



מגשימים – עקרונות מתקדמים

הדגמה מרכזית

שיעור 3 – המשך תכנות מונחה עצמים

תוכן עניינים

- 2 כללי
- 3 חלק 1 – חזרה (זריזה) על דברים משיעור קודם
- 4 חלק 2 – פונקציית אתחול ופונקציית ניקוי
- 7 חלק 3 – מימוש בנאים/מפרקים
- 10 חלק 4 – בנאי העתקה וסוגי העתקות
- 16 חלק 5 – שורת אתחול
- 17 חלק 6 – העמסת אופרטורים



When your hammer is C++,
everything begins to look like a thumb.

נושא ההדגמה: בניית מחלקות, דריסת אופרטורים/פונקציות

ראשי פרקים:

- בניית מחלקה בסיסית (חזרה על חומר של שיעור קודם)
- דריסת בנאי העתקה, מימוש אופרטור העתקה
- הסבר על העתקות באמצעות דוגמאות.

מטרות בשיעור:

השיעור משלב בתוכו מצגת תיאורטית, והדגמה מעשית. מטרתה של המצגת התיאורטית היא לתת הקדמה, סיפור מסגרת, וללוות את ההדגמה המעשית. החל **משקופית מספר 5** יש לפתוח פרויקט ב Visual Studio ולכתוב קוד שיתפתח לאורך השיעור.

ההדגמה מבוססת על **קוד שנכתב בשיעור הקודם**, בתיקיית "הדגמה" של שיעור קודם תוכלו למצוא את קבצי הבסיס שמהם אפשר להתחיל לכתוב את הקוד.

בתיקייה נמצא קוד בסיס של המחלקה Student, אנחנו נעבוד על המחלקות כדי להעביר את העקרונות של השיעור הקרוב.

חלק 1 – חזרה (זריזה) על דברים משיעור קודם

היה לנו המון תרגול בכתיבת מחלקות, אבל חשוב שנעשה חזרה זריזה לפני שממשיכים להתקדם בחומר, עברו על מקבץ השאלות הבא וודאו שהכל מובן.

מקבץ שאלות חזרה זריזות שמוודאות הכרה של הנושאים הקודמים:

מהי מחלקה? מה הקשר בין אובייקט למחלקה?

(מחלקה היא התבנית שעל פיה מיוצר האובייקט, אובייקט הוא המופע של המחלקה.
מחלקה היא ישות אבסטרקטית ואובייקט הוא ישות קונקרטית)

מהו כימוס? איך מתודות Get/Set עוזרות לנו ליישם את העיקרון הזה?

(כימוס הוא האפשרות לחשוף החוצה רק את מה שהמחלקה רוצה לחשוף. מתודות
get/set מאפשרות לחשוף גישה באמצעות מתודות שעליהן יש לנו שליטה [לדוגמא
בדיקת קלט].)

**מה יהיה מציין הגישה של שדות המחלקה? מה יהיה מציין הגישה של ממשק
המחלקה? ומה של פונקציות העזר?**

(שדות המחלקה יהיו פרטיים, ממשק המחלקה ציבורי, ופונקציות העזר פרטיות).

מה משמעות הוספת המילה const בסוף מתודה של מחלקה?

(הפונקציה לא יכולה לשנות את מצב האובייקט – את ערך השדות.)

מהו המצביע this?

(מצביע מהטיפוס של המחלקה, מועבר בצורה מרומזת ע"י הקומפיילר בכל קריאה כדי
שיהיה אפשר לדעת איזה אובייקט קרא למתודה ולגשת לשדות שלו.)

חלק 2 – פונקציית אתחול ופונקציית ניקוי

בשיעור קודם בנינו את מחלקת Student. השתמשנו במחלקה ע"י יצירת מופע שלה (אובייקט) ועריכת השדות שלו באופן יזום (המשתמש לא היה מחויב לאתחל את השדות) תוך שימוש בפונקציות ה set.

לדוגמא:

```
Student stud1;

// first student info
stud1.setFirstName("Shahar");
stud1.setLastName("Hasson");
stud1.setId(123456789);
stud1.setGrade(HISTORY_GRADE_IDX, 78);
stud1.setGrade(MATH_GRADE_IDX, 81);
stud1.setGrade(LITERATURE_GRADE_IDX, 90);
stud1.setGrade(ENGLISH_GRADE_IDX, 65);
```

כבר בשיעור הקודם אפשר היה לשים לב שזו לא דרך אידיאלית, ושכדאי לנו לבנות פונקציית אתחול אשר תאתחל את שדות האובייקט לפי הערכים שנשלחו (בדומה לתרגיל הבית והשיעור הראשון)

לפני שנכתוב את הפונקציה נציין שינוי קטן מהשיעור הקודם (נמצא כבר בקוד) – הטיפוס של שדה מערך הציונים שונה מהמערך המקורי (שאליו מוקצה זיכרון בצורה מקומית) למצביע (אליו נצטרך להקצות זיכרון באופן דינמי).

