המרכז לחינוך סייבר
CYBER EDUCATION CENTER
מגשימים
תכנית הסייבר הלאומית

עקרונות מתקדמים בתכנות

תרגול 3
VECTOR

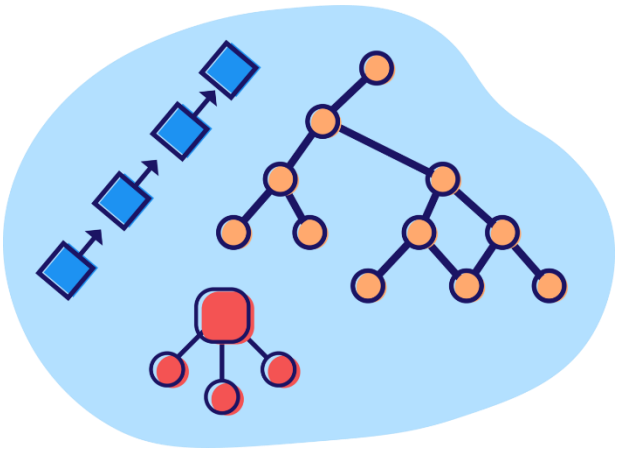
רקע

בתרגול מספר 1 השתמשנו במערך וברשימה מקושרת כדי לממש מחסנית ותור. ראינו שיש לשימוש במערך חסרון משמעותי על פני רשימה מקושרת – ברגע שהקצנו זכרון עבורו (הקצאה סטטית או הקצאה דינמית) הגודל שלו קבוע ולא ניתן לשינוי בקלות. מצד שני יש לו יתרונות במצבים מסוימים – כמו למשל האפשרות לגשת באופן ישיר לאיבר מסוים על פי האינדקס שלו.

מטרה

בתרגיל תממשו מבנה נתונים בשם וקטור (Vector) שיתנהג בצורה דומה למערך. הוא יאפשר גישה ישירה לאיבר על פי אינדקס, אך יחסוך לנו (למי שישתמש בו אחר כך) את ההתעסקות עם שינוי גודל המערך (כלומר זה יעשה מאחורי הקלעים).

עוד יתרון שיהיה לוקטור שלנו על פני מערך רגיל ב++C הוא שנדע בכל רגע נתון מה גודל המערך שלנו, וכך נוכל לזהות ולמנוע פניה לתא מחוץ לגבולות המערך (ובכך למנוע חולשת אבטחה מסוג buffer overflow)



אלה השלבים שתעברו:

בניית מחלקת Vector	מימוש הכנסה והוצאה	The Big 3	אופרטור הגישה	בונוס – תמיכה באופרטורים נוספים
• מימוש בנאי ומפרק • Getters/Setters	• מימוש push_back ו- pop_back • הגדלת קיבולת הווקטור	• מימוש בנאי העתקה • מימוש אופרטור השמה (=)	• הוספת תמיכה בגישה באמצעות אינדקס []	• אופרטורים -,+ =-,=+, • אופרטור הדפסה ל-stream <<

נתרגל מיומנויות חשובות:

- עבודה נכונה עם מחלקות
- מימוש נכון של העתקות בין אובייקטים
- ניהול זיכרון
- העמסת אופרטורים
- מימוש יעיל של מבנה נתונים

את התרגיל צריך להגיש ב-GIT: לינק להוראות שימוש ב-GIT.

כדאי לקרוא גם דגשים לתכנות נכון.

"Practice Makes Perfect…"

בהצלחה יא אלופות ואלופים!

שלב 1: יצירת מחלקת Vector

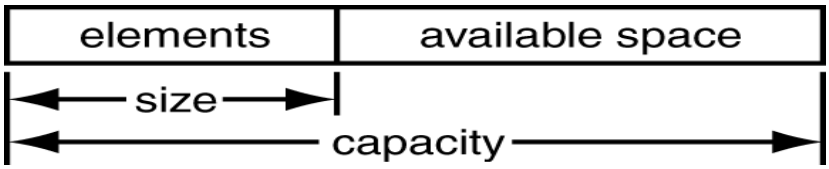
שלב 1 יצירת מחלקת Vector	שלב 2 הוספה והוצאה של איברי הווקטור	שלב 3 The Big 3	שלב 4 אופרטור הגישה	שלב בונוס עוד אופרטורים...

