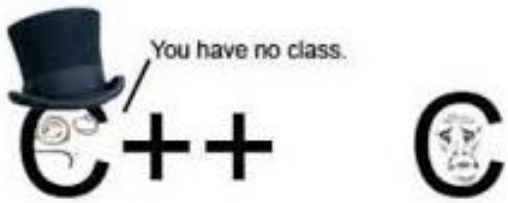
מגשימים – עקרונות מתקדמים

הדגמה חיה – מסמך לחניכים

שיעור 2 – תכנות מונחה עצמים

Contents



כללי

נושא ההדגמה: מבוא לתכנות מונחה עצמים, מחלקות וכימוס.

ראשי פרקים:

- בניית תכנית בצורה פרוצדורלית
- בניית התכנית כ Object Oriented
- הבנה של עיקרון הכימוס והטמעתו.

מטרות בשיעור:

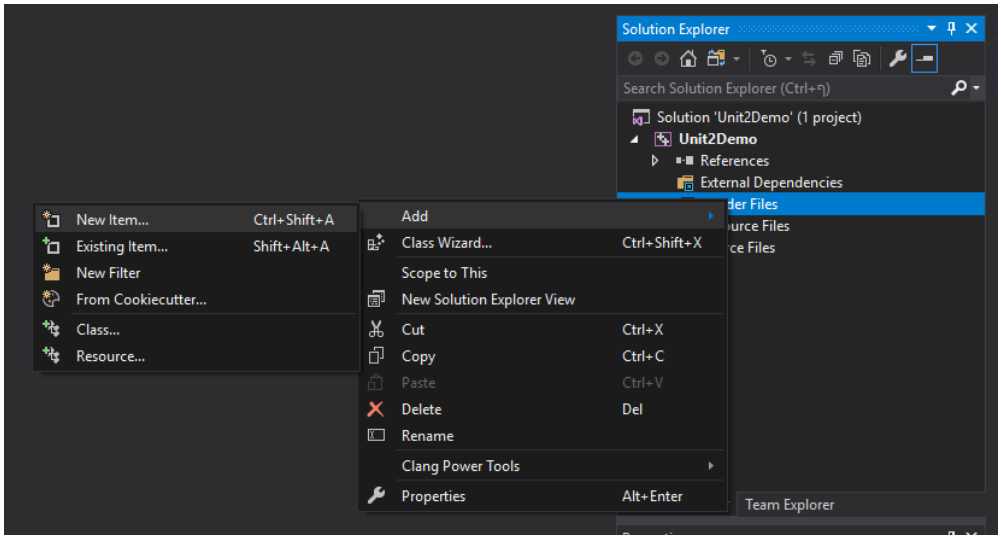
השיעור משלב בתוכו מצגת תיאורטית, והדגמה מעשית.
מטרתה של המצגת התיאורטית היא לתת הקדמה, סיפור מסגרת, וללוות את ההדגמה המעשית.
החל מ**שקופית מספר 7** יש לפתוח פרויקט בVisual Studio ולכתוב קוד שיתפתח לאורך השיעור.

התכנית שנבנה נכתבת בשני אופנים – פרוצדורלית ומונחית עצמים
דרך בניית התכנית נלמד את העקרונות החשובים בשיעור, כמו כן רמת הסדר והשמירה על הקונבנציות היא זו שנדרשת מאתנו גם בתרגילי הבית ולאורך כל השנה.

חלק 1 – פתיחת קובץ חדש

כבר יצא לנו להכיר את סביבת הפיתוח שלנו, גם משנה שעברה וגם מהתרגיל הקודם. אבל הנה תזכורת קטנה בכל זאת...
לאחר שפתחנו Empty Project חדש, נוסיף קובץ חדש מתוך הסרגל של הSolution Explorer .

נתחיל מלהוסיף קובץ cpp בשם program.cpp.



חלק 2 – הקדמה לחלק הראשון

לצורך החלק הראשון של ההדגמה אנחנו עובדים עם **קובץ אחד**, בהמשך השיעור ולאורך הסמסטר נקפיד על הפרדת התוכנית לקבצי header ומימוש, ולא נכתוב הכל בקובץ אחד. אולם בחלק הזה אנו רוצים בכוונה להתעלם מעקרונות OOP (כגון אבסטרקציה שמתבטאת בהפרדה לקובץ ממשק וקובץ מימוש)

בחלק הראשון של ההדגמה נממש את התוכנה שהופיעה בסיפור המסגרת בצורה **פרוצדורלית**, כלומר **כאוסף של פונקציות ומשתנים**. המטרה בחלק הזה היא לתת מוטיבציה לגישת OOP – תכנות מונחה עצמים, שמדמה את העולם בצורה קצת יותר טובה מאיך שהיינו רגילים עד עכשיו.

חשוב להבין שאין גישה מועדפת, ושזה תמיד תלוי בתוכנה שמפתחים, אבל חשוב להכיר את שתי הגישות ולהבין שגם אם שתיהן משיגות את המטרה, לכל אחת יש את היתרונות שלה.

צריך לזכור שתכנות פרוצדורלי לא בהכרח מבולגן, אפשר לעבוד מסודר ונקפיד על זה לאורך ההדגמה.

חלק 3 – כתיבת התכנית הפרוצדורלית

ננסה לחשוב אילו **משתנים** כדאי לשמור בשביל להשיג את מטרות התכנית? אפשר לחזור למצגת ולראות מה בדיוק דרשו מאיתנו.
בהמשך נראה שככל שננהל יותר משתנים בתכנית שלנו, התכנית תהפוך למסורבלת יותר.
אנו נגדיר משתנים של **שם פרטי ושם משפחה**, בנוסף נרצה לשמור את **תעודת הזהות** של התלמיד, ולכל תלמיד יש לשמור את **הציונים** שלו.

האם רשימה מקושרת היא מבנה הנתונים העדיף לשמירת הציונים? באופן כללי רשימה יכולה לעבוד, אבל מכיוון שאנחנו לא רוצים לעבוד עם זיכרון דינמי ומצביעים, נעבוד עם מערך, וזה מתאפשר מכיוון שמספר הציונים **קבוע מראש**, כלומר ניתן להגדיר מראש את המערך בגודל המתאים. (אפשר לחזור למצגת, ולראות שיש 4 ציונים סה"כ)

נתחיל מטיפול בסטודנט בודד, בהמשך נתרחב למספר סטודנטים.
נתחיל בלהצהיר על ת.ז ומערך הציונים.

```
int main()
{
        unsigned int studentId;
        unsigned int studentGrades[4];
        return 0;
}
```

כבר בקוד הבסיסי הזה יש כמה דברים חשובים.
ראשית נשים לב שה- **main מחזיר מספר**, אפשר נזכור שזהו קוד שמסמל אם התכנית הסתיימה בהצלחה, או האם התרחשה שגיאה מסוימת (0 אומר שהתכנית הסתיימה ללא שגיאה).
unsigned אומר שאנחנו לא מאפשרים אחסון של ערכים שליליים, ובמידה וערך המשתנה ירד מתחת לאפס הוא יהפוך לערך המקסימלי בטווח המשתנה. לדוגמא משתנה מסוג unsigned byte יכול לאחסן ערכים מספריים בטווח 0-255. במידה והוצב במשתנה 1- ערכו של המשתנה בפועל יהיה 255 (כאילו הלכנו צעד אחד מהסוף).