כלומר `unsigned int _grades[NUM_OF_GRADES]` הפך ל `unsigned int *_grades`

שאלה: לאור השינוי האם אפשר פשוט להכניס ערכים למערך? התשובה היא שלא, קודם צריך לבקש ממערכת ההפעלה זיכרון כדי שבו יהיה ניתן לאחסן את המידע.

בדר"כ נעדיף שהמשתמש לא יהיה חשוף לזה, כלומר שהוא יוכל פשוט לתת את הציונים ושאנחנו כבר נדאג לשים אותם. כלומר אנו רוצים למנוע מהמשתמש את הצורך בלהקצות זיכרון בעצמו (ואולי גם לשכוח לשחרר אותו)

לצורך כך נכתוב את פונקציית האתחול עם החתימה הבאה:

```
void init(const int id, const std::string firstName, const std::string lastName);
```

והמימוש של הפונקציה

```
void Student::init(const int id, const std::string firstName, const std::string lastName);
{
    this->_id = id;
    this->_firstName = firstName;
    this->_lastName = lastName;

    // allocates memory and assigns empty grades
    this->_grades = new unsigned int[NUM_OF_GRADES];
    for (int i = 0; i < NUM_OF_GRADES; i++)
    {
        this->_grades[i] = EMPTY_GRADE;
    }
}
```

שימו לב שלא ביצענו בדיקות קלט כגון בדיקה אם הציונים בין 0-100 או אם הת.ז היא 9 ספרות, אבל בתכניות שנכתוב בבית נקפיד על בדיקות מהסוג הזה.

תזכורת: פקודת new שקולה לפקודה malloc, והיא מבקשת ממערכת ההפעלה להקצות שטח מסוים בזיכרון. הפונקציה מחזירה מצביע לאזור בזיכרון שהוקצה.

בגלל שהייתה הקצאת זיכרון דינמית, אנו ניצור פונקציית ניקוי שמטרתה לשחרר את כל המשאבים הדינמיים של המחלקה. (על כל new שביצענו צריך להיות delete)

```
void clean();
```

החתימה של פונקציית הניקוי :

המימוש :

```
void Student::clean()
{
    delete[] _grades;
    _grades = nullptr;
}
```

נשים לב שאם הקצינו זיכרון באמצעות new[] (כמו שעושים בדר"כ עם מערכים) אנו משחררים באמצעות הפקודה delete[], הפקודה דואגת להחזיר את כל המשאבים למערכת ההפעלה בצורה תקינה ולהפוך את כל תאי המערך ללא זמינים.

בנוסף לאחר שחרור הזיכרון שעליו מצביע הpointer הצבנו **nullptr** למטרות אבטחת מידע (מי שרוצה מוזמן/ת לחקור על נושא hanging pointers, ולמה חשוב לשים nullptr/NULL במצביע לאחר ששחררנו זיכרון. תזכורת: nullptr דומה ל NULL שאנו מכירים משפת C רק שהוא typesafe ומתאים למצביעים. כשמשתמשים במצביעים כדאי להתרגל לעבוד איתו.

עכשיו נשנה את ה main שלנו כך שישתמש בפונקציית האתחול שכתבנו

```
// init first student info
stud1.init(123456789, "Shahar", "Hasson");
stud1.setGrade(HISTORY_GRADE_IDX, 78);
stud1.setGrade(MATH_GRADE_IDX, 81);
stud1.setGrade(LITERATURE_GRADE_IDX, 90);
stud1.setGrade(ENGLISH_GRADE_IDX, 65);

Student stud2;

// init second student info
stud2.init(111111111, "Beyonce", "");
stud2.setGrade(HISTORY_GRADE_IDX, 95);
stud2.setGrade(MATH_GRADE_IDX, 87);
stud2.setGrade(LITERATURE_GRADE_IDX, 90);
stud2.setGrade(ENGLISH_GRADE_IDX, 98);
```

כבר עכשיו הקוד נראה הרבה יותר מסודר, אבל האם זאת הדרך האידיאלית? מה קורה אם המשתמש לא קרא לפונקציית האתחול? מה יודפס אם נקרא ל `stud1.print()`?

זו הזדמנות להכיר את המושג unexpected behavior: במצב כמו זה אנחנו לא יודעים אם בטעות ניגש לחלק זיכרון שהוא Read Only (כאשר ננסה לעשות set לציונים של המערך) והתכנית תעוף. יכול להיות שיודפס "זבל", בכל מקרה אנחנו מבינים שהכי טוב אם המשתמש היה חייב לקרוא לפונקציה, כלומר לגרום לפונקציית האתחול להיקרא באופן אוטומטי.

מה לגבי פונקציית הניקוי? אם המשתמש שכח לקרוא לה מה הבעייתיות? במידה והיו הקצאות זיכרון דינמיות (שימוש ב new) תיהיה דליפת זיכרון, אם היו משאבים אחרים שהשתמשנו (קבצים, sockets וכו') אז תהליכים אחרים יכולים להיפגע מכך. לדוגמא אם לא ביצענו close על socket, ייקח זמן מה עד שניתן יהיה לפתוח socket שמשתמשת באותו port וכתובת ip.