ממשו את המחלקה Vector – וקטור של **מספרים שלמים**. המחלקה צריכה להכיל את השדות הבאים:

מצביע לתחילת המערך	`int *_elements`
גודל המערך , כמות האיברים הנגישים במערך	`int _size`
קיבולת המערך, כמות התאים המוקצים בזכרון (נגישים+לא נגישים)	`int _capacity`
פאקטור הגדילה- הגודל ההתחלתי של המערך אותו מקצים, וגם בכמה מגדילים את הקיבולת בכל פעם שהוקטור מתמלא	`int _resizeFactor`

* שימו לב!

$_size \; != \; _capacity$



* הטענה הבאה תמיד נכונה:

$_size \; <= \; _capacity$

שמירה של יותר קיבולת מהגודל שאנחנו צריכים בפועל מאפשרת לנו את הגמישות של הרחבת הוקטור. ברגע שהוקטור מלא ($size \; = \; capacity$) נצטרך להגדיל את הקיבולת שלו (על ידי הקצאת מקום חדש בזכרון, העתקת האיברים למקום החדש, ושינוי המצביע בהתאם), אבל זה יקרה רק בסעיף ג' .

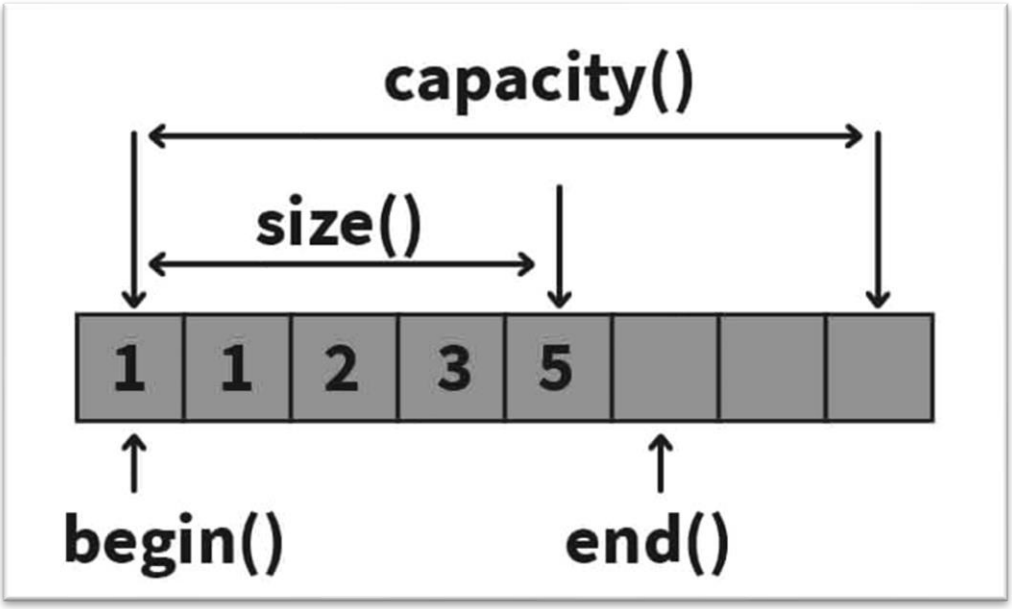
שלב 1: הוספת מתודות בסיסיות ל-Vector

שלב 1 יצירת מחלקת Vector	שלב 2 הוספה והוצאה של איברי הווקטור	שלב 3 The Big 3	שלב 4 אופרטור הגישה	שלב בונוס עוד אופרטורים...

