האוניברסיטה העברית בירושלים

בית הספר להנדסה ולמדעי המחשב ע"ש רחל וסלים בנין

סדנת תכנות בשפת C ו־C++ (67315)
תרגיל 6

תאריך הגשה: ה־29 ליוני, 2021, בשעה 23:59

1 רקע

בתרגיל זה תדרשו לעשות שימוש בכלים שרכשתם במהלך הקורס, כדי לממש container חדש ויעיל במיוחד. ניזכר ב־container, שסביר כי הוא המוכר ביותר לכם, `std::vector<T>`. נרצה לממש container הזהה לו לחלוטין מבחינת התנהגות, אך חסכוני יותר בזמני ריצה. קראו היטב את ההוראות המופיעות לאורך המסמך, בפרט לאלו הנוגעות לטיפוסים מ־STL בהם מותר (או, אסור), להשתמש בתרגיל זה, כמו גם לנושאי יעילות.

2 זיכרון סטטי וזיכרון דינמי

2.1 היתרונות והקשים של ניהול זיכרון ב־Stack וב־Heap

במהלך הקורס למדנו מהן הדרכים בהם נוכל לשמור בזיכרון ערכים ומבני נתונים. בפרט, דיברנו על שני מקטעים רלבנטים – ה־stack וה־heap. בפרט, ראינו כי:

- **זיכרון סטטי (שימוש ב־Stack):** ראינו שהזיכרון ב־stack זמין לנו "כברירת מחדל" בכל פונקציה, כשלכל פונקציה ה־stack ששייך לה. כמו כן, ראינו כי מדובר ב־"זיכרון לזמן קצר", שכן בעת יציאה מהפונקציה זיכרון זה משוחרר באופן אוטומטי.

- **זיכרון דינמי (שימוש ב־Heap)**: ראינו שהזיכרון ב־heap עומד לרשותנו רק כשנבקש זאת במפורש, באמצעות בקשה להקצאת זיכרון דינמי. כמו כן, ראינו כי להבדיל משימוש בזיכרון הסטטי, הזיכרון הדינמי אינו "קיים לזמן קצר בלבד", אלא קיים עד אשר נבקש ממערכת ההפעלה לשחררו באופן מפורש (ולא – ניצור דליפת זיכרון).

הבחירה באיזו דרך להשתמש – בזיכרון סטטי או דינמי, תלויה בסיטואציה הניצבת לפנינו, ולכל כלי יש את יתרונותיו וחסרונותיו. נציג כמה מהם:

- **שימוש בזיכרון סטטי:** מצד אחד, הגישה ל־stack מהירה באופן משמעותי מגישה ל־heap, ולכן ניתן לו עדיפות. מצד שני, הזיכרון שמוקצה ב־stack זמין, כאמור "לזמן קצר" בלבד. כלומר, עת ששמורה (scope) כלשהיא מסיימת את פעולתה, המשתנים שהוגדרו עבורה ב־stack משוחררים אוטומטית – וגישה אליהם מעתה ואילך תחשב

כקריאה בלתי חוקית. כמו כן, ל־stack גודל מקסימלי, שלא ניתן לחצות. למשל, כברירת מחדל, גודל המחסנית במחשבים המשתמשים ב־Windows כמערכת הפעלה הוא 1MB. ברורה, אפוא, המגבלה שבשימוש ב־stack לאחסון מידע רב.

- **שימוש בזיכרון:** מצד אחד, זיכרון דינמי מקנה לנו גמישות שכן אנו יכולים לבקש כמות גדולה הרבה יותר של זיכרון (בניגוד ל־stack, שכאמור מוגבל). ראינו שאפשר לנצל ייתרון זה בייחוד במקרים בהם איננו יודעים מהו גודל הקלט. אלא שמנגד, מאחר שמדובר ב־"זיכרון לזמן ארוך", על התוכנה לנהל את הזיכרון שאותו ביקשה ממערכת ההפעלה – וכידוע, אילו זו שוכחת לשחרר זיכרון שהקצתה, היא מביאה לכך שנוצרת דליפת זיכרון בתוכנית. יתרה מכך, כאמור לעיל, ניהול ה־heap, ובפרט הגישה לפריטים המאוחסנים בה, "מורכבת יותר". מכאן שגישה לזיכרון המאוחסן ב־heap תהא איטית יותר ותביא לכך שזמן הריצה של התוכנית שלנו יארך.

אנו נוכחים לראות כי לכל כלי הייתרונות והחסרונות שלו – ולנו האחריות להשתמש בכלים העומדים לרשותינו בתבונה.
עתה, נדבר באופן ספציפי על בעיה אחת שנתקלנו בה לאורך כל הקורס: שימוש במערכים ב־C ו־++C. עוד בתחילת הקורס ראינו כי מערך הוא למעשה קטע זיכרון **רציף** באורך n־פעמים טיפוס הנתונים המבוקש. כמו כן, ראינו כי כדי ליצור מערך עלינו לקבוע מה יהיה גודלו **בזמן קומפילציה**. כלומר, לא נוכל, למשל, לבסס את גודל המערך על קלט שקיבלנו מהמשתמש – שכן אורך המערך חייב להיות זמין למהדר עוד בזמן קומפילציה.
במצב זה, עמדו לפנינו האפשריות הבאות:

- **אם מדובר בגודל קבוע וידוע מראש**: ניתן להקצות את המערך על ה־stack. אלא שמעבר לחסרונות שבהקצאה על ה־stack שהזכרנו לעיל, הרי החיסרון המרכזי ברור ובולט ביותר: אנו חייבים לדעת מהו הגודל המקסימלי של הקלט כדי ששיטה זו תצלח. וודאי, ניתן לבחור במספר גדול מאוד (למשל $\mathbb{N} \ni n = 10000$), אך תמיד נמצא קלט בגודל $n+1$ שאיתו התוכנית שכתבנו לא תוכל להתמודד. במילים אחרות, כל עוד הקלט לא חסום מלעיל, מדובר בשיטה בעייתית.
נזכיר בהקשר הזה כי עוד בחלק של הקורס העוסק בשפת C, למדנו על המונח Variable Length Array (VLA). ראינו ש־VLA הוא מערך בגודל שאינו קבוע (כלומר, לא נקבע בזמן קומפילציה, אלא בזמן ריצה) ומוקצה על ה־stack. אלא שלאור הבעיתיות המובנית שבכלי זה (ההקצאה על ה־stack, שמוגבל מאוד מבחינת זיכרון) – השימוש שלו אינו מומלץ כלל ואף אסרנו את השימוש ב־VLA במסגרת קורס זה.

- **אם הגודל אינו ידוע מראש או שאינו קבוע:** נוכל לעשות שימוש בזיכרון דינמי. אלא ששימוש זה מביא עמו את החסרונות שהזכרנו באשר לשימוש ב־heap.

שתי האופציות הללו אינן ממצות את כל סט הכלים שהיה לנו, אך אלו האפשרויות המרכזיות שבהן נתקלנו. בתרגיל זה נממש מבנה נתונים המתנהג כמו ווקטור, אך מממש "מאחורי הקלעים" שיטה יעילה לניהול זיכרון, המנצלת את יתרונותיהם של ה־stack וה־heap וממזערת את חסרונותיהם – ובכך ננסה "ליהנות משני העולמות".

2.2 הגדרת טיפוס הנתונים Variable Length Vector

נגדיר את ה־container "וקטור באורך משתנה" (Variable Length Vector, או בקצרה VLVector) להיות טיפוס נתונים גנרי, הפועל על אלמנטים מסוג T ובעל קיבולת סטטית

$C \in \mathbb{N} \cup \{0\}$. ל־VLVector יהיה API[1] דומה לזה של `std::vector`, אך הייחודיות שלו ביחס לווקטור "רגיל", היא בכך שהוא עושה שימוש גם ב־stack וגם ב־heap לאחסון ערכיו.

2.3 אחסון סטטי (ב־stack) אל מול אחסון דינמי (ב־heap)

VLVector יפעל באמצעות האלגוריתם הנאיבי הבא על מנת "לתמרן" ביעילות בין שימוש ב־stack וב־heap:

- <u>הצהרה:</u> VLVector יהיה טיפוס גנרי (עם כותרת template) ויקבל שני פרמטרים גנרים: את טיפוס הנתונים שאותם הוא מאחסן, T, ואת $C.C \in \mathbb{N} \cup \{0\}$ מסמן כמה איברים, לכל היותר, יוכל הוקטור להכיל באופן סטטי (על ה־stack). לפיכך, ומטעמי יעילות, נדרוש כי כל מופע של VLVector יתפוס C ערכים **בדיוק** ב־stack.

