכניסה אותה למחרוזת המוצבעת ע"י line. המשתנה Max מגדיר את המספר המקסימלי של התווים אשר נרצה לקרוא.
הפונקציה תפסיק לקרוא מהקובץ כאשר תתקל ב '\n' או לאחר Max תווים.
הפונקציה מחזירה מצביע לתחילת line. כאשר נגיע ל – EOF הפונקציה מחזירה NULL.

<u>קריאה וכתיבה משוכללים ()fscanf , ()fprintf</u>

<u>הפונקציה ()fscanf :</u>
השימוש בפונקציה זו זהה לשימוש בפונקציה ()scanf וההבדל היחיד הוא שבפקודה זו מופיע גם מצביע לקובץ שהוא הקובץ שממנו יתבצע הקלט.
מבנה הפונקציה:

int fscanf(FILE *fp,char *format,...);

יש לדאוג שקובץ הקלט יתאים לפורמט של הקלט בפונקציה fscanf, ולבדוק את מספר הפרמטרים שנקלטו כפי שמבצעים כאשר קוראים לפונקציה scanf.

<u>הפונקציה ()fprintf :</u>
השימוש בפונקציה זו זהה לשימוש בפונקציה ()printf וההבדל היחיד הוא שבפקודה זו מופיע גם מצביע לקובץ שהוא הקובץ שאליו יתבצע הפלט.
מבנה הפונקציה:

int fprintf(FILE *fp,char *format,...);

דוגמא לקריאה מקובץ:

```
#include <stdio.h>
#define INPUT_FILE  "bank.txt"
#define MAX_LINE  1024

int main()
{
    FILE *finput;
    char buf[MAX_LINE];
    int lines;
    if ( (finput = fopen(INPUT_FILE,"r") ) == NULL)
    {
        fprintf(stderr,"Error opening %s for reading\n",INPUT_FILE);
        return 1;
    }
    fscanf(finput, “%d\n”, &lines);        // read first number, indicating how many lines to read
    for(int i=0; i<lines; i++)
    {
        fgets(buf, MAX_LINE, finput);
        fprintf(stdout, “Line %4d from file is: %s\n”, i, buf);
    }
    fclose(finput);
    return 0;
}
```

רשומות (מבנים – structures)

מבנה (struct) הוא אוסף של משתנים מטיפוסים זהים או שונים המקובצים יחד תחת שם אחד.
האפשרות לאגד מספר משתנים במבנה משולב מאפשרת גישה מהירה ויעילה לנתונים שבתוכם.
אופן הגדרת מבנה:

```
struct   <שם מבנה>   {
טיפוס   <שם משתנה>;
....
....
טיפוס   <שם משתנה>;
};
```

דוגמא: מבנה המייצג נקודה במרחב R^2.

```
struct point {
    double x;
    double y;
};
```

לאחר שהגדרנו מבנה, ניתן להגדיר בצורה מקוצרת משתני מבנה בעלי תבנית זהה באופן הבא:

```
struct   <שם סוג מבנה>   <רשימת משתני מבנה>;
```

לדוגמא:

```
struct point  point1,point2;
```

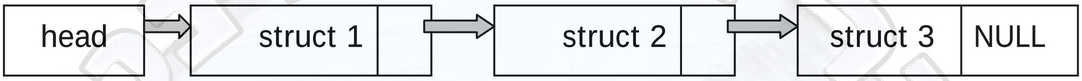
המשתנים point1 ו- point2 בעלי תבנית זהה.
כל אחד מהמשתנים במבנה נקרא שדה (field). על מנת לפנות לשדה מסוים במבנה נכתוב:
<שם מבנה>.<שם שדה>
לדוגמא, עבור גישה לשדה x של המבנה pt1 נכתוב: pt1.x;

<u>מצביעים למבנים-</u>
כפי שהגדרנו מצביעים לטיפוסים שהכרנו עד כה, ניתן להגדיר מצביעים למבנים. על מנת להעביר פוינטר למבנה כפרמטר לפונקציה, נעביר את כתובת המבנה למשתנה פנימי של הפונקציה שיוגדר לצורך כך כמצביע למבנה מאותו טיפוס.

כאשר נרצה לפנות למבנה המוצבע ע"י מצביע מוסיפים * לפני שם המצביע. לדוגמא על מנת לפנות לשדה hour המוצבע ע"י t נכתוב: *t.hour
נהוג להשתמש בסימן -> על מנת לפנות לשדה המוצבע ע"י מצביע למבנה.
לדוגמא, על מנת לפנות לשדה hour המוצבע ע"י t ניתן ונהוג לכתוב t->hour .

רשימות מקושרות (Linked list)

רשימה מקושרת מורכבת ממבנים , אשר כוללים את שדות המידע ושדה שהוא מצביע למבנה נוסף מאותו טיפוס . המבנה הנוסף מכיל אף הוא מצביע נוסף , וכך מתקבלת שרשרת של מבנים המקושרים באמצעות מצביעים .
נשים לב שהמצביע הוא מטיפוס המבנה שבו הוא מוגדר .
שרשרת במבנים יכולה לשמש אותנו בדיוק כמו מערך של מבנים , אלא שאין חובה להגדירם כרצף בזיכרון ואין חובה לציין את מספרם מראש . כך נוכל לצרף מבנה נוסף , ע"י הצבת כתובתו לשדה המצביע של המבנה האחרון בשרשרת .
לצורך מימוש הרשימה נשתמש בטיפוס "מצביע למבנה" , המצביע למבנה הראשון ברשימה מקושרת - נקרא לו head . ובמבנה האחרון ברשימה נציב בשדה המצביע למבנה את הערך NULL כדי לציין את סוף הרשימה (לפעמים נוח גם להחזיק מצביע נוסף לסוף הרשימה).



איך נממש רשימה מקושרת?
כל איבר ברשימה הוא מבנה. השדות בכל מבנה יהיו: שדה מידע (אחד או יותר, לפי הצורך), ושדה שהוא מצביע למבנה. הבעיה היא איך נדע כיצד לגשת לאיברי הרשימה?
לשם כך נגדיר מצביע לראש הרשימה, מצביע לסוף הרשימה, ונוח להחזיק גם משתנה שיאמר לנו כמה איברים יש ברשימה.
מצביע לתחילת הרשימה הוא למעשה הדבר היחיד שחייב להיות. שאר הנתונים הם אופציונליים. למה?
אם לא יהיה לנו מצביע לאיבר הראשון לא נוכל אף פעם לדעת היכן מתחילה הרשימה בזיכרון. לעומת זאת, את סוף הרשימה נוכל לדעת ע"י התקדמות על הרשימה עד שנגיע ל- NULL, ובדרך זו גם נוכל לספור את איברי הרשימה. למרות זאת "שווה" להחזיק עוד מצביע ועוד משתנה אם יש שימוש בהם בתוכנית.
מבחינה תכנותית, ניתן להגדיר מבנה אחד, שמגדיר איבר ברשימה (node), ואת המצביעים (להתחלה/לסוף) והמונה להגדיר כמשתנים בתוכנית. אך יהיה נכון יותר להיעזר במבנה נוסף (נקרא לו list) שיהיה המנהל של הרשימה והוא יחזיק את הנתונים על הרשימה ואת המצביעים.

לצורך בניית הרשימה נקצה כל מבנה בצורה דינמית ונוסיף אותו לסוף/תחילת הרשימה או למקום כלשהו בתוך הרשימה .

שיעור מספר 3

תוספות בסיסיות ב- ++C לעומת C

נושאי השיעור:

- .Enumerators
- שימוש ב- const.
- .inline functions
- .overloading functions
- .default arguments, anonymous arguments
- .reference

Enumerators

השימוש ב – enum אינו יחודי לתכנות ב- ++C, אך יותר נפוץ בשימוש.
קבועים ממוספרים (enumerated constants) מאפשרים ליצור טיפוסים חדשים, ולהגדיר משתנים מטיפוסים אלה, שערכיהם מוגבלים לקבוצה של ערכים אפשריים.
enum הינו טיפוס (type) כדוגמת int ו- char המאגד בתוכו אוסף של קבועים, כדוגמת אלו המוגדרים בשפת C ע"י define# , שלהם שמות המאפיינים את משמעותם.
המהדר (compiler) מתרגם את השמות בתוך הגדרת ה- enum לקבועים החל מ- 0. ניתן לשנות קביעה זו ע"י מתן ערכים כרצוננו.

דוגמא:

```
enum TrafficLight
{
    RED = 0,          // default value for RED is 0 anyway
    YELLOW,           // YELLOW=1
    AMBER = YELLOW + 2,        // AMBER=3;
    GREEN = 97         // initialize GREEN to be 97
};

int main()
{
    TrafficLight sign = RED;
    TrafficLight*  ps = &sign;
    if( *ps == YELLOW )
    {
        int convert_ok = AMBER;      // ok !!
        TrafficLight convert_not_ok = 2;     // Error: cannot convert from 'const int' to 'TrafficLight'
    }
    return 0;
}
```

בפועל, ניתן להציב כל ערך שלם למשתנה מסוג enum, אפילו אם אין הוא מוגדר כחלק מה- enum, למרות שמהדר טוב יזהיר אם תעשו זאת. משתנים ממוספרים הם בעצם מהטיפוס unsigned int, כך שהקבועים הממוספרים מתנהגים באותו אופן כמו המשתנים השלמים.

שימוש ב- const

השימוש במזהה (identifier) const על אובייקט שעליו הוא הוכרז מציין שאותו אובייקט נמצא
במצב של קריאה בלבד (read only) ולכן לא ניתן לשנותו.
השימוש ב- const הוא ++C הוא נרחב והוא משפר את נכונות הקוד.
הדרך היחידה לאתחל אובייקט (או משתנה) קבוע (const) היא בזמן יצירתו. כל נסיון לשנות את
ערכו בנקודה אחרת בתוכנית יגרור שגיאת הידור.
דוגמא:

אתחול:

```
const int year = 2002 ;        // initialization for constant variable
```

שימוש לא תקין:

```
const double pi ;              // Error: const object must be initialized if not extern
pi = 3.141592653589 ;

const int const_int=10;
int* int_ptr = &const_int;     // Error: cannot convert from 'const int *' to 'int *'
const_int++;                   // Error: cannot change constant value
```

שימוש ב- const במצביעים:

```
int i = 36 ;
const int ci = 77 ;            // const integer
int* const cpi = &i ;          // const pointer to integer
const int* pci = &ci ;         // pointer to const integer
const int* const cpci = &ci;   // const pointer to const integer
```

שפת ++C מאפשרת להחמיר את דרגת הגישה למשתנים / אובייקטים, אך לא להיפך.
אם נחזור למודל המחלקה – הנתונים שמורים במבנה פנימי ולא נגיש (private).
בנוסף אנו מגדירים פונקציות ממשק שלעיתים נרצה שיחזירו מצביע לנתונים אלו, או ייחוס (ידובר
בהמשך – reference) וזאת מבלי לאפשר שינוי הנתונים הפנימיים.
נרצה שלמעט שימוש בערך המוחזר לא תהיה אפשרות לשנותו ע"י משתמש חיצוני.
דרך נוחה ופשוטה לעשות כן מתוארת בדוגמא הבאה:

```
class String
{
    char* str;                 // private member
public:
    const char* get_string();  // public function
};

const char* String:: get_string()
{
    return str;                // converting from 'char*' to 'const char*' is allowed
}

void main()
{
    String s;
    char* c_str = s.get_string();          // Error: cannot convert from 'const char *' to 'char *'
    const char* cc_str = s.get_string();   // ok !!
    *cc_str = '\0' ;                       // Error: cannot change constant value
}
```

inline functions

בשפת ++C ניתן לעזור למהדר לבנות קוד יעיל יותר (optimized) בכמה צורות שונות.
אחת מהן הינה הוספת המילה השמורה inline לפני הגדרה ומימוש של פונקציות (הן פונקציות כלליות והן פונקציות השייכות למחלקה). בצורה זאת אנו מבקשים מהמהדר להמיר כל קריאה לפונקציה בקטע קוד המממש את הפונקציה. בכך נחסוך בזמן ריצה את התקורה (overhead) הנוספת של קריאה לפונקציה.
השימוש בפונקציות inline נפוץ ב- ++C מאחר ולאובייקטים יש בדר"כ כמה פונקציות שירות קטנות המבצעות פעולה אחת או שתיים.
השימוש בפונקציות inline דומה במעט לשימוש במקרו (macro), למעט העובדה שלמקרו חסרונות רבים.
לדוגמא :

```
#define MAX(a , b)  ((a>b) ? (a) : (b))

int p = 5, q = 8;
MAX(++p, ++q);   // will be → ((++p >++q) ? (++p) : (++q))
```

בדוגמא האחרונה אין בעיות הידור, אך הקטע קוד שנפרש בפועל הוא לא בהכרח מה שהתכוון המתכנת. למעשה, q מקודם פעמיים ו- p רק פעם אחת.
בדוגמא הבאה נראה כיצד נפרש קוד האסמבלי עבור קריאות לפונקציה עם ובלי inline :

```
inline int add(int i, int j)
{
    return (i+j) ;
}

void main()
{
    int sum = add(2,4) ;
    return ;
}
```

<u>פרישת קוד אסמבלי עם inline</u>

```
main:  push  eax
          mov eax, 4
          add  eax, 2
          mov dword ptr ds:[sum], eax
          ret
```

<u>פרישת קוד אסמבלי ללא inline</u>

```
main:  push  4
          push 2
          call  add
          add  esp, 8
          mov dword ptr ds:[sum], eax
          ret

add:   push  ebp
          mov ebp, esp
          mov eax, dword ptr [ebp+8]
          add  eax, dword ptr [ebp+0Ch]
          mov esp, ebp
          pop  ebp
          ret
```

overloading functions

ניתן להגדיר, לממש ולהשתמש בפונקציות בעלות אותו שם, אבל עם חתימת פרמטרים שונה.
כזכור, חתימת הפונקציה נקבעת על פי שמה והארגומנטים המועברים אליה, אך לא לפי הערך המוחזר שלה.
היתרון הוא בנוחות למשתמש בפונקציות אלו ונוחות למתכנת אשר אינו צריך "להמציא" שם לכל פונקציה, שעושה אותו דבר, אך עם פרמטרים שונים.

דוגמא משפת C: מתוך stdio.h - הצהרות על פונקציות הדפסה למיניהן ...

```
int  fprintf(FILE *, const char *, ...);
int  printf(const char *, ...);
int  vfprintf(FILE *, const char *, va_list);
int  vprintf(const char *, va_list);
int  _vsnprintf(char *, size_t, const char *, va_list);
int  vsprintf(char *, const char *, va_list);
int  _snprintf(char *, size_t, const char *, ...);
int  sprintf(char *, const char *, ...);
int  fwprintf(FILE *, const wchar_t *, ...);
int  wprintf(const wchar_t *, ...);
int  _snwprintf(wchar_t *, size_t, const wchar_t *, ...);
int  swprintf(wchar_t *, const wchar_t *, ...);
int  vfwprintf(FILE *, const wchar_t *, va_list);
int  vwprintf(const wchar_t *, va_list);
int  _vsnwprintf(wchar_t *, size_t, const wchar_t *, va_list);
int  vswprintf(wchar_t *, const wchar_t *, va_list);
```

דוגמא לשימוש בהעמסת פונקציות:

```
class String
{
    char* str;
public:
    ...
    void cat (char ch);              // concatenate character to local string
    void cat (char* str_to_cat);   // concatenate string to local string
};

main()
{
    String s;
    s.cat('H');
    s.cat("ello World!!");
}
```

default arguments

ב- C++ ניתן להגדיר ערכי ברירת מחדל (default arguments) לפונקציות.
הכללים להגדרה שכזאת הינם:
1. יש להשתמש ב- '=' כדי להשים ערכי ברירת מחדל.
2. ההשמה תתבצע אך ורק בהצהרה על הפונקציה (.h file), אך לא במימוש שלה.
3. השימוש הוא אך ורק לארגומנטים הכי ימניים של הפונקציה.

