סיכום C

28 ביוני 2021

1 שבוע 1:

1.1 הרצאה 1:

- בשפת c אנו אמורים להקצות את הזיכרון שאנו רוצים להשתמש בו.
- בשביל להדפיס אנו צריכים לייבא את הספרייה של ההדפסות כך: $\#include\ <stdio.h>$
- כל תכנית חייבת לכלול פונקציית $int\ main()$ שחייבת להחזיר ערך כלשהו.
- אנו מדפיסים בעזרת $printf()$.
- בדומה לג'אווה אנו נצטרך להגדיר שדות לפונקציות $int, void, flout$ וכולי.
- עבור כל אחד מהשדות: $1\ byte = char \leq short \leq int \leq long$ יש גם גירסה חיובית $unsigned$ עבור מספרים חיוביים בלבד.
- בכדי לגלות מה הגודל בזיכרון ששמור למצביע, נשתמש ב: $sizeOf()$. ערך ההחזרה שלו הוא lu המסמן $unsigned\ long$.
- $char$: ניתן להשתמש בו גם עם אותיות בעזרת טבלת $ascii$, כאשר הערכים נעים בין $A = 65$ ל $z = 122$ ונתחום אותו ב $'A'$. המספר המקסימלי שלו הוא 255 .
- כאשר נרצה להכניס ערכים ל $printf$ נוכל להשתמש ב $\%$ ואחריו להכניס d ל int. c ל $char$ וכו..
- עבור $signed$: אנו שומרים bit אחד שיסמן לנו את הסימן.

1.2 הרצאה 2 :

- $\boldsymbol{integer\ overflow}$: אם נכניס למשתנה מספר שאינו בטווח שלו, נקבל $overflow$.

- ***casting***: ניתן לעשות $cast$ למשתנים כך: $float\ x\ = (float)z/y$ וכך נוכל לשנות z, y שמוגדרים כ int ל $float$.
אך אם נשמור את x ל int נקבל עיגול כלפי מטה כ int.

- **ערכים בוליאנים:** בשפה אין לנו ערכים בוליאנים ולכן עבור $false$ נכתוב 0, ועבור $true$ נכתוב כל מספר שהוא לא 0.

- אם נרצה פונקציה שלא מחזירה ערך נוכל להשתמש ב $void$. פונקציה כזו נקראת פרוצדורה.

- בשביל לקרוא פונקציות אנחנו נצטרך להגדיר אותן קודם. אם נקרא לפונקציה שטרם הוגדרה, נקבל שגיאה.
ניתן לפתור את הבעיה ע"י $decleration$ כך שנכריז על הפונקציה לפניכן, אך נשאיר אותה ריקה. ואחרי הקריאה נכתוב את הפונקציה המלאה.

1.3 תרגול 1:

- ***const***: מילה שמורה שממירה לנו שגיאות זמן ריצה לשגיאת קומפלציה, על ידי זה שאנו מגדירים את המשתנה להיות קבוע כך שערכו לא ישתנה, בדומה ל $final$ ב $java$.

2 שבוע 2:

2.1 תרגול 2:

- **מצביעים:** כדי ליצור מצביע שמצביע למשתנה אחר , נוסיף ־ $*$ כך: $int^*p = \&y$ כך למעשה הצהרנו ש p הוא פויינטר שמצביע על הכתובת של y .
$*$: יש לה שני שימושים: **1:** הגדרת מצביע על ידי שמירת כתובת המשתנה למצביע.
2: דירפרנס, כלומר ־ אם יש לנו מצביע ואנו מאתחלים את הערך שלו למשתנה, אזי המשתנה יקבל את הערך שהמצביע מחזיק. כך: $int\ z =^* p$. וכך z יקבל את הערך שהפויינט p מחזיק
אם נכתוב $^*p = 4$ אזי הערך שעליו p מצביע ישתנה ל 4 (אך הכתובת תישמר).
$\&$: מעתיק את הכתובת של המשתנה.

- **מערכים:**
בשפת C יש שני סוגי מערכים ־ סטטיים ודינמים. למערכים סטטיים יש גודל קבוע **שלא יכול** להשתנות או להיקבע בזמן ריצה, אלא בזמן קומפילציה בלבד.
כיצד נצהיר על מערך: $int\ name[siza]$ הסוגריים המרובעות מסמנות שזהו מערך סטטי.
דרך נוספת להצהיר והגדיר את המערך היא כך: $int\ name[siza] = \{val1, val2, ...\}$ (במקרה זה אין צורך להגדיר $size$).
בנוסף: אי אפשר לערבב סוגי משתנים שונים, והמערך הנ"ל הוא מערך של int בלבד.
הערה: לא ניתן לגשת למערך כך ־ $arr[-1]$. בנוסף אם ננסה לגשת לתא מחוץ לגבולות המערך אנו לא נקבל שגיאת קומפלציה.
$sizeOf()$**של מערך:** נקבל את כמות התאים כפול כמות הבטים של כל תא. **עבור פויינטר למערך** ־ נקבל את גודל הפויינטר.

מערכים ופונקציות: כל פונקציה שמקבלת מערך, מקבלת אותו כפויינטר ולכן אם נרצה לדעת את גודל המערך, אנו נצטרך לשלוח לפונקציה גם את גודל המערך.

מערך דיפולטיבי: בסקופ גלובלי - אם נגדיר מערך ללא ערכים הוא יאותחל לערך הדיפולטיבי ($int = [0, 0...]$ וכו..).
לעומת זאת אם נאתחל כך **בסקופ לוקלי** - נקבל ערכי זבל (ערכים שאנו לא יודעים מה הם) ולא מערך של אפסים.
בכדי לאתחל מערך של אפסים נשתמש בסינטקס הבא: $int\ name[size] = \{0\}$ (עבור מספר שונה מ 0 זה לא יעבוד)

מצביע למערך: ניתן לשים מצביע שיצביע על האיבר הראשון במערך כך: $*p = &arr[0]$, ניתן להגיע כך גם לאיבר הבא במערך ע"י שימוש ב $\langle p+1 \rangle$ וכן הלאה.
ניתן גם להצביע כך: $p = arr$ בשני האופנים נקבל את אותו המצביע לאותו המערך.

- **סטרינגים:** בשפת C סטרינגים הם מערכים של $char$, ונאתחל אותו כך: $char\ str[] = "txt"$.
  או בעזרת פויינטר, כך: $char^*\ str = "txt"$ ההבדל הוא שאת הפויינטר אי אפשר לשנות וזוהי שגיאת זמן ריצה.
  מאחורי הקלעים: בסוף הסטרינג, בזיכרון המחשב יופיע התו "" המסמן שזהו סוף הסטרינג. לכן כל סטרינג יצטרך 1+ מקום בשביל התו הנ"ל
  ***const*:** בגדול כשנראה את המילה $const$ נסתכל שמאלה, ואם אין כלום משמאל אז נסתכל ימינה, וכך נדע על איזה ערך היא שומרת.
  יש שתי אופציות לשימוש ב $const$:
  לערך: נשתמש בסינטקס הבא: $int\ const\ ^*p$. וכך הערך int נשמר ואי אפשר לשנות אותו.
  לפויינטר: נשתמש בסינטקס הבא: $int^{*}\ const\ p$. וכך המצביע ישמור על הערך ולא יוכל להצביע על ערך אחר.(כי משמאל למילה נמצאת הכוכבית שמסמלת את הפויינטר).

