שיא הסי-ים

(C/C++)

מאת: יורם ביברמן

© כל הזכויות שמורות למחבר.
אין לעשות כל שימוש מסחרי בספר זה או בקטעים ממנו. ניתנת הרשות להשתמש בו לצורכי לימוד של המשתמש בלבד.

ליענן
שבכל אחד מאיתנו,
לנ.
של כולנו
ולמאבק איתם ובהם

בָּעֶרֶב
כְּשֶׁאָמְרָה לִי נַעֲרָתִי
לֵךְ
יָרַדְתִּי לָרְחוֹב לְהִתְהַלֵּךְ
וְהָיִיתִי הוֹלֵךְ וּמִסְתַּבֵּךְ
מִסְתַּבֵּךְ וְהוֹלֵךְ
וְהוֹלֵךְ וְהוֹלֵךְ וּמִסְתַּבֵּךְ

(נתן זך)

מבוא

ספר זה מסכם את הניסיון (תרתי משמע) המצטבר של כותב שורות אלה להורות את הקורס (ובהוראת הקורס) מבוא למדעי המחשב (הנקרא גם מבוא לתכנות). ניסיוני לימד אותי כי קורס זה הינו פרדוקסאלי במובן זה שלכאורה אף תלמיד ממוצע משנה ג׳ מסוגל היה, לכל הפחות מבחינת חומר הלימוד, ללמד את הקורס על נקלה, אך למעשה רבים וטובים מתקשים להנחיל את החומר המועבר בו לתלמידיהם. מניסיוני שלי אני יכול להוסיף, שאף ששנים רבות של הוראת הקורס מאחורי, אני ממשיך כל העת לשפר את כישורַי. להרגשתי, הסיבה לפרדוקסאליות היא שרכישת המיומנות התכנותית מחייבת את הלומד בביצוע ׳קפיצה׳ בלתי ניתנת להסבר, כלומר בביצוע מעבר לא רציף ממצב של מי שאינו יודע לתכנת, לכדי מי שמסוגל ׳לחשוב תכנותית׳, במילים אחרות לתרגם, כמעט מבלי משים, משימה המנוסחת במונחים מילוליים, לכְדי תכנית מחשב. הפער בין שתי העמדות הוא כמעט בלתי ניתן להבנה, או לאמפתיה (הן מצד זה, והן מצד זה), ולכן קשה עד מאוד למתכנת מיומן להיכנס לנעליו של זה מקרוב בא, באותה המידה שקשה ומתסכל לַתלמיד הנבוך לרכוש את המיומנויות הדרושות, שכן אין כל דרך ברורה שאם יפסע בה הוא מובטח שיגיע למטרה; על המתכנת הצעיר פשוט לטעות, טעייה כואבת, במבוך התכנותי עד שילמד את רזיו.

חוששני גם שלא לכל עמיתי המלומדים יש את אורך הרוח ללוות את תלמידיהם הנבוכים במסעם, בה במידה שלא לכל התלמידים יש את אורך הרוח, יכולת ההתמדה, והנכונות ליפול ולקום עד שהם ילמדו לזחול, ללכת, ולבסוף גם לרוץ. על-כן אני משתדל לחקוק הן על לוח לבי והן על זה של תלמידַי את המאמר ״לא הבישן למד, ולא הקפדן מלמד״.

לַתלמיד בראשית דרכו עלי לומר מייד, ייתכן שחומר הלימוד שיופיע בהמשך יהיה נהיר לך ׳כספר הפתוח׳, וייתכן שלא, ייתכן שהסבריו של המורה שילמד אותך יהיו פשוטים ובהירים, וייתכן שלא, בכל מקרה תגלה (למרבה המכאוב) כי קיים פער מפתיע ומתסכל בין היכולת *להבין* את תכניות המחשב שהוצגו בפניך, לבין היכולת *לכתוב בעצמך* תכניות. הדרך היחידה להתגבר על פער זה היא באמצעות תרגול, תרגול, תרגול. כל תכנית נוספת שתכתוב תקדם אותך יותר ממאה תכניות שתקרא, (קל וחומר מאלף קיטורים שתקַטֵר). מסיבה זאת כללתי בספר תרגילים לא מעטים שיאפשרו לך לנסות, ועל-ידי כך להעצים, את כוחך שלך בכתיבת תכניות.