דוגמאות:

```
int foo( int i, int j = 8, int k = 1000);            // j,k has default value
int bar( int i = 5, int j);                          // Error: missing default parameter for parameter 2
int date(int d = 1, int m = 1, int y = 2002);        // all arguments got default values

// here should be foo, bar, date implementation ...

main()
{
    foo(5);                 // == foo(5, 8, 1000);
    foo(5, 9);              // == foo(5, 9, 1000);
    date();                 // == date(1, 1, 2002);
    date(20, 11, 2003);     // == date(20, 11, 2003);
}
```

השימוש בהעמסת פונקציות וערכי ברירת מחדל גם יחד, עשוי להיות בעייתי ולכן יש להזהר:
דוגמא 1:

```
void calc (int, double);
void calc (long, float);

calc(100, 11.5);         // ok – call calc (int, double)
calc(250, 10.5F);        // ?? – what should be expended ? int→long or float→double ?
calc(70000L, 41.6);      // ?? – what should be truncated ? long→int or double→float ?
```

דוגמא 2:

```
int compare(int first, int second = 0);
int compare(int first);

compare(3);              // Error: 'compare' : ambiguous call to overloaded function
```

anonymous arguments

כדי להמנע מהערות המהדר על ארגומנטים הנשלחים לפונקציה אך אין בהם שימוש, ניתן להגדיר פונקציה ללא שמות לארגומנטים. הדבר משמש בעיקר כדי לבנות קטעי קוד זמניים עד למימוש שאר חלקי התכנית.

שימוש:

```
void under_construction(int, int, double, char*)
{
    return;
}
```

reference – (ייחוס)

משתנה ייחוס הינו מזהה (identifier) למשתנה / אובייקט קיים. לפיכך – לא ניתן להגדיר משתנה ייחוס ללא אתחולו. משתנה ייחוס מאותחל פעם אחת ולא ניתן לשינוי.
מגדירים משתנה ייחוס ע"י הסימן & מייד לאחר הגדרת הטיפוס.
דוגמא:

```
int i;
int & iref = i;
int & eight = 8;
```

המהדר מתרגם משתני ייחוס באופן הבא: int& iref ⇔ int* const
כלומר, זהו מעיין מצביע קבוע ולכן מרגע שאותחל משתנה זה, לא ניתן לשנותו.

השימוש במשתנה ייחוס דומה לשימוש במשתנה רגיל למעט העובדה שההתנהגות היא כאילו השתמשנו במצביע.
היתרונות לשימוש במשתנה ייחוס הינם:

- נוחות בשימוש.
- לא ניתן להעביר משתנה ייחוס "ריק" מאחר והוא חייב להיות מיוחס למשתנה קיים. השימוש במצביעים לחלופין, מסוכן יותר, שכן יש לבדוק את ערכו של המצביע (null =).
- בקריאה לפונקציה עם משתנה ייחוס לאובייקט, לא מועתק כל האובייקט אלא רק כתובתו.

דוגמאות:
1)

```
int & plus_plus(int & to_inc)
{
    to_inc ++;
    return (to_inc);
}

int  a, b = 8;
plus_plus(b);
a= plus_plus(b);
int & c = plus_plus(a);
c++;
cout << a << " " << b << " "<< c << endl;
```

מה תהיה ההדפסה בשורה האחרונה?

2)

```
class Date
{
    int d, m, y;
public:
    Date(int dd, int mm, int yy);
    bool equal (const Date & other);
};

Date today(8, 8, 2002), tomorrow(9, 8, 2002);
bool eq = today.equal(tomorrow);
```

טעויות נפוצות בשימוש עם משתני ייחוס:

- החזרת ייחוס לאובייקט שהוגדר בתחום (scope) של הפונקציה ממנו הייחוס הוחזר:

```
class String
{
    char* str;
private:
    String& copy();              // copy current String and return new string;
};

String& String::copy()
{
    String other;                // this is a local instance of String !!!
    other.str = new char[strlen(str)+1];
    strcpy(other.str, str);
    return other;
}

String str1;
String& str2 = str1.copy();    // Error: value of return value refer to local instance
```

- בלבול עם מצביעים (בגלל ה- "&").

שיעור מספר 4

המחלקה

נושאי השיעור:

- הגדרת מחלקה (class).
- תחומי גישה במחלקה – private, public.
- אופרטור התחום (::)
- מימוש מתודות (methods) של מחלקה וצורת הקריאה אליהן.
- this.
- הפרדת המחלקה לקבצי מקור וקבצי מימוש

המחלקה

המושג העיקרי בתכנות מונחה עצמי בשפת ++C (ולא רק בה) הינו "מחלקה" (class).
המחלקה הינה הרחבה של המבנה (struct) שבשפת C, הכוללת בנוסף לנתונים של המבנה גם פונקציות לטיפול בו. בנוסף לכך, כוללת שפה זו גם תמיכה מובנית בהסתרת מידע, אתחול והריסה אוטומטיים של עצמים, ירושה ופולימורפיזם, טיפול בחריגות ועוד.

להלן הגדרה של המחלקה Message:

```
class Message
{
private:
    int id;
public:
    void print(const char* msg);
};
```

ובאופן כללי יותר:

```
class <class_name>
{
    // variables and methods comes here ...
};
```

!! לא לשכוח ";" ...

תחומי גישה במחלקה – private, public

למחלקה Message שבדוגמא האחרונה שדה מסוג *int* ששמו *id* בחלק השמור (private) ופונקציה חברה *print(const char* msg)* הקרויה מתודה (method) בחלק הציבורי שלה (public).
המזהים (identifiers) public ו- private הינן מילים שמורות בשפה והן מציינות תחומי גישה למשתנים ומתודות.
public – ניתן לגשת לתחום זה של המחלקה ע"י כולם.
private – רק המחלקה עצמה רשאית לגשת לתחום זה.
ניתן להגדיר יותר מפעם אחת אזורי גישה במחלקה. ההגדרה תקפה עד להגדרה הבאה או עד סוף הגדרת המחלקה (כל הקודם זוכה).
ברירת המחדל של אזורי הגישה למחלקה הינה private, לכן, אם נשמיט את המזהה private בדוגמא האחרונה נקבל את אותה משמעות לשדה *id*.

נרחיב מעט את הדוגמא האחרונה ונדגים כיצד משתמשים במחלקה זו:

```
#include <iostream.h>

class Message
{
    int id;
    void internal_print(const char* msg) { cout << msg << endl; };
public:
    int get_id() { return id; }
    int set_id(int id_) { id = id_; }
    void print(const char* msg, int id_)
    {
        if(id_ == id)
            internal_print(msg);
    }
};

int main()
{
    Message msg;                // msg is object of type Message (msg also called instance of Message)
    msg.id = 2;                 // Error: cannot access private member id declared in class Message
    msg.set_id(2);              // ok! use of public method set_id
    msg.internal_print("hi");   // Error: cannot access private method internal_print declared ...
    int id = msg.get_id();      // ok! use of public method get_id
    Message* pmsg = &msg;       // declare pointer to Message and initialize it to point to msg
    pmsg->print("bye", 2);      // that is the way to use pointers
    return 0;
}
```

אנו רואים כי כעת יש לפונקציות של המחלקה מימוש בתוך המחלקה עצמה. שימו לב לכך שהפונקציה *print* קוראת לפונקציה *internal_print* למרות שהיא פונקציה פרטית. זה מתאפשר מכיוון שלכל פונקציה של המחלקה יש גישה לכל המשתנים והפונקציות של המחלקה, בין אם הם בחלק הפרטי או הציבורי.

<u>אופרטור התחום (::)</u>

כעת משהכרנו את עולם המחלקה (על קצה המזלג) עולה השאלה הבאה:
כיצד נייחס פונקציות בעלות אותו שם ופרמטרים למחלקות שלהן?
ברור שאם היו שתי מחלקות, A ו- B, ולשתיהן היתה מוגדרת פונקציה f() היינו צריכים להבדיל בין f() של A ו- f() של B בדרך כלשהי.
הדרך לעשות כן היא ע"י אופרטור התחום (::) אשר מגדיר משמאלו את התחום ומימינו את המשתנה / פונקציה המשויך לאותו תחום.
כאשר אין מצד שמאל כלום, התחום הוא התחום הגלובלי.
דוגמאות:

```
A::f()                  // f() is a function in the scope of A (struct, class, namespace, ...)
B::f()                  // f() is a function in the scope of B
::print();              // print() is a global function
Faculty::Student::name  // name is a variable declared in Student, which is declared in Faculty
```

<u>מימוש מתודות (methods) של מחלקה וצורת הקריאה אליהן</u>

לא כל המתודות יכולות להיות ממומשות בתוך הגדרת המחלקה. חלק מהמתודות הן ארוכות ובנוסף לסרבול שבכתיבתן במחלקה עצמה, יש לכך השפעות מזיקות לעיתים (יוסבר בהמשך). לכן, כאשר אנו באים לממש פונקציה מחוץ למקום בו היא הוגדרה במחלקה, עלינו להצהיר על הפונקציה לפי כל כללי השיוך שלה.
הפורמט הכללי של פונקציה כזו יהיה:

```
<return value> <scope>::<function name> ( <parameter 1>, <parameter 2>, ...)
{
   // function body goes here
}
```

לדוגמא, את הפונקציה *print* מהדוגמא הקודמת מממשים בצורה הבאה:

```
void Message::print(const char* msg, int id_)
{
    if(id_ == id)
        internal_print(msg);
}
```

הביטו בהגדרת המחלקות הבאה וענו על השאלות שלאחריה:

```
1   class A{
2       class C{
3           int n;
4       public:
5           int f();
6       }; // C
7   public:
8       int n;
9       class D{
10          int n;
11      public:
12          int f();
13      private:
14          class B{
15              int n;
16          }; // B
17      }; // D
18      D::B b;
19      int f();
20  }; // A
```

1. רשמו את שורת ההגדרה של כל הפונקציות f() כאילו שממומשו מחוץ להגדרת מחלקתן.
2. לאילו שדות ניתן לגשת באובייקט מסוג A? לאילו שדות של b (שורה 18) ניתן לפנות? כיצד?
3. אלו אובייקטים ניתן להגדיר ואילו לא? היכן ניתן להגדיר אותם?

this

נניח כי הגדרנו כמה עצמים מטיפוס Message:

```
Message m1, m2, m3, m4;
```

ואנו קוראים לפונקציה print של המחלקה Message עם האובייקט m2.

```
m2.print("m2", 2);
```

כיצד יודעת הפונקציה Message::print על איזה מהעצמים היא פועלת?
מהתבוננות בגוף הפונקציה, אין שום רמז להתייחסות למזהה כלשהו לעצם שעליו הופעלה הפונקציה.

תשובה: המהדר מבצע תרגום ביניים של קוד פונקציות המחלקה, ובמסגרת תרגום זה הוא מוסיף להגדרתן מצביע לעצם שעליו הוא פועל, כפרמטר ראשון. למעשה, הפונקציה print נראית כך לאחר תרגום הביניים של המהדר:

```
void print(Message* this, const char* msg, int id_)
{
    if(id_ == this->id)
        this->internal_print(msg);
}
```

כעת, בקריאה לפונקציה:

```
m2.print("m2", 2);
```

המהדר דואג לביצוע תרגום הביניים הבא:

```
Message::print (&m2, "m2", 2);
```

ניתן להתייחס בעת כתיבת הקוד של הפונקציה למצביע **this** באופן ישיר, אך לא ניתן לשנותו.
לא ניתן להשתמש בשם this עבור מרכיבים אחרים ב- c++ מאחר וזהו שם שמור.

שימושים נוספים ל- this :

1. כדי להבדיל בין משתנה מקומי (או פרמטר, אשר הינו משתנה מקומי) לבין חבר במחלקה ניתן להשתמש במפורשות במצביע this כדי "לעזור" למהדר להבין את "כוונת המשורר":

```
void Message::set_id (int id)
{
    this->id = id;        // id = id wouldn't be such a good idea …
}
```

2. ניתן להשתמש ב- this כערך מוחזר מפונקציה, ואז לאפשר קריאה נוספת לאותו עצם:

```
Message& Message::set_id(int id)
{
    this->id = id;        // same as before
    return (*this);
}

Message m;
m.set_id(2).print("m2", 2);
```

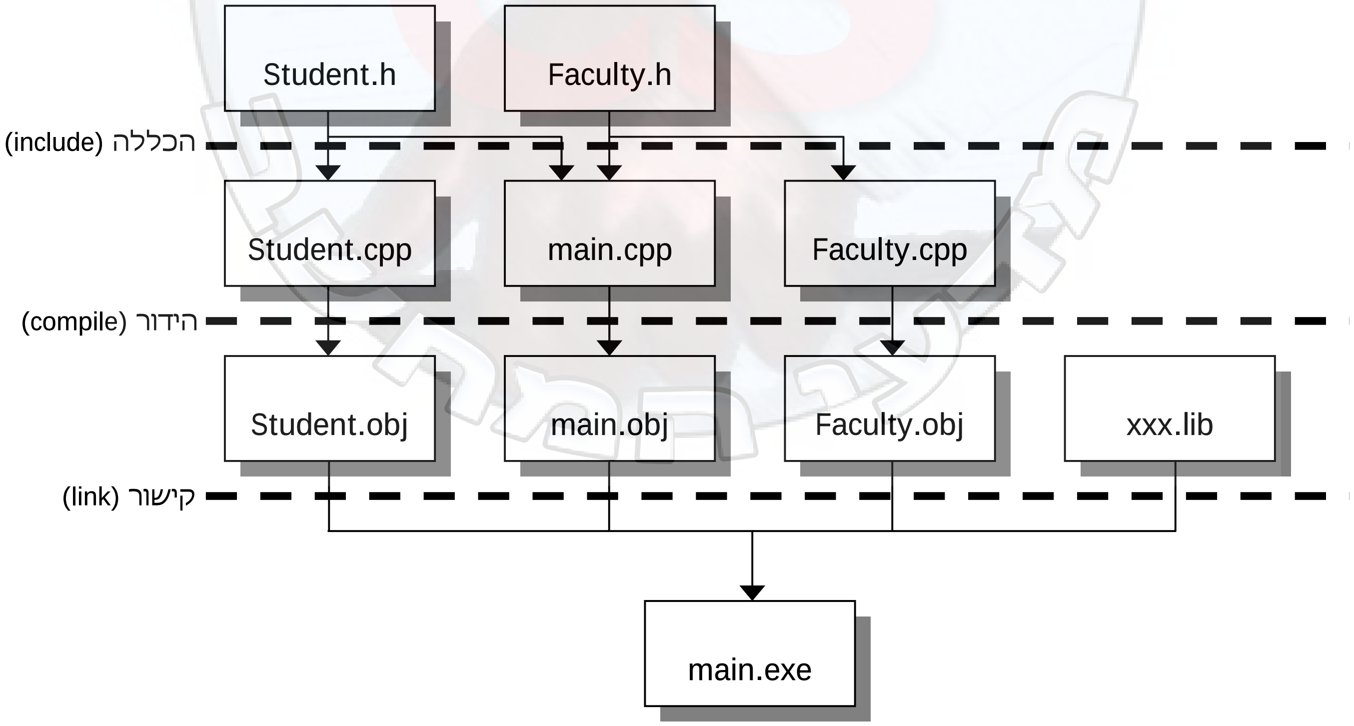
הפרדת המחלקה לקבצי מקור וקבצי מימוש

כאשר קוד פונקציה חברה במחלקה משתרע על פני למעלה ממספר בודד של שורות, יש להפריד את מימוש הפונקציה מההכרזה עליה.
קוד פונקציה המופיע בקובץ הכותר מתורגם להיות inline ולכן, מימוש פונקציות שלמות בקובץ הכותר תגרום ליצירת קוד בכל מקום שהפונקציה נקראת ובכך תגרום להגדלת המקום שתופסת התכנית בזכרון.
גם תהליך ההידור ייארך יותר מאחר וכל קובץ כותר המוכלל בקובץ מימוש יעבור הידור בזמן ארוך יותר.