- **קבצים** I/O:
  $stdin$ זו המערכת שדרכה אנו מקבלים קלט מהמשתמש.
  קבלת קלט מהמשתמש: ניתן לקבל בעזרת הפונקציה $getchar()$ שמחזירה $char$ מ $stdin$.
  או בעזרת הפונקציה $sscanf()$ עבור קלטים מסובכים יותר מאשר $char$ כגון סטרינג .
  הדפסה ל $stdout$: ניתן להדפיס בעזרת $putchar()$.
  $\boldsymbol{fgets(array, arraysize, fileToReadFrom)}$**:** זוהי פונקציה המקבלת מערך ואת גודלו בכדי לכתוב לתוכו את התוכן של הקובץ שהוא הפרמטר השלישי. (אנו צריכים ליצור מערך ריק לפני, בכדי לשמור לתוכו את התווים).
  פתיחת קבצים: נשתמש בפונקציה $fopen(path, "commend")$ ואנו שולחים לה נתיב ומקבלים $File^*$ של קובץ פתוח שאנו יכולים לעבוד איתו.
  הפונקציה הנ"ל מקבלת משתנה של פקודה : w, a, r כאשר: r - מסמלת קריאת הקובץ בלבד. w - דריסת קובץ וכתיבת קובץ חדש, או פתיחת קובץ במקרה שהוא לא קיים. a - אם יש קובץ קיים - נוסיף לסוף הקובץ הקיים **ולא** נדרוס אותו.
  אם יש בעיה בפתיחת הקובץ אזי $fopen()$ תחזיר לנו $NULL$.
  סגירת קבצים: בכדי לסגור קובץ נשתמש ב $fclose()$.
  קריאת וכתיבה לקבצים: קריאה: $fgetc(buffer,\ buffeSize,\ file)$ כאשר לתוך המשתנה הראשון נכניס באפר ריק שהפונקציה תקרא את הקובץ לתוכו. במשתנה השני נחזיק את גודל הבאפר ובמשתנה השלישי את הקובץ שממנו

נקרא. **כתיבה:** $fprintf()$ או $fputs()$. **העברת קבצים:** $fseek()$ או $rewind()$.

קבלת פרמטרים משורת הפקודה: את פרמטרים נקבל משורת הפקודה באופן הבא: $int\ main(int\ argc,\ char\ ^*argv[])$ כאשר $argc$ - כמות הפרמטרים שהתקבלו. ו $^*argv[]$ מערך של $char$. כאשר הערך הראשון הוא שם התכנית ולכן $argc \geq 1$ תמיד.

2.2 הרצאה 3:

- הזיכרון במחשב מנוהל בצורת מחסנית. עבור כל פונקציה שנגדיר נקבל מחסנית שתשמור עבור כל משתנה את המקום שלו, לפי הסדר.
- **מערך דו מימדי:** ניתן לכתוב בשני אופנים: $int\ arr[2][3] = \{\{2,3,4\},\{3,4,5\}\}$ או $int\ arr[2][3] = \{2,3,4,3,4,5\}$. בשתי השיטות הקומפיילר יודע שזה מערך דו מימדי.
- בשפת C לא ניתן להגדיר שני מערכים a, b ולעשות השמה $a = b$ משום שמערכים הם כתובות - שגיאת קומפלציה. בנוסף השוואת מערכים $a == b$ תחזיר לנו $false$ משום שנשמרים מצביעים, לכן צריך להשוות איבר איבר.
- כאשר אנו רוצים לאתחל מצביע שלא מצביע לכלום, נגדיר אותו בתור $null$.

3 שבוע 3:

3.1 הרצאה 4:

- $\boldsymbol{NULL\ POINTER}$**:** יש להיזהר לא להשים ערך לתוך מצביע, אלא לשים את הערך $NULL$. לאחר מכן נוכל להשתמש בו ולבדוק האם הערך שווה ל $NULL$ או השתנה.
- **הערה:** כאשר אנו מקדמים מצביע אנו צריכים לשים לב לאיזה ערך הוא מצביע. אם הוא מצביע ל int הוא יתקדם 4 בטים ואם הוא מצביע ל $char$ אנו נתקדם בייט אחד.
- **מערכים ומצביעים:** ניתן להגדיר מצביע p למערך, ולאחר מכן לגשת למערך על ידי $p[i] = val$ או על ידי $*(p+i) = val$.
- **העברת מעריכים ומצביעים לפונקציות:** כאשר אנו מעבירים לפונקציה מערך - הפונקציה מתייחסת אליו כאל מצביע למערך.
- **השמת משתנה בתוך סוגריים:** כאשר אנו עושים השמה למשתנה בתוך סוגריים, הסוגריים מקבלים את הערך של המשתנה בצד שמאל של ההשמה.
- $\boldsymbol{struct}$: מבנה נתונים שאנו מכינים לפי דרישה, ונאתחל אותם כך: $stryct\ name\{type\ x;\ type\ y;\};$ וזהו למעשה מבני נתונים המכיל שני אינטים - x, y. וכך למעשה ניתן לשמור כמה משתנים (גם מסוגים שונים) באופן רציף בזיכרון. **אתחול משתנה חדש:** כשנרצה לאתחל משתנה חדש ב $struct$ נגדיר אותו כך $struct\ nameOfstruct\ typeName\ =$ ... .

הערה: ניתן להציב $struct$ בתוך $struct$.

העתקה של סטראקט: מתרחשת בייט אחר בייט כל שדה (כולל מערכים). שינוי ההעתק לא משנה את המקור.

$\boldsymbol{vecEquals}()$: בכדי להשוות שני סטראקטים נצטרך להשתמש בפונקציה הנ"ל.

- $\boldsymbol{typedef}$: מילה שמורה שמאפשרת לנו להגדיר טיפוסים חדשים ולקצר אחכ ולא לכתוב את כל הטייפ מחדש כך $typedef\ type\ name$ ועכשיו כשנקרא ל $name$ הוא יקרא ל $type$ בלי צורך להגדיר אותו מחדש.

- **זיכרון דינמי:** כאשר אנו צריכים זיכרון אנו יכולים לקחת זיכרון מה $heap$ של המערכת ולשמור אותו שישתחרר בסוף זמן הריצה.

  יתרונות:

  1: ניתן לשחרר אותו תוך כדי זמן ריצה.

  2: אנו יכולים לשלוט מתי ואיפה לממש את הזיכרון.

  3: אנו יכולים להתאים את התכנית לפי כמות הזיכרון שיש לנו בכל מחשב ומחשב.

$\boldsymbol{malloc(int\ size)}$: זאת פונקציה המחזירה לנו פויינטר $void^*$ למקום בזיכרון שנמצא ב $heap$ ואנו נקבל אותו לפי בייטים (ולא $char, unt$ וכו). במקרה של חוסר מקום נקבל $NULL$.

הסינטקס הוא: $type\ *p\ = malloc(sizeOf(*type) \cdot length)$

$\boldsymbol{realloc(array, size)}$: פונקציה שלוקחת מערך קיים ומשנה את גודלו תוך כדי זמן ריצה. ומחזירה לנו מצביע חדש (קיימת אופציה שהמצביע ישתנה).

אם הפונקציה נכשלת היא מחזירה $NULL$. ואז היא תתנהג כמו $malloc$.

אם נכניס לערך $size$ את הערך 0 - היא תעבור כמו הפונקציה $free$.

$\boldsymbol{free(void^*\ p)}$: פונקציה המשחררת את הזיכרון אחרי השימוש ב $malloc$.

3.2 הרצאה 5:

- $\boldsymbol{segmentetion\ fault}$: שגיאה המתרחשת כאשר אנו ניגשים לזיכרון שאסור לנו להגיע אליו - לקרוא או להשתמש בו.

- **מחסנית - $\boldsymbol{stack}$:** המחסנית ממומשת בשיטת $FIFI$. ונשמרים בה משתנים לוקאלים, פונקציות שקראו לנו וכולי. למעשה זהו זיכרון מקומי, של פונקציות שאנו משתמשים בהן והוא נשמר עד סוף זמן השימוש.

  שגיאת זיכרון במחסנית תוגדר כ $stuck\ overflow$.

- $\boldsymbol{data\ sigment}$: זהו מקום בזיכרון ששומר משתנים גלובלים או סטטים, בנוסף סטרינגים גם ישמרו במקום זה (זו הסיבה שלא ניתן לשנות רק חלק מסטרינג).

- $\boldsymbol{static}$: כאשר אנו מגדירים משתנה כסטטי, הוא לא ישמר במחסנית ולא יעלם בסוף ריצת הפונקציה.