- <u>הוספת איבר(ים) לווקטור:</u> יהי $size \in \mathbb{N} \cup \{0\}$ כמות האיברים הנוכחית בווקטור (לפני פעולת ההוספה), ו־$k \in \mathbb{N}$ כמות האיברים שנרצה להוסיף בפעולת ההוספה.[2]

  - **אם** $size + k \leq C$**:** במקרה הזה הוספת k איברים לא **תחצה** את C (ה־threshold), לכן k הערכים החדשים ישמרו ב־stack.
  - **אם** $size \leq C \wedge size + k > C$**:** במקרה הזה כמות האיברים הנוכחית, $size$, קטנה מ־C, אלא שבתוספת k האיברים החדשים – נחצה את C. במקרה שכזה, לא נוכל לשמור את כל $size + k$ האיברים בזיכרון הסטטי, ולכן הווקטור יפסיק **באופן גורף** להשתמש בזיכרון סטטי ויעבור להשתמש בזיכרון דינמי. כדי לעשות זאת, הווקטור יקצה את כמות הזיכרון הנדרשת, כמפורט בחלק הבא, **ויעתיק אליו את כל הערכים שעד כה נשמרו על ה־stack** (**לא ניתן** להימנע מהעתקה, ראו את הנספח לפירוט). כלומר, כל $size + k$ האיברים – ישמרו בזיכרון הדינמי.
  - **אם** $size > C$**:** במקרה כזה ברור שבפרט גם $size + k > C$ – ולכן נמשיך להשתמש בזיכרון הדינמי. הזיכרון הדינמי יגדל בהתאם להסבר שמופיע בפרק הבא.

- <u>הסרת איבר(ים) מהווקטור:</u> יהי $size \in \mathbb{N}$ כמות האיברים הנוכחית בווקטור (לפני פעולת ההסרה), ו־$k \in \mathbb{N} \;\; S.t. \;\; k \leq size$ כמות האיברים שנרצה להסיר.

  - **אם** $size - k > C$**:** במקרה זה הסרת k האיברים לא תגיע ל־C, לכן נמחק את האיברים מה־heap, בהתאם להסבר שיופיע בפרק הבא, ובכל מקרה – נמשיך להשתמש בזיכרון הדינמי (כלומר הווקטור ימשיך להחזיק את כל ערכיו ב־heap).
  - **אם** $size - k \leq C$**:** במקרה זה נוכל לחזור ולהשתמש בזיכרון הסטטי – שזו שיטת האחסון העדיפה – לכן נעתיק חזרה את הערכים ל־stack, נמחק את הזיכרון הדינמי במלואו ונחזור להשתמש בזיכרון הסטטי (בלבד).

---

[1] תזכורת: API (Application Programming Interface) הוא מונח המתייחס, בעניינו, לרשימת הפעולות **הפומביות** של האוביקט, שאליהן ניתן לגשת. ראו: https://bit.ly/39LxnQt.

[2] ראינו שב־ `std::vector` ניתן לקרוא לפעולת `insert` כך שתוסיף איבר יחיד או מספר איברים (בעזרת iterator).

2.4 קיבולת הווקטור

כאמור, לווקטור שלנו, כמו גם ל־`std::vector`, תהיה פונקציית קיבולת, שתתאר את כמות האיברים המקסימלית שהווקטור יכול להכיל בכל רגע נתון.
נגדיר את $cap_C : \mathbb{N}\cup\{0\}\times\mathbb{N}\to\mathbb{N}$ להיות פונקציית הקיבולת, כך שבהינתן $C\in\mathbb{N}\cup\{0\}$ – קבוע המייצג את הזיכרון הסטטי המקסימלי של הווקטור, הפונקציה תקבל 2 ארגומנטים: $size\in\mathbb{N}\cup\{0\}$ – כמות האיברים **הנוכחית** בווקטור (לפני הוספה / הסרה של איברים) ו־$k\in\mathbb{N}$ כמות האיברים שנרצה להוסיף או להסיר. cap_C תחזיר את הקיבולת המקסימלית של הווקטור.
לנגד עיננו שתי מטרות: מצד אחד, נרצה לשמור על זמני ריצה טובים ככול האפשר. כך, נרצה שפעולות הגישה לווקטור, ההוספה לסוף הווקטור והסרת האיבר שבסוף הווקטור יפעלו כולן ב־$O(1)$ (לשיעורין). מהצד השני, לא נרצה להקצות **יותר מדי** מקום, שיתבזבז לשווא.
כאשר מדובר בזיכרון **סטטי** ($size+k\le C$), זה קל – הקיבולת של הווקטור היא C. תמיד.
עם זאת, מה תהיה קיבולת הווקטור כשהוא חוצה את C ועובר להשתמש בזיכרון דינמי? ניסיון נאיבי יהיה להגדיל את הווקטור כל פעם ב־k איברים. לדוגמה, בכל פעם שמוסיפים איבר חדש יחיד ($k=1$), נקצה את כל הווקטור מחדש עם $(size+1)\cdot sizeof(T)$ בייטים ונעתיק לתוכו את איבריו של הווקטור הישן. אלא, שראינו בקורס (וב־DAST) שגישה זו מובילה לזמן ריצה של $O(n)$ ולכן אינה מתאימה (הוכחה מתמטית מופיעה בנספח לתרגיל).
להלן הפתרון: נגדיר את פונקצית הקיבולת $cap_C : \mathbb{N}\cup\{0\}\times\mathbb{N}\to\mathbb{N}$, עבור $size\in\mathbb{N}\cup\{0\}$, כמות האיברים **הנוכחית** בווקטור, $k\in\mathbb{N}$ כמות האיברים שנרצה להוסיף לווקטור ו־$C\in\mathbb{N}\cup\{0\}$, קבוע הזיכרון הסטטי המקסימלי, כך:

$$cap_C\left(size,\,k\right)=\begin{cases}C & size+k\le C\\ \left\lfloor\frac{3\cdot(size+k)}{2}\right\rfloor & otherwise\end{cases}$$

הנימוקים בבסיס הגדרה זו של cap_C מופיעים בנספח המצורף לתרגיל זה. **רצוי <u>מאוד</u> שתעיינו בו ותבינו אותו.**
נעשה שימוש ב־cap_C כך:

- <u>הוספת איברים חדשים:</u> פונקצית הקיבולת תופעל אך ורק כשאין בווקטור מספיק מקום כדי להכיל את k האיברים החדשים (כלומר, אם יש מקום בזיכרון הסטטי / הדינמי לאחסון $size+k$ איברים, אין טעם להפעיל את cap_C. כפועל יוצא, במקרה שבו אנו עובדים עם זיכרון דינמי – לא נגדיל את הווקטור).

- <u>הסרת איברים:</u>

  – <u>במקרה שבו $size-k>C$:</u> מטעמי יעילות, לא נקטין את קיבולת הווקטור. כלומר, גם כאשר חישוב cap_C היה מביא לערך קיבולת אחר, קטן יותר, עדיין נבחר שלא להקטין את הווקטור כלל (ולכן אין טעם לחשב את cap_C במקרה זה).

  – <u>במקרה שבו $size>C\ \wedge\ size-k\le C$:</u> במקרה זה נעתיק את כל האיברים לזיכרון הסטטי ולאחר מכן נמחק כליל את הזיכרון הדינמי.

אם כן, לסיכום:

- יש להשתמש בזיכרון סטטי **כל עוד** כמות האיברים בווקטור **אינה חוצה** את C.