נניח שברצוננו לכתוב תכנית המנהלת חוג כלשהו באוניברסיטה, ולצורך כך נרצה לממש שתי מחלקות – Faculty ו- Student המייצגות חוג וסטודנט.
נייחד הכרזות על מחלקות והגדרות נוספות רלוונטיות בקבצי כותר שונים:
Faculty.h , Student.h
מימוש למחלקות אלו נשים בקבצי המימוש:
Faculty.cpp , Student.cpp
בנוסף נבנה תכנית המשתמשת באוייקטים מטיפוס המחלקות שהגדרנו בקובץ main.cpp .

בכל קובץ מימוש (.cpp) יש להכליל (#include) את קבצי הכותר הרלוונטיים כדי שלא ייוצרו שגיאות הידור.
בתהליך ההידור נבנים מקבצי המימוש קבצי קישור (object files), המקושרים יחדיו לאחר מכן יחד עם ספריות נוספות (במידת הצורך) בתהליך הקישור (link) ליצירת קובץ הרצה.

באופן תרשימי ניתן לתאר את מבנה קבצי התכנית ושלבי עיבודן כך:



שיעור מספר 5

בניית עצמים והריסתם

נושאי השיעור:

- סוגי משתנים, תחומי הגדרה.
- בנאים ומפרקים של מחלקה.
- בנאים כמתרגמים.
- בנאי ההעתקה (copy constructor).
- פונקציות קבועות (const functions) ועצמים קבועים.
- תחומי אחסון.

סוגי משתנים, תחומי הגדרה

"משך חייו" של משתנה / עצם (object) תלוי בסוגו והיכן ואיך הוגדר.
אלו שלושת סוגי המשתנים העיקריים בתכנית ++C טיפוסית:

משתנים אוטומטיים –
משתנים המוגדרים לוקלית לבלוק בו הם משמשים. משך חייהם הינו מרגע הגדרתם (תחילת הבלוק) ועד סוף הבלוק שבו הוגדרו. בלוק מוגדר לרב ע"י סוגריים מסולסלות ("{" , "}").
דוגמא למשתנים אוטומטיים:

```
if(i <= j)
{
    int automat= i*j;
    i--;     j++;
}
```

משתנים סטטיים –
משתנה סטטי הינו משתנה המשויך לתחום בו הוא מוגדר (בדומה למשתנה אוטומטי), אבל משך חייו הוא מתחילת הריצה ועד סופה.
משתנה סטטי המוגדר בבלוק, מוכר רק בתחומי הבלוק.
משתנה סטטי המוגדר כ- "גלובלי" (בראש קובץ מימוש) מוכר רק בתחום הקובץ עצמו.
דוגמא למשתנים סטטיים:

```
static int static_in_file;     // static in the scope of the file

void foo (int i, int j)
{
    if(i <= j)
    {
        static int static_in_block = i;      // static in the scope of the block
        i--;     j++;
    }
    static_in_block++;       // Error: 'static_in_block' - undeclared identifier
    static_in_file++;          // that's ok !!
}
```

משתנים דינמים –
משתנה דינמי הינו משתנה אשר מוקצה לו מקום בזכרון בזמן ריצת התכנית.
הצורה להקצות מקום ולשחררו ב- ++C היא ע"י האופרטורים new ו- delete אשר יתוארו בהמשך.
דוגמא להקצאת משתנה דינמי:

```
int* dynamic = new int[SIZE];      // allocate array of int of size SIZE and return pointer to it.
```

זמן המחייה של משתנים:

משתנים אוטומטיים – בתחום בו הם מוגדרים.
משתנים סטטיים וגלובליים – כל זמן הריצה.
משתנים דינמים – מהרגע בו הוקצו ועד שחרורם, או סיום התכנית.

בנאים ומפרקים של מחלקה

הבנאי (constructor)

ב- ++C קיים מנגנון אוטומטי לאתחול אובייקטים: constructor. זוהי פונקצית מחלקה מיוחדת הנקראת בזמן יצירת אובייקט חדש מהמחלקה. תפקידה לאתחל את האובייקט בערכים מתאימים לשדות השונים שבו. מכיוון שהמנגנון אוטומטי, אין צורך בפונקצית אתחול כגון ()init .

- ה- constructor מוגדר כפונקציה חברה במחלקה, ששמה (של הפונקציה) הוא כשם המחלקה. פונקציה זו יכולה לקבל פרמטרים כמו כל פונקציה אחרת, אך היא אינה מחזירה אף פרמטר, אף לא void.
- ניתן להגדיר מספר בנאים (function overloading) בתנאי שחתימת הפרמטרים שלהם שונה.
- כאשר לא מוגדר בנאי, המהדר מספק בנאי ברירת מחדל (default constructor), אשר אינו מבצע דבר. באם הוגדר בנאי כלשהו למחלקה, המהדר אינו מספק בנאי ברירת מחדל.
- לא ניתן לקרוא לבנאי מפורשות. המהדר הוא זה "ששותל" קריאה לבנאי בכל יצירת אובייקט.

המפרק (destructor)

המפרק זוהי פונקצית מחלקה מיוחדת בדומה לבנאי, הנקראת בסיום חיי האובייקט. מטרתה של הפונקציה - לאפשר פעולות ניקוי, אם נדרש, בסיום חיי האובייקט. לרב פעולות הניקוי הן שחרור כל הזכרון שהוקצה לטובת האובייקט.

- המפרק מוגדר כפונקציה חברה במחלקה, ששמה (של הפונקציה) הוא כשם המחלקה, בתוספת הקידומת "~".
- המפרק אינו מקבל פרמטרים ולפיכך הוא יחיד עבור מחלקה. המפרק אף אינו מחזיר ערך.

דוגמא לשימוש בבנאים ומפרק במחלקה המייצגת מחרוזת (String).

```
class String
{
    char* str;                  // private member to be allocated later on
public:
    String() { str = 0; }// overwrite default constructor with this constructor
    String( const char* s)  // constructor that accept string and copies it
    {
        str = new char[ strlen(s)+1 ];    // actual allocation
    strcpy(str, s);                         // copy string to allocated memory
    }
    ~String()                   // destructor
    {
    if(str)
        delete [] str;      // delete operator – release memory allocated for object
    }
};
```

בהמשך להגדרת המחלקה נראה כיצד להגדיר אובייקטים ולהפעיל את הבנאי המתאים:

```
int main()
{
    String s1("I know C++");             // call to String(const char*)
    String* s2;
    {                                     // new block starts here !!!
        String s3;                        // call to String()
        s2 = new String(".. So am I !!"); // once again - call to String(const char*)
    }                                     // block ends here !!! call to ~String() of s3 !!!
    delete s2;                            // call to ~String() of s2
    return 0;                             // end of main block - call to ~String() of s1
}
```

בנאים כמתרגמים (ממירי טיפוס)

למעשה, בנאי המקבל פרמטר יחיד מהווה פונקציה להמרת טיפוס. בדוגמא האחרונה, הבנאי של String המקבל *const char כפרמטר הוא פונקציה להמרה מ- *const char ל- String.
מכיוון שכך, ניתן להשתמש בבנאי במספר דרכים:

1. שימוש ישיר בהגדרת אובייקט:

```
String s ("my name is ...");
```

2. בהצבה עם המרה (casting):

```
String s;
s = String("my name is ...");
```

3. בהצבה עם המרה (casting) בסגנון C:

```
String s;
s = (String) "my name is ...";
```

4. בהצבה ללא המרה:

```
String s = "my name is ...";
```

במקרים 2-4 נוצר אובייקט זמני ולאחר מכן הוא מועתק ל- s.
על אופן ההעתקה (וכיצד הוא צריך להתבצע) נרחיב בהמשך.

בצורה שבה הוגדר הבנאי של String אנו מספקים למהדר את האפשרות להמיר, בצורה מרומזת ובצורה ישירה, כל משתנה מטיפוס *const char ל- String. כדי למנוע המרה לא נחוצה ע"י המהדר ושגיאות תכנות אפשריות, ניתן להגדיר בנאי אשר מהווה ממיר טיפוס בצורה מפורשת בלבד. מילת הקסמים היא explicit (מילה שמורה) והיא באה לפני שם הבנאי:

```
class String
{
    ...
    explicit String( const char* s);   // allow only explicit conversion from 'const char*'
    ...
};

String s1("Hi");                       // ok!
String s2 = "and";                     // Error: cannot convert from 'char *' to 'class String'
String s2 = String("goodbye");         // ok!
```

בנאי ההעתקה (copy constructor)

עד עתה ראינו בנאים שמבצעים אתחול של אובייקט על סמך רשימת פרמטרים הניתנים ביצירתו. באופן זה יכולנו להגדיר אובייקטים בצורות שונות, ע"י העברת פרמטרים שונים.
לעיתים נרצה לבנות אובייקט כהעתק של אובייקט קיים.
נרצה לעשות כן דרך הבנאי ולא בצורה של הגדרת אובייקט ולאחר מכן "תיקון" ערכו ע"י פונקציה המבצעת עדכון לאובייקט.
הצורה להגדיר בנאי העתקה (copy constructor) היא הצורה הבאה:

```
class X
{
    ...
    X (const X&);                       // copy constructor for class X
    ...
};
```

נחזור לדוגמא הקודמת של המחלקה String ונגדיר כעת בנאי העתקה:

```
class String
{
    char* str;                          // private member to be allocated later on
public:
    String() { str = 0; }               // overwrite default constructor with this constructor
    String( const char* s);             // constructor that accept string and copies it to
    String( const String& other);       // copy constructor
    ~String()                           // destructor
};

String :: String( const char* s)
{
    str = new char[ strlen(s)+1 ];      // actual allocation
    strcpy(str, s);                     // copy string to allocated memory
}

String  :: ~String()
{
    if(str)
    delete [] str;                      // delete operator – release memory allocated for object
}

String  :: String( const String& other)
{
    if(other.str)
    {
    str = new char [strlen(s)+1 ];      // actual allocation
        strcpy(str, other.str);         // copy string to allocated memory
    }
    else
        str = 0;
}
```

והפעלתו תהיה באופנים הבאים:

```
String s1 (“Hello studies!!!”);         // use of String(const char*)
String s2 (s1);                         // use copy constructor of s2 with s1 as argument
String s3 (String(“Goodbye world”));    // use of String(const char*) and then copy constructor
String s4 = s3;                         // use of copy constructor on assignment on definition (!!)
```

פונקציות קבועות (const functions) ועצמים קבועים

כאשר אנו מתבוננים לעומקן של הפונקציות החברות במחלקה מסוימת ואנו רוצים לחלק אותן לשתי קבוצות, חלוקות רבות אפשריות:
חלוקה אחת אפשרית היא של פונקציות נגישות מחוץ למחלקה (public) וכאלו שאינן (private).
צורה נוספת לחלק פונקציות אלו לשתי קבוצות עיקריות היא לקבוצה של פונקציות מסוג שאילתות (queries) ופונקציות המשנות תוכנו של אובייקט (modifiers).
queries functions – פונקציות שמטרתן להחזיר מידע על תוכן האובייקט או על מצבו.
modifiers functions - פונקציות אשר משנות את מצב האובייקט, ערכי שדה אחד או יותר בו, או אפילו מאפשרות גישה לא בטוחה לשדות של האובייקט.

פונקציה קבועה (constant function) הינה פונקציית שאילתא (query) והיא מאופיינת ע"י המילה const המופיעה מיד לאחר שורת הפרמטרים שמקבלת הפונקציה.
המשמעות בעת כתיבת const על כל פונקציה שאמורה להיות קבועה היא בכך שכאשר נגדיר אובייקט קבוע (const object) נוכל להשתמש אך ורק באותן פונקציות אשר הוגדרו כקבועות.

נדגים בעזרת הדוגמא הבאה:

```
class Person
{
    char* name;
    int id;
public:
    Person(const char* s) { name = 0; set_name(s); }
    ~Person() { if(name) delete [] name; }
    const char* get_name() const ;              // const function
    inline char* get_name() { return name; }    // non-const function, only for the sake of this example
    void set_name (const char*);
    inline int get_id() const { return id; }    // return value is copy of 'id'
    inline void set_id (int id) { this->id = id; }
};

const char* Person :: get_name() const
{
    return name;
}

void Person :: set_name (const char* s)
{
    if (name)
        delete [] name;
    name = new char [ strlen(s)+1 ];
    strcpy(name, s);
}

void main()
{
    Person p1 ("Bobo");
    const Person p2 ("Momo");   // const object of type Person
    p1.set_id (1234);           // ok, non const object
    p2.set_id (4321);           // Error: cannot convert 'this' pointer from 'const class Person' to 'class Person &'
    int p1_id = p1.get_id();    // ok, const function does not change object
}
```

תחומי אחסון

<u>Composition</u> – מצב שבו אובייקט הינו חלק מאובייקט אחר (למשל אובייקט מסוג String הינו אחד השדות באובייקט מסוג Person). אובייקט שכזה נוצר כאשר האובייקט שמכיל אותו נוצר, הוא חי כל עוד האובייקט שמכיל אותו חי והוא נמחק (המפרק שלו מופעל) כאשר האובייקט שמכיל אותו נמחק.
סדר האתחול בעת יצירת אובייקט X המכיל בתוכו אובייקט Y הינו:
מופעל הבנאי של Y ורק לאחר מכן הבנאי של X.
סדר ההריסה של אובייקט כזה הינו:
מופעל המפרק של X ורק לאחר מכן המפרק של Y.

נשאלת השאלה:
כיצד מאתחלים אובייקט המוכל באובייקט אחר, אם הבנאי שלו נקרא קודם?

תשובה:
ניתן להעביר לבנאי של אובייקט פנימי פרמטרים ע"י שורת אתחול הפרמטרים המופיעה לפני גוף הבנאי. שורה זו מאתחלת משתנים בפורמט הבא:

```
<class_name> :: <class_name> ( <parameter1>, <parameter2> )
    : <field_name1 (parameter1) > , <field_name2 (parameter2) >
{
    // constructor body
}
```

אתחול פרמטרים בשורת האתחול הינו הכרחי עבור מצב של הכלה של אובייקט, של אתחול משתנים קבועים (const fields), של משתני ייחוס, ושל מחלקות יסוד (ידובר בהמשך).
דוגמא לאתחול פרמטרים בשורת אתחול הפרמטרים:

```
class String { ... };
class Person
{
    String name;
    int id;
public:
    Person(const char* nam, int id_ ) ;
};

Person :: Person (const char* nam, int id_ ) : name(nam) , id(id_)
{
    // constructor body.
    // In this case – there is nothing to do.
}
```

לא הכרחי. ניתן לאתחול בתוך גוף הבנאי באופן הרגיל: id = id_;

הכרחי, לצורך אתחול name ע"י קריאה לבנאי String(const char*)

<u>Association</u> – מצב בו אובייקט מכיל ייחוס או מצביע לאובייקטים אחרים. במקרה כזה לא מופעל בנאי עבור המצביעים או הייחוסים, מאחר ואלו לא אובייקטים.
יש לשים לב, שבעוד שאת המצביעים ניתן לאתחל בגוף הבנאי, את משתני הייחוס (reference) יש לאתחל בשורת אתחול הפרמטרים.

שיעור מספר 6

העמסת אופרטורים (operator overloading)

נושאי השיעור:

- אופרטורים – כללי.
- אופרטורים לוגיים.
- אופרטורים אריתמטיים.
- פונקציות חברות (friend functions).
- אופרטורים אונריים.
- אופרטורים בינריים.
- אופרטור ההשמה (=).
- אופרטור האינדקס ("[]").

אופרטורים – כללי

כחלק מעקרון הפשטת נתונים (Data Abstruction), ניתן לספק משמעות לאופרטורים המופעלים על טיפוסים שהמשתמש הגדיר. הנוחות בהגדרת אופרטורים על פני שימוש בפונקציה ייעודית לכך היא רבה והיא תודגם בהמשך.