- **משתנים גלובלים וסטטים:** מקבלים ערך דיפולטיבי באתחול.

- **סטרינגים:** אם נגדיר סטרינג כמערך, כך: $char\ str[] = "someStr"$ - תתבצע העתקה שלו למחסנית בנוסף למיקום בדאטה סייגמנט.
  אך אם נגדיר כך: $char^*\ str = "someStr"$ הסטרינג ישמר בדאטה סייגמנט, ומצביע לשם ישמר במחסנית.
  לכן תמיד באופן השני, נעדיף לשמור את הסטרינגים כמשתנה $const$ כדי שיהיו לנו שגיאות קומפילציה ולא ריצה.
  $\boldsymbol{strcpy(str1, str2)}$: פונקציה להעתקת סטרינג.

3.3 תרגול 3:

- **מערכים ופונקציות:** כשאנו מעבירים מערך לפונקציה מתקבל לנו ויינטר, לכן תמיד נעדיף לשלוח לפונקציה את גודל המערך גם כן.

- **אריתמטיקה של פויינטרים:** אם נשמור פויינטר ונגדיר אחכ $p++$, הפויינטר יתקדם לפי גודל המערך (לדוגמה - 4 בייט עבור int) ולא רק בייט אחד.
  היא עובדת רק בין פויינטר ו int, או בין שני פויינטרים מאותו הסוג.
  הערה: על $void*$ לא מוגדרת אריתמטיקה.

- **הערה:** כשאנו מקצים זיכרון דינאמי תמיד נבדוק שהוא לא שווה ל $NULL$ והמערך אכן הוקצה, ונתקדם רק במקרים אלו. (ניתן להכניס ערך $NULL$ ל $free$).

- $\boldsymbol{calloc()}$: פונקציה שמקצה מערך דינאמי, אך בשונה מ $malloc$ מאתחלת מערך של אפסים.

4 שבוע 4:

4.1 הרצאה 6:

- **דירפרנס למצביע של סטרקט:** אם אנו רוצים לגשת לאיבר בסטרקט דרך פויינטר. נוכל לכתוב באופן הבא: $pointer \rightarrow var1[]$.

- **העתקת ערכי סטרקט:** כאשר אנו מעתיקים ומשתמשים בסימן "=" אנו נעתיק ערכים בלבד.

- $\boldsymbol{clonVec(vec\ *a,\ vec\ *b)}$: פונקציה שמעתיקה לנו את כל המערך. (אם אנו רוצים להעתיק מערך נצטרך להעתיק איבר איבר).

- **מצביע למצביע:** אם נגדיר משתנה עם שתי כוכביות, הוא ישמש כמצביע למצביע.

- **יצירת מערכים רב מימדיים:**
  1 - מערך סטטי: יושב ב $stack$, נגדיר כך - $int[size_1][size_2]$. החסרון הוא שאנו צריכים לדעת את הגודל מראש
  2 - מערך חצי דינמי: יושב ב $heap$, ראשית ניצור מערך סטטי שישב ב $stack$, ואחכ ניצור תת מערכים דינמים בעזרת $malloc$. נכתוב לולאה בגודל מספר המערכים שאנו רוצים, ועבור כל i ניצור מערך עם $malloc$. בכדי לגשת לאיבר במערך נצטרך שתי גישות.

3 - מערך דינמי: נגדיר כך $type \quad **arr$ ואז בדומה למערך חצי דינמי נרוץ בלולאה כאשר המערך הראשון הוא מערך דינמי של מצביעים לשאר השורות. כאן בכדי לגשת לאיבר במערך נצטרך שלש גישות, לכן דרך זו פחות יעילה מבחינת יעילות, אך רווחית מבחינה דינמית כי כך ניתן להגדיר מערך רב מימדי בזמן ריצה.
בשחרור - נשחרר את השורות ורק אח"כ נשחרר את מערך המצביעים לשורות.
4 : ניצור מערך חד מימדי דינמי, אשר מייצג לנו מערך דו מימדי. החסרון הוא - הגישה היא יותר מסובכת. נגדיר כך $type \ *arr = (type*)malloc(size_1 \cdot size_2 \cdot Sizeof(type))$.
וניגש אליו כך: $arr[i][j] \rightarrow arr[i8ncols + j]$.

- **שלשה מצביעים:** אם נרצה ליצור מערך של מטריצות, נצטרך לשמור מצביע עם שלש כוכביות.

4.2 הרצאה 7 - $Preprocesor$:

- **סולמית - #**: כאשר אנו רוצים לתת פקודה לפרה־פרוסור אנו צריכים לשים לפניה "#", כך לדוגמה הקומפיילר מבין להחליף את הקבוע במספר שהוגדר לו.
הפקודות הן $define, include$ ועוד.

- $MACRO$: כאשר אנו רוצים להגדיר פעולה שחוזרת על עצמה כמה פעמים, נוכל להגדיר אותה בראש הקובץ כך $\#define\ name(x)\ actionOnX$ ואז נוכל להשתמש במאקרו כפונקציה.

- $\boldsymbol{enum\{NAME = val\}\backslash const\ type}$: מנגנון של השפה בכדי להגדיר קבועים.

- $\boldsymbol{\#if}$: קומפילציה על תנאי. דרך זו מאפשרת לנו להדפיס רק אם תנאי כלשהו מתקיים.

- $\boldsymbol{assert()}$: בעזרת פקודה זו אנו יכולים לבדוק שהפונקציה מתבצעת, רק אם פקודת ה $assert$ מתקיימת.
אנו צריכים להוסיף את תקיית $\#includ\langle assert.h\rangle$

- $var = Exp_1?Exp_2 : Exp_3$: זהו תנאי כמו תנאי if : אם תנאי 1 מתקיים - אז המשתנה שווה לתנאי 2. אחרת - המשתנה שווה לתנאי 3 .

4.3 תרגול 4 - $VALGRIND$:

- $valgrind$: תכנה שבודקת האם יש דליפות זכרון בתכנית שלנו.

- **כיצד נשתמש:** נכתוב את הפקודה הבאה - valgrind /path/to/your/program arg1 arg2 (התכנה לא עובדת עבור ווינדוס).

- **פלט התכנית:** אנו צריכים לחפש את $lake - summery$ ואם המספרים אינם אפסים - יש דליפת זכרון.

- **כיצד נגלה מאיפה הדליפה:** בסוף התכנית תופיע שורת $use : -g$, נעתיק אותה לתחילת התכנית שלנו. הפלט שייצא יציין את המיקום המדוייק של דליפת הזכרון - היכן הוקצה הזיכרון ששכחנו לשחרר ומאיפה קראנו לו.

- כדי לבדוק ביעילות אנו צריכים להריץ את הטסטים עם $valgrind$.

5 שבוע 5:

5.1 הרצאה 8:

- **סוגי שגיאות:** ישנם שגיאות שהם באגים - שגיאות קוד. ויש שגיאות $exeptions$ שן שגיאות כתוצאה מקלט או זיכרון.
- $assert$ - **התמודדות עם באגים:** נוכל להשתמש במילה השמורה $assert$ בכדי להתמודד עם באגים (אך לא נשתמש בה לדגיאות קלט).
  החסרון: אנו יוצאים מיידית מהתכנית, ואין לנו זמן לטפל בשגיאה.

$\boldsymbol{stder}$: כאשר יש לנו $error$ אנו נדפיס ל $standart-error$. יש 3 פונציות שאפשר להשתמש בהן: $fprintf(), perror(), strerr()$.