המחמירים שבינינו עשויים להגיד שגם האינדקס של הציון במערך צריך להיות מסוג unsigned (מכיוון שהוא תמיד חיובי), הם צודקים, אולם אנו לא חייבים להיות קפדנים עד כדי כך.

גודל מערך הציונים הוא 4. נתרגל להשתמש בפקודות #define כדי להפוך את התכנית לקריאה יותר, ולכן מספרים קבועים בעלי משמעות יש ללוות באמצעות #define מתאים.

נוסיף לקוד שלנו פקודת #define בשביל גודל המערך.
האם רשמנו את הפקודה בצורה נכונה? לא! הוספת ';' זוהי שגיאה נפוצה

```
#define NUM_OF_GRADES 4;
```

הוספת ';' תיצור שגיאת קומפילציה

חשוב להבין שפקודות #define הן פקודות "טיפשות" במובן מסוים. פקודת #define מייצרת החלפה טקסטואלית – כלומר בכל מקום בקוד שיופיע NUM_OF_GRADES הוא יוחלף בערך שמימין להגדרה – במקרה שלנו 4;. הוספת ';' במקומות לא רצויים בקוד ישנה את המבנה שלו ועשוי ליצור שגיאות קומפילציה. הדרך הנכונה להגדרת קבועים היא ללא תוספת ';'.
נוסיף הגדרת קבועים למספר הציונים במערך (גודל המערך) ואת האינדקסים השונים.

```
// grades array access
#define NUM_OF_GRADES 4;
#define HISTORY_GRADE_IDX 0
#define MATH_GRADE_IDX 1
#define LITERATURE_GRADE_IDX 2
#define ENGLISH_GRADE_IDX 3
```

ותמיד נזכור שההגדרות נועדו לתת סדר, וכדי לעזור למי שקורא את הקוד להבין את הערכים ששלחנו (במקום שנראה 2 נוכל להבין שזהו המיקום של הציון בספרות).

נוסיף גם הגדרה לציון ריק (מסמל שהתלמיד לא קיבל ציון במבחן הספציפי), נסכים על ערך 200 (אפשר להסכים על כל ערך בתנאי שהוא חיובי [בגלל unsigned] ולא חופף לערכי ציונים [0-100])

```
#define EMPTY GRADE 200
```

בהזדמנות זו אפשר גם אפשר להיזכר בשלבי הקומפילציה (מי שסגור על זה יכול לדלג). ניתן גם לראות את השלבים השונים עבור תכנית קטנה כמו בקישור הבא
ישנם 3 שלבים שעוברים מהרגע שקוד הופך לאובייקט רשום במערכת ההפעלה שאותו ניתן להריץ.

1. שלב ה preprocess – בשלב זה מועתקים כל חלקי הקוד שיובאו בצורה חיצונית ע"י הפקודה #include , כלומר הם מודבקים כמו שהם בתוך הקובץ שייבא אותם. בנוסף מתבצעות כל ההחלפות הטקסטואליות של פקודות ה #define ונקראות פונקציות ה macro שהוגדרו (אין סיבה להיכנס לזה, אפשר לקרוא על כמה מתבניות ה macro המוכרות כאן)
2. שלב הקומפילציה - הקומפיילר עובר על הקוד ובודק שכל ה type-ים בתכנית מסכימים אחד עם השני (אין השמה של type לא נכון), שאין הגדרות כפולות, ושכל פונקציה קיבלה את מספר הארגומנטים הנכונים. ישנן המון בדיקות שנעשות בזמן קומפילציה, אין צורך להתעמק בשאר הבדיקות. הקומפיילר מתרגם את הקוד של התכנית לשפת assembly (שפת low level) שמוגדרת ע"פ ארכיטקטורת המעבד שמריץ את התכנית.
3. שלב ה assembly – הקוד שרשמנו מועבר משפת low level כגון אסמבלי, ומשם מתורגם לקוד בינרי = קוד שהמעבד יודע לקרוא ולבצע.
4. שלב הלינקג' – הקומפיילר מקשר בין ההצהרה של הפונקציה ובין המימוש שלה, צריך לזכור שרק פונקציות שנעשתה קריאה אליהן ילונקג'ו, ולכן לא נוכל לראות שגיאות לינקג' (לדוגמא שאין מימוש לפונקציה) במידה ולא קראנו לפונקציה.

נוסיף הצהרות של **שם פרטי ומשפחה** עבור תלמיד.
מרגע זה ננסה להימנע מהגדרת רצפי תווים כ- *char ונשתמש במחלקה שבנו בשבילנו שנועדה בדיוק לצורך הזה. זה הזמן להכיר להם את **מחלקת string** של הספרייה הסטנדרטית.

מה הבעיה בקוד הבא? למה הוא לא מתקמפל?

```
int main()
{
        int studentId;
        int studentGrades[4];
        string studentFirstName;
        string studentLastName;
        return 0;
}
```

יש כמה סיבות שהקוד לא מתקמפל, והראשונה היא שלא ייבאנו את הקוד של המחלקה string.
נוסיף את השורה #include <string> בתחילת הקובץ

```
#include <string>
```

נזכור שההבדל בין #include של קובץ עם גרשיים (לדוגמא #include "My_File.h") לעומת סוגריים משולשים (<>) הוא ש "" גורמים לקומפיילר לחפש קובץ בתיקייה המקומית של הפרויקט, וסוגריים משולשים גורמים לקומפיילר לחפש ספריות חיצוניות.

גם אחרי שהוספנו #include הקוד עדיין לא מתקמפל והסיבה היא שכרגע אנו יכולים להצהיר על משתנה מסוג string באמצעות הוספת ה namespace שלו, כלומר ניתן להצהיר על המשתנים כסוג std::string, ולא כסוג string בלבד.

נבצע comment על השורות הבעייתיות, נלמד קצת על הנושא namespace ונחזור.
ניתן להשתמש בקיצור דרך לביצוע comment על כמה שורות בבת אחת באמצעות לחיצה על Ctrl+k ואז על Ctrl+c
באותו אופן אפשר לבצע uncomment (של השורות המסומנות) באמצעות Ctrl+k ואז Ctrl+u
בסביבה של Microsoft Visual Studio. ניתן ללמוד קיצורים נוספים בקישור הבא
http://visualstudioshortcuts.com/2017/

namespaces

namespace הוא מושג מאוד חשוב להבניה וכתיבה של מערכות גדולות, אבל גם בגלל שאנחנו לא מתכננים מערכות גדולות, ובעיקר בגלל שיש לנו פחות משנה אנחנו נעשה רק סקירה קצרה של הנושא.

מה זה namespace?
מרחב לוגי שמאגד תחתיו פונקציות, משתנים (ובהמשך מחלקות).

namespaces עושים לנו סדר, ועוזרים לנו להימנע מכפילויות:

למשל אם כתבנו פונקציה אחת בשם print שתפקידה להדפיס למסך ופונקציה אחרת print שתפקידה להדפיס למדפסת.
אם יהיו 2 פונקציות עם אותו שם זה יהיה בעייתי.
ניתן להגדיר כל אחת ב-namespace שונה, וזה יאפשר לגשת אליהם ולזהות אותו גם בעזרת ה-namespace וגם על ידי שמן.