זה מוביל לנושא הבא של **בנאים (constructors) ומפרקים (destructors)**. יש לחזור למצגת כדי לראות קצת רקע תיאורטי.

חלק 3 – מימוש בנאים/מפרקים

נשנה את הקוד ככה שבמקום פונקציית init ו- clean נממש בנאי ומפרק.

נשנה את החתימות של פונקציות האתחול והניקוי כך שיהפכו להיות הבנאי והמפרק של המחלקה (גם בקובץ ה- cpp).

שאלת מחשבה, האם בנאי ומפרק צריכים להיות public או private? אם ה- C'tor וה- D'tor יהיו פרטיים לא יהיה ניתן ליצור אובייקטים מחוץ למחלקה. ישנה תבנית עיצוב בשם singleton שמשתמשת בקונספט הזה, אבל אנו לא נלמד עליה.

חתימת הבנאי

```
Student(const int id, const std::string firstName, const std::string lastName);
```

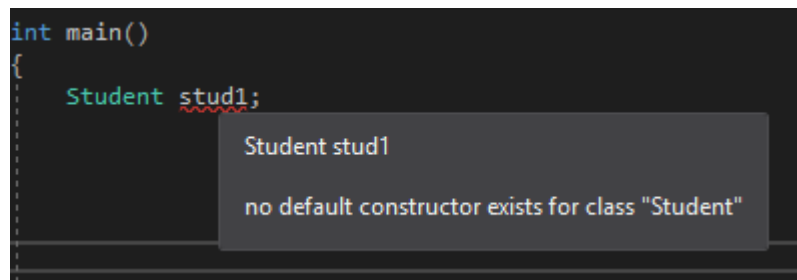
```
~Student();
```

חתימת המפרק

ניגש לקובץ cpp ונשנה את החתימה של הפונקציות init ו- clean לחתימות של הבנאי והמפרק.

נשים לב שהשם של הפונקציה זהו שם המחלקה, ושהמפרק מסומן כשם המחלקה עם התו '~' לפני.

כעת נחזור ל- main שלנו, נמחק את הקריאות לפונקציות האתחול והניקוי, ונשים לב לשגיאה שנוצרה.



```
int main()
{
    Student stud1;
```

Student stud1

no default constructor exists for class "Student"

הסיבה לשגיאה היא שכאשר מימשנו בנאי מסוים, הכתבנו את הדרך שבה מאתחלים את האובייקט. עד עכשיו היה ממומש בנאי ברירת מחדל (שלא מקבל פרמטרים וגם לא עושה כלום), ומהרגע שמימשנו את הבנאי שלנו (עם חתימה שונה), דרסנו את הבנאי המקורי. בהמשך המצגת נסביר על הנושא בצורה יותר מפורטת.

כרגע אפשר להסתפק בדבר הבא: מהרגע שהגדרנו בנאי משלנו נדרס הבנאי הדיפולטיבי, וניתן לאתחל את האובייקט רק באמצעות הבנאי שהגדרנו.

בתוך קובץ main.cpp נמחק את פונקציית ה- main שכתבנו, ונכתוב שלוש פונקציות חדשות במטרה לחדד את ההבנה של בנאים ומפרקים.

```
void f1()
{
    Student s(111222333, "Ash", "Ketchum");
}

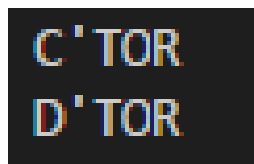
void f2()
{
    Student* s = new Student(333222111, "Justin", "Bieber");
    delete s;
}

void f3()
{
    Student* s = new Student(98754321, "Uri", "Kaduri");
}
```

נוסיף לבנאי המחלקה שורת הדפסה: `cout << "C'TOR" << endl;`
ובצורה דומה נוסיף למפרק `cout << "D'TOR" << endl;`

המטרה היא להבין מתי נקרא הבנאי, ומתי נקרא המפרק.

נקרא לפונקציה הראשונה מתוך ה- main.



התכנית תוציא את הפלט הבא:

הסיבה היא שבעת יצירת האובייקט נקרא בנאי המחלקה, ובעת היציאה מהפונקציה (סגירת סוגריים מסולסלים) נקראים כל המפרקים של האובייקטים שהוגדרו מקומית (לא דינמית באמצעות new) בפונקציה.

סגירת ה scope
של הפונקציה

```
void f1()
{
    Student s(111222333, "Ash", "Ketchum");
}
```

נקרא לפונקציה השנייה מתוך ה – main.

C'TOR
D'TOR

התכנית תוציא פלט זהה לפונקציה הראשונה:

שימו לב שבעת ביצוע פקודת מחיקה (delete) נקרא גם המפרק של האובייקט עליו מצביע s.

```
void f2()
{
    Student *s = new Student(333222111, "Justin", "Bieber");
    delete s;
}
```

כלומר קריאה ל delete היא דרך לעורר את מפרק המחלקה.

נקרא לפונקציה השלישית, והפעם נשאל מה לא בסדר בקוד?
תשובה: יש דליפת זיכרון.

פלט התכנית C'TOR

אין קריאה למפרק של המחלקה, רק לבנאי.