בנקודה זאת, טרם שאנו צוללים לים התכנותי, ראוי להוסיף גם מילת אזהרה: דעו לכם שמי שרוצה לעסוק בתכנות צפוי לבלות שעות רבות, וקשות, על שטויות: נקודה פסיק (;) שנשמטה, כוכבית (*) שהוצבה שלא במקומה, ושאר טעויות פעוטות, שעת אנו נתקלים בהם במקומות אחרים אנו לכל היותר מחייכים, ונדים בראשנו, אך בתכנית מחשב הם יכולים לעשות את כל הפער בין איגרא רמא, לבירה עמיקתא. בשעה יפה זאת של התחלה, עת כולנו נרגשים, ועם חיוך מנצח על השפתיים שהגענו עד הלום, אני רוצה להזהיר אתכם שכבר ראיתי לא מעט תלמידים עם דמעות בעיניים ומשׂכנעוּת בלב שהמחשב החליט באופן אישי ונחרץ להוציאם מדעתם, ויהי מה, שכן העוְלות שהוא מעולל להם (בדרך כלל בשעות הקטנות של הלילה) הן באמת מעל ומעבר לכל דמיון. לכל מי שייקלע למצב זה אני רוצה לומר כבר עתה שאכן לעיתים התנהגותו של המחשב יכולה להיות אכזרית, אך שהוא שומר לעצמו את הזכות לנהוג כך אך ורק כלפי מי שהוא חש שהינו טירון, שאצבעותיו עוד לא מיומנות בנגינה על המקלדת. כלפי מנוסים יותר הוא, בחיים, לא יעז לנהוג כך. על כן העצה היחידה היא לחרוק שיניים ולהמשיך לשחות. לעיתים, אם הדבר אפשרי, עצה ממתכנת מעט יותר מנוסה עשויה לחסוך לכם כאב רב, אולם אין מנוס מלהתמודד עם הקשיים בכוחות עצמכם; רק כך תלמדו.

צדו השני של המטבע הוא שלהתרשמותי אך מעטים 'אינם בנויים' או 'אינם מתאימים' לתחום המחשבים. מטֵבע הדברים יש מי שהדברים הולכים לו פחות בקושי, ויש מי שהם הולכים לו יותר בקושי, אולם, לדעתי, לא נדרש כשרון ייחודי כלשהו, ורוב מי שיתאמץ מספיק, ויתַרגל מספיק, בסוף גם יצליח. ראיתי כבר לא מעט תלמידים שנראו לי בינוניים בכישוריהם, אך עם כוח רצון עז, ששרדו, ולעומתם תלמידים מבריקים שלא הייתה להם הנכונות לשאת בעול הסיזיפי לעיתים וויתרו. בהקֵשר זה אני רוצה גם לומר שאם אתם רואים מישהו יושב מול המחשב עם חיוך נינוח על השפתיים, תוך שהוא מפזר לצדדים הערות מזלזלות על התכנית שיש לכתוב דעו שהוא: (א) משקר, או (ב) כבר היה בעבר במקום בו אתם מצויים עתה, ועל-כן עתה הוא כבר קדימה יותר; כל כמה שזה נראה לכם לא אפשרי, תוך זמן מה גם אתם תתייחסו לתכניות שנראות לכם עתה מחסום בלתי עביר, כאל משימה פשוטה למדי.