נכתוב את הקוד הבא. (צריך לבצע #include <iostream> כדי שאפשר יהיה לגשת ל std::cout)

מילה שמורה namespace

הפונקציה מוגדרת בתוך מרחב השמות

```
namespace first
{
	void myPrint()
	{
		std::cout << "first" << std::endl;
	}
}

namespace second
{
	void myPrint()
	{
		std::cout << "second" << std::endl;
	}
}
```

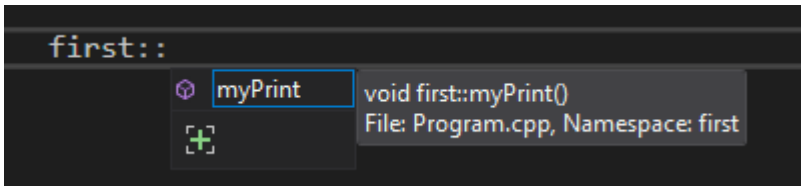
למעשה הגדרנו שני מרחבי שמות (namespaces) ובשניהם הגדרנו פונקציה בעלת אותו שם (myPrint).

כעת נראה שאפשר לגשת לכל אחד ממרחבי השמות ולעורר את הפונקציות שכתובות בו באמצעות האופרטור **::** (נקודתיים כפולים)

בפונקציית ה main נבצע קריאות לשתי הפונקציות דרך מרחב השמות

```
int main()
{
        int studentId;
        int studentGrades[4];
        string studentFirstName;
        string studentLastName;
        second::myPrint();
        first::myPrint();
        return 0;
}
```

הnamespace מופיע כמו משתנה רגיל במרחב הקוד שלנו, אבל כאשר נכתוב את האופרטור :: מימינו נוכל לגשת לכל הפונקציות והמשתנים שהוגדרו בתוכו.



זו הסיבה שאנו נדרשים לבצע std::cout כאשר רוצים לגשת לoutput stream שנמצא במרחב השמות של הספרייה הסטנדרטית (std).

ניתן (והמון מתכנתים מעדיפים את השיטה הזו) לכתוב גם את הקידומת (namespace) של משתנה או פונקציה, למשל std::string, , כי אז ברור בעת קריאת הקוד מאיפה השם string מגיע, וזה הופך אותו לברור יותר.

אם זאת, קיימת אופציה לקצר את הקוד הנכתב ולהשמיט את מרחבי השמות באמצעות ייבוא/שימוש ב namespace של ספריות חיצוניות.

נראה כיצד מייבאים מרחב שמות אל מרחב השמות של התכנית שלנו, כלומר להפוך את כל המשתנים/הפונקציות במרחב השמות שייבאנו לחלק מהתכנית שלנו (כאילו היינו מגדירים בעצמנו).

את הייבוא נבצע באמצעות הפקודה **using**.
using מייבאת את השם/מרחב השמות שמצד ימין אליה אל תוך מרחב השמות בה היא נכתבת.
כלומר אם נבצע מתוך התכנית שלנו `using namespace first;`
נוכל להשתמש בכל מה שמוגדר בתוך מרחב השמות כאילו הגדרנו אותו בעצמנו.

נוסיף עוד משתנה מסוג int שנקרא myInt בתוך מרחב השמות first.
לצורך בדיקה ניתן לנסות לקרוא לפונקציה myPrint ללא ה namespace ולראות שנקראת הפונקציה שציפינו, בנוסף נוכל לגשת למשתנה myInt ללא הוספת ה namespace.

ניתן לייבא חלק מתוך מרחב השמות, לדוגמא ניתן לייבא רק את הפונקציה myPrint באמצעות הפקודה using. לדוגמא: `using first::myPrint;` מייבא רק את הפונקציה myPrint מתוך מרחב השמות, אל המשתנה myInt עדיין נצטרך לגשת באמצעות ה namespace.

תמיד נשאף לייבא אך ורק מה שאנחנו צריכים, ולא לייבא namespace שלם ללא צורך. לדוגמא אחת הפקודות הנפוצות שעושים ב C++ היא `using namespace std;`. שימוש בכל מרחב השמות של הספרייה הסטנדרטית יזהם את מרחב השמות של התכנית שלנו בהמון שמות לא רצויים (תופעה זו נקראת namespace pollution).
לדוגמא אחרי ביצוע הפקודה `using namespace std;` לא נוכל להגדיר פונקציה משלנו בשם sort או להשתמש בשמות רבים.
לכן נזכור לייבא אך ורק את הדברים הדרושים.
בתכנית שלנו, למשל נבצע

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

כעת אפשר לגרום לשורות הקוד שלהן עשינו comment לעבוד, ע"י ביצוע using std::string כדי לייבא את הטיפוס string למרחב השמות של התכנית שלנו.

```
using std::string;
```

שוב, חשוב לזכור שאפשר ואפילו מומלץ לכתוב את התכנית ללא using כדי שיהיה ברור מאיפה כל פקודה נלקחה, וש - using היא אופציה ולא קונבנציה.

המשך חלק 3 – המשך כתיבת התכנית הפרוצדורלית

כעת כשיש לנו את כל המשתנים הדרושים כדי לבצע את מה שביקשו נציב ערכים לדוגמא (כלומר נציב ערכים של סטודנט פיקטיבי בשביל לבנות את הפונקציות בהמשך).

```
int main()
{
	int studentId = 123456789;

	int studentGrades[NUM_OF_GRADES];
	studentGrades[HISTORY_GRADE_IDX] = 78;
	studentGrades[MATH_GRADE_IDX] = 81;
	studentGrades[LITERATURE_GRADE_IDX] = 90;
	studentGrades[ENGLISH_GRADE_IDX] = 65;

	// same as writing
	// int studentGrades[NUM_OF_GRADES] = { 78, 81, 90, 65 };

	string studentFirstName = "Shahar";
	string studentLastName = "Hasson";


	return 0;
}
```

הפונקציה הראשונה שנכתוב היא averageGrade אשר מקבלת את מערך הציונים של התלמיד.

נצהיר על הפונקציה בתחילת הקובץ.

```
double averageGrade(unsigned int* grades);
```

את ההצהרה על הפונקציות אנו עושים כדי שנוכל לקרוא לפונקציה מכל שורות הקוד של הקובץ.
ב C/C++ לא ניתן לקרוא לפונקציה שהצהרתה נמצאת בשורה תחתונה מזו של הקריאה. כלומר אם הפונקציה לא מוגדרת מעל השורה שבה השתמשנו בה נקבל שגיאת קומפילציה.
כאשר אנו מצהירים על הפונקציה בתחילת הקובץ, הפונקציה נגישה מכל חלקי הקוד והקומפיילר דואג ללנקג' את ההצהרה על הפונקציה למימושה.