נחזור למצגת ונעבור לנושא הבא: העמסת פונקציות

חלק 4 – בנאי העתקה וסוגי העתקות

נתחיל בלהדגים מה עושה בנאי העתקה אשר ניתן לנו "בחינם", כלומר בנאי העתקה הדיפולטיבי.

נשתמש ב default copy constructor וננסה להעתיק אובייקט של סטודנט אחד, לתוך סטודנט של אובייקט אחר.

בנאי ההעתקה המתקבל כברירת מחדל מבצע **העתקה רדודה** (shallow copy). העתקה רדודה היא העתקת ביט אחרי ביט מכל שדה במחלקה (כמו memcpy). עבור משתנים פרימיטיביים כגון int אין שום בעיה להסתמך על העתקה רדודה, מכיוון שהעתקה זו תעתיק את הערך הנכון משדה אחד לשדה אחר. כאשר מדברים על מבנים דינמיים או מצביעים, העתקה רדודה כבר לא מספיקה. בהעתקה רדודה מועתקת **הכתובת** עליה מצביע הפוינטר, ולכן גם האובייקט שמועתק יצביע על אותו מקום בזיכרון.

כלומר נוכל לראות כיצד שינוי בציונים של סטודנט אחד, משנים את הציונים של הסטודנט השני. הסיבה היא שהמערך grades מצביע על אותו מקום בזיכרון.

נכתוב תכנית קצרה שמראה את הבעיה:

סטודנט A נוצר עם ערכים ראשוניים.
סטודנט B נוצר בעזרת בנאי העתקה (מועתק מ A)
משנים את הציון של A בהיסטוריה ל 60
משנים את תעודת הזהות של B למספר ת.ז אחר

באמצעות ההדפסות נוכל לראות ששינוי בציון של אחד משפיע על השני, אבל השדות הפרימיטיביים בלתי תלויים.

משמאל – קוד התכנית, בעמוד מימין הפלט, ונקודות שכדאי להדגיש. חשוב להדגיש את העובדה שהציון של B השתנה למרות שלא רצינו, אולם תעודת הזהות השתנתה כמו שציפינו.

```
int main()
{
    Student A(123456789, "Blinky", "Bill");
    cout << "Student A info" << endl;
    A.print();
    cout << "\n" << endl;

    // creates Student object B by copying A
    Student B(A);
    cout << "Student B info" << endl;
    B.print();
    cout << "\n" << endl;

    A.setGrade(HISTORY_GRADE_IDX, 60);
    B.setId(987654321);
    cout << "changed Student A History grade to 60,
            and B id to 987654321" << endl;

    cout << "Student A info" << endl;
    A.print();
    cout << "\n" << endl;

    cout << "Student B info" << endl;
    B.print();
    cout << "\n" << endl;

    system("pause");
    return 0;
}
```

שימוש בבנאי
העתקה

```
Student A info
Student id: 123456789
Name: Blinky Bill
*** Grades ***
History: Not Graded
Math: Not Graded
Literature: Not Graded
English: Not Graded
```

```
Student B info
Student id: 123456789
Name: Blinky Bill
*** Grades ***
History: Not Graded
Math: Not Graded
Literature: Not Graded
English: Not Graded
```

changed Student A History grade to 60, B id to 987654321

```
Student A info
Student id: 123456789
Name: Blinky Bill
*** Grades ***
History: 60
Math: Not Graded
Literature: Not Graded
English: Not Graded
```

ת.ז של A לא השתנתה

הציון של A השתנה

```
Student B info
Student id: 987654321
Name: Blinky Bill
*** Grades ***
History: 60
Math: Not Graded
Literature: Not Graded
English: Not Graded
```

ת.ז של B השתנתה

הציון של B השתנה!

כעת נראה כיצד לדרוס את בנאי ההעתקה, ולממש אותו בדרך הנכונה. נוסיף לקובץ ה header (Student.h) את החתימה של בנאי ההעתקה:



זה המקום לעשות חזרה ממש זריזה על מה זה reference:

- מוגדר ע"י הסימון &
- אם נשלח כפרמטר לפונקציה הקריאה לפונקציה נשארת אותו דבר, שליחת המשתנה עדיין נעשית כמו העברת ערך רגילה. (לא צריך להעביר כתובת)
- כאשר מוגדר כמשתנה הוא חייב להיות מאותחל באמצעות משתנה אחר.
- אפשר להחזיר רפרנס מפונקציה, אבל צריך לשים לב שהמשתנה לא נמחק בסוף הפונקציה
- משתנה ופוינטר דומים, אבל הם לא אותו דבר:
 - רפרנס לא יכול להכיל nullptr, וחייב להיות מאותחל
 - השימוש ברפרנס נעשה בצורה רגילה ללא צורך בכוכבית
 - לא ניתן להגדיר מערך של רפרנסים
 - לא חייב לתפוס מקום ולכן בדר"כ לא תופס. כתובת המשתנה היא כמו הכתובת של המשתנה אליו הוא מפנה.