- יש להשתמש בזיכרון דינמי **כל עוד** כמות האיברים **חצתה** את C.
- יש לתמוך במעבר מזיכרון סטטי לזיכרון דינמי, **ולהיפך**.
- יש לעמוד בחסם של $O(1)$ לשיעורין לפעולות הגישה וההוספה/ההסרה לסוף/מסוף הווקטור.
- זיכרו שאין מנוס מהעתקת האיברים בכל הגדלה או הקטנה של הווקטור.
- נחשב את cap_C אך ורק כשנידרש להגדיל או להקטין את הווקטור.
  עם זאת, נזכיר: כאשר עובדים עם זיכרון דינמי, קיבולת הווקטור יכולה **רק** לגדול. לא נקטין את הווקטור כשנשתמש בזיכרון דינמי.
- **שימו לב**: כשמסירים איברים עד כדי הגעה לערך הקטן או השווה ל־C (כלומר כאשר $size - k \leq C$) חוזרים לעשות שימוש בזיכרון הסטטי, **וה־**$capacity$ **הדינמי "מתאפס"**. כך, כשנגדיל שוב את הווקטור עד אשר $size+k > C$ לא נחזור להשתמש ב־$capacity$ הישן, אלא נחשב את הערך הנכון מחדש, באמצעות cap_C.
- **נדגיש**: המימוש שלכם **חייב** להשתמש ב־cap_C. ציונו של מימוש שיגדיל או יקטין את הווקטור בצורה שונה, או יחזיר ערכים לא תואמים – עלול להיפגע **משמעותית.**

דוגמה למעברים בין ערכי ה־$capacity$ **וסוגי הזיכרון (סטטי/דינמי) מופיעה בהמשך התרגיל.**

3 המחלקה vl_vector

בתרגיל זה הנכם נדרשים לממש, בקובץ vl_vector.h, מחלקה גנרית בשם vl_vector. ה־container שלכם ישמור ערכים מסוג T ועם קיבולת סטטית StaticCapacity (שניהם משתנים גנרים שהמחלקה מקבלת). ל־StaticCapacity נגדיר ערך ברירת מחדל של 16. עליכם לתמוך ב־API הבא:

	התיאור	הערות	זמן ריצה
פעולות מחזור החיים של האוביקט			
Default Constructor	בנאי שמאתחל vl_vector ריק.		$O(1)$
Copy Constructor	מימוש של בנאי העתקה.		$O(n)$ מספר האיברים בווקטור המועתק הוא n.
Sequence based Constructor	בנאי המקבל Input Iterator (מקטע $[first, last)$) של ערכי T ושומר את הערכים בוקטור. החתימה המלאה בהמשך.	הגדלת הווקטור, אם יש בה צורך – תעשה עבור כל n האיברים יחדיו, בפעם אחת, ולא עבור כל איבר לחוד.	$O(n)$ מספר האיברים ב־$[first, last)$ הוא n.
Single-value initialized constructor	בנאי המקבל כמות $count \in \mathbb{N} \cup \{0\}$ ואיבר v כלשהוא מסוג T . הבנאי מאתחל את הווקטור עם $count$ איברים בעלי הערך v.	יש לחשב את ההכנסה של האיברים כרצף ולא כהכנסה של איבר איבר	$O(count)$
Destructor	מימוש Destructor.		
פעולות			

`size`	הפעולה מחזירה את כמות איברים הנוכחית בווקטור.	ערך החזרה מטיפוס `size_t`.	$O(1)$
`capacity`	פעולה המחזירה את קיבולת הווקטור הנוכחית.	ערך החזרה מטיפוס `size_t`. נזכיר שלא צריך לחשב בפונקציה זו את cap_C.	$O(1)$
`empty`	פעולה הבודקת האם הווקטור ריק.	ערך החזרה `bool`.	$O(1)$
`at`	פעולה מקבלת אינדקס ומחזירה את הערך המשויך לו בווקטור.	הפעולה תזרוק חריגה במקרה שהאינדקס שגוי.	$O(1)$
`push_back`	הפעולה מקבלת איבר ומוסיפה אותו לסוף הווקטור.	הפעולה אינה מחזירה ערך.	$O(1)$ (amortized – לשיעורין)[3]
`insert (1)`	פעולה המקבלת איטרטור המצביע לאיבר מסוים בווקטור (position), ואיבר חדש. הפעולה תוסיף את האיבר החדש **לפני** ה־position (משמאל ל־position).	הפעולה תחזיר איטרטור המצביע לאיבר **החדש** (לאיבר שנוסף כעת).	$O(n)$ כאשר n הוא כמות האיברים בווקטור ($size$).
`insert (2)`	פעולה המקבלת איטרטור המצביע לאיבר מסוים בווקטור (position), ו־2 משתנים המייצגים Input Iterator למקטע $[first, last)$. הפעולה תוסיף את כל ערכי האיטרטור **לפני** ה־position.	הפעולה תחזיר איטרטור של הווקטור, המצביע לאיבר **<u>הראשון</u>** מרצף האיברים החדשים. תוכלו להסיק כיצד יש להגדיר את $first, last$ בעזרת החתימה של ה־sequence based constructor שבהמשך.	$O(n)$ כאשר n הוא כמות האיברים בווקטור, בחיבור כמות האיברים במקטע $[first, last)$
`pop_back`	הפעולה מסירה את האיבר האחרון מהווקטור.	הפעולה אינה מחזירה ערך. אם $size = 0$ יש לעצור מבלי לזרוק חריגה.	$O(1)$ (amortized)
`erase (1)`	הפעולה מקבלת איטרטור של הווקטור ומסירה את האיבר שהוא מצביע עליו.	הפעולה תחזיר איטרטור לאיבר **שמימין** לאיבר שהוסר.	$O(n)$ כאשר n הוא כמות האיברים בווקטור.
`erase (2)`	הפעולה מקבלת 2 משתנים המייצגים איטרטור של מופע ה־`vl_vector`, למקטע $[first, last)$. הפעולה תסיר את הערכים שבמקטע מהווקטור.	הפעולה תחזיר איטרטור לאיבר **שמימין** לאיברים שהוסרו.	$O(n)$ כאשר n הוא מספר האיברים בווקטור.
`clear`	פעולה המסירה את כל איברי הווקטור.		$O(n)$ כאשר n הוא מספר האיברים בווקטור.

[3] זמן ריצה לשיעורין.

כתזכורת נציג הגדרה לא פורמאלית לצורך התרגיל: אלגוריתם מתבצע בזמן ריצה לשיעורין $O(1)$ אם רובן המכריע של הקריאות אליו מתבצעות בזמן קבוע, בעוד חלק קטן מהקריאות אליו מתבצעות בזמן ארוך יותר. להרחבה, ראו: https://bit.ly/3jSVAsQ.

data	פעולה המחזירה **מצביע** למשתנה שמחזיק כרגע את המידע.	הפעולה תחזיר מצביע למשתנה שמחזיק את האיברים ב־stack או ב־heap, בהתאם למצב הנוכחי של ה־vl_vector.	$O(1)$
contains	הפעולה מקבלת משתנה מסוג T ומחזירה ערך בוליאני – האם הערך נמצא בווקטור?		$O(n)$ כאשר n הוא מספר האיברים בווקטור.
Iterator Support	על המחלקה vl_vector **לתמוך** ב־iterator (**לרבות typedefs**) בהתאם לשמות הסטנדרטים של C++ 14.	עליכם **לתמוך** ב־Random Access Iterator (const ו־non const).	על כל הפעולות הנדרשות לעמוד בזמן ריצה של $O(1)$.
Reverse Iterator Support	על המחלקה vl_vector **לתמוך** ב־reverse iterator[4] (**לרבות typedefs**) בהתאם לשמות הסטנדרטים של C++.	עליכם **לתמוך** ב־Random Access Iterator (const ו־non const).	
אופרטורים			
השמה	תמיכה באופרטור ההשמה (=).		
subscript	תמיכה באופרטור [].	האופרטור יקבל אינדקס ויחזיר את הערך המשוייך לו. **אין לזרוק חריגה במקרה זה.**	$O(1)$
השוואה	תמיכה באופרטורים ==, ! =	שני ווקטורים שווים אחד לשני אם ורק אם ערך ה־$size$ שלהם זהה, איבריהם שווים ומופיעים בסדר זהה. ניתן להניח כי אופרטור ההשוואה יופעל רק על ווקטורים שערכי ה־T וה־C שלהם זהים.	$O(n)$

דגשים, הבהרות, הנחיות והנחות כלליות:

- החתימה ל־Sequence based Constructor (שרלבנטית גם, בשינויים המתחייבים, ל־insert 2) היא:

```
template<class InputIterator>
vl_vector(InputIterator first, InputIterator last);
```

- **נדגיש שוב:** על המחלקה להיות **גנרית**. הערך הגנרי הראשון הוא טיפוס הנתונים שהמחלקה מאחסנת, אליו התייחסנו כ־T. הפרמטר הגנרי השני הוא הקיבולת המקסימלית שניתן לאחסן באופן סטטי, לה קראנו StaticCapacity (בחלק "התיאורטי" היא מסומנת כ־C). **ל־StaticCapacity יהיה ערך ברירת המחדל – 16.**

- **ניתן להניח** כי מופעים מסוג T תומכים ב־operator==, operator= וכן כי יש למופעי T בנאי דיפולטיבי ובנאי העתקה.