נשים לב שאנו מקבלים כפרמטר **מצביע** ל int (`unsigned int* grades`) שזה בעצם כתובת שמצביעה לאזור בזיכרון בו שוכן int.
אפשר יהיה לשלוח את כתובת המערך שהוגדר בmain, ובגלל שמדובר בכתובת של המשתנה אז המערך ישתנה במהלך הפונקציה, וגם יחזור עם השינוי (כלומר שינוי במערך במהלך הפונקציה ישנה גם את המערך ששלחנו ב main מכיוון ששניהם מצביעים על אותו אזור בזיכרון).
בשיעור הבא נתמקד בנושא הזיכרון יותר, ולכן אין סיבה להתעמק.

נממש את הפונקציה:

```
double getAverage(unsigned int* grades)
{
	double sum = 0;
	int numOfValidGrades = 0;
	for (int i = 0; i < NUM_OF_GRADES; i++)
	{
		if (grades[i] != EMPTY_GRADE)	// takes only valid grades
		{
			numOfValidGrades++;
			sum += grades[i];
		}
	}
	if (numOfValidGrades == 0)		// avoids division by zero
		return -1;
	return sum / numOfValidGrades;	// the average of all valid grades
}
```

אנחנו עוברים רק על הציונים האמיתיים, ולא מחשבים את הציונים הריקים. בנוסף אנחנו מתעדים חלקי קוד פחות טריוויאליים, ונמנעים מחלוקה באפס.

הדבר הבא שנעשה זה לכתוב פונקציה אשר מדפיסה את נתוני הסטודנט בפורמט נוח, לדוגמא:

```
Student Id: 123456789
Name : Shachar Hason
Grades :
History : 78
Math : 81
Literature : 90
English : 65
```

כדי לכתוב קוד קצת יותר אלגנטי נכתוב פונקציית עזר אשר מחזירה מחרוזת שמייצגת ציון ספציפי. (כלומר ממירה ל string) ציון בודד.

נצהיר עליה בתחילת הקובץ

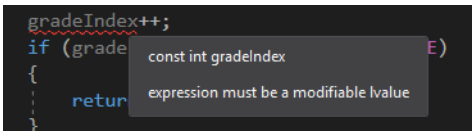
```
string getGradeString(unsigned int* grades, const int gradeIndex);
```

נשים לב שהוספנו את המילה **const**. המילה const היא מילה שמורה שמציינת לקומפיילר שלא ניתן לשנות את הערך שמצד ימין שלה. כלומר במקרה שלנו אנו מעבירים כפרמטר את האינדקס של הציון, אבל מונעים את האפשרות לשנות אותו במהלך הפונקציה.
נקפיד על סימון הקבועים לאורך כל הסמסטר, וחשוב לוודא שמנענו את האפשרות לשנות ערכים קבועים במהלך הפונקציה.

נממש את הפונקציה.

```
string getGradeString(unsigned int* grades, const int gradeIndex)
{
	if (grades[gradeIndex] == EMPTY_GRADE)
	{
		return "Not Graded";
	}
	else
	{
		return std::to_string(grades[gradeIndex]);
	}
}
```

אפשר ומומלץ להשתמש בפעולות שנמצאות בספרייה הסטנדרטית לדוגמא ()to_string.
בדר"כ קל למצוא אותן ב- Google, הן יעילות ובטוחות.



נוכל לראות שאם ננסה לשנות את המשתנה
$gradeIndex$ במהלך הפונקציה נקבל שגיאת
קומפילציה משום שהוא הוגדר כ const.
נצהיר על הפונקציה שמדפיסה את נתוני הסטודנט.

```
void print(const int id, const string fName, const string lName, unsigned int* grades);
```

ונממש

```
void print(const int id, const string fName, const string lName, unsigned int* grades)
{
	cout << "Student id: " << id << endl
		<< "Name: " << fName << " " << lName << endl
		<< "*** Grades ***" << endl;
	cout << "History: " << getGradeString(grades, HISTORY_GRADE_IDX) << endl
		<< "Math: " << getGradeString(grades, MATH_GRADE_IDX) << endl
		<< "Literature: " << getGradeString(grades, LITERATURE_GRADE_IDX) << endl
		<< "English: " << getGradeString(grades, ENGLISH_GRADE_IDX) << endl;
}
```

נחזור לשקופית מס' 6 מצגת ונראה שהדרישה האחרונה מהתכנית היא היכולת לתמוך בכיתות (של 10 תלמידים).

איך כדאי לשמור מידע על יותר מסטודנט אחד?
כנראה שחלקינו כבר מכירים את נושא ה struct ולכן כנראה חשבתם כבר על הפתרון הנכון, אבל אנחנו מנסים בכוונה להמשיך את התכנית בכיוון הפרוצדורלי, ולהימנע משימוש בישויות כגון מחלקה או struct.

נוסיף מידע על תלמידה נוספת בפונקציית ה main:

```
int student2Id = 111111111;
int student2Grades[NUM_OF_GRADES] = { 95, 87, 90, 98 };
string student2FirstName = "Beyonce";
string student2LastName = "";
```

אנחנו כבר מתחילים לראות את הבעייתיות, ומבינים שאם נמשיך ככה נצטרך להגדיר המון משתנים שלנהל אותם יהיה לא כיף, לכן ננסה לשפר את התכנית, ונגדיר מערכים עבור כל משתנה.
כל מערך יהיה באורך מספר התלמידים המקסימלי בבית הספר, והאינדקס יציין את המיקום של נתוני התלמיד בכל מערך.

```
#define EMPTY 200
#define MAX_NUM_OF_STUDENTS 30

int studentIdsArray[MAX_NUM_OF_STUDENTS] = { EMPTY };
unsigned int studentGradesTable[MAX_NUM_OF_STUDENTS][NUM_OF_GRADES] = { EMPTY };
string studentFirstNamesArray[MAX_NUM_OF_STUDENTS] = { "" };
string studentLastNamesArray[MAX_NUM_OF_STUDENTS] = { "" };
```

כעת ה main שלנו יראה ככה:

```
int main()
{
	// first student info
	int student1Idx = 0;
	studentIdsArray[student1Idx] = 123456789;
	studentGradesTable[student1Idx][HISTORY_GRADE_IDX] = 78;
	studentGradesTable[student1Idx][MATH_GRADE_IDX] = 81;
	studentGradesTable[student1Idx][LITERATURE_GRADE_IDX] = 90;
	studentGradesTable[student1Idx][ENGLISH_GRADE_IDX] = 65;
	studentFirstNamesArray[student1Idx] = "Shahar";
	studentLastNamesArray[student1Idx] = "Hasson";

	// second student info
	int student2Idx = 1;
	studentIdsArray[student2Idx] = 111111111;
	studentGradesTable[student2Idx][HISTORY_GRADE_IDX] = 95;
	studentGradesTable[student2Idx][MATH_GRADE_IDX] = 87;
	studentGradesTable[student2Idx][LITERATURE_GRADE_IDX] = 90;
	studentGradesTable[student2Idx][ENGLISH_GRADE_IDX] = 98;
	studentFirstNamesArray[student2Idx] = "Beyonce";
	studentLastNamesArray[student2Idx] = "";

	// and so on...

	return 0;
}
```

בשביל להוסיף כיתות לתכנית, נוסיף עוד מערך של כיתות, ובכל תא ששייך לסטודנט מסוים תופיע מספר הכיתה שלו, כלומר כל תא יגיד באיזה כיתה הסטודנט שאליו שייך התא.

```
int studentClassRoomsArray[MAX_NUM_OF_STUDENTS] = { EMPTY };
```