חשוב מאוד להבין את ההבדל בין שני המושגים by value ו-by ref אם הנושא לא ברור מומלץ להיעזר [בסרטון](#)

נתחיל לממש את הבנאי:

את השדות הרגילים אין בעיה להעתיק כמו שעשינו עד עכשיו. אבל יש בעיה להעתיק את המצביע/מערך ולכן נצטרך להקצות מקום נפרד למצביע חדש, ולהעביר אליו את כל המידע שהיה במצביע הקודם. העתקה מסוג זה (כזו שמעתיקה את תכולת המצביעים) נקראת העתקה עמוקה (copy deep).

```

Student::Student(const Student& other)
{
    // shallow copy fields
    this->_id = other._id;
    this->_firstName = other._firstName;
    this->_lastName = other._lastName;

    // deep copy dynamic fields (pointers/arrays)
    this->_grades = new unsigned int[NUM_OF_GRADES];
    for (int i = 0; i < NUM_OF_GRADES; i++)
    {
        // copies cell by cell
        this->_grades[i] = other._grades[i];
    }
}

```

כעת נריץ את אותה תכנית ונקבל את הפלט הבא

```

Student A info
Student id: 123456789
Name: Blikny Bill
*** Grades ***
History: Not Graded
Math: Not Graded
Literature: Not Graded
English: Not Graded

```

```

Student B info
Student id: 123456789
Name: Blikny Bill
*** Grades ***
History: Not Graded
Math: Not Graded
Literature: Not Graded
English: Not Graded

```

changed Student A History grade to 60, B id to 987654321

```

Student A info
Student id: 123456789
Name: Blikny Bill
*** Grades ***
History: 60
Math: Not Graded
Literature: Not Graded
English: Not Graded

```

```

Student B info
Student id: 987654321
Name: Blikny Bill
*** Grades ***
History: Not Graded
Math: Not Graded
Literature: Not Graded
English: Not Graded

```

השדות הפרימיטיביים של A ו-B

הציון של A השתנה

הציון של B לא השתנה

שימו לב שהשתמשנו במילה new על מנת להקצות מקום נפרד למערך של B כעת A ו B זהים, אבל לא תלויים אחד בשני (מצביעים על מקומות שונים בזיכרון)

בחלק הבא נראה לחניכים מקרים שבהם הקומפיילר עושה שימוש בבנאי ההעתקה ללא ידיעתנו.

נתחיל בלהוסיף לבנאי ההעתקה שלנו שורת הדפסה

```
cout << "Hey!!! Someone copied me!!!" << endl;
```

נחליף את ה main שלנו ב main הבא:

```
#include "Student.h"
#include <iostream>

using std::cout;
using std::endl;
using std::string;

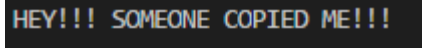
void dontCopy(Student s)
{
    // creates a student with default values
    Student newStudent(-1, "", "");

    newStudent.setId(s.getId());
    newStudent.setFirstName(s.getFirstName());
    newStudent.setLastName(s.getLastName());

    // grades are not copied
}

int main()
{
    Student stud(123456789, "Blinky", "Bill");
    dontCopy(stud);

    system("pause");
    return 0;
}
```


 HEY!!! SOMEONE COPIED ME!!!

הפלט של התכנית הוא:

הסיבה שבכל זאת נקרא בנאי ההעתקה היא שהעברנו את האובייקט שיצרנו בmain לפונקציה dontCopy.

כאשר בפונקציה מתקבל אובייקט כפרמטר נוצר העתק של האובייקט ע"י בנאי ההעתקה. במקרה שלנו יצרנו את האובייקט stud, העברנו אותו לפונקציה dontCopy אשר יצרה את האובייקט s באמצעות בנאי ההעתקה.

כדי להימנע מהעתקה ניתן לשלוח באמצעות reference או להעביר את הכתובת (פוינטר).

שאלה: האם אפשר לדרוס destructor?

התשובה היא שמכיוון שה destructor לא מקבל כלום (החתימה שלו לא משתנה), אי אפשר לדרוס אותו.