[4] reverse iterator הוא איטרטור שרץ על איברי ה־container בסדר הפוך. למשל, עבור [1, 2, 3, 4] תבוצע הריצה בסדר $1 \to 2 \to 3 \to 4$. מומלץ בחום לקרוא את המקור הבא, המציג דרך אפשרית למימוש (אתם רשאים להשתמש בה): https://en.cppreference.com/w/cpp/iterator/reverse_iterator.

- **בפתרונכם אינכם רשאים לעשות שימוש באף container של STL.** קרי, ניתן לעשות שימוש בכל אלגוריתם של STL, אך איננכם רשאים לעשות שימוש ב־containers. כך, בפרט, אין לעשות שימוש ב־`std::array`, `std::vector` ו־`std::list`. **שימוש ב־containers יגרור בהכרח ציון 0** (וממילא אינו יכול לעמוד במלוא ההגדרות של `vl_vector`). באופן דומה, למען הסר ספק, לא ניתן להשתמש בספריות חיצוניות.

- הווקטור יהיה בעלי הזיכרון של איבריו (כלומר, הוא יחזיר ב־ownership עליהם). כדי לעשות זאת, יש **להעתיק** את האיברים שמקבלים בפעולות ההוספה – ולא לשמור reference/pointer אליהם.

- בחלק הנוגע ל־iterators עליכם לתמוך ב־naming conventions של STL. בפרט, הגדירו טיפוסים רלוונטים אם צריך, כפי שלמדנו בכיתה ובתרגול.[5] בפרט, בחלק הנוגע ל־reverse iterator שימו לב שאתם עומדים ב־convention הרלבנטי. תוכלו לראות דוגמה ל־conventions האלו ב־`std::vector`[6] (לדוגמה, המתודה המקבילה ל־`begin` עבור reverse iterator היא `rbegin`).

- כאשר מקבלים Input Iterator ניתן להניח על איבריו כי אכן כולם מאותו טיפוס הנתונים – וכי טיפוס נתונים זה הוא טיפוס הנתונים ששומרים בווקטור (T). כמו כן, ניתן להניח כי $begin \leq end$.

- ניתן לטעות ולחשוב, בשוגג, כי ישנה התנגשות בין החתימות של Sequence based Constructor ושל Single-value Initialized Constructor. מאחר שניתן להניח כי Single-value Initialized Constructor יקבל **בהכרח** ערך *count* מטיפוס `size_t` – בהכרח לא תהיה התנגשות (למה? שאלה טובה לקראת המבחן ☺).

- ה־API הנ"ל מציג לכם את שמות הפונקציות המחייבות, הפרמטרים, ערכי החזרה וטיפוסיהם. בעת מימוש ה־API, עליכם ליישם את העקרונות שנלמדו בקורס באשר לערכים קבועים (constants) ומשתני ייחוס (references). **שימוש בקונבנציות אלו הוא חלק אינטגרלי מהתרגיל.** עיקרון זה נכון בפרט גם לגבי התמיכה ה־iterator וה־reverse iterator.

- **שימו ♡**: לפני שתיגשו לחיבור הפתרון, חישבו על **כל** הכלים שרכשתם בקורס. בפרט, כשאתם שוקלים האם האופציה X מתאימה למימוש – חישבו בין היתר איזה תכונות יש לה? היכן היא מוקצת? מה הייתרונות שלה? מה היא דורשת מכם מבחינת מימוש. חשוב לנו להדגיש: תרגיל זה מתוכנן כך שהוא נוגע במרבית החומר של הקורס. שימוש נכון בכלים שונים שלמדנו לא רק שיקצר את מרבית הפונקציות לאורך של כמה שורות בלבד, אלא יאפשר לכם לקבל "במתנה" חלק נכבד מהמימוש.

4 בונוס (25 נק')

פרק זה הוא חלק רשות, המציע בונוס של עד 25 נקודות נוספות לציון התרגיל. נבהיר כי בהחלט ניתן לקבל 100 בתרגיל גם מבלי להגיש חלק זה. הציון המקסימלי ל־"תרגיל 6", אפוא, הוא 120.

[5] נעיר כי כדי לממש container "חוקי", התואם באופן מלא ל־STL convention, הייתם נדרשים להוסיף מספר הגדרות ומתודות נוספות. **איננו דורשים זאת בתרגיל זה.** ההגדרות להן אתם נדרשים הן אלו שנלמדו בהרצאה ובתרגולים בלבד. להרחבה, ראו: https://en.cppreference.com/w/cpp/named_req/Container.

[6] ניתן לעיין בממשק של std::vector בקישור הזה: https://en.cppreference.com/w/cpp/container/vector

4.1 רקע

בחלק הנוגע לשפת C בסדנה, ראינו שמחרוזות מיוצגות כמערך של תווים. כך, למשל:

Hello, World!												
H	e	l	l	o		W	o	r	l	d	!	\0

בהמשך, ראינו שעבודה עם מחרוזות בשפת C אינה כה טריוויאלית (למשל חיבור בין מחרוזות הצריך הקצאת מחרוזת דינמית, שמוש ב־`strcpy` וכו'). לקושי זה מצאנו פתרון ב־C++, כשחקרנו את ה־API של `std::string`.
אלא, שכאשר חושבים על ניהול זיכרון אופטימלי, הקשיים שתיארנו לעיל מהם "סובל" `std::vector` רלבנטים בהחלט גם ל־`std::string`. לכן, עולה השאלה האם אפשר להשתמש ב־VLVector שכתבנו על מנת לייעל, מבחינת שימוש בזיכרון, באופן **ספציפי** את העבודה עם מחרוזות? נראה שכן.

4.2 המחלקה `vl_string`

נגדיר את ה־container "מחרוזת מאורך משתנה" (Variable Length String, או בקצרה VLString) להיות טיפוס נתונים גנרי המייצג **סוג ספציפי** של Variable Length Vector.[7]
VLString יחזיק **תווים** (כלומר, "במונחי VLVector", $T = char$) ויהיה בעל קיבולת סטטית C. ל־VLString יהיו, מצד אחד, תכונות דומות לאלו של מחרוזת, כך שיהיה ניתן למשל לבצע פעולות שרשור או הדפסה. אלא, שמהצד השני, הוא יציע אלגוריתם זהה לזה של VLVector מבחינת ניהול זיכרון, ובכך יהנה מהיתרונות ניהול הזיכרון של VLVector.
ממשו, בקובץ `vl_string.h`, מחלקה גנרית בשם `vl_string`. מבנה הנתונים שלכם יקבל פרמטר גנרי **יחיד**, והוא קיבולת סטטית `StaticCapacity`. ל־`StaticCapacity` נגדיר ערך ברירת מחדל של 16. עליכם לתמוך ב־API הבא:

	התיאור	הערות	זמן ריצה
פעולות מחזור החיים של האוביקט			
Default Constructor	בנאי שמאתחל `vl_string` עם **מחרוזת ריקה**.		$O(1)$
Copy Constructor	מימוש של בנאי העתקה.		$O(n)$ מספר התווים במחרוזת הוא n.
Implicit Consstructor	בנאי **implicit** המקבל פרמטר יחיד מסוג `const char *` ושומר את תוויו ב־`vl_string`.	קיבולת המחרוזת תחושב באמצעות cap_C כאשר $k = n$ (כלומר k האיברים שמוסיפים הם n התווים שבמחרוזת).	$O(n)$ מספר התווים במחרוזת הוא n.
פעולות			
`vl_vector` methods	על המחלקה לתמוך **בכל הפעולות** של `vl_string`.		
`contains`	הפעולה מקבלת פרמטר יחיד מסוג `const char *` ובודקת האם הוא נמצא ב־`vl_string`.	פעולה זו באה "במקום" הפעולה הסטנדרטית של `vl_vector`.	$O(n)$ מספר התווים במחרוזת הוא n.

[7] למדנו בקורס כלים רבים – חשבו איזה כלי תואם בדיוק לסיטואציות שבהן נדרש לממש קשר שבו "B הוא סוג ספציפי של A".