וב - main (מספרי הכיתות הם 0-2):

```
studentClassRoomsArray[student1Idx] = 0;
studentClassRoomsArray[student2Idx] = 1;
```

כל מה שנשאר זה לכתוב פונקציה אשר מציגה את כל הסטודנטים בכיתה.

```
void printStudentInClass(const int classNum)
{
	cout << "Class room number " << classNum << " students info:" << endl;

	for (int studentIdx = 0; studentIdx < MAX_NUM_OF_STUDENTS; studentIdx++)
	{
		if (studentClassRoomsArray[studentIdx] == classNum)
		{
			print(studentIdsArray[studentIdx],
				studentFirstNamesArray[studentIdx],
				studentLastNamesArray[studentIdx],
				studentGradesTable[studentIdx]);
		}
	}
}
```

אז מה ראינו? ראינו שאפשר לעשות את מה שאנחנו רוצים גם בגישה הישנה, והאמת שרוב המפתחים עובדים ככה. עכשיו נחזור למצגת כדי לראות כמה נקודות בעייתיות שהיו לנו בקוד, ולהתחיל ללמוד גישה חדשה שנקראת Object Oriented.

חלק 4 – כתיבת התכנית בגישת תכנות מונחה עצמים

4.1 הגדרת המחלקה Student

נתחיל באיך מגדירים מחלקה ב C++. באיזה סוג של קובץ צריכה להיות מוגדרת המחלקה? האם בקובץ header או האם בקובץ cpp?
התשובה היא שמחלקה היא ישות רעיונית (אבסטרקטית) היא רק מגדירה את המאפיינים והממשק של האובייקט, כלומר צריכה להיות בקובץ ה header.

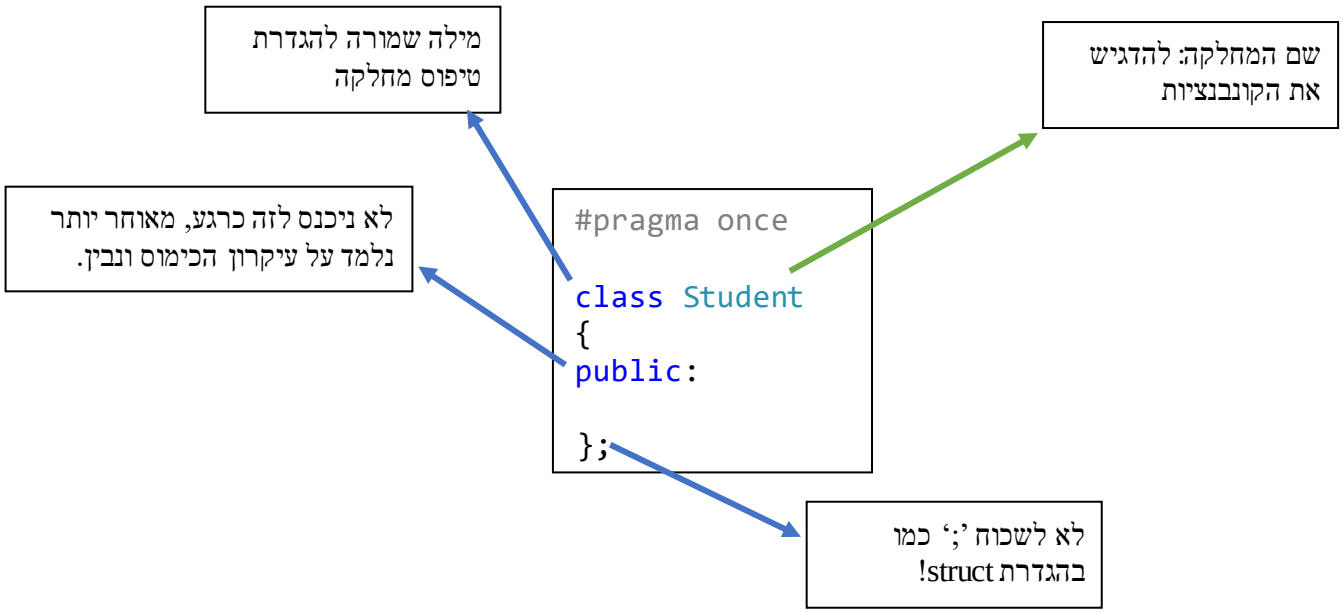
נפתח קובץ חדש בשם Student.h. נקפיד על קונבנציות, קובץ המכיל מחלקה יהיה באות גדולה.

מיד בפתיחת הקובץ נשים לב שה Visual Studio הוסיף עבורנו את השורה `#pragma once`
השורה #pragma once שקולה לפקודות הבאות.

```
#ifndef UNIT_2_DEMO_STUDENT_H  // equivalent to #pragma once
#define UNIT_2_DEMO_STUDENT_H

#endif
```

מה המטרה של השורות האלו?
בתהליך ה preprocess (שקורה לפני תהליך הקומפילציה), מועתקים חלקי הקוד אל תוך הקבצים אשר ביצעו #include.
פקודות אלו דואגות שלא נעתיק פעמיים את תכולת הקובץ, ומונעות הגדרה כפולה של מבנים/מחלקות/פונקציות אשר גוררות שגיאות קומפילציה.
(דוגמא למצב כזה היא ביצוע שתי פקודות #include לאותו קובץ ברצף).

נגדיר מחלקה המייצגת ישות של תלמיד – מחלקת Student



4.2 הגדרת שדות המחלקה

באופן דומה לאיך שהגדרנו את המשתנים, נגדיר את השדות של המחלקה כלומר המאפיינים של תלמיד.
נזכור לבצע #include <string> בשביל שדות השם פרטי/משפחה. נוסיף את פקודות ה #define שעשינו שעזרו לנו לגשת למערך בצורה ברורה ומסודרת.

```
#include <string>

// grades array access
#define NUM_OF_GRADES 4
#define HISTORY_GRADE_IDX 0
#define MATH_GRADE_IDX 1
#define LITERATURE_GRADE_IDX 2
#define ENGLISH_GRADE_IDX 3

// initial grade value
#define EMPTY_GRADE -1

class Student
{
public:
	// fields
	int id;
	std::string firstName;
	std::string lastName;
	unsigned int grades[NUM_OF_GRADES];

};
```

אנחנו לא משתמשים בפקודה **using** מתוך קובץ h, משום שייבוא מרחב שמות (או שם ספציפי) יגרור ייבוא שלו לכל מי שמייבא את הקובץ שלנו. כלומר אם השתמשנו ב using כל מי שעשה #include Student.h ייבא את השם שבו השתמשנו מבלי שביקש.

בנוסף חשוב להדגיש שבקבצי h אנו עושים #include לספריות שדרושות בשביל ממשק המחלקה, למשל בדוגמא שלנו אנו צריכים גישה ל std::string ולכן ביצענו #include, אולם אין צורך לבצע #include <iostream>, כי זה קשור למימוש, אותו אנו לא חושפים החוצה.