בנוסף ממשו את הפונקציות הבאות:

בנאי המקבל את הגודל ההתחלתי של הוקטור. הבנאי מקצה דינאמית מערך בגודל הנתון. בארגומנט שנקבל כאן נשתמש גם כדי להקצות עוד קיבולת בכל פעם שהוקטור יתמלא. אם ערך הפרמטר קטן מ-2 אז הבנאי יתייחס כאילו הוא קיבל את הערך 2 כפרמטר.	`Vector( int n)`
מפרק אשר משחרר זכרון שהוקצה דינאמית	`~Vector()`
מחזירה את גודל הוקטור.	`int size()`
מחזירה את קיבלות הוקטור.	`int capacity()`
מחזירה את פקטור הגדילה.	`int resizeFactor()`
מחזירה האם הוקטור ריק (אין איבר נגיש)	`bool empty()`

בסיום החלק הראשון סיימנו עם ההגדרה של המחלקה והממשק הבסיסי שלה, בסעיפים הבאים נממש פונקציות עם התנהגות מורכבת יותר כמו הוספת איברים נגישים, והגדלת הקיבולת.



שלב 2: הוספה והוצאה של אלמנטים ב-Vector

שלב 1 יצירת מחלקת Vector	**שלב 2** **הוספה והוצאה של איברי הווקטור**	שלב 3 The Big 3	שלב 4 אופרטור הגישה	שלב בונוס עוד אופרטורים...

הוסיפו לממשק המחלקה את הפונקציות הבאות:

מוסיפה איבר בסוף הוקטור (סוף = אחרי האיבר הנגיש האחרון)	`void push_back(const int& val)`
מוחקת את האיבר הנגיש האחרון בוקטור ומחזירה אותו. במידה והוקטור ריק, מדפיסה: "`error: pop from empty vector`" ומחזירה **-9999** .	`int pop_back()`
מוודאת שקיבולת הוקטור תהייה לפחות n. משנה את קיבולת הוקטור ע"י הקצאת זכרון במידה וצריך (על פי פאקטור הגדילה).	`void reserve( int n)`

הערה לגבי push_back:

כל עוד $size < capacity$, נוכל פשוט להפוך איבר לא נגיש לאיבר נגיש. אבל מה קורה כאשר הוקטור מלא, כלומר כאשר $size = capacity$?

במקרה זה אנחנו נקצה מערך גדול יותר (בגודל $capacity + resizeFactor$) ונעתיק את כל איברי המערך למערך החדש. (לא לשכוח לשחרר זכרון דינאמי שכבר אינו בשימוש, ולעדכן מצביעים במידת הצורך, וכמובן לעדכן את `capacity`)

הערה לגבי reserve:

יתכן שיהיה צורך להגדיל את ה-$capacity$ בכפולה של $resizeFactor$.

אם למשל כרגע $capacity = 10$ ו- $resizeFactor = 5$, וקראו לפעולה כך: $reserve(23)$

יש להקצות עוד $resizeFactor * 3$ מקומות. משום שאם נגדיל בפחות מזה,הקיבולת עדיין תהיה קטנה מ-23.

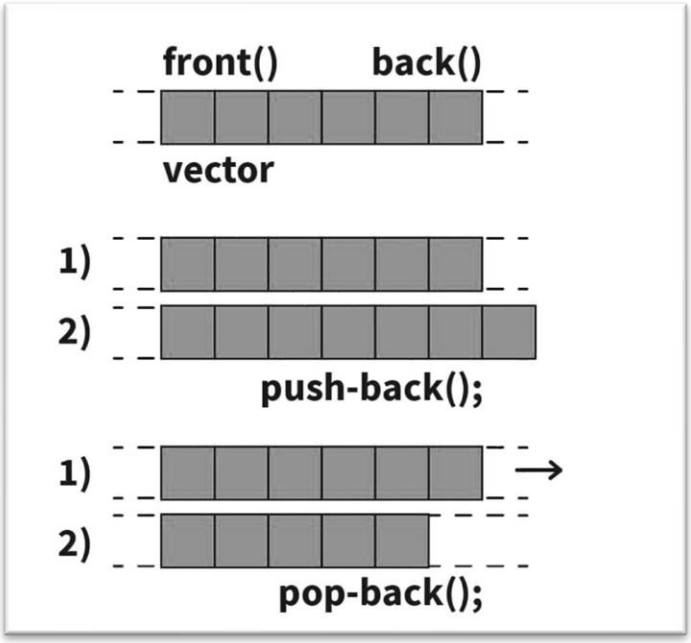
שלב 2: הגדלת קיבולת ה-Vector

שלב בונוס	שלב 4	שלב 3	שלב 2	שלב 1
עוד אופרטורים...	אופרטור הגישה	The Big 3	הוספה והוצאה של איברי הווקטור	יצירת מחלקת Vector