- **טיפול בשגיאות:** 0 - מסמן הצלחה, אחרת, נחזיר 1 או 1- . בפונקציה שמחזירה מצביע: כשיש כשלון נחזיר $NULL$.
- $\boldsymbol{exit}()$: פונקציה המאפשאת לנו לצאת באמצע התכנית. אנו נעדיף לא להשתמש בה אלא לפעפע שגיאה ל $main$.
- $\#include\ <errno.h>$: זהו int שמוגדרת בספרייה והוא משתנה גלובלי של שגיאות של הספריות הסטנדרטיות (פתיחת קבצים וכו), ואנו יכולים להשתמש בו כשאנו רוצים להדפיס שגיאות.
  נדפיס כך: $fprintf("\%s", strerror(errno))$ ואם נקבל שתי שיאות הוא ידפיס את האחרונה.
- **שלבי הקומפלציה: 1:** $preprocessor$ - עורך את הטקסט. **2:** $compiler$ - הופך את הקוד עם סיומת c לקוד מכונה עם סיומת o. **3:** $linker$.
- **קובץ** $header$: זהו קובץ $stack.h$ שבו נשים את כל ההכרזות על הפונקציות, וכך יהיה לנו יותר קל לקמפל את הקוד. את המימוש נבצע ב $stack.c$.
  נצטרך לכלול את הקובץ $stack.h$ בכל הקבצים שאנו רוצים להשתמש בו . בכדי לכלול אותו נשתמש ב $\#ifndef$ ואחכ נעשה לו $\#define$ כדי שלא יוגדר לנו פעמיים (אחרת נקבל שגיאת קומפלציה).
- **שגיאות של** $linker$**:** שגיאה על פונקציה שחסרה. פונקציות כפולות.
- $\boldsymbol{make}$: כלי לקמפול של מספר קבצים. ב $clion$ אנו משתמשים בו בעזרת $cmake$.
- **ספריות:** יש את הספריה הסטנדרטית, ספריות גרפיות וכו.

יש שני סוגים של ספריות: $static$: ספריה שאנו מצמידים ל $executable$ שלנו בזמן קומפילציה. $shared$: עובדות רק בזמן ריצה ולא מוצמדות ל $executable$.

יתרונות של ספריה סטטית: אחרי הקמפול אנו יכולים לשכוח אותה ולא צריכים לדאוג לה יותר.

יתרונות של $shared$**:** מאפשרת חלוקת קוד. מקטינה את $executable$ כי הספריה לא מוצמדת אליו. אין צורך לקמפל מחדש את הספריה.

5.2 תרגול 5:

- **מערכים דו מימדיים בפונקציות:** יש כמה דרכים:
  $void\ foo(type\ name[][size])$
  $void\ foo(type\ *name[])$
  $void\ foo(type\ **name$

- **מערך דינאמי דו מימדי אפקטיבי:** נגדיר מערך רציף בזיכרון באורך שנרצה, כך: $size_1 \cdot size_2 \cdot sizeOf(type)$ ואז כשנרצה לגשת לאיבר מסויים במערך ניגש כך $arr[i \cdot ncols + j]$.
  כך מעשה אנו חוסכים והופכים שלש גישות זיכרון לאחת. אך החסרון הוא שהקוד פחות קריא.

- **כלל ימין שמאל:** זהו כלל שעוזר לנו לפרש ביטויים עם פויינטרים. אנו מסתכלים על הביטוי ומתחילים מהשם של המשתנה, אח"כ נביט ימינה ואח"כ שמאלה. אם אנו נתקלים בסוגריים "(" אנו מסתכלים שמאלה, ואותו הדבר לגבי צד שמאל. אחכ נחזור ימינה ונמשיך אחרי הסוגר ")". וכן הלאה וכן הלאה.. וכך נוכל לפרש את הביטוי.

5.3 הרצאה 9:

- **יצירת ספריה:** בכדי לייצר ספרייה אנו צריכים לכתוב ar.

- **משתנים סטטים:** נגדיר בעזרת המילה השמורה $static$. ותלוי איפה הוא מוגדר:
  מחוץ לפונקציה - משנה סטטי הוא משתנה שרואים אותו רק בתוך אותו קובץ, הוא לא גלובלי.
  בתוך הפונקציה - משתנה סטטי מאותחל פעם אחת בתחילת התכנית, ולא בכל קריאה לפונקיציה.

- משתנה סטטי יושב במקום נפרד בזיכרון. והוא נגיש רק בתוך המקום שבו הוא מוגדר.

- בדיפולט, משתנה סטטי שלא מאותחל מוגדר להיות 0.

- ***module***: קובץ ה c שלנו.

- ***exstern***: כאשר אנו רוצים להכריז על משתנה בקובץ h נשתמש במילה $extern$, ונכתוב אותה לפני המשתנה בקובץ h.

6 שבוע 6:

6.1 הרצאה 10:

- ***Generic Progreming*** : כאשר אנו רוצים לבנות מבנה נתונים גנרי שקובע את סוג ה $data$ בזמן ריצה, נשתמש ב $void*$. בנוסף נשמור מצביע לגודל ה $data$ וכך נדע כמה זיכרון להקצות בכל פעם, (משום שאנו לא יודעים מה סוג הדאטה).

- **מצביעים לפונקציות:** זה **לא** משתנה, מצביע לפונקציה הוא מצביע לכתובת שבה הפונקציה שמורה בזכרון המחשב. זה שימושי כאשר יש לנו כמה מצבים ובכל מצב אנו צריכים לקרוא לפונקציה אחרת. דוגמה נוספת - חישוב אינטגרל נומרי.
כיצד נממש: עבור פונקציה $func$ שמקבלת int ומחזירה $void$. נגדיר את המצביע כך: $void\ (*funcPtr)(int) = \&func;$ (לא חייבים להשתמש ב &). בכדי לקרוא לפונקציה נממש כך: $(*funcPtr)(val)$ (לא חייבים להשתמש ב $*$ לפני הפויינטר).

- **ארגומנטים של פונקציה:** כאשר אנו משאירים את שורת הארגומנטים ריקה, הקומפיילר לא יודע כמה ארגומנטים הפונקציה מקבלת. לכן אם אנו רוצים לכתוב פונקציה שלא מקבלת ארגומנטים כלל, נשים בשורת הארגומנטים את המילה $void$.

6.2 הרצאה 11:

- **מצביע לפונקציה:** זה מצביע לקוד ולא לדאטה, אין התעסקות בהקצאות זיכרון והמצביע הוא השם של הפונקציה. בדומה למצביע רגיל ניתן להעביר מצביעים בין פונקציות ולשמור אותם במערכים.

- ניתן להשתמש ב $typedef$ כדי לחסוך בכתיבה של מצביעים לפונקציה.

- $\boldsymbol{MAP}$: פונקציה שמקבלת רשימה ופונקציה, ומחילה על כל הרשימה את הפונקציה.

- $\boldsymbol{qsort}()$: פונקציה שמקבלת מערך, גודל של המערך, גודל של כל איבר ופונקציית $compare()$. וממיינת לנו את המערך ב $quick\ sort$. אנו צריכים להגדיר את פונקציית ההשוואה שלנו לפי הערכים שאנו רוצים למיין - $int, char...$.

- **מצביעים לפונקציה בתוך סטראקט:** נוכל לשים משתנה בסטראקט שמקבל מצביע לפונקציה, ואח"כ נשתמש במצביע באופן ספציפי לפי השימוש הנצרך.

- $\boldsymbol{enum}$: דרך נוספת להגדיר קבועים, זה מבנה נתונים שמקבל ערכים קבועים. נגדיר כך: $enum\ name\ \{var1, var2...\}$. ניתן לתת לקבועים שונים את אותם ערכים. אם לא נאתחל קבוע מסויים - הוא יקבל את הערך הקודם 1+ .

6.3 תרגול 6:

- $\boldsymbol{void*}$: כדי לממש אותו נצטרך לעשות $cast$, לא נוכל לממש דירפרנס, ולא נוכל להחזיק בו מצביע לפונקציות.

- $\boldsymbol{NAN}$**:** כאשר אנו משתמשים ב $flout$ קיימת האפשרות שנקבל ערך שאינו מספר - NAN, ניתן לבדוק זאת עם המאקרו $isnan(arg)$ שיחזיר לנו 0 אם המספר אינו NAN.

- $bool$: קיימת ספרייה שניתן לכלול אותה ולהגדיר דרכה משתנים בוליאני ב c.