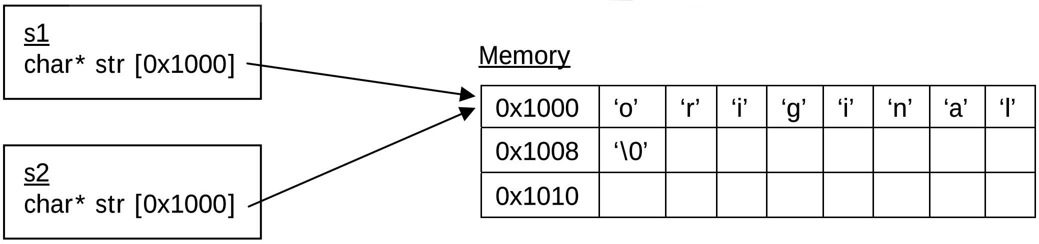
לדוגמא, עבור המחלקה String משיעור קודם, היינו רוצים לבצע את הפעולות הבאות:

```
String s1("Shabat "), s2("Shalom");
if(s1 == s2)          // same as declaring function String::equal(const string&) and using s1.equal(s2)
    s1 += s2;         // same as declaring function String::concatinate(const string&) and using s1.concatinate(s2)
```

פעולת השוואה למשל, היא "טבעית" ובעלות משמעות עבור מחרוזות כמו גם עבור טיפוסים אחרים המוגדרים ע"י משתמש.

אופרטור נוסף הוא אופרטור ההשמה (=) הקיים באופן טבעי עבור כל טיפוס, אך הוא מבצע פעולות סטנדרטיות של השמה של שדה אחד למשנהו (העתקה).
לא תמיד נרצה את המימוש הסטנדרטי הנ"ל, מה גם שהוא עלול להיות הרה אסון.
להלן דוגמא בהנחה שלא הוגדר אופרטור השמה ע"י המתכנת:

```
{
    String s1("original");    // allocation at constructor + copying string
    String s2;                // empty string – no allocation on constructor
    s2 = s1;                  // should be allocation + copy, but in fact there is a copy of pointer
}                             // here comes s1, s2 destructor ...
```



0x1000	'o'	'r'	'i'	'g'	'i'	'n'	'a'	'l'
0x1008	'\0'							
0x1010								

הבעייה תווצר עם הפעלת המפרק של s1 ו- s2. יתבצע שחרור כפול של אזור בזכרון שהוקצה פעם אחת בלבד. יתרה מכך, שינוי מחרוזת אחת ישפיע על המחרוזת השניה.

הדרך לפתור בעייה זו, היא כמובן להגדיר אופרטור = ולממשו בצורה הנכונה.
באופן כללי, אם המשתמש הגדיר מחלקה X:

```
Class X
{
    ...
};
```

הוא יכול באמצעות מנגנון העמסת אופרטורים, לקבוע על אובייקטים מהמחלקה

```
X   x1, x2, x3;
```

את המשמעויות של האופרטורים, כגון:

- אופרטורים לוגיים:

```
x1 < x2
x2 == x3
x2 >= x1
```

- אופרטורים אריתמטיים:

```
x1 + x2
x2 – x3
x3 * x3
x2 / x1
```

- אופרטורי הצבה ואופרטורי הצבה משולבים:

```
x3 = x1
x3 = x1 + x2
x2 += x3
x3 *= x2
```

- אופרטורים אונרים:

```
x1++
--x2
!x3
~x2
-x1
```

- אופרטורי המרה:

```
(int) x1
(double) x2
```

- אופרטורי קלט-פלט:

```
cout << x1 << x2
cin >> x2 >> x3
```

וכן אופרטורים נוספים.

מימוש אופרטור הוא כמימוש כל פונקציה אחרת.
אופרטור יכול להיות חבר מחלקה או סתם פונקציה גלובלית.
בהמשך נראה כיצד להגדיר אופרטורים (אונרים ובינארים) בתוך ומחוץ למחלקה.
כדי להגדיר אופרטור אנו משתמשים במילה השמורה operator .
דוגמאות:

```
String operator+ (const String& left, const String& right);   // global binary operator + to add two strings
bool operator< (const String& left, const String& right);     // global binary operator < to compare two strings
Date& Date::operator= (const Date& other);                     // internal assignment operator for class Date
Complex operator~ (const Complex& self);                       // global unary operator ~ to deal with Complex types
```

אופרטורים לוגיים

קבוצת האופרטורים הלוגיים כוללת אופרטורים להשוואה בין אובייקטים שונים המחזירים תשובה בוליאנית, כלומר אמת או שקר.

לדוגמא עבור המחלקה Rational המייצגת מספר רציונלי (p/q כאשר p ו- q מספרים טבעיים), נגדיר את האופרטורים הלוגיים :

```
class Rational
{
    int m_n;  // numerator
    int m_d;  // denominator
public:
    Rational(int n=0, int d=1)
        {m_n=n; m_d=d;}

    // logical compare operators
    inline bool operator<(const Rational &r)    const  { return m_n*r.m_d < r.m_n*m_d; }
    inline bool operator>(const Rational &r)    const  { return r < *this;}
    inline bool operator<=(const Rational &r)  const  { return !(*this > r); }
    inline bool operator>=(const Rational &r)  const  { return !(*this < r); }
    inline bool operator!=(const Rational &r)   const  { return (*this < r) || (r < *this); }
    inline bool operator==(const Rational &r)  const  { return !(*this != r); }
};
```

ונוכל להשתמש בהם באופן הבא:

```
Rational r1(2,3) , r2(4,5) ;
...
if(r1 == r2)                              // call function r1.operator==(r2)
    cout << "r1 equals r2" << endl;
else
    cout << "r1 differs from r2" << endl;
```

לחלופין, נוכל להגדיר את אותם אופרטורים כגלובליים, באופן הבא:

```
class Rational
{
    int m_n;  // numerator
    int m_d;  // denominator
public:
    Rational(int n=0, int d=1)
        {m_n=n; m_d=d;}
};

// logical compare operators
inline bool operator<(const Rational &l, const Rational &r)     { return l.m_n*r.m_d < r.m_n*l.m_d; }
inline bool operator>(const Rational &l, const Rational &r)     { return r < l;}
inline bool operator<=(const Rational &l, const Rational &r)   { return !(l > r); }
inline bool operator>=(const Rational &l, const Rational &r)   { return !(l < r); }
inline bool operator!=(const Rational &l, const Rational &r)    { return (l < r) || (r < l); }
inline bool operator==(const Rational &l, const Rational &r)   { return !(l != r); }
```

שאלה:
מה הבעייתיות בהגדרת אופרטורים גלובליים? האם הקוד, כפי שכתוב לעיל יעבור הידור?
תשובה תנתן בהמשך, וגם פתרון לבעייה.

באופן כללי, התחביר של אופרטור לוגי עבור מחלקה X הוא כדלהלן:

אופרטור בתוך מחלקה:

```
                <
                >
                >=
bool operator   <=   (const  X& right) const
                ==
                !=
```

אופרטור גלובלי:

```
                <
                >
                >=
bool operator   <=   (const  X& left, const  X& right);
                ==
                !=
```

<u>אופרטורים אריתמטיים</u>

קבוצת האופרטורים האריתמטיים כוללת אופרטורים לביצוע פעולות מתמטיות בין עצמים שונים המחזירים אובייקט ע"י ערך (by value). הערך המוחזר הוא העתק של אובייקט המוגדר זמנית בתוך פונקציית האופרטור (על המחסנית).

לדוגמא עבור המחלקה Rational לעיל, נגדיר את האופרטורים האריתמטיים בצורה גלובלית:

```
// arithmetic operators
Rational operator+(const Rational &l, const Rational &r)
{
    Rational temp(l.m_n * r.m_d + r.m_n*l.m_d, l.m_d * r.m_d);
    Return temp;
}

Rational operator-(const Rational &l, const Rational &r)
{
    Rational temp(l.m_n * r.m_d - r.m_n*l.m_d, l.m_d * r.m_d);
    Return temp;
}

Rational operator*(const Rational &l, const Rational &r)
{
    Rational temp(l.m_n * r.m_n, l.m_d * r.m_d);
    Return temp;
}

Rational operator/(const Rational &l, const Rational &r)
{
    Rational temp(l.m_n * r.m_d, l.m_d * r.m_n);
    Return temp;
}
```

באופן כללי, התחביר של אופרטור אריתמטי עבור מחלקה X הוא כדלהלן:

אופרטור בתוך מחלקה:

```
             +
             -
X operator   *   (const  X& right) const
             /
             %
```

אופרטור גלובלי:

```
             +
             -
X operator   *   (const  X& left, const  X& right);
             /
             %
```

פונקציות חברות (friend functions)

בפיסקה זו נתייחס למושג "פונקציה חברה" (friend) במובן של פונקציה "ידידה" ולא במובן "השייכות" הרגיל של פונקציה חברה (member function), כפונקציה השייכת למחלקה.

לעיתים המתכנת מעוניין לאפשר למחלקה או לפונקציה (ובכלל זה לאופרטורים גלובליים) לגשת לשדות המצויים בחלק הפרטי (private) של המחלקה אותה הוא כותב.
כדי לאפשר גישה סלקטיבית זו, ניתן להצהיר על מחלקה או פונקציה, כ"חברה" של המחלקה.
מילת הקסמים היא friend והיא באה לפני חתימת הפונקציה.
על ההצהרה להיות כתובה בהגדרת המחלקה.

השימוש העיקרי הוא בד"כ באופרטורים גלובליים, אשר יש לאפשר להם גישה לנתונים פרטיים.
שימוש בפונקציות חברות בצורה נרחבת אינו רצוי ורצוי להמעיט בו.

עבור אותם אופרטורים אריתמטיים שהוגדרו קודם, נגדיר במחלקה Rational:

```
class Rational
{
    int m_n;  // numerator
    int m_d;  // denominator
public:
    Rational(int n=0, int d=1)
        {m_n=n; m_d=d;}

    friend Rational operator+(const Rational &l, const Rational &r);
    friend Rational operator-(const Rational &l, const Rational &r);
    friend Rational operator*(const Rational &l, const Rational &r);
    friend Rational operator/(const Rational &l, const Rational &r);
};
```

בצורה זו אנו פותרים את הבעייה שהוצגה קודם ומאפשרים גישה ישירה לאופרטורים גלובלים.

אופרטורים אונריים

אופרטורים אונריים הם אופרטורים הפועלים על אופרנד יחיד והם מספקים הפשטה על אובייקטי המחלקה. לדוגמא, עבור המחלקה Rational היינו רוצים לאפשר פעולות כגון:

```
Rational  r1 (4,5), r2;
r2 = ~r1;     // inverse: r2 = 5/4
r2 = -r1;     // negate: r2 = -4/5
```

האופרטור הראשון, "~" מחזיר את השבר ההופכי.
האופרטור השני הוא "-" אונרי, והוא מחזיר את השבר הנגדי.

האופרטורים "++" ו- "--" הינם אופרטורים אונריים והם יכולים לבא לפני אובייקט (prefix - ++X) או לאחריו (postfix - X++). כדי להבדילם, יש צורך בחתימת פונקציה שונה, ולכן האופרטורים התחיליים (prefix) מוגדרים כפונקציה ולה פרמטר אחד והוא האובייקט שעליו היא פועלת, בעוד שלאופרטורים הסופיים (postfix) יש שני פרמטרים - האובייקט שעליו היא פועלת ו- int.

```
X& operator++( const X& self);  // prefix
X   operator++( X& self, int);     // postfix
```

מימוש חלקי של Rational עם אופרטורים אונריים יראה כך:

```
class Rational
{
    int m_n;  // numerator
    int m_d;  // denominator
public:
    Rational(int n=0, int d=1)
        {m_n=n; m_d=d;}

    // unary operators
    Rational  operator~() const            // inverse
        { return Rational(m_d, m_n); }
    Rational  operator-() const            // negate
        { return Rational(-m_n, m_d); }
    Rational &operator++(void)             // prefix ++
        { m_n += m_d; return *this; }
    Rational  operator++(int)              // postfix ++
    {
        Rational temp(*this);              // save value before increment
        m_n += m_d;                        // increment
        return temp;                       // return value before increment
    }
    Rational &operator--(void)             // prefix --
        { m_n -= m_d; return *this; }
    Rational  operator--(int)              // postfix --
    {
        Rational temp(*this);
        m_n -= m_d;
        return temp;
    }
};
```

<u>אופרטורים בינריים</u>

אופרטורים בינריים הם אופרטורים הפועלים על שני אופרנדים.
כאשר אופרטורים אלו מוגדרים במחלקה, פעולת האופרטור מתבצעת על האובייקט שמשמאל לאופרטור, כאשר הפרמטר שהוא מקבל הוא האובייקט שמימין לאופרטור:

```
bool  Rational :: operator==(const Rational& other);
Rational r1(4,6), r2(5,7);
bool equal = (r1 == r2);          // bool equal = ( r1.operator==(r2) )
```

כאשר אופרטורים אלו מוגדרים מחוץ למחלקה, בצורה גלובלית, פעולת האופרטור מתבצעת על שני אובייקטים: הפרמטר הראשון הינו האובייקט שמשמאל לאופרטור והפרמטר השני הוא האובייקט שמימין לאופרטור:

```
Rational   operator+(const Rational& left, const Rational& right);
Rational r1(4,6), r2(5,7);
Rational sum = r1+r2;             // Rational sum = operator+(r1, r2);
```

אופרטור ההשמה (=)

אופרטור ההשמה (assignment operator) מועמס מראש לכל טיפוס, למעט מערכים. הפעולה שהוא מבצע הינה העתקת השדות של האובייקט לאובייקט עליו התבצעה פעולת ההשמה. כאשר האובייקט מכיל שדות שאף אחד מהם אינו מצביע, פעולת ההעתקה מספיקה. לדוגמא, במחלקה Rational אין מצביעים ולכן אין צורך להגדיר את האופרטור "=" שוב. האופרטור הקיים מספיק מאחר והוא מעתיק שני שדות של integer.

במחלקה String שהספקנו להכיר, אופרטור ההשמה הבסיסי אינו מספיק מאחר ויש להעתיק את המחרוזת, ולא את המצביע אליה! לצורך כך יש להקצות למחרוזת מקום בזיכרון והעתיקה לשם.

אופרטור ההשמה צריך לבצע את הפעולות הבאות, לפי הסדר:
1- בדיקה מפני הצבה עצמית.
2- שחרור זכרון קודם.
3- הקצאת זכרון חדש והעתקת הערך החדש.
4- החזרת האובייקט שעליו נעשתה פעולת ההשמה, כייחוס (reference).