בנוגע לכתיבת תכניות. מהיכרותי את טבע האדם, והסטודנט, אני יודע שעת תלמיד מתבקש להתמודד עם תרגיל חדש, על חומר שזה עתה נלמד, הוא, בדרך כלל, פועל בשיטת שלושת השלבים: (א) הוא מציץ בדף התרגיל, ומזדעזע קשות מאכזריותו של מי שניסח אותו. (ב) הוא דוחף את דף התרגיל עמוק ככל שניתן לתִיקו, ומשתדל, בדרך כלל בלי הצלחה יתרה, לשכוח ממנו לכמה זמן, עד שהוא יירגע מהתרגיל הקודם. (ג) מספר ימים לפני מועד ההגשה הוא שולף בכאב לב את דף התרגיל ומנסה להתמודד אִתו; אך אז כבר אין די זמן לאתר את המורה ולהתייעץ עמו, יש לעבוד בשעות בהם הדעת כבר אינה צלולה, ומקַדם הלחץ כבר בשיאו, במיוחד שמסביב כבר נשמעים דיבורים של תלמידים שלכל הפחות נשמעים כמי שכמעט סיימו את העבודה על התרגיל. כאן עלי להזהירכם משני כשלים נוספים: (א) לעולם אל תעתיקו תרגיל, אל 'תציצו' בתרגיל של חבר 'רק כדי לקבל רעיונות', ואל תעבדו עם חבר ששולט בחומר טוב מכם וש- 'רק יסייע לכם'. אתם יורים כך לעצמכם כדור ברגל! מורה הקורס עייף מלשבת ב-'משפטי שלמה' כאלה או אחרים בהם עליו להכריע מי כתב תכנית כלשהי, ובאיזה מידה הוא עשה זאת באופן עצמאי. מנגד, כמיטב יכולתנו אנו מנטרים את הנושא ובמידת הצורך מקשים את לבנו, ומענישים בחומרה את מי ששגה. (ב) לעולם אל תתנפלו על המחשב טרם שכתבתם את התכנית בנחת על דף עם עפרון **ומוחק**. רק אחרי שכתבתם, תיקנתם, שיניתם, ואתם משוכנעים שהפעם יצאה תחת ידכם יצירת מופת שאפילו מחשב אכזרי ייאלץ להודות שהיא מושלמת, הדליקו את המחשב. סביר להניח שתגלו שעדיין יש בתכניתכם שגיאות רבות, אך מנגד שיש בה כבר גם גרעין של תכנית סבירה שניתן לשפץ. התנפלות על המחשב עם תכנית שלא נבדקה על-ידכם בראשכם שוב ושוב, היא מתכון לתסכול, זעם, ובמקרה הטוב לתכנית מסורבלת שנראית כמו טלאי על טלאי (ובמקרה הרע לקטסטרופה). הקפידו עד מאוד להגיש את כל התרגילים הניתנים לכם. הניסיון מלמד שתלמיד שלא הגיש שני תרגילים רצופים יכול לראות את עצמו כמי שמצוי עם יותר מאשר רגל אחת בחוץ.

בקורס זה נלמד את שפת C++ או ליתר דיוק חלק משפה זאת. שפת C++ היא שפה עשירה מאוד, ועל כן בזמן המוקצה לקורס זה לא נוכל, וגם לא נזדקק, להיכרות מלאה עמה. יחד עם זאת אני ממליץ לתלמיד השקדן, בהמשך הדרך, אחרי שהיכרותו עם המרכיבים הבסיסיים של השפה תשתפר, לקרוא בספרים, וללמוד את היבטים נוספים של השפה, וכל המרבה הרי זה משובח. בפרט ובמיוחד לא נכיר חלק לא מבוטל של השפה המאפשר תכנות בגישה הקרויה 'תכנות מונחה עצמים' (Object Oriented Programming). בקורס זה נתמקד בגישה תכנותית הקרויה 'תכנות מודולארי' (Modular Programming). נראה לנו כי מכיוון שבעולם התכנות קיימות פרדיגמות שונות, והתכנות המודולארי הוא פרדיגמה חשובה, אזי מתאים להיכנס לעולם התכנות דרך שער זה, ואת הגישה מונחית העצמים להותיר לקורס המשך,

בפרט לכזה במבני נתונים, בו גם טבעי לתכנת עם אובייקטים. מנגד, בַּקורס לא נסתפק רק בהיכרות עם שפת C++ אלא נשאף לרכוש את מיומנות התכנות המודולארי. מניסיוני, רכישת צורת החשיבה המודולארית היא אחד האתגרים המרכזיים הניצבים בפני התלמיד, אחרי שהוא התגבר על האתגר הבסיסי של השימוש בפקודות שמעמידה לרשותו השפה.