4.2 הגדרת מתודות המחלקה

נוסיף את ההצהרה על המתודות של המחלקה:

```
public:
        // fields
        int id;
        std::string firstName;
        std::string lastName;
        unsigned int grades[NUM_OF_GRADES];

        // methods
        double getAverage() const;
        void print() const;
        std::string getGradeString(const int gradeIndex) const;// helper method

};
```

שימוש במילה const בסוף מתודה

לפעמים נרצה להגדיר מראש מתודות ש**לא יכולות לשנות את ערך השדות באובייקט**, ולשם כך נוסיף לחתימה שלהן את המילה const בסוף.
פונקציות כמו getAverage או פונקציות הדפסה לא משנות את השדות של האובייקט אלא רק משתמשות בהן, ולכן יוגדרו כ const.
גם זה יהיה סטנדרט לאורך השנה, ונקפיד עליו!.

4.3 מימוש מחלקת Student

בחלק הזה נתחיל לממש את המתודות של המחלקה.

נפתח קובץ חדש בשם Student.cpp, נסביר שעל קבצי מימוש אין צורך להגדיר #ifdef או #pragma once משום שאליהם לא מבצעים #include.

נתחיל לערוך את הקובץ שפתחנו, ותחילה נרשום #include לקובץ שכתבנו (באמצעות "" כדי שנכלול קבצים מהתיקייה המקומית)

```
#include "Student.h"
```

בתחילת קובץ ה cpp נבצע את כל הפעולות שיקלו על המימוש, למשל using.

```
#include <iostream>

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

נעבור למימוש מתודות, נתחיל מהפונקציה getAverage.

חשוב לכלול את המילה const במידה והפונקציה הוגדרה כך

חתימת הפונקציה

פנייה ל namespace של מחלקת Student

ערך ההחזרה של הפונקציה

```
double Student::getAverage() const
{
	double sum = 0;
	int numOfValidGrades = 0;
	for (int i = 0; i < NUM_OF_GRADES; i++)
	{
		if (grades[i] != EMPTY_GRADE) // takes only valid grades
		{
			numOfValidGrades++;
			sum += grades[i];
		}
	}
	if (numOfValidGrades == 0)
	{
		return 0; // avoids division by zero
	}
	return sum / numOfValidGrades; // the average of all valid grade
}
```

(זהו אותו קוד שכתבנו קודם, רק עכשיו הוא ממומש במתודה)
כאשר רוצים לממש פונקציה שמוגדרת בתוך מחלקה (מתודה) יש לכלול את **החתימה** המלאה של הפונקציה, **ערך ההחזרה** שלה, את המילה **const** (במידה והוגדרה כך) ואת ה **namespace** של המחלקה. אם אחד הדברים יהיה חסר או שונה מאיך שהוגדר במחלקה, הקומפיילר יחשוב שאנו מגדירים פונקציה חדשה באותו הקובץ ולא מממשים את הפונקציה שרצינו.
נממש את הפונקציות print ופונקציית העזר getGradeString

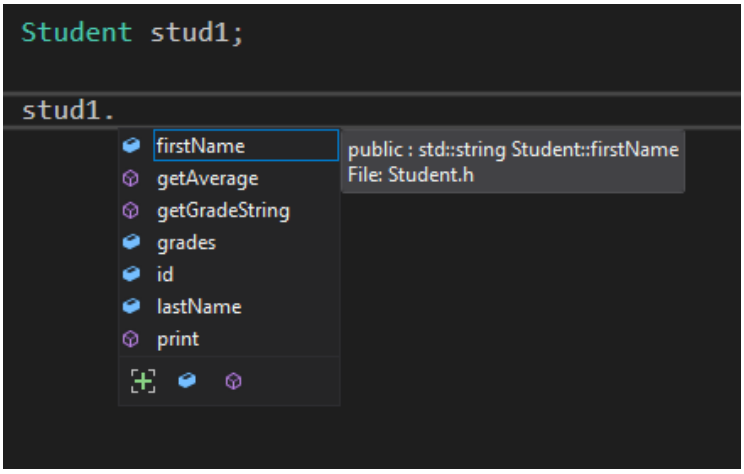
```
void Student::print() const
{
	cout << "Student id: " << id << endl
		<< "Name: " << firstName << " " << lastName << endl
		<< "*** Grades ***" << endl;
	cout << "History: " << getGradeString(HISTORY_GRADE_IDX) << endl
		<< "Math: " << getGradeString(MATH_GRADE_IDX) << endl
		<< "Literature: " << getGradeString(LITERATURE_GRADE_IDX) << endl
		<< "English: " << getGradeString(ENGLISH_GRADE_IDX) << endl;
}

string Student::getGradeString(const int gradeIndex) const
{
	if (grades[gradeIndex] == EMPTY_GRADE)
	{
		return "Not Graded";
	}
	else
	{
		return std::to_string(grades[gradeIndex]);
	}
}
```

שימו לב שבשונה מאיך שכתבנו את התכנית קודם\ כאן המתודות לא צריכות לקבל את כל הארגומנטים, מכיוון שהן חלק מאותה ישות שמחזיקה בשדות האלו.
אם בתכנות פרוצדורלי לא הייתה הפרדה בין המידע והפונקציות שיש בתכנית וכולם היו יכולים לגשת לכולם, פה אנחנו מאגדים את כל מה שרלוונטי לישות של תלמיד תחת אותו שם – Student.

לפני שנעבור ליצור אובייקטים, **נחזור למצגת** כדי ללמוד כמה דברים תיאורטיים.

ניצור קובץ חדש עם פונקציית main (נדאג לבצע comment לקוד הקודם כדי שלא יהיו שתי פונקציות main באותו פרויקט), ושם נראה כיצד יוצרים מופעים למחלקה Student – כלומר אובייקטים.

ניצור מופע של המחלקה Student ונראה לחניכים שבאמצעות האופרטור '.' ניתן לגשת לכל השדות והמתודות של האובייקט



נציב ערכים בשדות של האובייקט ונדגיש שהשדות מהווים את המצב שלו.

נזכור שמה שמבדיל בין אובייקטים אלו השדות, אולם כיצד האובייקט מגיב ומה הוא עושה (התנהגות) זהה. לדוגמא הממוצע מחושב באופן זהה עבור כל אובייקט של Student, וגם ההדפסה נעשית בצורה דומה – המימוש של הפונקציות זהה.

```
int main()
{
	Student stud1;

	// first student info
	stud1.firstName = "Shahar";
	stud1.lastName = "Hasson";
	stud1.id = 123456789;
	stud1.grades[HISTORY_GRADE_IDX] = 78;
	stud1.grades[MATH_GRADE_IDX] = 81;
	stud1.grades[LITERATURE_GRADE_IDX] = 90;
	stud1.grades[ENGLISH_GRADE_IDX] = 65;

	Student stud2;

	// second student info
	stud2.firstName = "Beyonce";
	stud2.lastName = "";
	stud2.id = 111111111;
	stud2.grades[HISTORY_GRADE_IDX] = 95;
	stud2.grades[MATH_GRADE_IDX] = 87;
	stud2.grades[LITERATURE_GRADE_IDX] = 90;
	stud2.grades[ENGLISH_GRADE_IDX] = 98;

	return 0;
}
```

כעת נראה איך לעורר מתודות של אובייקט.