6.3.1 סטריגים $< string.h >$:

- כל הפונקציות הללו צריכות לקבל בסוף המערך $\backslash 0$ כדי שהן יידעו שזה סטרינג.
- $\boldsymbol{strcpy(dest\ src)}$: פונקציה המעתיקה את המקור ליעד, אנו צריכים לתת מקום ליעד לפני ההעתקה.
- $\boldsymbol{strcat(dest,\ src)}$: מחברת את המקור לסוף של היעד. - מחזירה פויינטר למקום האמיתי של היעד.
- $\boldsymbol{strcmp(str1,\ str2)}$: פונקציה שמשווה בין שני סטרינגים ומחזירה 0 במצב של שוויון.

7 שבוע 7 - cpp:

7.1 הרצאה 12:

- $\boldsymbol{extern\ varType}$: מילה שמורה שעושה $declare$ ומיידעת את הקומפיילר שהמשתנה קיים. אם נאתחל אותו אזי גם יהיה $define$.
  אנו יכולים לכתוב $extern\ int\ x$ מספר פעמים בקובץ משום שהוא מוגדר במקום ספציפי בזיכרון.
- $\boldsymbol{switch}$: לולאה המקבלת ערך ומכניסה אותו ל $case$ המתאים, לבסוף יש לנו $deafult$ שיתבצע במקרה הדיפולטיבי.
- $\boldsymbol{xor}$: אופרטור ^ המייצג שרק אחד מהתנאים מתקיים, אם שניהם מתקיימים נקבל $false$.
- $\boldsymbol{0b}$: כשנכתוב כך לפני מספר הוא ייוצג בייצוג בינארי.
- $\boldsymbol{Bitwise\ operator}$: עוברים כל ביט ומשווים לפי האופרטורים הרלוונטים $<<$, $>>$, $|$, $\&$. בשונה מאופרטורים לוגים.
  נשתמש ב $\&$: כדי לבדוק ערך של ביט. וב xor : כדי לשנות ערך של ביט.
- $\boldsymbol{Shift\ left}$: אופרטור: $binary\ num << val$. המייצג הזזת הביטים שמאלה, המספר שכתוב מימין לאופרטור מייצג את מספר ההזזות. (הזזה ב x מדמה הכפלה ב 2^x . כמו הוספה עשרונית שמעלה בחזקת 10).
- **דגלים בביטים:** אנו יכולים לשמור בבייט בודד 8 דגלים בינארים. לכן לפעמים נעדיף דגלים בינארים.
- $\boldsymbol{masks}$: יצירת דגלים והפעלת מניפפולציות עליהם נקראת מסיכה.
- $\boldsymbol{union}$: מאפשר לכתוב באותו המקום בזיכרון בפרשנויות שונות.
- $\boldsymbol{restrict}$: מודיע לקומפיילר שלא נשתמש בפויינטר שלנו בדרך מסויימת, וכך הוא מבצע אופטימזציות שיאפשרו לקוד לרוץ מהר יותר.

7.1.1 cpp:

- ***classes***: אנו יכולים להגדיר מחלקה שבה מה שנגדיר כ $public$ יהיה גלוי לכולם, את ה $class\ members$ נגדיר כ $private$ והם יהיו פרטיים. ברירת המחדל היא $private$.
  בנוסף נגדיר $constractor$ שנושא את השם של המחלקה ומאתחל את ה $class\ members$, הסקופ שלהם הוא כל ה $class$.

- ***const***: ניתן להגדיר פונקציות במחלקה כקבועות כך שלא ישנו לנו את הערך, נכתוב $const$ אחרי שם הפונקציה.

- **הגדרת מטודות:** בכדי להגדיר מטודה שקשורה למחלקה, נכתוב את שם המחלקה לפני הגדרת הפונקציה כך: $class\ name\ ::\ func()$.

- ***struct***: ב cpp סטראקט משמש כמחלקה $public$.

- $class$ יכול לקבל כמשתנה $class$ אחר.

- ***default constractor***: כאשר לא נגדיר קונסטרקטור, הקומפיילר יאתחל לנו קונסטרקטור דיפולטיבי. ניתן גם להגדיר בו ערכים דיפולטיבים למשתנים.

- ***initialize list***: ניתן לאתחל משתנים באופן הבא $int\ x(val)$ וזה שווה ל $int\ x = val$. ניתן גם לאתחל ככה משתנים בקונסטרקטור.

- ***Header initialization***: מאפשר אתחול של משתנה מסויים בקלאס לערך קבוע, ואז בכל פעם שנקרא לקונסטרקטור וניצור אובייקט של הקלאס הזה, הערך שנתנו יאותחל להיות הערך של המשתנה.

- ***constractor delegation***: כאשר יש לנו כמה קונסטרקטורים, כשאנו מאתחלים אובייקט נכתוב אחריו באיזו קונסטרקטור אנו מתכוונים להשתמש (לפני הסוגריים המסולסלים).

7.2 הרצאה 1:

- ***Destructors***: עושה את הפעולה ההפוכה מה $constractor$, הוא הורס ־ משחרר א ת הזיכרון. הוא נקרא אוטומטית בסוף ה $scope$. נסמן אותו כך $\tilde{}ClassName()$.

- ***friend*** : אנו יכולים להכריז שמחלקה A היא חברה של מחלקה B ואז הסקופ של המחלקה A יהיה פתוח בפני המחלקה B, כולל שדות פרטיים. ניתן להגדיר כך גם מטודות ששייכות לקלאס.

- ***this***: זהו פויינטר למחלקה עצמה. ניתן לייחס אותו דם לדאטה וגם לקוד.

- ***static***: משתנה שלא מאותחל בכל קריאה לפונקציה, אלא באתחול האובייקט בלבד.

- **פונקציה סטטית:** לא משתנה בכל אינסטנס, והיא משותפת לכל מי שנמצא בקלאס.

- ***const***: מילה שמורה שנוסיף בסוף של פונקציה, והיא מגדירה שלא נוכל לשנות אף אחד מה *class members*.
(לדוגמה - פונקציית *get*).
אם מטודה מוגדרת ב *const* היא יכולה לקרוא למטודות שמוגדרות כ *const* בלבד.

- ***alias*** – &: כך אנו נותנים שם נרדף למשתנה, נגדיר כך $int\ i\ =\ val$ אחכ $int\&\ ref\ =\ i$ וכך למעשה כשנשנה את ref גם i ישתנה. אנו צריכים לאתחל את המשתנה ולתת לו משתנה אחר משום שזה שם נרדף, אחרת זה לא יעבוד.

7.3 תרגול 1:

- ***namespace***: מילה שמורה, דרך לאגד בקבוצה משתנים ופונקציות. נוכל לארגן כך משתנים תחת שמות - כמו תקיות. נממש כך: $namespace :: operator$.
משתני גלובלי הוא כמו *namespace* ריק, לכן ניגש למשתנה גלובלי כך: $global\ operator ::$

- ***using***: מילה שמורה, אם נשים אותה לפני ה *namespace* נוכל לגשת למשתנה ב *namespace* עם המילה הזו.
הערה: אין להשתמש ב *using namespace name*, אלא ב $using\ name :: varName$

- ***string***: זאת מחלקה המייצגת את האובייקט סטרינג, אין צורך להשתמש ב $char*$, אלא נוכל לכתוב את *string* בתור ה *type* של המשתנה. בנוסף קלאס זה מייתר את השימוש בתו $\backslash 0$. נממש בעזרת $\#includ\langle string\rangle$.
בספרייה זו יש פונקציות השוואה בין סטרינגים ועוד.

- ***innitial list***: אתחול דאטה ממבר בשורה אחת. אתחול כזה מאתחל את המשתנים באותו הרגע שהוא יוצר אותם, בניגוד לבניה דרך קונסטרקטור שמקצה מקום ורק אחכ מאתחל את הערך של המשתנים.
האתחול מתבצע לפי הסדר שמופיע בבנאי **ולא** לפי הסדר שכתבנו אותם בשורת האתחול.

8 שבוע 8:

8.1 הרצאה 2:

- בתוך מטודה סטטית לא ניתן לשים *this* משום שאין לה אף מופע.