התחביר הכללי של האופרטור "=" על מחלקה X:
X& operator= (const X& right);
דוגמא – המחלקה String ומימוש אופרטור ההשמה עבורה:

```
class String
{
    char *str;
    void copy(const char *s)
    {
        if(s==0 || strcmp(s,"")==0)            // empty parameter string
            reset();
        else
        {
            str = new char[strlen(s)+1];
            strcpy(str, s);
        }
    }
    void reset()   { str=0;}
public:
    String(const char *s) { copy(s); }
    ~String()
    {
        delete [] str;
        reset();
    }
    String &operator=(const String& s)  // assignment operator
    {
        if(m_str!=s.m_str)                  // check for self assignment
        {
            delete [] m_str;
            copy(s.m_str);
        }
        return *this;
    }
};
```

אופרטור האינדקס ("[]")

אופרטור האינדקס (subscript) מאפשר גישה טבעית לאיבר מסוים במערך.
דוגמא לשימוש באופרטור האינדקס על מערך שלמים ועל מערך אובייקטים:

```
int         int_array[5];     // array of size 5 of integers
Rational    rat_array[5];     // array of size 5 of Rational objects

int         i = int_array[3];
Rational    r = rat_array[2];
```

עבור מחלקות מסוימות יש משמעות לאופרטור האינדקס.
למשל עבור מחלקה הממששת מחרוזת (String) הגישה עם אופרטור האינדקס היא למקום מסוים במערך התווים השמור בתוך האובייקט:

```
class String
{
    char *str;
public:
    String(const char *s);
    ~String();
    char &operator[](int index)
    {
        static char c;
        if(index>=0 && index<len)
            return str[index];
        else {
            cerr << "Error:out of range " << endl;
            return c;              // we should use an exception
        }
    }
    char operator[](int index) const
    {
        if(index>=0 && index<len)
            return str[index];
        else {
            cerr << "Error:out of range " << endl;
            return 0;              // we should use an exception
        }
    }
};
```

שימו לב שהגדרנו שתי גרסאות של האופרטור "[]".
<u>הראשונה</u> (ללא const) משמשת עבור אובייקטים שאינם const ולכן ניתן לשנותם. הערך המוחזר הוא ייחוס למקום הנדרש במערך התווים str. שינוי של הערך המוחזר יגרור שינוי במחרוזת עצמה!
<u>השנייה</u> (עם const) משמשת עבור אובייקטים קבועים (const) והיא אינה מחזירה ייחוס למקום הנדרש במערך התווים str, כי אם העתק של התו במערך התווים.

```
const String  c_str("this is a const string");
String        r_str("this is a regular string");
r_str[0]    = 'T';        // correct to upper case – call to 'char& String:: operator[](0)'
char c      = c_str[4]    // get 4th letter – call to 'char String:: operator[](4) const'
```

שיעור מספר 7

קלט-פלט ועבודה עם קבצים

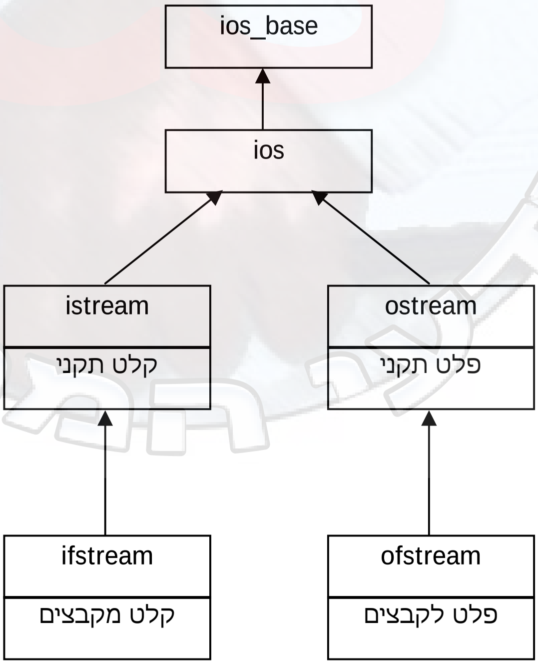
נושאי השיעור:

- קלט ופלט ב- ++C.
- העמסת אופרטורי הקלט והפלט.
- פונקציות קלט ופלט שימושיות.
- מחלקות לטיפול בקבצים.
- קריאה וכתיבה עם פורמט מקבצים.
- קריאה וכתיבה ל/מקבצים בינארים.

קלט ופלט ב- ++C

מנגנון הקלט והפלט ב- ++C הוא מנגנון מבוסס זרמים (streams): מקור הקלט ויעד הפלט הם מעין "זרמים" של נתונים פנימה והחוצה.
זרמי הקלט והפלט הם זרמים של בתים: התכנית יכולה לקרוא ולכתוב נתונים כבתים בודדים או כנתונים מורכבים יותר: שלמים, מחרוזות, אובייקטים וכו'.
זרמי הקלט והפלט מיוצגים ב- ++C בספרייה התקנית ע"י מחלקות.
שני צידי הקלט-פלט הם אובייקטים.

הירככייה חלקית של מחלקות הקלט-פלט מתוארות בתרשים הבא. היררכיה זו הינה יחס של ירושה בין מחלקות, אותה נלמד בהמשך.



המחלקות istream ו- ostream מטפלות בקלט מהמקלדת ופלט למסך בהתאמה.
האובייקטים cin ו- cout הם מטיפוסי המחלקות הללו, בהתאמה.
המחלקות הללו, בנוסף להגדרת האופרטורים "<<" (ostream - cout) ו- ">>" (istream - cin), מגדירות אופרטורים נוספים וכן מתודות נוספות. בסעיפים הבאים נכיר אופרטורים ומתודות אלו.

<u>העמסת אופרטורי הקלט והפלט</u>

המחלקה ostream מגדירה אופרטור "<<" עבור כל טיפוס בסיסי קיים, כלומר:

```
class ostream
{
    ...
    ostream& operator<< (int i);
    ostream& operator<< (const char* str);
    ...
};
```

צורת ההגדרה של האופרטורים מאפשרת שרשור של קריאות להדפסה למסך בשורה אחת, כלומר, במקום לכתוב:

```
int i=9;
double d=0.8;
cout << i ;        // call 'ostream& ostream :: operator<< (int)'
cout << "  " ;     // call 'ostream& ostream :: operator<< (const char*)'
cout << d ;        // call 'ostream& ostream :: operator<< (double)'
cout << '\n' ;     // call 'ostream& ostream :: operator<< (char)'
```

ניתן לכתוב:

```
int i=9;
double d=0.8;
cout << i << "  " << d << endl;
```

כאשר endl הינו תחליף לכתיבת התו '\n'.
במידה ונרצה להעמיס את אופרטור הפלט כדי שיתמוך בהדפסת פלט של אובייקטים ממחלקות אותן הגדרנו, עלינו להגדיר אופרטור גלובלי באופן הבא:
עבור מחלקה X יש להגדיר:

```
ostream& operator<< (ostream& os, const X& to_print);
```

בגוף האופרטור ניתן להשתמש באופרטורים המוגדרים כבר של cout ובכך להדפיס את תוכן האובייקט. דוגמא:

```
class Point
{
    int start, end;
public:
    explicit Point(int st=0, int en=0) : start(st), end(en) { }
    friend ostream& operator<< (ostream& os, const Point& to_print);
};
ostream& operator<< (ostream& os, const Point& to_print)
{
    os << '[' << to_print.start << ',' << to_print.end <<']'  ;
    return os;
}

int main()
{
    Point p(1,3), q(2,5);
    cout << "first point = " << p << endl << "second point = " << q << endl;
}
```

המחלקה istream מגדירה אופרטור "<<" עבור כל טיפוס בסיסי קיים, כלומר:

```
class istream
{
    ...
    istream& operator>> (int& i);
    istream& operator>> (double& d);
    istream& operator>> (char& c);
    ...
};
```

באותו אופן שבו פועל cout פועל cin. ניתן לשרשר קריאות של קלט מהמקלדת בשורה אחת.

```
int i ;
double d ;
cin >> i >> d ;      // read from KB integer and then double
```

על אובייקטים מהמחלקות היורשות מ- ios מוגדר האופרטור "!", אשר מחזיר ערך בוליאני לגבי מצבו של האובייקט. למשל, האופרטור הנ"ל יחזיר ערך TRUE אם הופעל על אובייקט מטיפוס istream (cin) כאשר יש שגיאה בקלט.
דוגמא:

```
int n;
double d;
cout << "Enter and integer and a double: ";
cin >> n >> d;
if( !cin )
    cout << "Error in input! " << endl ;
else
    cout << "you entered: " << n << ", " << d << endl ;
```

<u>פונקציות קלט ופלט שימושיות</u>

- הפונקציות ()int bad ו- ()int good בודקות האם הפעולה האחרונה שנעשתה על אובייקט הקלט-פלט הצליחה ומחזירות ערך מתאים.

- הפונקציה ()get בגרסה הקוראת תו בודד, קוראת את התו הבא מהקלט, ללא דילוג מעל תווים "לבנים". בכך היא שונה מהאופרטור "<<".

- הפונקציה ()get בגרסה הקוראת מחרוזת שלמה, קוראת שורה שלמה עד לתו סיום מחרוזת ('\n' , '\0').

- הפונקציה ()getline קוראת מחרוזת שלמה (בדומה ל- ()get), אבל היא מחליפה את התו '\n' בתו סיום מחרוזת רגיל – '\0'.

- הפונקציה ()eof בודקת האם התו האחרון שנקרא הינו תו סיום קלט (ctrl+z).

- הפונקציה ()width מאפשרת לקבוע את מידת הריווח בעת הדפסה לאובייקט הפלט.

מחלקות לטיפול בקבצים

המחלקות ifstream ו- ofstream מייצגות זרמי קלט-פלט בהתייחס לקבצים:

- ifstream מייצגת זרם קלט מתוך קובץ, ומאפשרת לקרוא ממנו נתונים בדומה לקלט מתוך אובייקט מהמחלקה istream. עקב יחס הירושה שביניהן, השימוש בהן די דומה.

- ofstream מייצגת זרם פלט אל קובץ, ומאפשרת לכתוב אליו נתונים בדומה לפלט מתוך אובייקט מהמחלקה ostream. גם כאן, עקב יחס הירושה שביניהן, השימוש בהן הוא אחיד.

פתיחת קובץ תתבצע ע"י הפונקציה ()open שהגדרתה:

```
void open (const char* s, ios_base::openmode mode);
```

openmode הוא טיפוס enum המוגדר במחלקה ios_base וכולל את הקבועים:

in - פתיחה לקריאה
out – פתיחה לכתיבה
trunc – דריסת קובץ בפתיחה לקריאה, אם הוא קיים כבר
app – הוספה לקובץ בפתיחה לקריאה, אם הוא קיים כבר

ניתן להרכיב שילוב ערכים בין הקבועים הללו, ע"י שימוש באופרטור "|".
לדוגמא:

```
ofstream fs;
fs.open (“file.txt”, ios_base::out | ios_base::app);    // open “file.txt” for writing, in append mode
```

ברירת המחדל של הפונקציה ()open היא ios_base::in עבור אובייקטים מטיפוס ifstream ו- ios_base::out עבור אובייקטים מטיפוס ofstream.
ניתן להגדיר אובייקט מהמחלקות הללו בעזרת בנאי המקבל שם קובץ לפתיחה:

```
char buf[1024];
ifstream fin (“input.txt”);       // open file “input.txt” for reading – ios_base::in
ofstream fout (“output.txt”);  // open file “output.txt” for writing – ios_base::out

while (!fin.eof())
{
    fin.getline(buf, 1024);      // read one line from file into buffer
    fout << buf;                     // write buffer into output file
}
```

סגירת קובץ תתבצע ע"י הפונקציה ()close שהגדרתה:

```
void close (void);
```

בניגוד לשימוש בקבצים בשפת C, כאשר שם חייבים לסגור את הקובץ ע"י קריאה לפונקציה ()fclose לפני שהתוכנית מסתיימת, המפרק של המחלקות ifstream ו- ofstream דואג לבצע עבורנו סגירה של הקבצים שנפתחו.

קריאה וכתיבה עם פורמט מקבצים

השימוש באופרטורים לקלט ופלט בקבצים נעשה בדומה לשימוש באופרטורים ">>" ו- "<<" במחלקות istream ו- ostream.
יש לשים לב שצורת הקריאה מקובץ והכתיבה אליו נעשית עם עריכה (formatting).
כאשר קוראים מתוך קובץ טקסט, בנוסף לקריאת תוכנו של הקובץ תו אחרי תו, מתבצעת המרה של הנתונים שנקראו מהקובץ לערך המתאים שהתבקשנו לקרוא.
לדוגמא – מימוש אפשרי של האופרטור ">>" המופעל על טיפוס שלם:

```
istream& istream :: operator >> (istream& os, int& val)
{
    // we assume fp is a 'FILE *' type varaible that ostream object holds as a private member,
    // and that it points to the next position to read from, in the file
    int ch;
    while (isspace(ch=fgetc(fp))) { }              // read white characters
    if(eof())                                      // end of file reached
    {
        bad_flag = true;
        return os;
    }
    val = ch – '0';                                // read first character and translate it
    while(!(isspace(ch=fgetc(fp))) && !eof())      // read the rest of the characters and translate them
    {
        val *= 10;
        val += ch – '0';
    }
    // here we should move fp one character backward, since we read one extra character
}
```

כפי שניתן לראות, קריאת תו מקובץ טקסטואלי, אינה דבר של מה בכך והיא דורשת עבודה לא קטנה. יתרה מכך, נניח שברצוננו לאתחל אובייקט בעל שדות רבים מתוך קובץ. נממש לצורך כך אופרטור גלובלי עבור המחלקה שלנו ושם ייראה הקוד, פחות או יותר כך:

```
istream& operator >>(istream& is, const MyClass& x)
{
    is >> x.member1;
    is >> x.member2;
    is >> x.member3;
    is >> x.member4.internal_member1;
    is >> x.member4.internal_member2;
    is >> x.member5;
    // ...
    return is;
}
```

הרבה תכניות בנויות על אינטראקציה עם המשתמש. להרבה משתמשים חשוב לקבל תגובה מהירה מהתכנית אותה הם מפעילים.
צורה כזאת של עבודה עם קבצים היא איטית יחסית ונדרשת דרך מהירה יותר לקרוא ולכתוב נתונים מ/לקבצים.

קריאה וכתיבה ל/מקבצים בינארים

כאמור – נדרשת האפשרות לקרוא מקבצים (ולכתוב אליהם) בצורה מהירה יותר ויעילה יותר.

כאשר קובץ מכיל ערכים בצורה טקסטואלית, עליו להכיל מספר תווים עבור כל טיפוס וכן תווים "לבנים" להפרדה בין ערך למשנהו.
לדוגמא: למרות שבפועל משתנה מסוג double מיוצג ע"י 4 בתים, ייתכן שיתפוס מספר בתים רב יותר בקובץ. אם נתבונן במספר 123456.000001 הרי שמספר זה יתפוס 14 בתים בקובץ טקסט, כולל התו "רווח" שיבוא לאחר המספר. ייצוג בזבזני זה היה נמנע, וכך גם זמן ההמרה הנדרש כדי להמיר "מחרוזת" למספר, אם הקובץ היה מכיל 4 בתים עם המספר עצמו. מאחר והפורמט לקריאה מקובץ ידוע גם כך למתכנת, אין טעם לבזבז זמן יקר על המרה ממחרוזות לערכים.
בנוסף, מכיוון שאובייקט הוא בעל גודל ידוע מראש, ניתן לקרוא נתונים ישירות לתוכו (יודגם בהמשך).

קריאה וכתיבה לקבצים לא טקסטואלים, נקראת קריאה וכתיבה לקבצים בינארים.
הפונקציות לקריאה וכתיבה הן ()read ו- ()write אשר מקבלות מצביע לחוצץ (buffer) בתור פרמטר ראשון וכמות בתים הנדרשת לקריאה/כתיבה והן מבצעות קריאה/כתיבה של הבתים הללו מ/לקובץ:

```
istream& read (char* buf, int size);
ostream& write (const char* buf, int size);
```

דוגמאות:

```
class Point
{
    int x, y;
public:
    explicit Point(int xx=0, int yy=0) : x(xx), y(yy) {}
};

int main()
{
    ifstream fin( “input.txt” );
    ofstream fout( “output.txt” );
    Point triangle[3] , p(2,5) ;
    fin.read( (void*)triangle, sizeof(Point)*3 );// read data for three objects at once
    fout.write( (void*)p , sizeof(Point) );           // write one Point object to output file
    return 0;
}

*********************************************************************
int main()
{
    ifstream fin( “input.txt” );
    int ** matrix = new int* [100];                         // allocate array of 100 pointers
    for(int i=0; i<100; i++)
    {
        matrix[i] = new int[100];                           // allocate 100 rows of 100 columns each
        fin.read( (void*)matrix, sizeof(int)*100 );         // read data for 100 integers at once
    }
    // do something with matrix and finally release it’s memory
    return 0;
}
```

שיעור מספר 8

תבניות - Templates

נושאי השיעור:

- מבוא.
- תבניות של פונקציות.
- תבניות של מחלקה.
- דברים שכדאי לדעת על תבניות

מבוא

תבניות (templates) הוא מנגנון מתקדם ב- ++C המספק בידי המתכנת עוצמה תכנותית והשגת יעילות גבוהה במיוחד בכתיבת תכניות. מנגנון התבניות דורש הבנה מעמיקה בכדי לעשות בו שימוש נכון.
באופן כללי, המנגנון של תבניות מאפשר להגדיר תבנית לפונקציות ומחלקות אשר שונות אחת מהשנייה בטיפוס של התבנית. המהדר פורש עבור המתכנת את הקוד המתאים בהתאם לשימוש בתכנית.