כאמור, בַּקורס נלמד את שפת C++. למעשה מרביתו המוחלטת של החומר שילמד משותפת לשפת C++ ולשפת C, ורק כמה נושאים משניים שנכיר נכללים בשפת C++, אך לא בשפת C. דווקא הנושא כמעט הראשון שנכיר: קלט ופלט, שייך לרשימת הנושאים בהם השפות נבדלות זו מזו; ולמעשה אחת הסיבות המרכזיות בגינן החלטתי ללמד בקורס הנוכחי את שפת C++ היא כדי לחסוך מהסטודנט הנבוך, את המפגש עם פקודות הקלט\פלט המכוערות של שפת C כבר בתחילת דרכו. מעבר לנושא זה קיימים עוד כמה נושאים שוליים בהם השפות נבדלות (משתנים מטיפוס bool, פרמטרי הפניה, קבועים); עת נגיע לנושא כזה אציין זאת מפורשות. אומר זאת אחרת: להערכתי, תלמיד שירכוש שליטה בחומר הנלמד בקורס הנוכחי צריך להיות מסוגל יחסית בקלות ובזריזות לעבור לתכנת בשפת C. כמות ההתאמות או השינויים שידרשו ממנו יהיו מעטים.

הקורס בו אנו עוסקים נקרא מבוא למדעי המחשב, ולא קורס בשפת C++, או קורס בתכנות; ולא בכדי. חלקים נרחבים מהקורס מוקדשים לנושאים נוספים במדעי המחשב; בעיקר להיכרות עם מבני נתונים שונים, אלגוריתמים, ושאלת זמן הריצה של תכניות. חלק מרכזי של הקורס עוסק באלגוריתמים רקורסיביים שונים, ובטיפול בסיסי במבני נתונים כגון רשימות ועצים. ההיכרות עם הנושאים הכלליים נעשית תוך התמקדות במימושם בשפת C++. המטרה היא להביא את התלמיד לכדי מיומנות תכנותית מעמיקה בנושאים אלה, ולצִדה להיכרות בסיסית עם ההיבטים התיאורטיים שלהם. ההיבטים התיאורטיים נלמדים בצורה לא פורמאלית, תוך 'נפנוף ידיים' מתוך תקווה שכך נקל יהיה עליכם לקבל את האינטואיציה הדרושה, אותה תבססו בהבנה פורמאלית יותר של הנושאים בקורסים עתידיים.

נושא נוסף שנלמד בצורה די יסודית הוא אופן הקצאת הזיכרון לתכניות. השאיפה היא להקנות לתלמידים הבנה *עקרונית* של האופן בו מוקצה זיכרון על הערמה ועל המחסנית במהלך קריאות לפונקציות, ועת התכנית מקצה זיכרון דינמי.

אחד הגורמים שהביאוני לכתיבת ספר זה הייתה התחושה ששפת C++, יחד עם הנושאים הנוספים אותם יש צורך לכסות בקורס זה, רחבים מכדי שהזמן שעומד לרשותי בכיתה יאפשר לי להורות אותם בַּנינוחות בה הייתי חפץ לעשות זאת. מכאן אני גוזר המְלצה לַתלמידים לקרוא את חומר הלימוד בְּנינוחות, גם אם להרגשתכם באותה שעה, ההסברים בכיתה היו יחסית נהירים לכם; קל וחומר אם לאו. לא מן הנמנע שקריאה נינוחה תעזור לכם להבהיר נושאים שאולי אפילו לא הבחנתם שלא היו די ברורים לכם, או שלא כל השתמעויותיהם היו לנגד עיניכם. מסיבה זאת אני גם נוטה להמליץ לחזור ולקרוא בשנית נושאים אותם למדתם בעבר. ייתכן כי בשעה שלמדתם את החומר בראשונה היו היבטים משניים שבצוק העיתים לא הובנו על-ידכם די צרכם. עתה, שהחומר הבסיסי הוטְמע על-ידכם טוב יותר, מן הסתם בעקבות תרגולו, תוכלו להתפנות לקלוט נקודות שלא נקלטו בסערת הקרב הראשונה. מנגד, אין טעם לעסוק בשינון החומר, או התכניות המוצגות, די להבינם.

אחת ההתלבטויות המרכזיות שעמדו בפני עת ניגשתי לכתוב ספר זה הייתה הבאה: אני מטיף לתלמידי השְכם והערב כי הנכס החשוב ביותר עִמו הם יכולים לצאת משערי האקדמיה הוא היכולת לקרוא חומר כתוב באנגלית. אני מוסיף כי הם