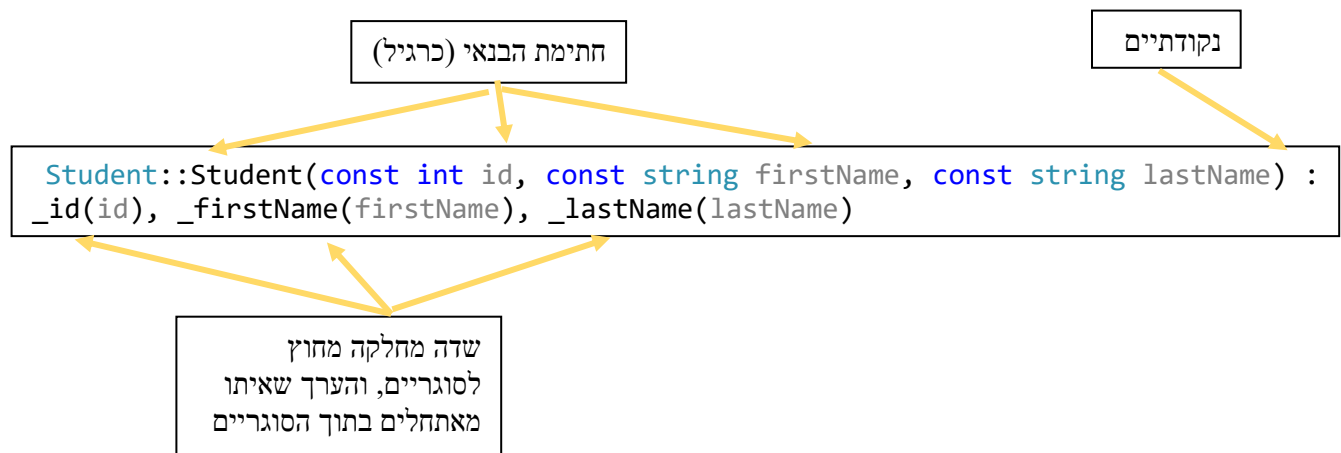
חלק 5 – שורת אתחול

נציג את הדרך לאתחול דברים לפני בניית האובייקט, כלומר דרך להגיד לקומפיילר לבצע אתחול של שדות/משתנים אחרים לפני שנכנסים לבנאי.

Initialization line - שורת אתחול מגדירים בבנאים. התחביר הוא כזה (קובץ cpp):

```
Student::Student(const int id, const string firstName, const string lastName) :_id(id), _firstName(firstName), _lastName(lastName)
```

שקול לזה (שורת האתחול למטה למטרת סדר)



שורת האתחול מבצעת את האתחולים עוד לפני שנכנסים לבנאי. עבור שדות רגילים, אין שום הבדל בין ביצוע שורת אתחול לבין ביצוע אתחול רגיל שעשינו עד עכשיו

```
// initialize fields
this->_id = id;
this->_firstName = firstName;
this->_lastName = lastName;
```

שורת אתחול הופכת להיות חשובה כשנלמד על הורשה, ושמרצה לאתחול שדות קבועים (כי ערכם צריך להיות מאותחל לפני הבנאי).

חלק 6 – העמסת אופרטורים

קודם כל נבין את הבעיה, וננסה לבצע השוואה בין שני סטודנטים באמצעות אופרטור '=='.

יגרור שגיאת קומפילציה

```
int main()
{
    Student stud(123456789, "Blinky", "Bill");
    Student stud2(stud);

    if (stud == stud2)
    {
        cout << "same student" << endl;
    }
    else
    {
        cout << "different student" << endl;
    }
    system("pause");
    return 0;
}
```

no operator "==" matches these operands -- operand types are: Student == Student

השגיאה שתתקבל:

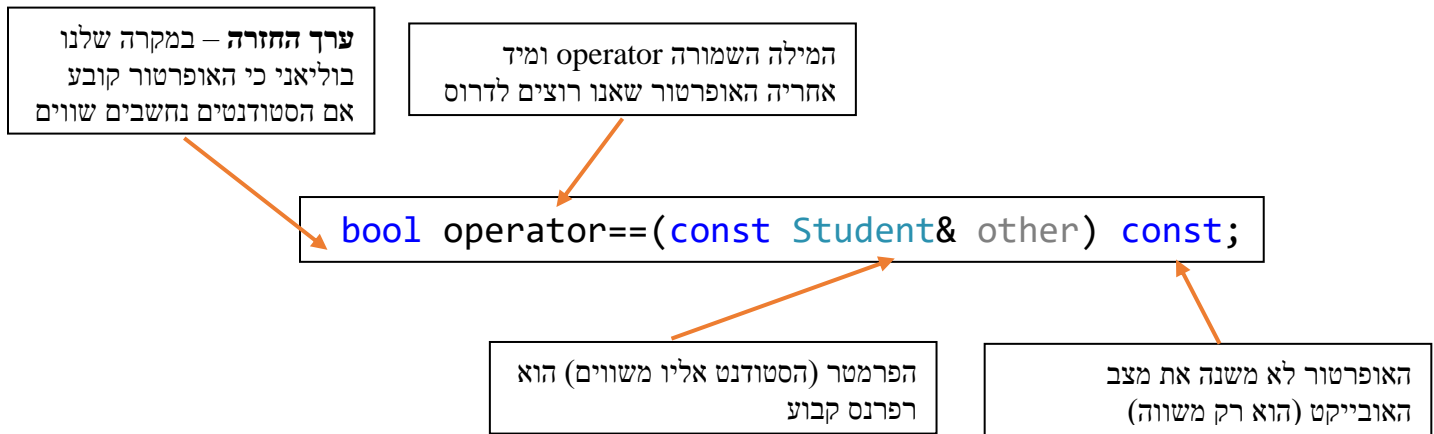
הקומפיילר אומר שהוא לא מכיר אופרטור "==" בין אובייקטים של Student, כלומר הוא לא יודע כיצד לבצע את ההשוואה.