נניח שברצוננו לכתוב פונקציה המחליפה ערכים של שני משתנים שלמים:

```
void swap (int & a, int & b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

וכזאת המחליפה ערכי משתנים מסוג float:

```
void swap (float & a, float & b)
{
    float temp = a;
    a = b;
    b = temp;
}
```

נקל להבחין כי למעט טיפוסי המשתנים (int ו- float) אין כל שוני בין הפונקציות הנ"ל.
הגדרת תבנית לפונקציה זו היתה חוסכת למתכנת כתיבה כפולה של הפונקציה.
אין המדובר רק ב- "copy and paste" , מאחר ובנוסף לכך שתיקון טעות גורר תיקון בכמה פונקציות זהות (מקור לשגיאות), תבניות מאפשרות שימוש חוזר בקוד כתוב עבור טיפוסים חדשים המוגדרים ע"י המשתמש.

באותו אופן, ניתן להדגים בניית מחלקות על סמך תבנית. חשבו על מימוש של מחלקה המיצגת רשימה מקושרת של מבנים מסוג Point. מבנה יחיד ברשימה הוא מהצורה:

```
struct Node
{
    Point *val ;
    Node *next ;
};
```

באותו אופן, רשימה מקושרת של מבנים מסוג String תשתמש במבנה מהצורה:

```
struct Node
{
    String *val ;
    Node *next ;
};
```

היינו רוצים להגדיר רשימות מקושרות של איברים מטיפוסים שונים מבלי לכתוב את הקוד של מחלקה כזו מחדש בכל פעם.

תבניות של פונקציות

template של פונקציה מאפשר להגדיר עבור הפונקציה
- פרמטרים.
- משתנים מקומיים.
- ערך מוחזר

כבעלי טיפוס כללי.

המילה השמורה template מורה למהדר להתייחס להגדרה הבאה אחריו כאל תבנית ולא לפרוש קוד עבורה, אלא אם נדרש לעשות כן.

נדגים את השימוש בתבניות, ע"י מימוש של הפונקציה bubble_sort הממיינת איברים מטיפוס כללי. תחילה נבנה תבנית לפונקציה המחליפה שני איברים:

```
template <class T>
void swap ( T  & a, T & b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

swap() מקבלת שני פרמטרים מטיפוס T – טיפוס כללי המוכרז כ- template :

```
template <class T>
```

הכרזה זו מציינת שבהגדרת הפונקציה העוקבת, T ישמש כשם טיפוס כללי.
כותרת הפונקציה כוללת פרמטרים מטיפוס ייחוס ל- T:

```
void swap ( T  & a, T & b)
```

הטיפוס המדויק של a ו- b ייקבע בשימוש בפונקציה.
בגוף הפונקציה מבצעים את ההחלפה ע"י שימוש במשתנה עזר מקומי מסוג T.

בקוד המשתמש בפונקציה נקבע כיצד יפרוש המהדר את הקוד של התבנית, ע"ס הטיפוסים המועברים לתבנית בעת השימוש.
לדוגמא:

```
int x=12, y=8;
swap (x, y);      // swap two integers
```

גורמת לכך שהמהדר יפרוש את הקוד:

```
void swap (int & a, int & b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

ומיד יבצע קריאה אליה, תוך כדי ביצוע בדיקות הידור.

כעת נכתוב את bubble_sort כ- template של פונקציה המבצעת מיון בועות:

```
template <class T>
void bubble_sort (T * start, T * end)
{
    int size = end – start;       // number of elements
    for (int i=0 ; i<size ; i++)
        for (int j=0; j<size-i-1 ; j++)
            if (start[j] > start[j+1])
                swap (start[j] , start[j+1]);
}
```

שימוש לדוגמא:

```
int arr1[7] = { 3, -4, 1, 17, 15, 12, -20 };
bubble_sort (arr1, arr1+5) ;          // sort only first 5 elements --> { -4, 1, 3, 15 ,17, 12, -20 }
```

בדוגמא האחרונה המיון היה על חלק מהמערך כולו.
באותו אופן ניתן למיין מערך של String:

```
String arr2[4];              // assume default constructor exist
arr2[0] = “what”;            // assume operator = is defined to take ‘const char * ’
arr2[1] = “a”;   arr2[2] = “beutiful”;    arr2[3] = “day”;
bubble_sort (arr2, arr2+3) ;
```

ההבדל בדוגמא זו לקודמת, הוא בכך שהפעלת האופרטורים (">", "=") על טיפוס מסוג מחלקה, מפעילה את האופרטורים שהמחלקה הגדירה.
אם אין אופרטור ">" המוגדר על המחלקה String, המהדר יספק הודעת שגיאה בעת הקריאה לפונקציה bubble_sort האומרת שלא מוגדר אופטור מתאים.

<u>תבניות של מחלקה</u>

template של מחלקה מאפשר להתייחס לטיפוס מסוים במחלקה באופן כללי. בכך מנגנון זה מאפשר לכתוב תבנית של מחלקה שתתאים למספר רב של טיפוסים.

דוגמא שכיחה היא המחלקה Vector המייצגת מערך של טיפוסים כלשהם. בעזרת מנגנון ה- template ניתן להגדיר מחלקות בעלות טיפוסי איברים שונים.

```
template <class T>
class Vector
{
    T *m_data;
    int m_size;
    void reset() { m_size=0; m_data=0; }
    void allocate(int size) { m_size=size; m_data=new T[m_size]; }
    void copy(const T v[] );
public:
    explicit Vector(int size=0, T initVal=T());
    Vector(const Vector& v);                        // copy constructor
    ~Vector() { delete [] m_data; reset(); }
    int size() const { return m_size;}
    void set(T val);
    Vector &operator=(const Vector &v);
    T& operator[](int i)  { return m_data[i]; }
    const T& operator[](int i) const  {return m_data[i];}
};
```

ומימוש הפונקציות של תבנית המחלקה, מחוץ להגדרתה:

```
template <class T>
Vector<T> & Vector<T> :: operator=(const Vector<T> &v)
{
    if(this!=&v)       // protect against self assignment
    {
        delete [] m_data;
        allocate(v.m_size);
        copy(v.m_data);
    }
    return *this;
}

template <class T>
void Vector<T> :: copy(const T v[] )
{
    for(int i=0; i<size(); i++)
        m_data[i]=v[i];
}

template <class T>
Vector<T> :: Vector(int size, T initVal)
{
    if(size!=0)
    {
        allocate(size); set(initVal);
    }
    else reset();
}

template <class T>
Vector<T> :: Vector(const Vector& v)       // copy constructor
{
    if(v.size()!=0)
    {
        allocate(v.size()); copy(v.m_data);
    }
    else
        reset();
}

template <class T>
void Vector<T> :: set(T val)
{
    for(int i=0; i<size(); i++)
        m_data[i] = val;
}
```

ושימוש אפשרי של מחלקה זו:

```
int main () {
    Vector <int > vi (8);        // Vector of integers, with initial size of 8 integers
    Vector <double > vd;       // Vector of integers, with initial size of 8 integers
    vi [3] = 20;                 // use operator []
    vd.set(3.6);                 // use 'Vector<double>::set(double)'
    return 0;
}
```

דברים שכדאי לדעת על תבניות

1. כאשר המהדר מבצע הידור של מחלקה חדשה הנוצרת מתבנית, הוא אינו פורש מיידית את כל הפונקציות, אלא רק את אלו שהמשתמש קורא להן.
לכן, ייתכן שפרמטר template מסויים לא מקיים את כל האילוצים המוגדרים בכל הפונקציות ובכל זאת לא תתקבל שגיאת הידור, עד לקריאה לפונקציה כזו.

2. ניתן להגדיר אובייקט template מקונן, כלומר, כזה שהפרמטר שלו הוא template בעצמו, ובכך ליצור מבנים מורכבים:
לדוגמא, בכדי להגדיר מטריצה של שלמים ניתן לכתוב:

```
Vector <Vector<int> > matrix(9)
```

matrix הינו אובייקט הכולל וקטור של 9 וקטורי שלמים.

3. אחת התופעות הנובעות משימוש בתבניות היא "ניפוח קוד", כלומר, הגדלת נפח הקוד של התכנית. הדבר נובע מכך שהפעלה חוזרת של התבנית על כל טיפוס שונה יוצרת ורסיה שונה של פונקציות מתוך התבנית.

4. כאשר נקראת פונקצית תבנית, המהדר יוצר אותה ומיד לאחר מכן מנסה להדר אותה.
למשל, עבור הפונקציה ()swap שהגדרנו קודם, אם יתבצעו הקריאות הבאות:

```
int in1=8, in2=9;
swap (in1, in2);

long lo1=11, lo2=14;
swap (lo1, lo2);
```

יווצרו שתי ורסיות לפונקציה: אחת עבור int ואחת עבור long.
נסיון להפעיל את הפונקציה על שני טיפוסים שונים תגרור הודעת שגיאה:

```
swap (lo1, in2);          // Error !
```

המהדר לא יודע לאיזו פונקציה מהקיימות לבצע קריאה. שתיהן עדיפות במידה שווה.

5. ניתן להגדיר פונקציות חריגות שלא כתבנית (כ- template), למרות שקיימת עבורן תבנית.
הסיבה לצורך הזה היא שלעיתים התבנית לא מבצעת את עבודתה "כנדרש" על טיפוסים מסויימים. למשל – פונקציה בשם compare יכולה להשוות טיפוסים מסוג: int, float וכו' אך השוואה של *char מתייחסת לרב למחרוזות והכוונה היא להשוואה שלהן ולא של המצביע אליהן. כדי לפתור דילמה זו, מגדירים את (*compare(char*, char והמהדר ייחפש אותה קודם לפני שהוא מרכיב תבנית עבורה.

6. ניתן להגדיר יותר מאשר טיפוס אחד לתבנית באופן הבא:

```
template <class T, class S>
class XXX
{
    T   first_type;
    S   second_type;
    ...
};
```

7. ניתן להגדיר ערכי ברירת מחדל לתבנית באופן הבא:

```
template <class T, int SIZE=10>
class Vector
{
    T m_data[SIZE];
    ...
};
```

וכן להגדיר את טיפוס התבנית כברירת מחדל:

```
template <class T=int>
class Vector
{
    T *m_data;
    ...
};
```

ולהשתמש באופן הבא:

```
Vector<int> v1 ;          // Vector of integers
Vector<> v2 ;             // Vector of integers – now using default type
```

שיעור מספר 9

STL (Standard Template Library)

נושאי השיעור:

- namespaces .
- הארכיטקטורה של STL.
- שימוש במכולות.
- איטרטורים.
- אלגוריתמים.

namespaces

namespace (מרחב שמות) הינה מילה שמורה והיא באה לפני הגדרה של תחום (scope) בדומה להגדרת מחלקה:

```
namespace MyNameSpace {
    // all sort of definitions – functions, varaibles, classes, etc...
};
```

בתוך מרחב שמות ניתן להגדיר טיפוסים, מחלקות, מבנים, פונקציות וכמעט כל דבר העולה על הדעת. השימוש למרחב שמות הוא בעיקר כדי להכליל הגדרות בתוך תחת שם אחד, שהוא שמו של מרחב השמות (MyNameSpace בדוגמא לעיל).

כדי להשתמש בהגדרה ממרחב שמות מסוים יש לציין פנייה מפורשת אליה עם שם מרחב בשמות ואופרטור התחום:

```
namespace NS1        // namespace with name NS1
{
    class X {  // ...   };

    template <class T>
    void sort(T * start, T* end, int size) { ... };

    ostream MyCout;
};
```

והשימוש בבגדרות שבתחום שמות הנ"ל:

```
int main()
{
    NS1::MyCout << "using ostream object declared inside NS1 namespace\n";
    NS1::X  x_obj[10];
    NS1::sort (x_obj, x_obj+10, sizeof(NS!::X) );
    Return 0;
};
```

מאחר ויש אי נוחות בחזרה על התבנית ***<namespace name>::*** בכל פעם שפונים להגדרה ממרחב הכתובות, ניתן להשתמש במילה השמורה **using** כשלאחריה שם ההגדרה המלא ממרחב השמות, או מרחב השמות כולו, כדי להצהיר (בכל שלב של הקוד) על שימוש מקוצר בהגדרה מסוימת. השימוש תקף החל ממקום הגדרתו עד לסוף הקובץ, או עד שהוצהר על שימוש במרחב שמות אחר או בשם דומה ממרחב שמות אחר.

דוגמאות:

```
namespace NS1           // namespace with name NS1
{
    class X { // ... };
    int f( );
};

namespace NS2           // namespace with name NS1
{
    class X { // ... };
    int f( );
};

using NS2::f();

int main()
{
    f();            // call NS2::f();
    NS1::f();       // call NS1::f();

    using namespace NS1;         // from now on calls will refer namespace NS1

    f();            // call NS1::f();
    X x;            // declare NS1::X object

    Return 0;
}
```

<u>הארכיטקטורה של STL</u>

STL (Standard Template Library) היא ספרייה המכילה הגדרות כלליות של
- תבניות של אלגוריתמים כלליים כמו מיון, חיפוש, מיזוג, החלפה, הזזה וכו'.
- תבניות של מיכלים (containers) אשר מהווים בעצם מבני נתונים שונים לאחסון מידע. ביניהם ניתן למצוא: vector, list, map, set, queue, deque, stack ועוד.
- תבניות של איטרטורים, אשר הינם הפשטה של מצביעים לאיברים במיכלים

בתכנון STL נבחרה הגישה בה מחלקות במיכלים אינן יורשות מבסיס משותף (קיום פונקציות וירטואליות מונע לרוב שימוש במנגנון inline ולכן לא מאפשר יעילות מכסימלית), והאלגוריתמים מוגדרים כ- templates של פונקציות גלובליות.
בצורה זו מושגת יעילות מכסימלית בקוד של התבניות.

כל מרכיבי STL מוגדרים תחת מרחב השמות std. כל שמות המרכיבים בספרייה נכתבים באותיות קטנות (ללא אות רישית גדולה). דוגמאות לשמות: std::vector, std::list<int> וכו'.

השימוש בתבניות של STL מקל על עבודת התכנות, מאחר והמתכנת מתמקד בפחות עבודה. העובדה שמבני הנתונים מוכנים לשימוש והאלגוריתמים השימושיים ביותר קיימים ונבדקו בקפידה לאיתור שגיאות, מונע שגיאות רבות מצד המתכנת ומאפשר לו התמקדות בקוד המשתמש בהם בלבד (לא שזה מעט עבודה...).

שימוש במכולות

נדגים את השימוש במכולות ע"י המכולה list. תחילה נדגים שימוש כללי ללא איטרטורים.
בהמשך נבין מהם איטרטורים וכיצד להשתמש בהם על מכולות.
באופן כללי, הגדרת מכולה מתבצעת בצורה שנלמדה לגבי תבניות של מחלקות:

```
list <int> li;                 // define object of type list that holds list of integers
vector <Rational * > vec_rat;  // define object of type vector that holds array of pointers to Rational class
```

list
הפונקציות העיקריות של המחלקה list :

size() – מחזירה את מספר האיברים ברשימה.
empty() – בודקת האם הרשימה ריקה ומחזירה ערך בוליאני.
pop_front(), pop_back() – מוציאה את האיבר הראשון/אחרון ברשימה, אך לא מחזירה אותו.
front(), back() – מחזירה את האיבר הראשון/אחרון ברשימה, אך לא מוציאה אותו מהרשימה.
push_front(), push_back() – מכניסה איבר לתחילת/סוף הרשימה.
clear() – מוחקת את כל תוכן הרשימה.
sort() – ממיינת את הרשימה בסדר עולה. אובייקטים נדרשים להגדיר אופרטורי השוואה ביניהם.

דוגמא:

```
#include <list>           // note!! No ".h" in list file !!
#include <iostream>       // note!! No ".h" in iostream file !!