הוסיפו גם את שתי הפונקציות הבאות לשינוי גודל המערך ($size$)

משנה את גודל המערך ל n. • אם n קטן או שווה ל-$capacity$, משנה את הגודל ל-n (ולמעשה מוחקת/מוסיפה איברים). • אם n גדול מ-$capacity$, משנה את הקיבולת של הוקטור (כך ש-n לא יהיה גדול מ-$capacity$. הגדילה צריכה להיות לפי פקטור הגדילה. יש לנו כבר פונקציה שמגדילה...) ומשנה את הגודל להיות n (כלומר מוסיפה איברים).	`void resize( int n )`
מציבה ערך בכל איברי הוקטור הנגישים	`void assign( int val )`
אותו הדבר, הפעם הפונקציה גם מציבה את val בכל איבר חדש שהיא מוסיפה.	`void resize( int n, const int& val )`

אם לא ברור מהו ההבדל בין $resize$ ל-$reserve$ תוכלו לחפש באינטרנט.



שלב 3: The Big 3

שלב 1 יצירת מחלקת Vector	שלב 2 הוספה והוצאה של איברי הווקטור	שלב 3 The Big 3	שלב 4 אופרטור הגישה	שלב בונוס עוד אופרטורים...

שימו 💙

בשביל לבצע את השלבים הבאים תצטרכו ללמוד על העמסת אופרטורים.

ניתן לגשת אל תיקיית התוכן של השיעור ולצפות בסרטון שממשיך את ההדגמה שהייתה בביתה

https://drive.google.com/drive/u/0/folders/1sc_ExCug_UwFxnqDqm6tnPmkt13U3h8P

The big three - היזכרו מה הם שלושת הפונקציות/אופרטורים המכונים כך. האם יש צורך לממש אותם בעצמנו במחלקה Vector ? אם כן, עשו זאת. אם לא, הסבירו למה.

☝ טיפים חשובים

- במידה והחלטתם/ן לממש את פונקציות ה-big 3 הקפידו:
  - לדייק בחתימה של המתודות (להקפיד על `const` ו-ref איפה שצריך)
  - לא לחזור על קוד שכבר כתבתם/ן
  - הימנעו מדליפות זיכרון – על כל `new` שעשינו צריך להיות `delete` מתאים.

לקריאה נוספת

https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)

שלב 4: אופרטור הגישה

שלב 1 יצירת מחלקת Vector	שלב 2 הוספה והוצאה של איברי הווקטור	שלב 3 The Big 3	שלב 4 אופרטור הגישה	שלב בונוס עוד אופרטורים...

נרצה להשתמש במחלקה כמו שהיינו משתמשים במערך, כלומר לגשת ישירות לאינדקס מסוים בעזרת האופרטור [ ] העמיסו את האופרטור [ ] במחלקה Vector, כך שעבור אובייקט בשם v, כאשר נבצע $v[n]$ יוחזר רפרנס לאיבר במקום ה- n.

במידה ו-n חורג מהגבולות של הוקטור יש להדפיס הודעה מתאימה ולהחזיר את האיבר הראשון.

זוהי אינה הדרך האידאלית לטיפול במקרה כזה, אבל עדין לא למדנו איך לטפל בשגיאות כמו שצריך.

☝ טיפים חשובים
• **בדיקת קלט** – וודאו שלא מתבצעת גישה עם אינדקס לא חוקי • **הדפסת שגיאות** – את ההדפסות יש לבצע אל ה-stream הייעודי `cerr` ולא `cout` • **בדקו את עצמכם/ן** – וודאו שניתן לגשת לאיברי הוקטור בדומה למערך רגיל.

שימו 💙

מצורף קובץ `Vector.h` המגדיר את הממשק כפי שנדרש בתרגיל. אפשר להוסיף פונקציות או שדות עזר אך לדעתנו אין צורך.