- ***class reference***: ניתן להחזיר ממטודה רפרנס למשתנה.

- ניתן לשלב *const* ורפרנס יחד. באותו האופן ניתן לשלב *const* עם *class member*.

8.2 הרצאה 3:

- **רפרנס למשתנה בפונקציה:** ניתן להשתמש ברפרנס למשתנה כדי להשתמש בפונקציה $swap(\&a,\ \&b)$ על ידי השמת רפרנס בפרמטרים של הפונקציה, אנו למעשה נעבוד עם המשתנים עצמם ולא עם עותק שלהם.

- $assert(a == b)$**:** פונקציה זתחזיר שגיאה אם ההשוואה לא מתקיימת.

- **העברה של** $class$ **עם רפרנס:** אנו יכולים לשלוח לפונקציה רפרנס של אובייקט, בנוסף אנו יכולים להגדיר רפרנס לאובייקט כ $const$ כך שנוא לא ישתנה.

- **מבר קלאס** $const$**:** זהו ממבר קלאס שלא ניתן לשנות אותו.

- **החזרת** $const$**:** פונקציה שמוגדרת כ $const$ לא יכולה להחזיר משתנה שאינו $const$, לכן נגדיר את ערך ההחזרה שלה להיות $const$ גם כן.

8.3 תרגול 2:

- $\boldsymbol{stream}$**:** מקשר לנו בין הדאטה שאנו רוצים להדפיס או לקבל, לבין מקום שדורש קלט\פלט.

- **הדפסה:** נבצע באמצעות הפקודה הבאה ל $stream$ כך: $std :: cout << somthing\ to\ print$

- **סוגי סטרימים:** למעשה הם קלאסים שתומכים בין דאטה לתכנה (קריאה או כתיבה).
  $istream$: $input\ stream$ קוראים למקום הזה.
  $ostream$: $output\ stream$ מכניסים מידע לסטרים והוא כותב את זה אח"כ.
  בעיקר אנו נעבוד עם קבצים $of\backslash if\backslash f - stream$ לקריאה או כתיבה לקבצים.
  בכדי לגשת אליהם נצטרך להשתמש ב $std ::$ משו שהם חלק מהספריה הסטנדרטית.

- **כיצד נשתמש ב** $stream$**:** נייבא את הספריה $\langle iostream \rangle$.

- **ירידת שורה:** נממש באמצעות $<< endl$.

- **אינפוט סטנדרטי:** כך נקבל אינפוט מהמשתמש. נממש באמצעות: $cin >> to\ insert$.

- קריאה וכתיבה לקבצים:
  ניצור קובץ באופן הבא: $std :: ofstream\ outFile(fileName, std :: ios :: out);$.
  נקרא מקובץ באופן הבא: נגדיר פתיחת קובץ באופן הבא: $std :: ifstream\ inFile(fileName);$. אחכ נגדיר $int\ i$ ולולאה כך: $while(inFile >> i)$ וכך למעשה נקרא את הקובץ עד שנגיע לסופו.
  לבסוף צריך להשתמש ב $in\backslash outFile.close()$ בכדי לסגר את הקובץ.

- **פתיחת קובץ וכתיבה לסופו:** נשתמש ב $ios :: app$, כך למעשה נכתוב לסופו של הקובץ ולא נדרוס את תחילתו כשנכתוב.
  $ios :: ate$ **:** שם א הסמן בסוף הקובץ, אך אח"כ ניתן להזיז אותו ולכתוב ביכן שנרצה. בניגוד ל appשיחזיר אותנו לסוף כל פעם.

- **שילוב בין** $mods$**:** נוכל לכתוב בכמה מודים - $out\backslash in\backslash app...$ על ידי שימוש באופרטור or בינארי.

- $\boldsymbol{tellg}()$: אומר לנו כמה תווים קראנו והיכן נמצא הסמן.
  $\boldsymbol{seekg(n, location)}$: נותן הוראה לשים את הסמן במקום של ה $location$, ולזוז ממנו n צעדים. (כשנרצה להישאר ב $location$ נשים $n = 0$).

- **פונקציות לבדיקת קובץ:** הפונקציות הבאות יחזירו לנו תנאי בוליאני האם הפתיחה, הכתיבה לקובץ וכו... הצליחה $bad(), fail(), eof()$. לאחר מכן נכבה את הדגלים עם הפונקצייה $clear()$.

9 שבוע 9:

9.1 הרצאה 4:

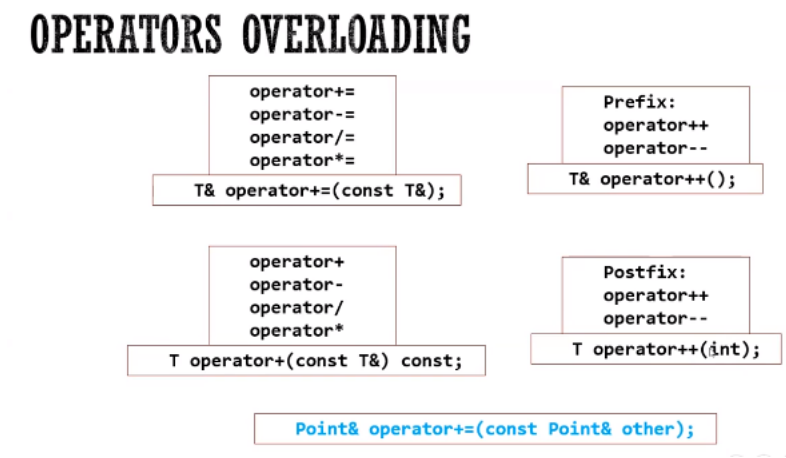
- ***meutable*:** מילה שמורה שנשים לפני משתנה, וכך נוכל לשנות משתנים אפילו אם אנחנו נמצאים בתוך מטודה שמוגדרת כ $const$.
- ***operator overloading*:** המילה השמורה - $operator$, מאפשרת לנו להשתמש באופרטורים גם עבור ערכים שאינם מספרים.

9.2 הרצאה 5:

- ***operator*:** ניתן להשתמש בפונקציה אופרטור בשני אופנים:
  $operatot++()$: מסמנת pre, כלומר - העלאת ערך והחזרת הערך החדש.
  $operatot++(int)$: מסמנת $post$, כלומר - שמירת עותק של המשתנה, העלאת הערך של המשתנה, ושליחת העותק עם הערך ההתחלתי.
- ***copy constractor*:** בנאי שמעתיק את המופע הנוכחי, נממש בעזרת הצבת $(*this)$ בתור פרמטר שהקונסטרקטור מקבל.
- **הקצאת ושחור זיכרון:** אנו נקצה זיכרון ב cpp בעזרת המילה השמורה ***new***. אחרי הקצאת זיכון אין צורך לבדוק אם ההקצאה שווה ל $nullptr$, משו שיחזור לנו $exeption$ אוטומטית במקרה של חריגה. נאתחל מערך כך: $arr = new\ valType[n]$.
  שחרור הזכרון מתבצע באמצעות המילה השמורה $delete$. בכדי לשחרר מערך נכתוב כך: $delete[]\ arrName$.
- **כלל השלש:** הכלל אומר שאם שינינו אחד מהשלשה הבאים: $copy\ constractor, copy\ assignment, defoult\ distractor$ נצטרך לדרוס ולכתוב את שלשתם מחדש.

9.3 תרגול 3 cpp:

- ***operator overloading*:** סיכום של שימוש בדריסת אופרטורים:



באופן כללי: כשנרצה להשתמש באופרטור השמה - אנו רוצים שתהיה לנו היכולת לשרשר כמה דברים יחד. לכן אנו צריכים להחזיר רפרנס.

10 שבוע 10:

10.1 הרצאה 6 :

- ***copy assignment*:** ממומש באמצעות $oporator = (toCopy)$. הפונקציה הזו תעתיק לנו את האובייקט החדש, לאובייקט הקיים לאחר שתמחק את האובייקט הקיים. כאשר אנו משתמשים ב $copy\ assignment$ חשוב לזכור לטפל בהעתקה זל האובקייקט לעצמו, לכן תמיד נבדוק $this! = toCopy$ ואח"כ נמשיך בהעתקה.