using namespace std;      // from now on no need of "std::" prefix

int main()
{
    list <int> li;         // define list of integers
    li.push_front(4);      // insert integer to front of list ( |->[4]->END )
    li.push_front(5);      // insert integer to front of list ( |->[5]->[4]->END )
    li.push_back(9);       // insert integer to back of list  ( |->[5]->[4]->[9]->END )
    li.sort();             // sort the list ( |->[4]->[5]->[9]->END )

    cout << "front of list = "  << li.front()              // get integer that is in front of the list (4)
         << ", back of list = "  << li.back()              // get integer that is in last in the list (9)
         << ", size of list = "   << li.size()  << endl;   // get list's number of elements (3)

    li.push_back(li.front());    // ( |->[4]->[5]->[9]->[4]->END )

    cout << "list has " << li.size() << " members\n";
    li.clear();                  // empty list
    cout << "after clear() list has " << li.size() << " members\n";

    return 0;
}
```

והפלט של התכנית:

```
front of list = 4, back of list = 9, size of list = 3
list has 4 members
after clear() list has 0 members
```

איטרטורים

איטרטורים משמשים כמעיין מצביעים לאיברים בתוך מכולות.
מאחר ואין באפשרות המתכנת לקבל מצביע אמיתי לתך מבנה הנתונים (data hiding) ומאחר וגם אם היה באפשרותו, עדיין מבנה הנתונים הפנימי לא ידוע לו (שמות השדות, צורת המימוש וכו'), נדרשת צורה שונה לאפשר למשתמש במכולה לעבור על אבריה. צורה זו ממומשת ע"י הגדרת מחלקה מקוננת בתוך כל מכולה הקרויה iterator. מחלקה זו הינה חברה (friend) של המכולה ולכן יכולה לגשת למבנה הנתונים שלה. המחלקה iterator מגדירה בתוכה אופרטורים להשוואה (==, =!, ...) , אופרטורים של קידום (++, - -) וכן אופרטורים של גישה (<-, *).

הרבה מהפונקציות של המכולה משתמשות באיטרטורים, או מחזירות איטרטורים:
למשל:

begin() – מחזירה איטרטור לתחילת מבנה הנתונים.
end() – מחזירה איטרטור לאיבר שאחרי האיבר האחרון במבנה הנתונים.
insert() – מכניסה איבר חדש למיקום שהאיטרטור שמועבר אליה מצביע.
erase() – מוחקת נתונים מהמכולה מאיטרטור עד איטרטור.

דוגמא:

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list < int > li;
    li.push_front(4);         // insert integer to front of list ( |->[4]->END )
    li.push_front(5);         // insert integer to front of list ( |->[5]->[4]->END )
    li.push_back(9);          // insert integer to back of list  ( |->[5]->[4]->[9]->END )

    list < int >::iterator it = li.begin();
    list < int >::iterator it_end = li.end();

    while( it != it_end)      // use 'list<int>::iterator::operator !=()'
    {
    int temp = *it;       // use 'list<int>::iterator::operator *()'
    if(temp >=5)
    {
        li.insert(it, ++temp);   // insert one more member to list and leave loop
        break;
    }
    it++;                 // use 'list<int>::iterator::operator ++()'
    }
    li.erase( li.begin(), li.end() );      // same as calling clear()
    return 0;
}
```

אלגוריתמים

האלגוריתמים ב- STL ממומשים ע"י פונקציות template הפועלות על סדרת איברים כללית. ה"דבק" המחבר בין המיכלים לאלגוריתמים הוא האיטרטורים: האלגוריתם מקבל כפרמטר template את טיפוס האיטרטור המתאים ו/או את טיפוס האיבר, ובהתאם הוא פועל עליו.

לדוגמא, אלגוריתם החיפוש ממומש ע"י הפונקציה ()find המוגדרת כך:

```
template <class InIter, class T>
InIter find (InIter first, InIter last, const T& val);
```

וניתן להפעילו לדוגמא כך:

```
vector <int> v(100);
for( int i=0; i<100; i++ )
    v[i] = i*i ;

vector <int> :: iterator it ;
it = find (v.begin() , v.end(), 256);
if ( it != v.end() )            // found !
{
    int result = *it ;
}
```

כדי להשתמש באלגוריתמים יש להכליל את הקובץ <algorithm> בתחילת התכנית.
אלגוריתמים נוספים ב- STL :

replace (begin, end, what, to) - הפונקציה מקבלת איטרטור להתחלת רצף איברים (begin), איטרטור לסוף רצף איברים (end) איבר לחפש ולהחליף (what) באיבר אחר (to).
הפונקציה מחליפה את כל האיברים ברצף הדומים ל- what ב- to.

sort (begin, end) - הפונקציה מקבלת איטרטור להתחלת רצף איברים (begin) ואיטרטור לסוף רצף איברים (end) והיא ממינת את האיברים בטווח הנ"ל. גרסה נוספת של sort קיימת, המקבלת פרמטר נוסף שהוא פרדיקט (אובייקט המשמש להשוואה בין איברים) ובעזרתו משווה איברים.

שיעור מספר 10

ירושה (inheritance)

נושאי השיעור:
- מבוא.
- ירושה בסיסית ומרובה.
- אתחול מחלקות.

ירושה - מבוא

ירושה הינה יחס בין מחלקות: מחלקה יכולה לרשת ממחלקה אחרת את מכלול הטיפוסים, הקבועים, המשתנים והפונקציות המוגדרות בה.

המחלקה היורשת יכולה להוסיף הגדרות משלה, וכן לשנות את משמעות הפונקציות שירשה.
קיימים מונחים נרדפים לציון ירושה:
- המחלקה המורישה נקראת **מחלקת יסוד** או **מחלקת בסיס (Base)**.
- המחלקה היורשת נקראת **מחלקה נגזרת (Derived)**.

ירושה מאפשרת :
- הרחבת מחלקה קיימת.
- הגדרת מכנה משותף למספר מחלקות.
- פולימורפיזם (רב-צורתיות).

שתי האפשרויות הראשונות מונעות שכפול קוד, ולכן הופכות את התכנית לקלה יותר לתחזוקה.
המנגנון השלישי- פולימורפיזם- מהווה את הבסיס לתכנות מונחה עצמים ובו ניגע בהמשך.

תחביר ירושה בסיסית:

```
class BaseClass { ... };

class DerivedClass : public BaseClass { ... };
```

שם המחלקה היורשת מופיעה מצד שמאל כבהגדרת מחלקה רגילה ולאחריו יש נקודותיים (":") כשמימינם מופיעה שם המחלקה ממנה יורשים כשלפניה צורת הירושה.

לעת עתה אנו נתמקד בצורת ירושה שהיא public ואותה נכיר בהמשך יותר. נראה גם שאפשריות צורות ירושה שונות (protected, private), אך הן פחות "פופולריות".

ירושה בסיסית

מחלקה מסוימת יכולה לרשת ממחלקה אחרת ובכך לקבל את מכלול ההגדרות של המחלקה המורישה. נתבונן בדוגמא הבאה, על מחלקה בשם Person המייצגת אדם, ועל מחלקה היורשת ממנה – Student המייצגת סטודנט (שהינו "סוג של אדם"):

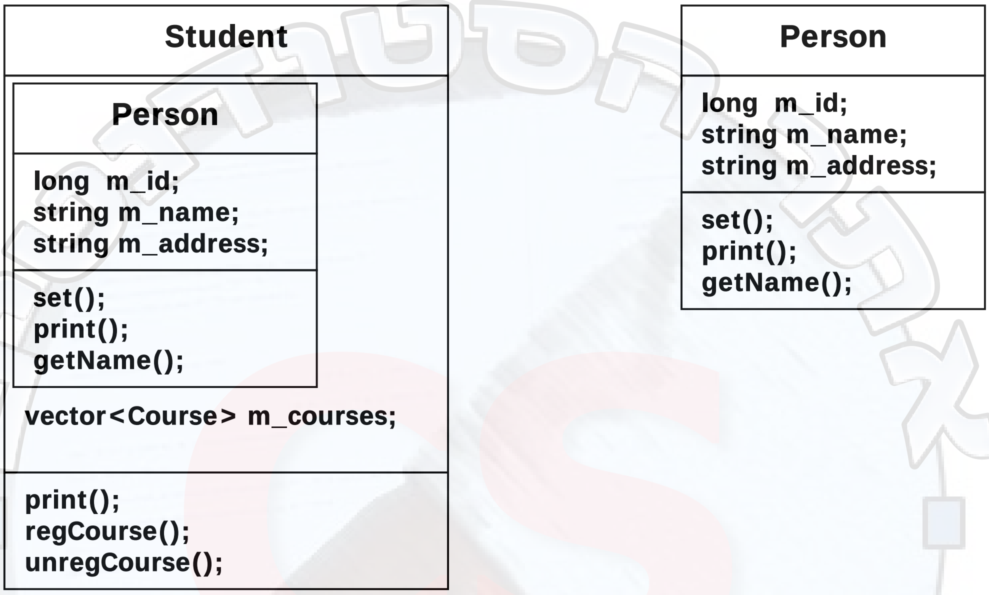
```
class Person
{
public:
    Person() {m_id=0;}
    Person(const long id, const char* name, const char *addr) :
        m_id(id), m_name(name), m_address(addr)  {}
    void set(const long id, const string &name, const string &addr)
    { m_id = id; m_name=name; m_address=addr; print();}
    string getName() const { return m_name; }
    void print() const
    {
        cout << "\n\nPerson: ";
        cout << m_id << "," << m_name << "," << m_address << endl;
    }
private:
    long  m_id;
    string m_name;
    string m_address;
};
```

המחלקה המייצגת סטודנט:

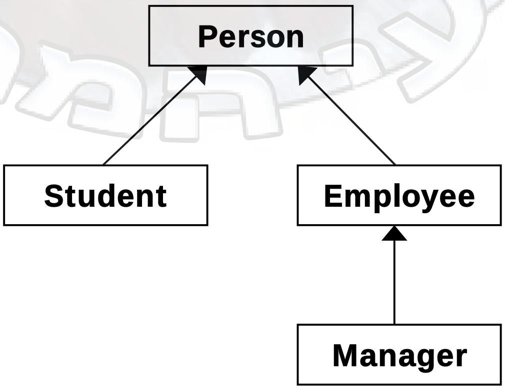
```
class Student : public Person
{
public:
    enum Course { MATH=1000, CS1, CS2, ALGORITHMS, DATAST, C, CPP, JAVA, OS, ASSEMBLY};
    Student() {}
    Student(const long id, const char* name, const char *addr) :
        Person(id,name,addr) {}        // initialize Person object with parameters
    void regCourse(Course c) { m_courses.push_back(c); }
    void unregCourse(Course c)         // search for certain course in vector and remove it
    {
        vector<Course>::iterator i = find(m_courses.begin(), m_courses.end(), c);
        if(i!=m_courses.end())
            m_courses.erase(i);
    }
    void print() const                 // print details: first Person details, and then Student details
    {
        Person::print();
        cout << "Student: " ;
        if(!m_courses.empty())
        {
            cout << "Courses = ";
            for(int i=0; i<m_courses.size(); i++)
                cout << m_courses[i] << ",";
        }
    }
private:
    vector<Course> m_courses;
};
```

המחלקה Student מוגדרת כיורשת מהמחלקה Person . לפיכך, Student כוללת את כל ההגדרות שב- Person.
המחלקה Student מגדירה פונקציות חדשות – regCourse ו- unregCourse מאחר והן ייעודיות לאובייקט מסוג Student. בנוסף המחלקה מגדירה פונקצית print אשר דורסת (overwrite) את ההגדרה של הפונקציה print של המחלקה Person.
שימו לב לכך שבפונקציה print של Student יש קריאה מפורשת לפונקציה print של Person. לא ניתן לקרוא לה בצורה אחרת מאחר ואז תווצר קריאה רקורסיבית.

מבחינה תרשימית, ניתן לתאר את המחלקות באופן הבא:



מבחינת יחסי ירושה נהוג לשרטט את היחס בין מחלקת יסוד למחלקה היורשת ממנה באופן הבא:



ירושה מרובה

מחלקה יכולה לרשת ממספר מחלקות אחרות ובכך לקבל את מכלול ההגדרות שלהן.
אין זהו חומר בסיסי בתכנות מונחה עצמים ולכן לא נרחיב בנושא זה, אך נציין את הרעיון הכללי בירושה מרובה ונביא את התחביר (syntax) שלו.
ירושה מרובה הינה ירושה של מחלקה אחת ממספר (2 ויותר) מחלקות.
לדוגמא, כאשר מייצגים כלים חשמליים במחלקה אחת וכלי דקורציה במחלקה אחרת, המידע הרלוונטי לכלים חשמליים דקורטיביים (כמו למשל שעון מעוצב) מצוי בשתי המחלקות גם יחד.
לכן ניתן להגדיר מחלקה לכלים חשמליים דקורטיביים היורשת משתי המחלקות הנ"ל בירושה מרובה.
התחביר של מחלקה X היורשת ירושה מרובה משתי מחלקות Y, Z הינו:

```
class X : public Y,  public Z
{
   // ...
};
```

אתחול בירושה

תזכורת:
Composition – מצב שבו אובייקט הינו חלק מאובייקט אחר. אובייקט שכזה נוצר כאשר האובייקט שמכיל אותו נוצר, הוא חי כל עוד האובייקט שמכיל אותו חי והוא נמחק (המפרק שלו מופעל) כאשר האובייקט שמכיל אותו נמחק.
במצב של ירושה, כאשר מוגדר אובייקט ממחלקה נגזרת, נוצר גם האובייקט ממחלקת הבסיס (בדיוק כמו composition).
סדר האתחול אם כן בעת יצירת אובייקט ממחלקה X היורשת ממחלקה Y הינו:
מופעל הבנאי של Y ורק לאחר מכן הבנאי של X.
סדר ההריסה של אובייקט כזה הינו:
מופעל המפרק של X ורק לאחר מכן המפרק של Y.

האתחול של אובייקט ממחלקת הבסיס נעשה ע"י קריאה לבנאי שלו משורת האתחול בבנאי של המחלקה הנגזרת.
דוגמא:

```
class Y {
    public:
        Y(int a){ ... }
};

class X : public Y
{
    public:
        X(int b, int c) : Y(b) { ... }
}
```

שיעור מספר 11

פולימורפיזם (polymorphism)

נושאי השיעור:

- פולימורפיזם – רב-צורתיות
- פונקציות וירטואליות (מדומות)
- Static Casting

פולימורפיזם - רב צורתיות - polymorphism

פולימורפיזם מאפשר לנו להתייחס לאובייקטים שונים התייחסות זהה שמובילה לתוצאות שונות.
ראשית, בואו נראה איזה בעיות מאפשר לנו הפולימורפיזם לפתור ואיזה יתרונות גלומים בו.

בחברה מסוימת יש 5 מנהלים, 5 מזכירים, 10 שליחים ו- 100 אנשי מכירות.
לצורך ניהול מאגר המידע שלנו, נבנה מחלקת אב בשם Employee הכוללת את כל הדברים המשותפים לכל העובדים, ולמחלקה זו יהיו 4 מחלקות בת יורשות: Manager, Secretary, Salesman, Messenger.
נניח שמחלקת האב נראית כך:

```
class Employee
{
      char *name;
      int age;
public:
      void show()
      {
            cout << name << ” “ << age;
      }
};
```

כלומר, לכל אחד מהעובדים יש את שתי התכונות הנ"ל ואת המתודה show() .
בכלים שיש לנו כרגע, כדי להפעיל על כל העובדים את המתודה show נצטרך ליצור מערך, ולעבור עליו בלולאה:

```
Manager * p_man [5];
Secretary * p_sec [5];
Salesman * p_sal [100];
Messenger * p_mes[10];

// initialize arrays
int i;
for(i=0; i<5; i++)
    p_man[i]->show();

for(i=0; i<5; i++)
    p_sec[i]->show();

for(i=0; i<100; i++)
    p_sal[i]->show();

for(i=0; i<10; i++)
    p_mes[i]->show();
```

צורת עבודה זו מסורבלת ובעייתית.
נרצה שיהיה לנו מערך יחיד המשותף לכל העובדים למרות שהם מטיפוסים שונים!
הדבר אפשרי מאחר ולכל העובדים יש משהו משותף: הם מחלקות נגזרות של Employee.