אסור(!!) לשנות את שם המחלקה ואת הממשק שלה. כלומר, אין לשנות ואת החתימות של הפונקציות הנתונות.

שלב בונוס: אופרטורים נוספים

שלב 1	שלב 2	שלב 3	שלב 4	שלב בונוס
יצירת מחלקת Vector	הוספה והוצאה של איברי הווקטור	The Big 3	אופרטור הגישה	עוד אופרטורים...

תמיכה באופרטורים נוספים

נרצה להוסיף אפשרות של חיבור וחיסור בין שני ווקטורים.

הוסיפו את החתימות, ודרסו אופרטורים כרצונכם כך שבהינתן שני ווקטורים v_1, v_2 התכנית תתמוך במקרים הבאים:

נניח שמתקיים: $v_1 = \{1, 2, 3, 4\}, v_2 = \{10, 20, 30, 40\}$.

לאחר ביצוע השורה **v3 = v1 + v2** הווקטור **v3** יכיל **{11,22,33,44}**.

לאחר ביצוע השורה **v3 = v2 – v1** הווקטור **v3** יכיל **{9,18,27,36}**

לאחר ביצוע השורה **v1 -= v2** הווקטור **v1** יכיל **{-9,-18,-27,-36}**, הווקטור **v2** לא משתנה

לאחר ביצוע השורה **v1 += v2** הווקטור **v1** יכיל **{11,22,33,44}**, הווקטור **v2** לא משתנה.

תמיכה באופרטורים ההכנסה ל-stream

דרסו את אופרטור ההכנסה ל stream **(**stream insertion**)** אשר יאפשר הדפסה נוחה של הווקטורים שבנינו.

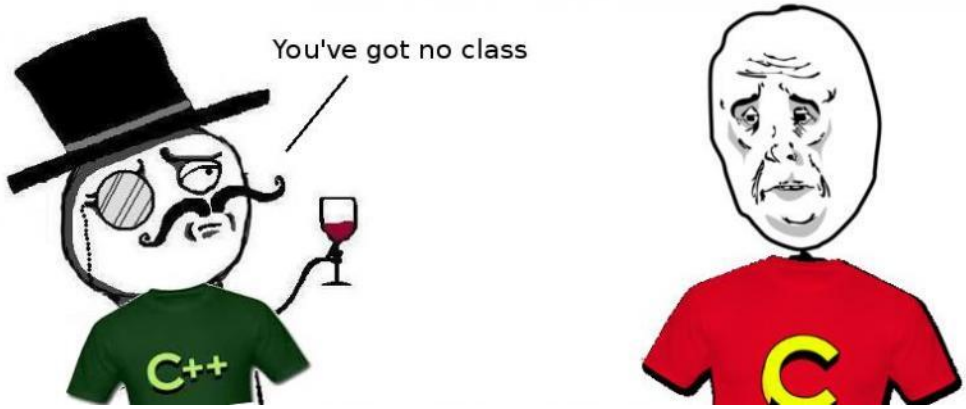
תוכלו לדעת שביצעתם/ן את המשימה כנדרש אם התכנית תתמוך בהדפסה של הווקטור באמצעות **std::cout**

כלומר בהינתן ווקטור $v1$ התכנית תתמוך בשורה **std::cout << v1** ותדפיס את תכולת הווקטור בצורה הבאה:

```
Vector Info:\nCapacity is {capacity}\nSize is {size} data is {val1, val2, val3,…,valn}\n
```

* כדי להצהיר על חתימת אופרטור >> תצטרכו לקרוא קצת על פונקציות **friend**.

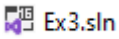
(היעזרו בקישור הבא)