- כאשר אנו משתמשים ב " = " באיתחול, מה שיקרא זה ה $copy\ constractor$. בהשמה יקרא ה $copy\ assignment$.

- **חריגות:** כאשר יש לנו חריגות בהקצאת זכרון בעזרת new, החריגה שמוחזרת היא $std :: bad\ alloc$. לכן כשנרצה להקצות זיכרון נוכל לעזות זאת בעזרת בלוק $try - cach$ ולתפוס את החריגה שנזרקת.
  throw: מילה שמורה שנכתוב בתוך הבלוק try, ואז היא תיזרק לנו במקרה של $catch$.

- **אפורטורים נוספים:**
  $(\boldsymbol{index})$***opoerator***[]: מקבלת אינדקס, ומחזירה את מה ששמור במערך במקום של האינדקס.
  $(\boldsymbol{index})$***at***: באותו האופן כמו אופרטור [] , אך היא תזרוק לנו חריגה אם נבקש לגשת למקום בזיכרון שלא מוקצה לנו. במימוש נצטרך לבדוק שהאינדקס לא גדול מגודל המערך, ואם כן - נזרוק חריגה.

10.2 הרצאה 7 - STL:

- ***STL*:** ספרייה המכילה אלגוריתמים ומבני נתונים. היא מכילה קונטיינרים, איטרטורים ואלגוריתמים.

- **קונטיינרים:** יש שני סוגים של קונטיינרים:
  $sequence$ - יש חשיבות לסדר (רשימה. ווקטור: מחליף לנו מערך, נממש כך: $std :: vector < type >$. ווקטור דו כיווני - $deque$: הוא יודע לעשות גם $push\ front$ בזמן של $O(1)$.

$associetive$ - אין חשיבות לסדר (map: נממש כך - $std :: map < type1, type2 >$, החיפוש מתבצע בעזרת חיפוש המפתח, בשביל שהמפה תצליח למיין אנו צריכים לממש את האופרטור "$<$" אם הוא לא קיים, בנוסף כל איבר יכול להופיע פעם אחת לכל היותר. set: כל איבר יכול להופיע פעם אחת לכל היותר. קיים מימוש של מולטי מאפ\סט שלהם ניתן להכניס איבר אחד כמה פעמים. $unorderd\ set\backslash map$: ממיינת באמצעות פונקציית האש).

- **איטרטורים:** מאפשרים לנו לרוץ על פני כל הקונטיינרים בצורה אחידה. למעשה הוא מצביע ונממש אותו כמו פויינטר אריתמטיקס, יש לנו שתי מטודות $begin(), end()$ שמסמנות לשים מצביעים לפני האיבר האשון ואחרי האיבר האחרון בהתאמה. נאתחל כך: $std :: set < type > iterator\ i = setName.begin()$ למעשה כך אנו נותנים פקודה לאיטרטור להצביע לתחילת הסט (ניתן באותו האופן להצביע לסוף).

- **אלגוריתמים:** חלק מתקיית STL ונמצאים בה מספר אלגוריתמים שיכולים לרוץ עם איטרטורים ולבצע פעולות.

- $\boldsymbol{sort(start, end)}$: מקבל התחלה וסוף של איטרטור וממיין אותו, אמור להיות מוגדר לנו היחס "$<$" בכדי שנוכל למיין לפי יחס סדר.

- $\boldsymbol{find(start, end, element)}$: אנו מגדירים טווח התחלה וסוף של איטרטור ואיבר לחפש. הפונקציה מחזירה לנו איטרטור למקום שבו האיבר נמצא, או איטרטור לסוף הקונטיינר אם האיבר לא נמצא.

10.3 תרגול 4:

- כאשר אנו מקבלים $by\ value$ אנו קוראים ל $copy\ constructor$, בניגוד ל $by\ reference$.

- כאשר אנו עושים בלוק $try - catch$ אנו נעדיף לשלוח את ה $exeption$ עם רפרנס.

- כשאנו מעבירים לפונקציה ארגומנט $by\ reference$, הקומפיילר מכין משתנה חדש, ומעתיק לתוכו את המשתנה ששלחנו. כלומר - ברגע שנקרא לפונקציה עם ערך מסויים הוא יועתק אוטומטית והפונקציה תעבוד עם ההעתק.

- $\boldsymbol{what()}$**:** פונקציה שנממש אותה עבור חריגות כך: $e.what()$ והיא תחזיר לנו את החריגה שבגינה נפל הקובץ.

11 שבוע 11:

11.1 הרצאה 8 - איטרטורים:

- **איטרטור קונסט:** כמו באופרטור [], גם באיטרטורים יש לנו גישה והשמה. כדי לממש איטרטור $const$ נשתמש ב $cbegin(), cend()$ במקום $end(), begin()$. נגדיר אותו כקונסט כך: $std :: vector < type >:: const\_iterator$.

- **סוגי איטרטורים:** יש כמה סוגים של איטרטורים, וכל אחד מהם עובד עם אופרטורים אחרים ובאופן אחר:

STL: Iterator Types

	Output	Input	Forward	Bi-directional	Random
Read		`x = *i`	`x = *i`	`x = *i`	`x = *i`
Write	`*i = x`		`*i = x`	`*i = x`	`*i = x`
Iteration	`++`	`++`	`++`	`++, --`	`++, --, +, -, +=, -=`
Comparison		`==, !=`	`==, !=`	`==, !=`	`==, !=, <, >, <=, >=`

- Output: write only and can write only once (one-pass)
- Input: read only and can read only once (one-pass)
- Forward supports both read and write (multi-pass)
- Bi-directional support also decrement (multi-pass)
- Random supports random access (just like C pointer, multi-pass)

19

- ***iterator traits*:** בכדי שאיטרטר יקרא איטרטור סטנדרטי, נצטרך להגדיר לו את התכונות הבאות: סוג הדאטה, מהו הרפרנס, מהו פויינטר, מרחק בין איטרטורים, מאיזו קטגוריה האיטרטור שלנו $(input\backslash output...)$. כולם מסוג $typedef$. כך:

```
class Iterator {
private:
    Node* node_;

public:
    // ITERATOR TRAITS
    typedef int value_type;
    typedef int& reference;
    typedef int* pointer;
    typedef std::ptrdiff_t
        difference_type;
    typedef std::forward_iterator_tag
        iterator_category;
```

ולאחר מכן נממש את המטודות ששייכות לסוג האיטרטור שהגדרנו, מה שמופיע בטבלה לעיל.

- ***cons iterator*:** כאשר נממש איטרטור קונסט, אנו נצטרך לממש $convertion$ שמקבל איטרטור והופך אותו לקונסט איטרטור.

11.2 הרצאה 9 - $Templates$:

- ***overload resolution*:** ב $c++$ אנו יכולים להגדר פונקציות שונות בעלות אותו השם, שכל אחת מהן מקבלת פרמטרים אחרים. הקומפיילר ימפה אותן, ויידע בכל פעם באיזה פונקציה אנו משתמשים.

הערה חשובה: פונקציות בעלות אותו השם **חייבות** להחזיר את אותו טייפ, אם כל אחת מחזירה ערך שונה נקבל שגיאת קומפלציה.

- ***Function template*:** המילה השמורה $template$ מאפשר לנו ליצור פונקציות גנריות שיקבלו כל טיפוס שנשים בה במהלך ריצת התכנית.

נממש כך: נכתוב מעל הפונקציה את השורה הבאה $template\ <typename\ T>$ (אפשר להשתמש במילה השמורה $class$ במקום $typename$) ואח"כ נכניס לפונקציה את הארגומנטים עם טייפ T. כשנקרא לפונקציה הקומפיילר יממש

את הגרסה הרלוונטית לפי הטייפ של הארגומנטים.
הערה: אנו נצטרך לממש $copy\ constructor$ ואופרטור השמה, אם הם לא קיימים במחלקה שלנו..