```
	stud1.print();
	cout << "The average grade is " << stud1.getAverage() << endl;

	cout << "\n\n";

	stud2.print();
	cout << "The average grade is " << stud2.getAverage() << endl;

	system("pause");
```

נשים לב שניגשים אל הפונקציה מתוך האובייקט עצמו.
ניזכר בבעיה שעלתה במצגת:
כיצד הקומפיילר מצליח לשמור עותק בודד של מתודה בזיכרון (ולא לשמור העתק של הקוד עבור כל אובייקט)?.
אם יש מתודה אחת בלבד בזיכרון, אז הקומפיילר צריך לדעת איזה אובייקט קרא לה כדי לגשת לשדות שלו.
הקומפיילר מאפשר זאת באמצעות הוספה של המצביע **this** לכל קריאה.
הטיפוס של this הוא מהסוג **מצביע למחלקה** (בדוגמא שלנו Student*), ובכל קריאה למתודה הקומפיילר דואג להעביר את הכתובת של האובייקט שקרא לה.

כלומר לכל מתודה שהגדרנו נוסף אוטומטית פרמטר חדש (אנחנו לא רואים את זה אבל הקומפיילר הוסיף אותו) והחתימה של המתודה נראית כך:

```
void print(Student* this) const;
```

שזה שקול לחתימה

```
void print() const;
```

ובmain הקומפיילר הופך את הקריאה

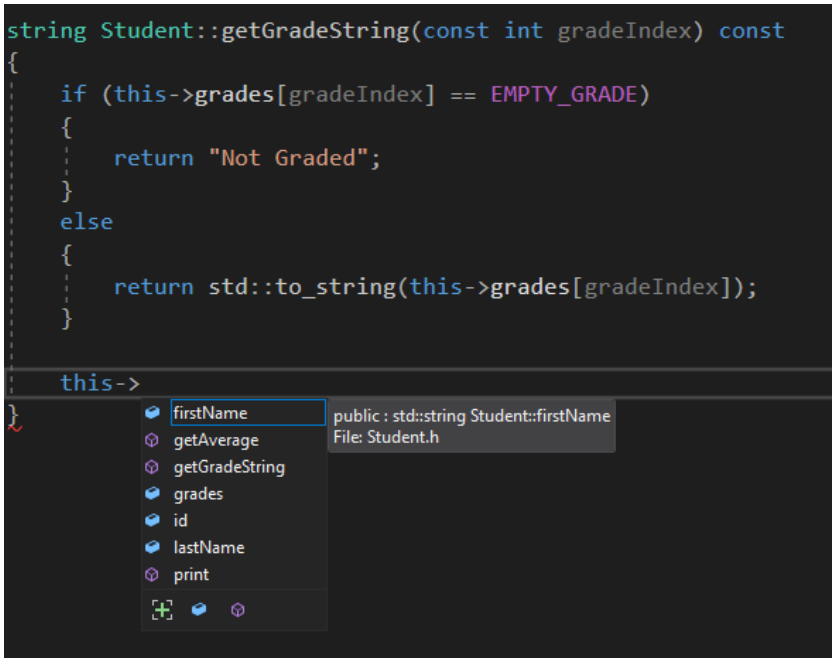
```
stud1.print();
```

לקריאה

```
stud1.print(&stud1);
```

ומעביר באופן מרומז את הכתובת של האובייקט שקרא לפונקציה, ושם יוכל להיעזר ב this בתוך הפונקציה כדי לגשת לשדות של האובייקט הספציפי שקרא לה.

נוכל לחזור לקובץ Student.cpp ולראות באחת הפונקציות שקיים מצביע בשם this ובו ניתן להשתמש כדי לגשת לשדות.



כעת נחזור למצגת בשביל ללמוד על כימוס.

4.4 הוספת כימוס למחלקה

אחרי שהסברנו על כימוס, נשנה את המחלקה כך שתיישם את העיקרון.
נעבור לקובץ ה h, נגדיר את שדות המחלקה כפרטיים, ואת מתודות הממשק של המחלקה כציבוריות.
את פונקציות העזר, אלו שלא חלק מהממשק שנחשוף החוצה נגדיר כפרטיות גם כן, משום שהן נועדו לשימוש פנימי של המחלקה ולא נועדו להיחשף החוצה.
נקפיד על הקונבנציה של קו תחתון לפני שם שדה פרטי

```
#pragma once
#include <string>

// grades array access
#define NUM_OF_GRADES 4
#define HISTORY_GRADE_IDX 0
#define MATH_GRADE_IDX 1
#define LITERATURE_GRADE_IDX 2
#define ENGLISH_GRADE_IDX 3

// initial grade value
#define EMPTY_GRADE -1

class Student
{
public:
	// methods
	double getAverage() const;
	void print() const;

private:
	// fields
	int _id;
	std::string _firstName;
	std::string _lastName;
	unsigned int _grades[NUM_OF_GRADES];

	//methods
	std::string getGradeString(const int gradeIndex) const;// helper method

};
```

מתחת למילה: public יוגדרו כל השדות/מתודות הציבוריים של המחלקה.

מתחת למילה: private יוגדרו כל השדות/מתודות הפרטיים של המחלקה.

נשמור על קונבנציה של קו תחתון לפני שם משתנה פרטי

הדבר הבא יהיה להגדיר את פונקציות ה Get של המחלקה (Getters)
האם הן צריכות להיות ציבוריות או פרטיות?
המתודות צריכות להיות ציבוריות על מנת שאפשר יהיה לגשת אליהן מבחוץ.

```
// getters
int getId() const;
std::string getFirstName() const;
std::string getLastName() const;
int getHeight() const;
```

Getters הן מתודות המוגדרות כ const, משום שהן לא משנות את מצב השדות.
המתודות מחזירות את ערכי השדות.
נשאר לנו לכתוב Getters לציונים.
האם לפי דעתכם/ן נכון להגדיר גישה למערך הציונים כלומר לחשוף את המצביע למערך?

```
unsigned int* getGrades() const;
```

במידה וחושפים כתובת של משתנה, אנו פותחים דרך לשנות אותו גם מחוץ לתחומי המחלקה.
משתמש אשר ביצע קריאה למתודה מקבל את המצביע למערך, ויכול לשנות את הערכים שבתוכו.
הנושא חשוב מכיוון שהוא מפתח אינטואיציה שתעזור לחניכים.

לכן חשוב לשים לב: מתי חשפנו מצביע? ומתי חשפנו ערך?

אנו נתמודד עם הבעיה באמצעות זה שנחשוף רק ציון ספציפי (תא במערך) באמצעות שליחת index.

כלומר חתימת הפונקציה תיראה כך:

```
unsigned int getGrade(const int grade_idx) const;
```

בצורה דומה נגדיר את מתודות ה Set (Setters) של המחלקה

```
// setters
void setId(const int newId);
void setFirstName(const std::string newFirstName);
void setLastName(const std::string newLastName);
void setHeight(const int newHeight);
void setGrade(const int grade_idx, const unsigned int new_grade);
```

שימו לב להוספת const לפני פרמטר שלא משתנה לאורך הפונקציה. פונקציות set לא צריכות להחזיר כלום.

הפונקציה set_grade תקבל את האינדקס של הציון, ואת הציון החדש.

למה לא הוספנו const אחרי החתימה של הפונקציה ?
התשובה היא שהמילה const תמנע מהמתודה אפשרות לשנות את ערך השדות = מצב האובייקט,
ופונקציות set בדיוק עושות את זה – לשנות את ערך השדות.
כלומר פונקציות set **לא** יכולות להיות מוגדרות כקבועות.

נממש את פונקציות ה get