אופרטורים			
vl_vector operators	על המחלקה לתמוך **בכל האופרטורים** של vl_vector.		
$operator+=$	יש לתמוך בפעולות חיבור עם vl_string, const char *, const char	פעולת ה־+= מוגדרת כפעולת **שרשור**. אין צורך בסימטריות.	
$operator+$	יש לתמוך בפעולות חיבור עם vl_string, const char *, const char	פעולת ה־+ מוגדרת כפעולת **שרשור**. אין צורך בסימטריות.	
Implicit casting opreator	יש לתמוך ב־implicit casting operator ל־const char *.		

דגשים, הבהרות, הנחיות והנחות כלליות:

- אין צורך לממש פעולות נוספות שלא מופיעות בטבלה שלעיל. כך, למשל, אין צורך לתמוך ב־Single-value initialized constructor.
- הנכם נדרשים לעשות שימוש ב־vl_vector שהגדרתם בפרק הקודם לתרגיל. הניחו שהקובץ vl_vector.h נמצא באותה התיקיה של vl_string.h.
- **שימו לב:** הן בבנאי והן באופרטורי השרשור, שרשור תו יחיד שקול להוספת איבר יחיד לווקטור, בעוד ששרשור מחרוזת שקול להוספת Sequence (Iterator; למה?).
- **שימו לב:** באופן זהה ל־String literals, מחרוזת תמיד כוללת את תו ה־NULL ('\0') בסופה – כך גם בפרט כשהיא ריקה. אילוץ זה משפיע על המתודות של vl_vector ומחייב לשנותן באופן שבו יתמודדו עם תווים בודדים, מבלי להתייחס לתו ה־NULL. כך למשל:
  - size צריכה לשקף את אורך המחרוזת, באופן השקול ל־strlen.
  - capacity תחושב בהתאם לתוכן של vl_string – כלומר בהתאם לכל התווים שהוא מחזיק, **לרבות תו ה־NULL.**
  - הוספת או הסרת תווים למחרוזת (לדוגמה פעולות ה־push_back ו־pop_back) יוסיפו, או יסירו, בהתאמה, את **התו האחרון** מהמחרוזת, ולא את ה־terminator (כלומר, לא את \0).
  - אתחול מחרוזת ריקה (בבנאי הדיפולטיבי) תהא מחרוזת שמכילה רק את תו ה־NULL.
  - לא ניתן (ולכן גם לא מצופה מכם) לשנות את התנהגותן של Insert (2) ושל Erase (2) (למה?).
- כל ההנחיות הנוגעות ל־vl_vector רלבנטיות גם כאן. בפרט, גם בחלק זה אין להשתמש באף container של STL וכמובן שאין להשתמש ב־std::string, יש לתמוך ב־const ו־reference במידת הצורך, וכך גם תוכלו להוסיף מתודות כראות עיניכם, בהתאם לנלמד בקורס.
- בחלק זה – ובחלק זה בלבד – מותר לייבא את cstring (השם "המקביל" ל־string.h ב־C++) ולעשות שימוש "בפונקציות C" המוגדרות בו.

5 נהלי הגשה

- זיכרו לוודא שתרגילכם עובר קומפילציה **במחשבי בית הספר** ללא שגיאות **ואזהרות**, וכנגד מהדר בתקינה שנקבעה בקורס (C++14). אזהרות יביאו בהכח לגריעת ניקוד (בהתאם לחומרת האזהרות). תרגיל שאינו עובר הידור, ינוקד בציון 0. בנוסף, נזכיר שיש **לתעדף** פונקציות ותכונות של C++ על פני אלו של C. למשל, נעדיף להשתמש ב־`new` ו־`delete` על פני `malloc` ו־`free`.

- כאמור בהנחיות להגשת תרגילים – הקצאת זיכרון דינמית מחייבת את שחרור הזיכרון. **עליכם למנוע בכל מחיר דליפות זיכרון מה־container שלכם**. תוכלו להיעזר ב־valgrind כדי לאתר דליפות זיכרון. דליפות זיכרון יאבדו ניקוד **משמעותי**.

- לתרגיל זה **לא ניתן פתרון בית ספר**. כחלופה לכך, ציידנו אתכם בנספח המכיל מספר דוגמאות לשימוש ב־vl_vector ו־vl_string. **אין להגיש את את קובצי הדוגמה.**

- אנא וודאו כי התרגיל שלכם עובר את ה־Pre-submission Script **ללא שגיאות או אזהרות**. קובץ ה־Pre-submission Script זמין בנתיב.

```
~labcc2/presubmit/ex6/run <path_to_sibmission>
```

בהצלחה!!

6 נספח א' – חומרי עזר

לתרגיל זה לא מסופק פתרון בית ספר. במקום זאת, סיפקנו לכם מספר חומרי עזר:

6.1 תוכנית לדוגמה – Highest Student Grade

יצרנו עבורכם תוכנית לדוגמה העושה שימוש בכמה מהתכונות הבסיסיות של הווקטור. כך, אם זו מומשה נכון, תוכלו לקמפל ולהריץ בהצלחה את התוכנית. התוכנית highest_student_grade.cpp, מצורפת כחלק מקובצי התרגיל ואין להגישה.
התוכנית קולטת רשימה של סטודנטים מהמשתמש דרך ה־CLI, ולאחר מכן מדפיסה את הסטודנט עם ממוצע הציונים הגבוה ביותר. לשם כך, התוכנית מגדירה מחלקה בשם Student, שלה 2 שדות – "שם פרטי" ו־"ממוצע ציונים". כמו כן, לשם שמירת הסטודנטים שנקלטו על ידי המשתמש, התוכנית עושה שימוש ב־vl_vector. נביט בדוגמת הרצה:

```
$ ./HighestStudentGrade
Enter a student in the format "<name> <average>" or an empty string to stop:
Mozart 70.5
Enter a student in the format "<name> <average>" or an empty string to stop:
Beethoven 95
Enter a student in the format "<name> <average>" or an empty string to stop:
Liszt 83.0
Enter a student in the format "<name> <average>" or an empty string to stop:
<< Note: This is an empty line >>
-----------------------------------------------------------------
Total Students: 3
Student with highest grade: Beethoven (average: 95)
```

שימו לב שהקלט שצבוע בירוק הוא קלט שהזין המשתמש. כמו כן, השורה לפני שורת הפסים ריקה מאחר שהמשתמש הזין קלט ריק. נדגיש:

- התוכנית מבצעת בדיקות קלט בסיסיות בלבד. תוכנית זו אינה מתיימרת להיות פתרון מלא ומקיף, אלא להציג שימוש בסיסי ב־vl_vector שיצרתם.
- אנו ממליצים כי תעיינו בקפידה בתוכנית, הכוללת הערות המסבירות את הנעשה שלב שלב. תוכנית זו תוכל לסייע לכם בהבנת המשימה.

6.2 קובצי ה־Presubmission

קוד המקור של תוכנית ה־Presubmit זמינה עבורכם, ותוכלו למצוא שם בדיקות בסיסיות של הווקטור, **לרבות בדיקת Resize בסיסית**. מומלץ בחום שתעיינו בתוכנית זו. אתם רשאים בהחלט לבצע שינויים בתוכנית זו בכדי ליצור Tests משלכם.

6.3 דוגמה מספרית לגדילת וכיווץ הווקטור

כדי לוודא שהאופן שבו קיבולת הווקטור גדלה או מתכווצת ברור ונהיר לכולם, נתאר להלן מקרה אחד של הגדלה וכיווץ. בעמודה השמאלית ניתן לראות את הפעולה המבוצעת (בחלק המקרים היא כתובה בקוד, ובחלק בתיאור מלולי). בעמודות שמימינה, מופיעים ערכי ה־*size* וה־*capacity* שמתקבלים **כתוצאה** מביצוע הפעולה. נעיר כי רצף הפעולות שלהלן זמין כקוד ב־Presubmit תחת הפונקציה TestResize.

פעולה	**קיבולת (*capacity*)**	**גודל (*size*)**	**הצדקה**
`vl_vector<int> vec;`	16	0	ערכי ברירת מחדל
`vec.push_back(1);`	16	1	
Insert 16 additional items, **one by one**	25	17	נוסיף את האיברים אחד אחר השני עד שנגיע לאיבר ה־16. שם נדרש מעבר לזיכרון דינמי (כי $16+1>C$) וחישוב הקיבולת: $cap_{16}(16,\,1)=\left\lfloor\frac{3\cdot(16+1)}{2}\right\rfloor=25$
Insert 13 additional items, **using an iterator** (at one single call to "`insert`")	45	30	כמות האיברים שנרצה לשמור בווקטור (30) חוצה את הקיבולת, לכן נחשב לפי cap_C: $cap_{16}(17,\,13)=\left\lfloor\frac{3\cdot(17+13)}{2}\right\rfloor=45$
Erase 13 items, **one by one**	45	17	כשמסירים איברים (אך לא "חוזרים" לזיכרון הסטטי) ה־$capacity$ לא קטן.
`vec.clear();`	16	0	הקיבולת מאותחלת חזרה ל־C.
Insert 17 items, **one by one**	25	17	ה־$capacity$ לא יהיה 45 כיוון שבשלב הקודם חזרנו להשתמש בזיכרון הסטטי, פעולה ש־"אתחלה" את ה־$capacity$ הדינמי.
Removing **one** item	16	16	מאחר ש־$17-1=16=C$ יש לחזור לעשות שימוש בזיכרון הסטטי.