- כאשר אנו יוצרים קבצי $template$ אנו צריכים להכריז עליהם בקבצי ה h.

11.3 תרגול 5:

- כאשר אנו מאתחלים איטרטור אנו צריכים לאתחל את המטודה $begin()$ ו $end()$. עבור רשימות ניתן לשלוח מצביע לתחילת המערך ולאיבר האחרון (למעשה נשלח לסוף המערך - אחרי המערך).
- אם יש לנו גישה לפויינטר, אנו יכולים לממש אותו כאיטרטור אם הזיכרון רציף. ואין צורך להכין את כל הגרסאות של האיטרטור, לדרוס את המטודות, לממש איטרטור טריידס, בנוסף לא צריך ליצור מחלקת איטרטור.
- כשאנו ממשישם אופרטור הדפסה אנו צריכים להגדיר אותו כ $friend$ כדי שתהיה לו גישה לשדות הפרטיים של המחלקה.
- $friend$ היא אינה מטודה, ואין לה גישה למופע ספציפי. אלא גישה למתשנים פרטיים של המחלקה בלבד.

כל פעם שאנחנו יוצרים איטרטור אנחנו צריכים לממש 6 מטודות: $begin(), end(), const\ bedin(), const\ end(), cbegin(), cend()$.
בנוסף אנו צריכים לממש $typedef$ שנקראים $iterator, const\_iterator$.

12 שבוע 12:

12.1 הרצאה 10 - $Templates$:

- כשאנו מגדירים $template$ אנו מניחים שהוא תומך בטיפוסים שיש להם $copy\ constructor$ ואופרטור =.
- פונקציה שמוגדרת $template$ תיווצר בגרסה הרלוונטית רק שהיא תיקרא עם הטייפ הזה, עד אז היא נשארת רק בגדר הכרזה.
- ניתן להגיד לקומפיילר בדיוק מה אנו רוצים להכניס לפונקציה $template$ בך: $f < type > (arg...)$ כך למעשה נגיד לקומפיילר איה טיפוס אנו מכניסים.
- ניתן להגדיר $template$ עם כמה משתנים גנרים: $template < class\ T, class\ C >$.
- ניתן באותו האופן להגדיר מחלקה שהיא $template$, נוסיף מעל שם המחלקה את התבנית, בנוסף מעל כל מטודה שנמצאת מחוץ למחלקה נוסיף את התבנית גם כן.
- כשאנו יוצרים מחלקה $template$ אנו לא יוצרים קובץ cpp אלא קובץ h בלבד. משום שהמימוש מתבצע בקריאה ועד אז הכל בגדר הכרזה.

- כשנקרא למחלקה נקרא לה עם הטייפ כך: $name < type > className$. ואם יש לה כמה טיפוסים גנרים אז נשים טיפוסים לכולם.
- ניתן לתת ערכים דיפולטיבים למחלקה, והערך הדיפולטיבי יכנס כאשר נקרא לאובייקט בלי לשים ערך חדש. ניתן לשים ערך חדש ולדרוס את הערך הדיפולטיבי. $template < class\ T = defaultType, class\ C >$
- ***Template Specielization*:** מאפשר לנו לתת למטודה בודדת טייפ ספציפי. נאתחל לייד השם של המטודה או המחלקה כך: $< type >$.

12.1.1 ירושה:

- **ירושה:** יצירה של אובייקט חדש, תוך כי ירושת תכונות בסיס והרחבה שלהן.
  נממש כך: אם מחלקה A יורשת ממחלקה B, לייד שם המחלקה A נכתוב: $Class\ A:\ public\ B$.
- ***protected*:** מטודה שתוגדר בתור $protected$ תוכל לאפשר גישה לכל המחלקות שיורשות ממנה.
- ירושה מעגלית תגרום לשגיאה.
- **יש כמה סוגים של ירושה :** $public, private, protected$ הדיסולט במחלקה הוא $private$ ובסטראקט $public$.
  המילה שאיתה נירש, תגדיר את רמת ההגנה של מה שהוגדר $public, protected$, הכל יעלה ברמת הגנה לפי המילה שנשים. לדוגמה - אם נירש עם המילה $private$ כל המשתנים שמוגדרים $public, protected$ יעברו להיות $private$.

12.2 הרצאה 11 - ירושה:

- למחלקה המורישה אנו נקרא $base\ class$ ולמחלקה היורשת $derived\ class$.
- **קונסטרקטורים ודיסטרקטורים בירושה:** קונסטרקטורים - יקרא של האב ואחכ של הבן. דיסטרקטורים - יקרא קודם של הבן ואחכ של האב.
- כשאנו משתמשים ב $initializer\ list$ הקומפיילר מאתחל לפי הסדר של השדות.
- **קריאה למטודות שירשנו:** את הקראיה למטודות של המחלקה המורישה נממש כך: $base\ class\ name :: base\ class\ func()$.
- ***virtual*:** מילה שמורה, כשנשים אותה לפני מטודה במחלקת האב, הקומפיילר ידע שהוא יכול לקרוא למטודה המתאימה ביותר מבחינתו, לפי סוג האובייקט - הפעולה נקראת **רזולוציה דינמית**.
  בשיטה זו הקריאה תתבצע בזמן ריצה ולא בזמן קומפלציה.
  אם נסמן כ $virtual$ מטודה במחלקת האב, כל מחלקה יורשת שממשמת מטודה עם אותו השם גם תזכה בתהילה ותממש $virtual$.
- **קריאה לפונקציה וירטואלית מתוך קונסטרקטור או דיסטרקטור:** תיקרא לפונקציה ממחלקת האב, משום שהקומפיילר לא מבדיל וקורא לפונקציה של מחלקת האב. לכן לעולם לא נקרא לפונקציות וירטואליות מקונסטרקטור ודיסטרקטור.
- **דיסטרקטור וירטואלי:** בקלאס וירטואלי - אנו חייבים לממש דיסטרקטור וירטואלי.

12.3 תרגול 6 :

- לא יהיה כלום כי אין כלום.

13 שבוע 13:

13.1 הרצאה 12 - 21\6 - מחלקות אבסטרקטיות:

- **מחלקות אבסטרקטיות -** $pure-virtual$**:** אם נרצה ליצור מחלקה שאין לה מופע. נגדיר מטודה כוירטואל, ונשווה אותה ל 0, כך: $virtual\ returnValue\ funcName() = 0;$. כך למעשה לא נוכל להחזיק אובייקט של המחלקה אלא רק מצביעים או אוביקטים של המחלקות היורשות. כך אנו מסמנים שכל מחלקה יורת צריכה לממש את המטודות הללו, ואם לא גם המחלקה היורשת תהיה אבסטרקטית.

- ***ovveride*:** מילה שמורה שנשים אחרי הסוגריים העגולות של הפונקציה שאותה דרסנו, בכדי לסמן לקומפיילר שזאת מטודה ששינינו. במקרה של טעות תהיה לנו שגיאת ריצה ולא שגיאת קומפלציה.

- ***final*:** מילה שמורה שנשים בקלאס הבסיס כשנרצה לחסום אופציה של ירושה או $overriding$, ניתן להגדיר מטודות ומחלקות כ $final$. מחלקה שתוגדר כ $final$ לא תוכל לשמש כמחלקת אב.

13.2 הרצאה 13 - $Smart\ Poiters$:

- ***std :: uniqueptr*:** פויינטר שמקבל מצביע אחר ומקבל עליו בעלות, הוא אחראי על שחרור הזיכרון שלו. הוא נהפך לבעלים הבלעדיים של הפויינטר והוא אחראי על שחרור הזיכרון. הוא מאפשר $move\ constructor$ ו $move\ assignment$ בלבד.

13.3 תרגול 7:

- **ההבדל בין קופי קונסטרקטור לאופרטור השמה הוא:** באופרטור השמה אנו בודקים שהם לא שווים, מוחקים את כל האיברים מהווקטור ואז משווים את הווקטור שלנו לחדש.
  קופי קונסטרקטור - מעתיק את מה שקיבלנו לאובייקט חדש.