```
int Student::getId() const
{
	return this->_id;
}

string Student::getFirstName() const
{
	return this->_firstName;
}

string Student::getLastName() const
{
	return this->_lastName;
}

unsigned int Student::getGrade(const int grade_idx) const
{
	if (grade_idx >= NUM_OF_GRADES or grade_idx < 0)
	{
		std::cerr << "grade index must be between 0 to 3" << endl;
	}

	return this->_grades[grade_idx];
}
```

בדיקת תקינות של הקלט

הדפסת הודעת שגיאה לstream ייעודי

לשים לב שגם שמממשים פונקציה מקפידים על אותה חתימה (כולל המילה const) ועל ה namespace.
שימו לב שהשתמשנו ב- this.

אחד היתרונות במתן גישה באמצעות פונקציות ציבוריות כמו get הוא שניתן לבצע בדיקת קלט – כל קלט שמכניסים לפונקציית הget יכול לעבור בדיקה שלנו.
בדוגמא שלנו דאגנו שלא יוכלו לשלוח אינדקס לא תקין של ציון במערך.

כדאי גם להתרגל לתעד את השגיאות של התכנית באמצעות stream ייעודי שנקרא std :: cerr אשר נועד לתיעוד שגיאות. (הדפסה אליו תדפיס גם את השגיאה למסך).

כעת נעבור למימוש מתודות ה Set.

```
void Student::setId(const int newId)
{
	this->_id = newId;
}

void Student::setFirstName(const std::string newFirstName)
{
	this->_firstName = newFirstName;
}

void Student::setLastName(const std::string newLastName)
{
	this->_lastName = newLastName;
}

void Student::setGrade(const int grade_idx, const unsigned int new_grade)
{
	if (grade_idx >= NUM_OF_GRADES or grade_idx < 0)
	{
		std::cerr << "grade index must be between 0 to 3" << endl;
	}
	if (new_grade < 0 || new_grade > 100)
	{
		// writes to cerr - a stream dedicated to error audit
		std::cerr << "grade must be between 0 to 100" << endl;
	}
	else
	{
		this->_grades[grade_idx] = new_grade;
	}

}
```

בדיקת תקינות של הקלט

הדפסת הודעת שגיאה
לstream ייעודי

בדוגמא שלנו דאגנו לבדוק קלט כך שלא יוכלו להכניס אינדקס לא תקין או ציונים לא תקינים (שליליים או גבוהים מ-100).

גם כאן הדפסנו שגיאה באמצעות std :: cerr (נועד לתיעוד שגיאות).

נדגים שימוש בפונקציות בקובץ ה main.

נמחק את הקוד הקודם, ונראה שלא ניתן לגשת למשתנים פרטיים של המחלקה. אם מנסים, מקבלים שגיאה שאומרת שהמשתנה לא נגיש (inaccessible)
את המתודות הפרטיות לא נוכל אפילו להשלים באמצעות ה intellisense (שמשלים את הקוד אוטומטית)

```
int main()
{
    Student stud1;

    stud1._id = 555;
```

int Student::_id
fields

member "Student::_id" (declared at line 40 of "c:\Users\Tomeriq\Documents\Visual Studio 2017\Projects\Magshimim\Develop\Unit2Demo\Unit2Demo\Student.h") is inaccessible

נעשה שימוש בפונקציות ה set כדי להציב ערכים בשני האובייקטים.

```
int main()
{
        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);
        stud1.print();

}
```

נוכל לראות שהערכים שהוכנסו באמצעות פונקציות הset מודפסים באמצעות הפונקציה.
נראה גם שהצלחנו לזהות הכנסה של קלט לא תקין.

4.5 הגדרת המחלקה ClassRoom

הדבר האחרון שנעשה הוא ליצור מחלקה שמייצגת כיתה.

בתוך הכיתה יהיה מערך של 10 אובייקטים מסוג Student, לצורך פשטות וכדי שלא נצטרך להיכנס לנושא הקצאות זיכרון (יהיה בשיעור הבא) נגדיר שסטודנט מתחיל עם תעודת זהות 0 (זה הערך ההתחלתי ששם הקומפיילר למשתנה ב ++C), ונאפס את כל תעודות הזהות ל0 לפני הכנסת הסטודנטים.

נפתח קובץ header חדש בשם ClassRoom.h
נגדיר את הממשק והשדות של המחלקה.

```
#pragma once

#include "Student.h"

#define MAX_STUDENTS_IN_CLASS 10

class ClassRoom
{
public:
	void init()
	void printStudents() const;
	void addStudent(const Student stud);
private:
	Student _studentsInClass[MAX_STUDENTS_IN_CLASS];
};
```

לא לשכוח לבצע #include למחלקת Student

הפונקציה מקבל אובייקט מסוג Student

הגדרת מערך שבו כל איבר הוא מסוג Student

בממשק המחלקה הגדרנו שתי פונקציות:

1. addStudent – מקבלת אובייקט מסוג Student ומוסיפה אותו למערך
2. printStudent – מדפיסה את כל התלמידים בכיתה

המחלקה מאוד פשוטה, אבל היא מדגימה שמחלקה אחת יכולה להחזיק אובייקטים מסוג של מחלקה אחרת. כיתה מורכבת מתלמידים, ולכן ברגע שהגדרנו מהו תלמיד, אפשר היה להגדיר מהי כיתה.

נממש את המחלקה:

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

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

void ClassRoom::init()
{
	for (int i = 0; i < MAX_STUDENTS_IN_CLASS; i++)
	{
		this->_studentsInClass[i].setId(INITIAL_ID_VALUE);	// resets all students id to zero
	}
}

void ClassRoom::addStudent(const Student stud)
{
	for (int i = 0; i < MAX_STUDENTS_IN_CLASS; i++)
	{
		if (this->_studentsInClass[i].getId() == 0) // finds the first empty place for new student
		{
			this->_studentsInClass[i] = stud;
			break;		// added student, breaks the loop
		}
	}
}

void ClassRoom::printStudents() const
{
	for (int i = 0; i < MAX_STUDENTS_IN_CLASS; i++)
	{
		if (this->_studentsInClass[i].getId() != 0)		// print only real students
		{
			cout << "Student number " << i << "info:" << endl;
			this->_studentsInClass[i].print();  // calls the print method in Student class
			cout << endl;
		}
	}
}
```

כדי להוסיף תלמיד חדש הנחנו שהערך ההתחלתי של שדה תעודת הזהות באובייקטים שבמערך הוא 0. לכן כדי למצוא מקום פנוי במערך אנחנו הולכים קדימה עד שפוגשים באובייקט שערך ה id שלו הוא 0.

לאחר מכן אנחנו מעתיקים את האובייקט שקיבלנו לתוך המערך במקום שמצאנו.

שאלה שיכולה להעלות היא מה אם לא קראנו לפונקציית האתחול? שאלה מצוינת, בשיעור הבא נסביר איך מתמודדים עם זה.