6.4 תוכנית לדוגמה לחלק הבונוס – Append String

לאלו מכם שמעוניינים לממש את חלק הבונוס, יצרנו תוכנית לדוגמה, העושה שימוש בסיסי ב־vl_string. **אין להגיש תוכנית זו.**
התוכנית קולטת רשימה של מחרוזות מהמשתמש דרך ה־CLI, ולאחר מכן מדפיסה את השרשור שלהן. נביט בדוגמת הרצה:

```
$ ./AppendString
Enter a string to append, or an empty string to stop:
I love
Enter a string to append, or an empty string to stop:
bonuses in exercises
Enter a string to append, or an empty string to stop:
<< Note: This is an empty line >>
----------------------------------------------------------------
String: I love bonuses in exercises
```

שימו לב שהקלט שצבוע בירוק הוא קלט שהזין המשתמש. כמו כן, השורה לפני שורת הפסים ריקה מאחר שהמשתמש הזין קלט ריק. נדגיש:

- התוכנית מבצעת בדיקות קלט בסיסיות בלבד. תוכנית זו אינה מתיימרת להיות פתרון מלא ומקיף, אלא להציג שימוש בסיסי ב־vl_string שיצרתם.

- אנו ממליצים כי תעיינו בקפידה בתוכנית, הכוללת הערות המסבירות את הנעשה שלב שלב. תוכנית זו תוכל לסייע לכם בהבנת המשימה.

7 נספח ב' – שיקולים לקביעת פונקצית קיבולת הווקטור

כאמור, לווקטור שלנו, כמו גם ל־`std::vector`, יש פונקציית קיבולת, המתארת מהי כמות האיברים המקסימלית שהוא יכול להכיל בכל רגע נתון. נגדיר את $cap_C : \mathbb{N}\cup\{0\}\times\mathbb{N} \to \mathbb{N}$ להיות פונקציית הקיבולת של הווקטור, כך שבהינתן $size \in \mathbb{N}\cup\{0\}$ – כמות האיברים הנוכחית שהווקטור מכיל (לפני פעולת ההוספה), $k \in \mathbb{N}$ כמות האיברים שרוצים **להוסיף** לווקטור ו־$C \in \mathbb{N}\cup\{0\}$ – קבוע הזיכרון הסטטי המקסימלי של הווקטור, הפונקציה תחזיר את הקיבולת המקסימלית של הווקטור.
כאשר מדובר בזיכרון סטטי ($size+k \le C$), זה קל – הקיבולת של הווקטור היא C. תמיד. עתה, מהי הקיבולת של הווקטור כשהוא חוצה את C ועובר להשתמש בזיכרון דינמי? האם בהכרח נרצה להגדיר $cap_C\left(size,\, k\right) = \begin{cases} C & size+k \le C \\ size+k & size+k > C \end{cases}$? נראה להלן שלא.
הנחת המוצא שלנו היא שאנחנו רוצים לשמור על זמני ריצה טובים ככול האפשר. המטרה שלנו, אפוא, תהיה שבמימוש אופטימלי פעולות הגישה לווקטור, ההוספה לסוף הווקטור וההסרה מסוף הווקטור יפעלו כולן ב־$O(1)$ לשיעורין. אנו נתייחס רק לפעולת ההוספה לסוף הווקטור, כשזמן הריצה של פעולת ההסרה מהסוף יגזר משיקולים אלו באופן טריוויאלי.
תחילה, כאשר הווקטור עושה שימוש בזיכרון הסטטי, הוקצו עבורו מראש $C \cdot sizeof(T)$ בייטים שזמינים לו סטטית. מכאן ששמירת איבר חדש בסוף הווקטור תעשה בנקל ב־$O(1)$.
השאלה העיקרית היא, כיצד נקבע את קיבולת הווקטור בהקצאות דינמיות? כדי להקל על הדיון, נסמן ב־$\phi : \mathbb{N}\cup\{0\}\times\mathbb{N} \to \mathbb{N}$ את פונקציית הקיבולת **עבור זיכרון דינמי**, כך ש־

$$cap_C\left(size,\, k\right) = \begin{cases} C & size+k \le C \\ \phi(size,\, k) & size+k > C \end{cases}$$

כך, הדיון יוכל מעתה להתמקד רק ב־ϕ (ולמעשה "כאילו" מדובר בווקטור "דינמי רגיל", כדוגמת std::vector).

7.1 ניסיון ראשון – $\phi(s,\, k) = s + k$

כדי לא להקשות על הקריאה עם משתנים נוספים, נסתכל על המקרה הפרטי $k = 1$. ההתאמה ל־k יחסית טריוויאלית. נגדיר $\phi(s) = s + 1$ (כי כאמור $k = 1$). במקרה זה הנחנו, אפוא, **שקיבלת הווקטור (כשהוא משתמש בזיכרון דינמי) תהיה כמות האיברים הנוכחית שלו ועוד 1 (האיבר החדש שנוסף)**. דהיינו, ϕ יחזיר בדיוק את כמות האיברים החדשה שתהיה בווקטור, לאחר הוספת האיבר.
לפיכך, אם גודל הווקטור לפני הוספת האיבר הוא $C < size \in \mathbb{N}$, אזי כאשר נוסיף איבר חדש לסוף הווקטור, נצטרך להקצות זיכרון מחדש, כך שעתה נקבל $(size+1)\cdot sizeof(T)$ בייטים. על פניו, נשמע שזה דיי פשוט, לא? נבצע הקצאה דינמית שתוסיף לנו $sizeof(T)$ בייטים, נכתוב עליהם את האיבר החדש – ובא לציון גואל. או... שלא?

כל פעם שנגדיל את הווקטור, נצטרך להעתיק את איבריו.[8] [9] העתקת האיברים היא פעולה לינארית ולכן תבוצע, כמובן, ב־$O(n)$. מכאן שניסיון זה לא עומד בדרישות זמן הריצה.

7.1.1 מדוע לא ניתן להגדיל את זיכרון הווקטור עם realloc?

בשלב זה ניתן לתהות מדוע לא להשתמש ב־realloc כדי להגדיל את זיכרון הווקטור? כפי שציינו בשיעור הנוגע להקצאות דינמיות ב־C++, שימוש בפונקציות ניהול זיכרון דינמי של C ב־C++ עלול להוביל לטעויות ובאגים חמורים מבלי להפעיל את הכלים הנכונים לכך (שלצערינו, לא נלמדים בקורס).

בתור התחלה, נזכיר את ההתנהגות של realloc: בהינתן שהוקצו כבר k בייטים (למשל דרך malloc), ו־realloc התבקשה להקצות עוד l בייטים נוספים[10] אזי realloc תנסה בהתחלה לבחון האם יש מקום לעוד l בייטים שימוקמו **באופן רציף לאחר** k **הבייטים שכבר הוקצו**. אם התשובה לכך חיובית – l הבייטים הנוספים האלו "ישויכו" לתוכנית והיא תוכל להשתמש בהם ללא כל העתקה. ברם, אם לא קיימים l הבייטים הללו – realloc תחפש מקום **רציף** אחר בזיכרון **לכל** $k+l$ **הבייטים המבוקשים.** הבעיה המרכזית, אם כן, היא המקרה שבו אנו נדרשים להעתיק את כל ה־$k+l$ בייטים, הכוללים בתוכם גם את האוביקטים "הישנים".
משהנקודה הזו הובהרה – חישבו, עתה, על חמשת הנקודות הבאות (וזו לא רשימה מלאה):

- כאשר מועתק אוביקט ב־C++, כחלק מה־object life cycle היינו מצפים שיקרא בנאי ההעתקה או ה־assignment operator שלו (בהתאם להקשר). האם זה יקרה כשנקרא ל־realloc – שפשוט מעתיקה בלוק רציף של זיכרון?