נספחים

הגשה ב-GIT

- את הפרויקט יש לנהל ב-Git, לפתוח repository חדש בתוך קבוצת ה-gitlab שלנו ושל המדריך/ה, ולהגיש לינק לפרויקט ב-NEO (אפשר לעשות comment עם הלינק או להגיש מסמך txt עם הלינק בפנים).
- יש להעלות ל-repository את כל הקבצים הרלבנטיים לתרגיל (קבצי txt, מסמכים, ומשאבים אחרים שבהם השתמשנו).
  חשוב להעלות את פרויקט ה-Visual Studio השלם ולהתעלם מקבצים לא נחוצים (הנחיות במסמך הבא), במידה ולא הועלה הפרויקט השלם, אין להעלות את שאר הקבצים שיוצר Visual Studio – הם רבים מאוד, הם לא מכילים מידע נחוץ להרצת הפרויקט אצל המדריך, ורק יוצרים בלגן.
- הבחירה אילו קבצים להעלות ל-repository נעשית באמצעות הפקודות add ו-rm. אופציה נוספת (מומלצת) היא להוסיף קובץ gitignore. אשר יתעלם מהקבצים הלא נחוצים. במידה ותרצו תוכלו להיעזר בסרטוני עזר בנושא GIT.
- כסיימתם/ן, בדקו שניתן להריץ את הפרויקט בקלות – בצעו Clone אל תיקייה במחשב אשר שונה מזו שעבדתם/ן, ותראו שהפרויקט נפתח ע"י לחיצה על קובץ ה-sln ויכול לרוץ בלי בעיה



דגשים:

- את הפרויקט יש לפתוח בקבוצת ה-gitlab שאליה משותף/ת המדריך/ה כ-Maintainer.
- יש לוודא שכל הקבצים הרלבנטיים נוספו ל-repository (באמצעות הפקודה add), במידת הצורך ניתן להוריד קבצים מיותרים (באמצעות הפקודה rm)
- יש לבצע commit עבור כל סעיף, ובנקודות שבהן הוספנו שינויים חשובים (לפי הדגשים שהועברו בכיתה).
- עבור כל commit, זכרו לכתוב הודעה קצרה ואינפורמטיבית, שאפשר יהיה להבין מה היה השינוי בקוד.
- יש לדחוף את הקוד (באמצעות הפקודה push) ל-repository בסיום העבודה שלנו, חשוב שבסיום העבודה שלנו, ובמידה ונפנה למדריך/ה, ב-repository יהיה הקוד המעודכן ביותר.
- במידה ושכחנו או שאנחנו לא בטוחים איך מעלים קובץ, או מתעלמים מקבצים, כדאי לצפות בסרטוני ההדרכה בנושאי GIT. ניתן לגשת לסרטונים בלשונית ה-resources שבכיתת ה-NEO.
- בסיום העבודה יש להגיש לכיתת ה-NEO קישור ל-repository.

כללי

1. יש לבדוק שכל **המטלות מתקמפלות ורצות ב-**VS2019. מטלה שלא תעבור קומפילציה אצל הבודק לא תיבדק **והניקוד שלה יהיה 0** ☹
2. יש לבדוק שהקוד שכתבתם עובד. יש להריץ בדיקות שלכם ולוודא שהקוד ברמה טובה.
3. כאשר אתם מתבקשים לממש פונקציה, ממשו בדיוק את הנדרש. אין להוסיף הדפסות וכדו'. אם הוספתם תוך כדי הבדיקות שלכם הדפסות, אנא דאגו להוריד אותם לפני ההגשה.
4. להזכירכם! העבודה היא עצמית, ואין לעשות אותה ביחד.
5. על כל שאלה או בעיה יש לפנות למדריך, לפחות 36 שעות לפני מועד ההגשה.

דגשים לתכנות נכון

- כדאי לקמפל כל מספר שורות קוד ולא לחכות לסוף! הרבה יותר קל לתקן כאשר אין הרבה שגיאות קומפילציה. בנוסף קל יותר להבין מאיפה השגיאות נובעות.
- כדאי לכתוב פונקציה ולבדוק אותה לפני שאתם ממשיכים לפונקציה הבאה. כלומר, כתבו תכנית ראשית שמשתמשת בפונקציה ובודקת האם היא עובדת כראוי. חישבו על מקרי קצה ונסו לראות מה קורה.
- בכל פעם שאתם מתקנים משהו, זכרו שיכול להיות שפגעתם במשהו אחר. לכן עליכם לבדוק שוב מהתחלה.
- חשפו החוצה רק את הממשק המינימלי הדרוש (minimal API), הגדירו את שדות המחלקה כפרטיים, וכמה שפחות מתודות כציבוריות.