אפשר לראות את אותה הבעיה כאשר מנסים השוואה באמצעות '<', '<=' או '>=', '>='. אפשר לראות שגם לבצע '++' או אפילו לנסות להדפיס את האובייקט של Student באמצעות cout יגרור את אותה שגיאה (לדוגמא: `cout << stud << endl;`)

כעת נראה כיצד מממשים את האופרטורים, כלומר איך אומרים לקומפיילר איזה קוד לבצע בעת השימוש באופרטור הזה. אופרטור הוא כמו מתודה רגילה במחלקה, היא צריכה להיות מוצהרת, יש לה חתימה וערך החזרה. כדי לראות את חתימות האופרטורים ניתן להיכנס [לקישור הבא](#)

נגדיר עבור המחלקה שלנו את אופרטור ההשוואה '==' (public) כי אחרת לא ניתן יהיה להשוות אובייקטים מחוץ למחלקה).

בדומה לדריסה של בנאי ההעתקה, יש חשיבות **לחתימה** וערך החזרה של האופרטור על מנת לדרוס אותו. צריך להיות מדויקים בטיפוסים שמצהירים עליהם:



כעת אפשר לממש אותו בקובץ ה `cpp` נוכל להגיד ששני סטודנטים נחשבים זהים אם יש להם אותו ת.ז.

```
bool Student::operator==(const Student& other) const
{
    return this->_id == other._id;
}
```

כעת ניתן לראות ששגיאת הקומפילציה נעלמה, ושהסטודנטים אכן נחשבים זהים.

נממש כמה אופרטורים נוספים.

נחזור למצגת ונראה שכעת ביקשו שניתן יהיה להשוות סטודנטים ע"פ גובה. כלומר נוסיף שדה פרטי בשם מסוג `int` שמציין את גובה התלמיד (בסנטימטרים).

בנוסף רוצים שניתן יהיה להוסיף לסטודנט נקודה לכל ציון באמצעות האופרטור `++`

נתחיל בהצהרות:

```
// operators
bool operator==(const Student& other) const;
bool operator<(const Student& other) const;
bool operator>(const Student& other) const;
Student& operator++(int);
```

שימו לב שבאופרטור ++ יש כמה דברים לא אינטואיטיביים:

- האופרטור מקבל int שבו לא נעשה שימוש, זוהי דוגמא שמראה כמה חשוב לחפש ב Google את החתימה לפני שדורסים אופרטור.
- חוזר רפרנס לאובייקט Student. הסיבה היא שיהיה ניתן לבצע את האופרטור כמה פעמים ברצף: לדוגמא $((stud++))++$ בכל פעם חוזרת תוצאת הביניים, והפונקציה פועלת על האובייקט שחזר מהתוצאה של הפעלת האופרטור הקודם.

אם היה חוזר void לדוגמא, אז $stud++$ היה מוחלף ב void, ואז הקומפיילר היה רואה ניסיון לבצע $void++$ וזו שגיאת קומפילציה.

```
bool Student::operator<(const Student &other) const
{
    return this->_height < other._height;
}
bool Student::operator>(const Student &other) const
{
    return this->_height > other._height;
}
Student& Student::operator++(int)
{
    for (int i = 0; i < NUM_OF_GRADES; i++)
    {
        if (this->_grades[i] < 100)
        {
            this->_grades[i]++;
        }
    }
    return *this;
}
```

נממש בצורה הבאה:

במימוש של ++operator אנו מחזירים את תוצאת הביניים שהיא בעצם האובייקט שעליו מתבצע האופרטור. כלומר אחרי שעלו הציונים בנקודה אפשר להחזיר את האובייקט שעליו מצביע this כתוצאת הביניים. כדי להגיע לאובייקט שעליו התבצע האופרטור אנו עושים dereference (כלומר משתמשים ב* כדי להיכנס לתכולת המצביע) של this.

כעת נוכל לתרגל את השימוש באופרטורים באמצעות main קצר

```
#include "Student.h"
#include <iostream> // std::cout, std::endl

using std::cout;
using std::endl;

int main()
{
    // initiates a Student object
    Student A(111111111, "Beyonce", "", 169);
    Student B(222222222, "Rihanna", "", 173);

    A.setGrade(HISTORY_GRADE_IDX, 90);
    A.setGrade(MATH_GRADE_IDX, 77);
    A.setGrade(LITERATURE_GRADE_IDX, 88);
    A.setGrade(ENGLISH_GRADE_IDX, 100);

    A.print();
    B.print();

    if (A < B)
    {
        cout << B.getFirstName() << " is taller than " << A.getFirstName() << endl;
    }
    if (A > B)
    {
        cout << A.getFirstName() << " is taller than " << B.getFirstName() << endl;
    }

    cout << "\nAdded 3 points to all student's grades" << endl;
    A++;
    (A++)++;

    // the following lines are equivalent:
    if (A == B)
    {
        cout << "SAME" << endl;
    }

    if (A.operator==(B))
    {
        cout << "SAME" << endl;
    }

    A.print();

    return 0;
}
```

נשים לב כיצד השתנו הציונים, ושביצענו השוואה בין אובייקטים של סטודנט באמצעות האופרטורים שדרסנו.