- בהמשך לנקודה הקודמת, נזכיר שתפקידו של בנאי העתקה, בפרט, הוא לבצע deep copy (אם יש צורך). פעולה זו לא תבוצע במקרה של קריאה ל־realloc – מה שיביא לבעיות חמורות בניהול הזיכרון (כדי לחדד את הקושי: חזרו למחלקת MyString שהוצגה בהרצאות).

- באותו ההקשר – בעת ההעתקה אנחנו נדרשים להשמיד את האוביקט הקודם – ובמקרים אלו, כפי שראיתם, C++ אמורה לקרוא ל־־destructor. אלא, כשנשתמש ב־realloc, שרק מעתיקה רצף של בייטים ולא מודעת ל־C++ objects life cycle, האם היא אכן תדע איך לקרוא ל־destructor של האוביקט?

- זכרו שמחלקות עלולות לזרוק חריגה (exception) בזמן הקריאה לבנאי – איך realloc אמורה להתמודד עם מקרה שכזה?

- ההנחה הסמויה בעת השימוש ב־realloc היא שאוביקטים שמורים **באופן רציף** בזיכרון. אלא, האם באמת ניתן להניח שהאוביקטים שאנו עובדים איתם שמורים באופן **רציף** בזיכרון? מה יקרה, למשל, אם נדרוס את operator new ו־operator delete (ראינו בתרגול שניתן לעשות זאת) ונציע מימוש אחר לניהול הזיכרון?

---

[8] מאחר ש־T עשוי להיות אוביקט, שימוש ב־realloc לא יסייע לנו. מסיבות דומות, נזכיר שיש לתעדף שימוש באופרטורים של C++ על אלו של C ולכן אין להשתמש בפונקציות הקצאת הזיכרון של C, אלא יש להשתמש בכלי הקצאת זיכרון של C++.

[9] ניתן לתהות האם אין אלגוריתם שלא מצריך העתקה. כשאתם שוקלים זאת, כדאי לחשוב האם הוא עונה על שאר הדרישות שהצבנו. למשל, האם פעולת הגישה שלו ממומשת ב־$O(1)$?

[10] הדוגמה תואמת לקטע הקוד הבא:

```
char* obj = malloc(k);
char* tmp = realloc(obj, k + l);
```

7.2 ניסיון שני – $\phi(s) = (s+k)\cdot 2$

שוב, לצורכי הנוחות נסתכל על המקרה הפרטי $k=1$. נגדיר $\phi(s) = (s+1)\cdot 2$. כלומר הגדרנו את "אסטרטגית הגדילה" של הווקטור באופן שבו נגדיל את קיבולת הווקטור כל פעם פי 2, ולכן לא נבצע הקצאה מחדש בכל פעם שהמשתמש יבקש להוסיף איבר חדש. אנו טוענים כי הגדרה זו תביא לכך שפעולת ההוספה תפעל ב־$O(1)$ לשיעורין. נגדיל ונטען טענה חזקה יותר (שתשמש אותנו עוד רגע) – יהי $m \in \mathbb{R}$ פרמטר הגדילה, כך ש־$m > 1$ (ובענייננו $m = 2$). נטען כי פעולת ההוספה תבוצע ב־$O(1)$ לשיעורין.

הוכחה: יהי וקטור עם זיכרון דינמי[11] לו פרמטר גדילה $1 < m \in \mathbb{R}$. יהי $n \in \mathbb{N}$ כמות האיברים שנרצה להוסיף לסוף הווקטור. הוספת n האיברים תדרוש $\lfloor log_m(n) \rfloor$ הקצאות מחדש, כאשר ההקצאה ה־i תהיה פרופורציונלית ל־m^i. לכן, כל פעולת הוספה מבוצעת ב־

$$c_i = \begin{cases} m^i + 1 & \exists p \in \mathbb{N} \ \ S.t. \ \ i-1 = m^p \\ 1 & otherwise \end{cases}$$

כן, בסך הכל, הוספת n איברים פועלת בסיבוכיות זמן הריצה של:

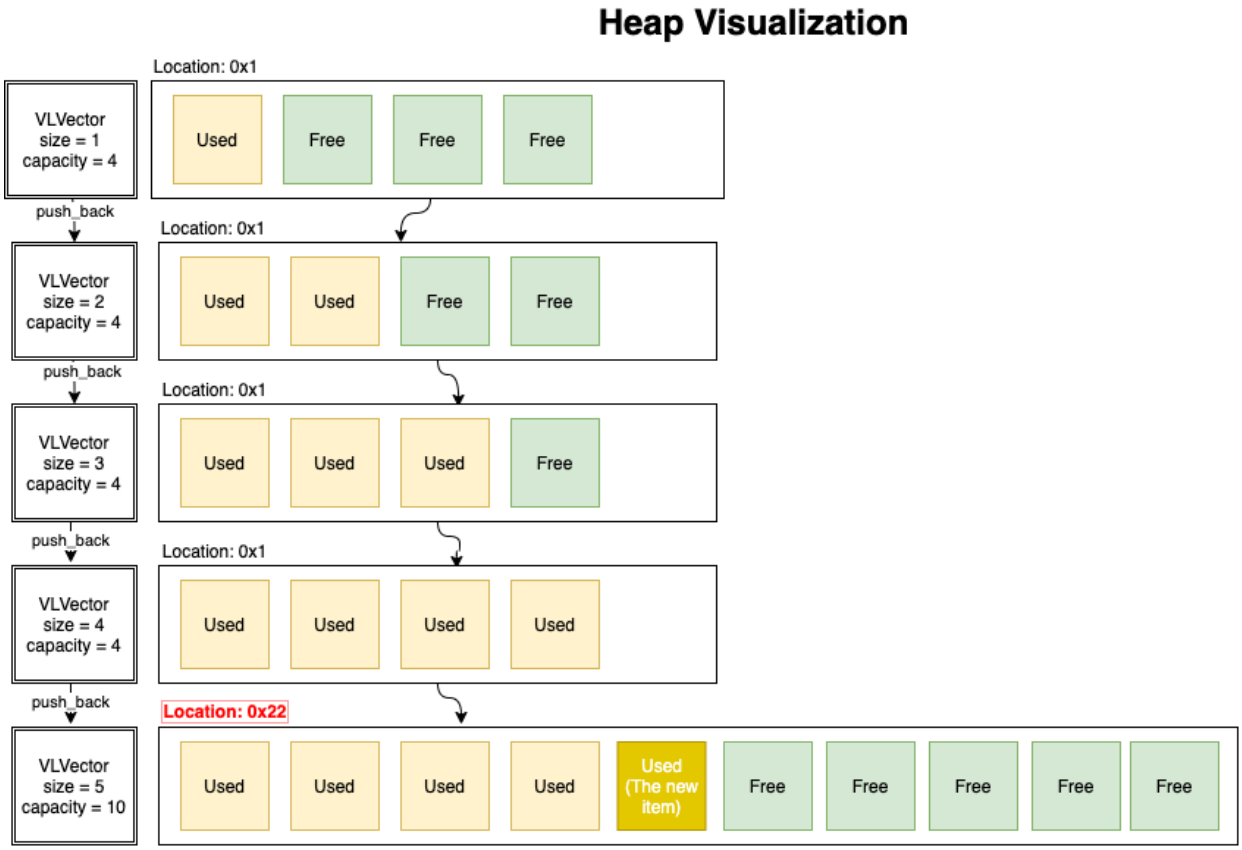
$$T(n) = \sum_{k=1}^{n} c_k \leq n + \sum_{k=1}^{\lfloor log_m(n) \rfloor} m^k \leq n + \frac{m \cdot n - 1}{m-1}$$

לפיכך, כשנחלק את $T(n)$ ב־n, עבור n פעולות הוספה, נקבל שכל פעולה מבוצעת בזמן ריצה לשיעורין של $\frac{T(n)}{n} \leq \frac{n-1}{n\cdot(m-1)} + 2 \in O(1)$. ■

המסקנה מהטענה לעיל היא שבענייננו, כאשר $m = 2$, נקבל $\frac{3n-1}{n} < 3 \ \forall n \in \mathbb{N}$ ולכן הצלחנו להגדיר את ϕ כך שתעבוד בזמן ריצה לשיעורין החסום על ידי $O(1)$. נזכיר שוב ש־ϕ הוגדרה כאן עבור $k=1$, אך החישובים שהראנו רלבנטים גם עבור $1 < k \in \mathbb{N}$ אחר. הבעיה עם פרמטר הגדילה 2 היא לא זמני ריצה – **אלא שימוש לא יעיל בזיכרון**. נקצר ונסביר את העיקרון הכללי, מבלי להעמיק בחישוב שעומד מאחוריו. נניח שמדובר בווקטור "רגיל" (כמו `std::vector`, דהיינו ווקטור ללא זיכרון סטטי[12]) המחזיק ב־$capacity$ התחלתי $C \in \mathbb{N}$. כשנידרש להגדיל את הווקטור לראשונה, כחלק מפעולת "הוספה לסוף הווקטור", הוא יצטרך לבקש ממערכת ההפעלה $2C$ בייטים חדשים לאחסון הנתונים. שימו לב לאילוסטרציה הבאה (ובפרט לכתובת בכל שלב):

[11] לאו דווקא כזה המצוייד גם בזיכרון סטטי. זו הוכחה יפה גם עבור `std::vector`.