יילמדו בבית-ספרנו נושאים רבים, מגוונים, ומעשירים. אחר הם ייבחנו על אותם נושאים, ואז הם יישכחו את תלמודם. מה אם כן יישאר לַתלמידים? מה הם בכל אופן יילמדו? הם יילמדו ללמוד. הם יעשירו את צורת החשיבה שלהם, ובדרך מסתורית כלשהי החומר הרב שהם יילמדו, ואחר יישכחו, יקֵל עליהם ללמוד נושאים אחרים לגמרי, אשר יעסיקו אותם אחרי תום לימודיהם. אולם הבסיס לַיכולת ללמוד, הוא היכולת לקרוא; וכאשר במקומותינו אומרים 'היכולת לקרוא' אין הכוונה לקרוא את התנ"ך בשפת המקור (מעשה יפה כשלעצמו, אך לא רלוונטי לענייננו), אלא הכוונה היא לקרוא חומר טכני הכתוב באנגלית. ובכן, מחד גיסא אני מטיף לרכישת מיומנות הקריאה באנגלית, ומאידך גיסא אני מציע לתלמידים חומר בעברית. זו אכן דילמה. להגנתי אני אומר כי בסמסטר הראשון של שנת הלימודים הראשונה מגיעות לתלמידים הקלות והנחות מתוקף מעמדם כ- 'אפרוחים'. הנחות שהולכות ובטלות ככל שהם גדלים. עלי להודות שאיני חש שלם לחלוטין עם טענה זאת, ועל כן אני קורא לכם: למרות שהספר שלפניכם כתוב בעברית, לִמדו ובעיקר הִתרגלו לקרוא חומר באנגלית. זה מאוד חשוב!

ומילה אחרונה לתלמידים שהחלו אך זה עתה ללמוד במוסד להשכלה גבוהה: כשאני הייתי בצבא, נהגו המ"כים לשאוג על הטירונים: "צאו מההלם!". המ"כים צדקו בכך שרבים מהטירונים אכן היו שרויים בהלם מסוים עם גיוסם לצבא, בשל המעבר החד מחיי התיכוניסט המצוי לזה של הטירון הבזוי. המ"כים שגו בכך שהם שאגו על הטירונים לצאת מההלם במקום לסייע להם בכך. להתרשמותי, גם תלמידי שנה א' מצויים בהלם מסוים סביב המעבר החד מהצבא, הטיול בחו"ל, או כל מקום אחר ממנו הם הגיעו, לחיים אקדמיים תובעניים מאוד (במיוחד בתחום המחשבים). מטבע הדברים, עם הזמן מסתגלים הסטודנטים למעמדם החדש, ואורך החיים הנגזר ממנו (שעות מול מסך המחשב, במקום מול זה של הסינימטק); אולם ההסתגלות דורשת זמן, ושלב משְברי הינו בלתי נמנע. אלה שיכולים להרשות לעצמם, לפחות בסמסטר הראשון, להתרכז בלימודים, טוב יעשו אם יניחו בצד עיסוקים אחרים, לכל הפחות עד שייקלו מעט המים.

לסיכום, אתם יוצאים מכאן למסע לא קל. מעניין, אולי גם מעשיר, אך כזה המבטיח לא מעט דם ,יזע ודמעות. אני מודה שלא הייתי רוצה להתחלף אתכם, זה לא תענוג להיות תלמיד לתואר ראשון, בטח ובטח שלא במחשבים, ובטח ובטח ובטח שלא תלמיד שנה א' למחשבים. ומנגד, כשלעצמי, אני שמח על המקום בו אני ניצב, ושאליו לא יכולתי להגיע בלי להיות קודם גם במצבכם...

דרך צלֵחה.

<u>תודות והפצרות</u>

כתיבתו של ספר זה התאפשרה רק תודות לדורות של תלמידים שהניחו את זְקנם (מרצונם, או שלא מתוך בחירה) תחת תערו של כותב שורות אלה, ואִפשרו לו לרכוש ולשכלל את ניסיונו. על-כן ראשונים לכל ראויים הם לתודתי. כמו כן רציתי להודות למורים הרבים שלימדו לצדי לאורך השנים, ותרמו לנושאים המוצגים בספר זה. בפרט אודה ל: דני ברנד, קלרה קדם, שוקי שגיב, מיכל פרנס, תמי וילנר, והצעירים חסרי המנוחה דן צפריר וגיל בז'רנו שגם הקציעו את שליטתי שלי בשפת C++. לבסוף אודה למוסדות להשכלה גבוהה השונים בהם שהיתי, ועודני שוהה, ושאפשרו לי להגיע עד הלום.