עוד נקודה שאפשר להתעכב עליה היא שהקומפיילר משנה את שורת השימוש באופרטור לשורת קריאה לפונקציה, כלומר כאשר הקומפיילר רואה `a==b` הוא למעשה הופך את השורה ל `a.operator==(b)`, כמו קריאה רגילה למתודה בשם `operator==` שאליה מועבר הפרמטר `b`.

```
// the following lines are equivalent:
if (A == B)
{
    cout << "SAME" << endl;
}

if (A.operator==(B))
{
    cout << "SAME" << endl;
}
```

כעת נרצה לממש את אופרטור ההעתקה (`operator=`). בדומה לבנאי ההעתקה, גם אופרטור העתקה ניתן לנו ב"חינם" ע"י הקומפיילר, ומבצע `shallow copy` כברירת מחדל. כדי לבצע `deep copy` נצטרך לדרוס את האופרטור.

חשוב להבין את הנקודה הבאה, כברירת מחדל ניתנים לנו 3 דברים חינם ע"י הקומפיילר (ממומשים עבורנו):

- מפרק (משחרר את המשאבים המקומיים של המחלקה – כלומר כל מה שהוקצה דינמית באמצעות `new/malloc` לא ישוחרר)
- בנאי העתקה (מקבל אובייקט מאותו סוג ומבצע `shallow copy`)
- אופרטור השמה/העתקה (מעתיק את מצב האובייקט מהאובייקט שמימין לאופרטור ב `shallow copy`)

אנו קוראים לפונקציות האלו `The Big 3`, ברגע שיש צורך לדרוס אחת מהפונקציות, אז כנראה שצריך גם לדרוס את השתיים האחרות. לדוגמא אם יש צורך לבצע `deep copy` בבנאי ההעתקה, כנראה שצריך גם לעשות את זה באופרטור העתקה, ולשנות את המפרק כך שישחרר את הזיכרון.

החתימה של אופרטור השמה/העתקה:

```
Student& operator=(const Student& other);
```

נשים לב שיש המון דמיון לאופרטורים שכבר ראינו, וערך ההחזרה הוא כזה שמאפשר את הפעלת האופרטור בשרשרת:

אם לדוגמא האופרטור לא היה מחזיר כלום (void) הייתה בעיה לבצע יותר מהעתקה אחת.

$a = b = c \nrightarrow (\text{void}) = c \rightarrow \text{compilation error}$

לדוגמא הפקודה $a = b = c$ תהפוך להיות (מבחינת הקומפיילר) ל- $(\text{void}) = c$ משום שערך ההחזרה של האופרטור הוא void.

לעומת זאת, אם מתוך הפעלת האופרטור תוחזר תוצאת הביניים, נוכל להפעיל את האופרטור בשרשרת.

כלומר $a = b = c$ יהפוך להיות (מבחינת הקומפיילר) $a = d$ כאשר d שווה לתוצאת הביניים כלומר לתוצאה של $(b = c)$

נעבור למימוש:

את אופרטור ההעתקה ניתן לממש בדיוק באותה דרך שמימשנו את בנאי ההעתקה, אולם יש להגן על עצמנו ממקרי קצה מסוימים:

- מישוה ניסה להעתיק את אותו האובייקט לעצמו (למשל $A=A$)
- האובייקט שאליו מעתיקים הכיל בעבר ערכים ושדות דינמיים, יש לשחרר את הזיכרון הישן לפני שדורסים אותו עם זיכרון חדש, כלומר למנוע זליגת זיכרון.

```
Student& Student::operator=(const Student& other)
{
    if (this == &other) // tries to copy the object to itself
    {
        return *this;
    }

    delete[] this->_grades; // release old memory

    // shallow copy fields
    this->_id = other._id;
    this->_firstName = other._firstName;
    this->_lastName = other._lastName;
    this->_height = other._height;

    // deep copy dynamic fields (pointers/arrays)
    this->_grades = new unsigned int[NUM_OF_GRADES];
    for (int i = 0; i < NUM_OF_GRADES; i++)
    {
        // copies cell by cell
        this->_grades[i] = other._grades[i];
    }

    return *this;
}
```

כעת שיש לנו את אופרטור ההשמה ממומש, ניתן לממש את בנאי ההעתקה בשורה אחת, וודאו שאתם/ן מבינים/ות למה השורה נכונה.

```
Student::Student(const Student& other)
{
    *this = other; // uses copy operator - copy other object to this object
}
```

השורה נכונה בגלל שכאשר יש ברשותנו אופרטור העתקה תקין ניתן להשתמש בו כדי להעתיק אל תוך האובייקט שנבנה (שעליו מצביע this) את התוכן של האובייקט האחר.

עוד נקודה בקשר לאופרטור העתקה:

יש מקרים בהם נרצה למנוע העתקה של אובייקט (לדוגמא אובייקט המכיל כמות עצומה של מידע כמו שרשראות D.N.A). אם נממש את אופרטור העתקה כפרטי, נוכל למנוע ממשתמשים להעתיק את האובייקט.

כעת נחזור למצגת כדי לעבור על משהו קטן ולסכם את השיעור.

ניתן להיעזר [בקישור הבא](#) לצורך חזרה על נושא Deep Copy vs Shallow Copy