[12] ההוכחה עבור וקטור שיש לו גם זיכרון סטטי, כמו המימוש שהגדרנו ל־VLVector, זהה.

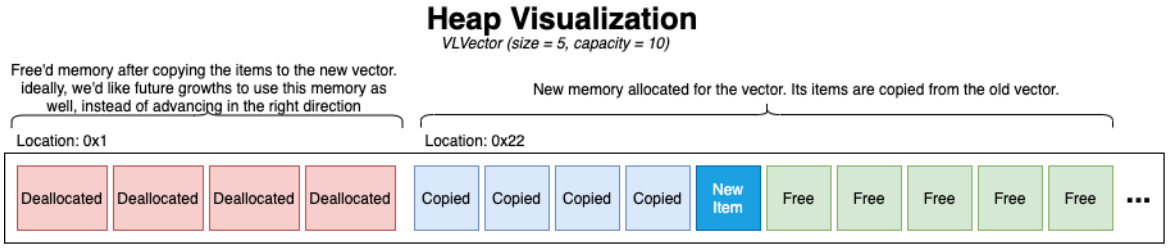


במקרה הזה, נקבל שהווקטור החדש שהקצנו תופס $2C$ בייטים (כי $m = 2$), **אך <u>לפניו</u> – וזו הנקודה – ישנם C בייטים, שאותם תפס הווקטור הקודם, ושאותם נרצה לשחרר**. לכן בסוף פעולת ההכנסה יש לנו $2C$ בייטים בשימוש על ידי הווקטור החדש, ו־C בייטים שהיו בשימוש על ידי הווקטור הישן וכעת הם deallocated.

אם כך, היכן "הבעיה" – הרי אותם אותם C שוחררו, אזי הם זמינים לשימוש חוזר, לא? התשובה היא שכדי לעשות שימוש אפקטיבי בזיכרון, **נרצה "למחזר" זיכרון**. כלומר, נרצה להגיע מתישהוא למצב שבו "צברנו" מספיק deallocated memory **רציף**, באמצעות שחרורי וקטורים קודמים, כך שביחד יוכלו להכיל מופע של וקטור גדול יותר. אם נגיע למצב כזה, נוכל "למחזר" את אותו זיכרון שעבר deallocation ולהקצות שם את הווקטור החדש, הגדול יותר. וזיכרו: לא נוכל לעולם "לצרף" את אותו deallocated memory לווקטור הנוכחי, כי אנחנו רוצים להעתיק את הערכים, אז בשעת ההקצאה של הווקטור החדש, והגדול יותר, הווקטור הישן **עדיין קיים בזיכרון** ולכן לא ניתן למזג בין קטעי הזיכרון לכדי וקטור אחד.

במילים אחרות, אידאלית, היינו רוצים שהווקטור יוכל לא רק לגדול "ימינה" (כלפי זיכרון חדש, שהוא עוד לא קיבל), אלא גם "שמאלה" (כלפי זיכרון שכבר היה בשימוש בעבר, ועבר deallocation).[13] ראו את האילוסטרציה הבאה של ה־heap, לאחר הגדלת קיבולת הווקטור:

[13] שימו לב שאנחנו דנים במצב "האידיאלי", בו בקשת הזיכרון לא "הכריחה" את מערכת ההפעלה להעביר את כל הווקטור לבלוק אחר בזיכרון (ואז כלל לא נוכל לשקול את מקרה זה, שהרי אנו מסתמכים כאן על רציפות הזיכרון).



שימו לב לתאים המופיעים כ־deallocated. אנו נרצה לאפשר לווקטור ב־"גדילות" עתידיות להשתמש בשטח זה, שהצטבר עם הזמן, במקום לבקש זיכרון חדש ממערכת ההפעלה.
למרבה הצער, נראה שעם פרמטר גדילה של $m = 2$ זה לא יתאפשר: כאשר נחשב את הערך של C במקרה הכללי, בהינתן פרמטר הגדילה $m = 2$ נקבל:

$$\sum_{k=0}^{n} 2^k = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$$

משמעות הדבר היא כי כל הקצאת זיכרון חדשה לווקטור שנבקש ממערכת ההפעלה תהיה גדולה **ממש** מכל יתר פיסות הזיכרון שהקצנו לווקטור בעבר **ביחד**. מכאן שמערכת ההפעלה לא תוכל לעולם "למחזר" את ה־deallocated memory ששיחררנו בעבר, שהרי גם כולו **יחדיו** לא מספיק לגודל החדש שנבקש. לכן, בלית ברירה, מערכת ההפעלה תצטרך "לזחול" קדימה בזיכרון ולבקש זיכרון חדש. מערכת ההפעלה לא תוכל לנצל את פיסות הזיכרון שעברו deallocation בשלבים קודמים, "לחזור אחורה" ולנצל אותן.
החישוב המלא מוביל לכך שבחירת $m < 2$ תבטיח שנוכל בשלב **כלשהו** לעשות שימוש חוזר בזיכרון ששחררנו. לדוגמה, אם נבחר $m = 1.5$ כפרמטר הגדילה נוכל להשתמש שוב בזיכרון שעבר deallocation לאחר 4 "הגדלות"; בעוד אם נבחר $m = 1.3$ נוכל לעשות שימוש חוזר בזיכרון ששוחרר בעבר לאחר 2 "הגדלות" בלבד.

7.3 ניסיון שלישי – $\phi(s) = \left\lfloor \frac{3\cdot(s+k)}{2} \right\rfloor$

המסקנה של שתי הטענות לעיל היא שנרצה לבחור פרמטר גדילה בטווח $1 < m < 2$. מצד אחד, ככול ש־m קרוב ל־1 מספר הפעמים שנוכל "למחזר" זיכרון ישן תגדל; אך כמות הפעמים שנאלץ לבצע הקצאות מחדש תגדל ולכן זמן הריצה יארך. מנגד, בחירת m שקרוב יותר ל־2 תשפר את זמני הריצה אך תמזער את כמות הפעמים שנוכל "למחזר" זיכרון ישן.
ניתן להוכיח מתמטית (נימנע מלעשות זאת כאן) שפרמטר הגדילה האופטימלי קרוב לערך של יחס הזהב, קרי $\frac{1+\sqrt{5}}{2} \approx 1.618$. מטעמים אלו, במימוש שלנו נבחר בערך 1.5 שהוא יחסית קרוב ליחס הזהב ופשוט לחישוב. בערך זה עושים שימוש במימושים רבים (למשל במימוש של `ArrayList<T>` ב־Java, המקביל לווקטור). נגדיר, אפוא, את ϕ בתור: $\phi(s) = \left\lfloor \frac{3\cdot(s+k)}{2} \right\rfloor$.

7.4 מסקנות

נגדיר את פונקצית הקיבולת $cap_C : \mathbb{N} \cup \{0\} \times \mathbb{N} \to \mathbb{N}$, עבור $size \in \mathbb{N} \cup \{0\}$, כמות האיברים **הנוכחית** בווקטור, $k \in \mathbb{N}$ כמות האיברים שנרצה להוסיף לווקטור בפעולת ההוספה ו־$C \in \mathbb{N} \cup \{0\}$, פרמטר (קבוע) הזיכרון הסטטי המקסימלי שזמין לווקטור, בתור:

$$cap_C\,(size,\, k) = \begin{cases} C & size + k \leq C \\ \left\lfloor \frac{3\cdot(size+k)}{2} \right\rfloor & otherwise \end{cases}$$