כלל הברזל קובע כי מצביע (ייחוס) של מחלקת יסוד יכול להצביע (להיות מיוחס) לאובייקט ממחלקה נגזרת (אך לא להיפך).
כלומר:

```
Employee *e= new Employee;
Manager *m = new Manager;
e=m;  //ok, Employee points to Manager
m=e;  //wrong!!! Compilation error!!!
```

הסיבה לכך היא של Manager יש את כל התכונות והיכולות של Employee ולכן Employee יכול להצביע ל Manager. ל Employee אין את כל התכונות והיכולות של Manager ולכן Manager לא יכול להצביע על Employee.
כאשר מצביע ממחלקת יסוד מצביע לאובייקט ממחלקה נגזרת הוא רואה אצל האובייקט המוצבע רק את החלק של התכונות והיכולות שלו עצמו.
מהאמור לעיל נובע כי ניתן לבצע הצבעה בצורה הבאה:

Employee *e1 = new Manager;

<u>מה קורה מבחינת התייחסות לאלמנטים (משתנים ומתודות) של העצם?</u>
יש לנו שלושה סוגים של אלמנטים אצל הבן. נראה מה יקרה כשנרצה להשתמש בהם:

1. אלמנט שהבן ירש מהאב ולא נגע בו, נשאר כמו שהוא (כמו מתודת show).
2. מתודה שהבן דרס (override), למשל מתודת ()calcSalary .
   זוהי מתודה שהבן ירש מהאב, אבל אצל האב היא עושה משהו אחד ואצל הבן משהו אחר.
   איזו מתודה תופעל כשנכתוב: ()e->calcSalary ? של הבן או של האב?
   כדי לענות על שאלה זו יש להבין את נושא הפונקציות הוירטואליות. ניגע בנושא זה בהמשך.
3. מתודה חדשה אצל הבן, שרק לבן יש אותה, למשל המתודה ()setBonus. זוהי מתודה שלא קיימת אצל האב בכלל, והיא נוצרה לראשונה אצל הבן.
   האם אפשר לעשות: (e->setBonus(1000 ?
   תשובה: לא. כי e הוא מצביע מסוג Employee וככזה הוא רואה רק את החלק שלו בבן.
   אז מה עושים אם בכל זאת רוצים ש-e יהיה מסוגל לראות לרגע את המתודות של Manager?
   יש דרך לבצע casting, אותה נלמד בהמשך.

<u>סיכום ביניים:</u>
עפ"י עקרון הפולימורפיזם, מצביע של מחלקת אב יכול להחזיק אובייקט מסוג האב או לחילופין אובייקט מסוג מחלקה יורשת כלשהי. כאשר צריך להחזיק מאגר גדול של עצמים שונים, היורשים מאב משותף, בונים מערך של מצביעים מסוג האב, אשר יחזיקו אובייקטים מהמחלקות הרצויות. לאחר מכן נוכל לבצע פעולות על כל האובייקטים, בלי חשיבות לסוגם, בעזרת לולאה.
כלומר:
פולימורפיזם מאפשר לנו להתייחס לאובייקטים שונים התייחסות זהה שמובילה לתוצאות שונות.

נתחיל בפתרון הבעיה הראשונה – מתודה שהבן דרס. כדי לדעת כיצד לפתור זאת, נלמד את נושא הפונקציות הוירטואליות (מדומות).

פונקציות וירטואליות (מדומות)

כאמור, הבעייה הראשונה שלנו היא כאשר מחלקה נגזרת דורסת פונקציה מסוימת של מחלקת הבסיס. אנו רוצים שלמרות שהמצביע על אובייקט ממחלקה נגזרת הוא מטיפוס מחלקת הבסיס, הפונקציה שתופעל תהיה הפונקציה של המחלקה הנגזרת.
בזמן הידור אין באפשרותנו (או באפשרות המהדר) לדעת על איזה אובייקט נצביע בזמן ריצה ולכן כדי לגרום לתכנית לבדוק בזמן ריצה את הטיפוס המוצבע, יש להוסיף את המילה השמורה **virtual** לפני הגדרת הפונקציה במחלקת הבסיס.
פונקציות אשר מסומנות במילה virtual נקראות פונקציות וירטואליות (מדומות).
מעתה, בכל פעם שנפנה עם מצביע מממחלקת הבסיס אל אובייקט מסוג מחלקה נגזרת, יבדוק המחשב האם הפונקציה נדרסה אצל הבן. אם כן – תופעל הפונקציה של הבן ואם לא – תופעל הפונקציה המקורית של האב.
בכך למעשה פתרנו את הבעיה שהוצגה מקודם – כיצד להפעיל מתודות של האב שהבן דרס.

לכאורה, תמיד כדאי לציין פונקציות כוירטואליות, אך למעשה הדבר מכביד על המחשב, היות ועבור כל פונקציה וירטואלית המחשב יורד בעץ ההורשה ובודק האם היא נדרסה. תהליך זה גוזל זמן.
הערה: אם מתודה אינה וירטואלית, הרי שמתודה שדרסה אותה, לא תוכל להיות וירטואלית.

דוגמא:

```
Employee*  staffArr [5];
StaffArr[0]=new Employee ();
StaffArr[1]=new Manager ();
StaffArr[2]=new Secretary ();
StaffArr[3]=new Employee ();
StaffArr[4]=new Messenger ();

for (int i; i<5; i++)
{
  staffArr [i].show();         // will activate Employee::show()
}
for (int i; i<5; i++)
{
  staffArr [i].calcSalary();  // will activate each virtual function according to the object type
}
```

ניתן להפעיל על המערך הנ"ל את מתודת calcSalary שהיא נדרסה אצל כל אחד מהבנים ואצל כל אחד עושה משהו אחר. להבדיל מהמתודה show אשר תעשה לכל האובייקטים אותו הדבר, תדפיס את השם ואת הגיל, המתודה calcSalary תבצע עבור אובייקט מסוג מנהל את המתודה שלו, עבור מזכירה את המתודה שלה וכו'. כלומר, יתבצעו פעולות שונות עבור כל אחד מהאובייקטים.

Static Casting – המרה לשם הפעלת מתודות שקיימות רק אצל הבן

ראינו כיצד מצביע מסוג אב המחזיק אובייקט מסוג בן מסוגל להפעיל מתודות שהיו קיימות אצל האב בין אם הן נדרסו ע"י הבן ובין אם לאו.
אך מה קורה כאשר רוצים להפעיל מתודה שקיימת רק אצל הבן?
מצביע מסוג אב "חושב" שהוא מחזיק אובייקט מסוג אב ולכן אינו מכיר את המתודות הייחודיות לבן.
אך היות ואנו יודעים כי מדובר באובייקט מסוג בן, נוכל לאלץ את המחשב להפעיל מתודות אלה ע"י כך שנגרום למצביע מסוג האב להתנהג כאילו היה מצביע מסוג בן.
אנו מבקשים מהמחשב "לסמוך עלינו" ומורים למצביע אב להפוך לרגע למצביע מסוג הבן.
פעולה זו נקראת "המרה" (Casting) ונתקלנו בה בעבר כאשר המרנו סוגי משתנים פשוטים.
כיצד מתבצע הדבר?
לפני שם המצביע נוסיף סוגריים ובתוכם נציין את סוג המצביע אליו אנו ממירים אותו. את כל זה נתחום בסוגריים וקיבלנו מצביע מסוג אב שהומר למצביע מסוג בן, כך שהוא יכול להפעיל מתודות ייחודיות.

```
((subclass*)superClassPointer)->method();
```

ראינו שההמרה עובדת כאשר מצביע מסוג אב אכן מחזיק אובייקט מסוג בן. אך מה יקרה אם נבצע המרה שעה שלמעשה מצביע מסוג אב מחזיק אובייקט מסוג אב (או מסוג בן אחר – שאין לו את אותה מתודה)?
לכאורה אין טעות – אך ברור שהמצב לא תקין. תארו לכם שבתוכנה עם מנהל ועובדים – כל העובדים יקבלו רכב ולא רק המנהלים...

ברור אם כך שכאשר עובדים עם מערך של מצביעים מסוג אב אשר חלקם מחזיקים בנים מסוגים שונים, אנו זקוקים לכלי שיאפשר לנו לדעת האם ואיזה המרה יש לבצע.

ניתן לחשוב על מספר דרכים, אך הנה דרך פשוטה לעשות זאת:
נוסיף לאב תכונה (אשר תעבור בירושה לכל הבנים). תכונה זו תהיה משתנה פשוט מסוג int (או short) אשר יקבל ערך מספרי בבנאי. ערך זה ייצג את סוג העצם.
לדוגמא: עובד פשוט יקבל קוד 0, מנהל יקבל קוד 1, מזכירה 2 וכך הלאה.
כאשר נעבור בלולאה על מערך של מצביעים מסוג אב, נוכל לשאול כל אחד מהם, עפ"י הערך של תכונה זו, מאיזה מחלקה הוא נגזר באמת, ובהתאם לכך נבצע המרות ונדע להפעיל מתודות ייחודיות.
בכך בעצם פתרנו לחלוטין את הבעיה המקורית מתחילת השיעור – אנו יכולים להפעיל את כל המתודות של אובייקט בן המוחזק ע"י מצביע מסוג האב!

שיעור מספר 12

פולימורפיזם - המשך

נושאי השיעור:

- מחלקות אבסטרקטיות
- מתודות מדומות טהורות
- מפרקים מדומים

מחלקות אבסטרקטיות

יש מקרים שבהם מספר מחלקות יורשות ממחלקת אב אחת, אשר היא למעשה אבסטרקטית.
לדוגמא: **animal**. הרי ברור שבמציאות לא קיים יצור כזה – בע"ח. זהו רק כינוי אבסטרקטי, לא מוחשי, לכלל החיות.
דוגמא נוספת: **צורה** – משולש, מלבן, משושה – כולן צורות. לכולן אפשר לחשב היקף ושטח. אך מה ההיקף של "צורה"?

כדי להמחיש רעיון זה מבחינה תכנותית נגדיר מחלקה אבסטרקטית, אשר ממנה עצמה אי-אפשר ליצור אובייקטים, אך מחלקות אחרות יוכלו לרשת ממנה ולתת משמעות למתודות שהן יקבלו בירושה.
כמו-כן, בהחלט ניתן ליצור מצביעים מסוג המחלקה הזו, אך אי אפשר לתת להם אובייקט ממשי שעליו הם יצביעו, היות ובלתי אפשרי לקרוא לבנאי של מחלקה אבסטרקטית, למרות שהוא קיים.

לשם מה אנו צריכים להגדיר מחלקה אם אי-אפשר ליצור ממנה מופעים?
כדי לאפשר שימוש בפולימורפיזם; נוכל ליצור מערך של מצביעים מסוג המחלקה האבסטרקטית, אשר יחזיקו אובייקטים של בנים שונים, שיממשו, כל אחד בצורה שונה, את המתודות האבסטרקטיות, דבר אשר לא היה מתאפשר אילולא ירשו מאב משותף.
<u>לדוגמא</u>:
לכל צורה (משולש, מרובע, משושה וכד') ניתן לחשב היקף, אך לא ניתן לחשב היקף של "צורה". נבנה מחלקה אבסטרקטית של צורות ובה מתודה אבסטרקטית לחישוב היקף, שלה לא יהיה מימוש. כל מחלקה יורשת, שהיא צורה ספציפית, תדרוס את אותה מתודה ותיתן לה מימוש. לאחר מכן נבנה מערך של מצביעים מסוג "צורה" אשר יחזיקו צורות שונות, ונוכל לעבור בלולאה על המערך ולומר לכל הצורות לחשב את היקפן, למרות שלכל אחת יש נוסחה שונה.

דגשים

1. ניתן להגדיר מתודה כאבסטרקטית אך ורק בתוך מחלקה אבסטרקטית. מרגע שהגדרנו מתודה כמדומה טהורה, המחלקה כולה אבסטרקטית.
2. כאשר מגדירים מתודה כאבסטרקטית, אנו למעשה מאלצים את המחלקות היורשות לדרוס אותה. כלומר, אם נבנה מחלקת Animal אבסטרקטית, נרצה שבכל מחלקה שתירש ממנה תהיה מתודת move כי כל חיה מסוגלת לנוע, אך כל מחלקה תממש אותה בדרך משלה (דג בשחייה, ציפור בתעופה וכד').
3. למתודה אבסטרקטית אין בלוק פקודות היות ואין לה מימוש.
מכיוון שהיא אבסטרקטית, מחלקה שתירש אותה לא תוכל להשאיר אותה ריקה, ותיאלץ לממש אותה (לכל הפחות בעזרת בלוק פקודות ריק).
4. מתודות אבסטרקטיות נקראות גם "מתודות וירטואליות טהורות" (pure virtual functions) היות ולעומת מתודות וירטואליות רגילות, את אלה לא רק אפשר, אלא גם **צריך** לדרוס.

5. **<u>חשוב</u>!** אם יש במחלקה אפילו מתודה אבסטרקטית אחת, חובה להגדיר את המחלקה כולה כאבסטרקטית, ולא ניתן ליצור ממנה אובייקטים.

<u>מתודות מדומות טהורות</u>

כיצד מגדירים פונקציה וירטואלית טהורה?
1. מוסיפים את המילה virtual בתחילת הגדרת הפונקציה.
2. בסוף שורת הגדרת הפונקציה, לא בונים בלוק פקודות אלא רושמים **=0;**

יחד עם זאת, ניתן לספק מימוש לפונקציה וירטואלית טהורה, כדי שכל פונקציה שתירש ממנה, תיישם מימוש זה.

מחלקה שיורשת ממחלקה אבסטרקטית, חייבת לתת מימוש לכל המתודות המדומות הטהורות, ולא – גם היא עצמה תחשב למופשטת.

דוגמא: Aliens
<u>Alien</u> – מחלקת חיזרים אבסטרקטית הכוללת תכונות של צבע ומידת עוינות. כמו-כן מתודת הדפסה.
<u>Klingon</u> – לחיזרים אלו תכונה נוספת – עובי המצח.
<u>Vulcan</u> – לחיזרים אלו תכונה נוספת – אורך האוזניים.

```
class Alien {
public:
    virtual int hostility() = 0;       // pure virtual method
    virtual string color() = 0;          // pure virtual method
    virtual void print() {
        cout << "Hostitlity is: " << hostility() << endl;
        cout << "Color is: " << color() << endl;       }
};

class Klingon : public Alien {
    int forehead_thickness;
public:
    int hostility() { return 8; }
    string color() { return "Brown"; }
    void print() { Alien::print();   cout << " forehead thickness is: " << forehead() << endl; }
    int forehead() { return forehead_thickness; }
};

class Vulcan : public Alien{
    int ears_len;
public:
    int hostility() { return 3; }
    string color() { return "Black"; }
    void print() { Alien::print();   cout << " ears len is: " << ears() << endl; }
    int ears() { return ears_len; }
};
```

מפרקים מדומים

כאשר נשתמש בפולימורפיזם וניצור מצביע מסוג מחלקה אבסטרקטית שיחזיק אובייקט של מחלקת בן, הרי באיזשהו שלב נפסיק להשתמש במצביע והאובייקט יימחק מהזיכרון בעזרת delete.
ברור שכדי למחוק אובייקט מסוג בן, עלינו להשתמש ראשית במפרק שלו, ואז במפרק של האב. רק כך יתפנה הזיכרון לחלוטין.
בפועל מופעל למעשה רק המפרק של האב ואין כלל קריאה למפרק של הבן, אשר עשוי להכיל פקודות חשובות.
כדי לפתור בעייה זו **-** במחלקות אבסטרקטיות, על המפרק להיות וירטואלי!

עבור מחלקה X המפרק הוירטואלי מוגדר באופן הבא:

```
class X {
   // ...
   virtual ~X() { ... }
   // ...
};
```