למרות המחשבה והעבודה הרבה שהושקעה בכתיבת ספר זה אין לי ספק שהוא כולל בחובו גם פגמים שונים ומשונים. אודה על כן לכל מי שיאות להאיר את עיני על קלה כחמורה:

א. שגיאות כתיב, כתב, ניסוח או שפה.

ב. חומר שאינו מוסבר בצורה די בהירה.
ג. חומר שאינו מוסבר כלל וראוי היה שיוסבר.
ד. חומר שמוסבר אולי כהלכה אך באופן שגוי!

הערות תתקבלנה בברכה באמצעות הדואר האלקטרוני:

yoramb@hadassah.ac.il

תוכן עניינים

1. הקדמה

פרק זה כשמו כן הוא: פרק הקדמה. מטרתו ללמד אתכם *מעט* אודות המכונה בה תשתמשו במהלך לימודיכם: המחשב. יאמר מיד, כשם שאתם יכולים לנהוג במכונית, או לתפעל מכונת כביסה, בלי להכיר כלל את עקרונות פעולתם של אותם מכשירים, כך אתם יכולים גם לתפעל את המחשב בלי להבין כיצד הוא פועל; אולם לדעתי יועיל לכם להבין, ולו באופן כללי ועל קצה המזלג, כיצד פועל המחשב, מדוע אתם מבצעים צעד זה או אחר, ולְמה אותו צעד גורם במחשב. על-כן בחרתי לפתוח את מסענו בהסבר קצר וכללי אודות המחשב. אני מודע לכך שמי שהיכרותו עם עולם המִחְשוב התמצתה עד היום בתפעול מכשיר בנק אוטומטי או בשימוש במעבד תמלילים, עלול, בשלב זה, שלא להבין חלק מהדברים, על כן אני ממליץ לכם לקרוא פרק זה עתה, אך לשוב אליו אחרי שתצברו ניסיון מסוים בתפעול המחשב ובתְכנותו. אני תקווה כי מה שלא יובן עתה יובן אז בצורה מלאה יותר.

נפתח בשאלה: מהו המחשב? במה הוא שונה ממכונת הכביסה (ודומה, אולי, ללגו)?

ביסודו כמכשיר אבסטרטי מורכב המחשב משני מרכיבים א. **זיכרון** (memory, ולעיתים נדייק ונאמר **זיכרון ראשי** main memory כדי להבדיל בין הזיכרון הראשי לזיכרון המשני), ב. **מעבד** (processor ולעיתים נדייק ונאמר **מעבד ראשי** או Central Processing Unit ובקיצור CPU. ללשון הדיוק נזדקק עת נרצה להבחין בין המעבד הראשי למעבדי עזר נוספים שעשויים להיות מותקנים במחשב).

כדי להפוך את המכשיר האבסטרקטי לפרקטי, יחוברו למחשב רכיבי **ציוד היקפי** (peripheral device) נוספים כגון מקלדת, מסך, דיסק קשיח (hard disk), כונן תקליטונים, עכבר, מדפסת. ברכיבים אלה נדון בהמשך שכן הם חיוניים לשם תפעולו של המחשב, אולם הם אינם חלק מהמחשב ביסודו, במובנו הצר ביותר.

1.1 הזיכרון

הזיכרון כשמו כן הוא: תפקידו לזכור (במילים אחרות: לשמור) את מה שאוחסן בו. הזיכרון שומר את מה שאוחסן בו רק כל עוד המחשב פועל (עת מכבים את המחשב תוכנו של הזיכרון אובד). מה הזיכרון זוכר? לא זיכרונות ילדות, אלא: (א) **נתונים** (data) כגון המספר 17, או השם 'יוסי כהן', (ב) **תכניות** (programs): תכנית היא סדרה של **פקודות** (statements) המובַנות למחשב, כגון: (1) חבר את המספרים 3879 ו- 17, (2) שמור את הסכום בזיכרון, (3) הגדל את הסכום באחד, (3) אם הסכום אינו מתחלק ב- 121 אזי אפֵס (הכנס אפס) את תא הזיכרון בו שמרת את הסכום.

הזיכרון אינו יודע לעשות דבר עם הנתונים והתכניות השמורים בו פרט לשמירתם.

המחשבים בהם אנו משתמשים כיום נקראים **מחשבי פון-נוימן**, על-שם האדם שהגה אותם. אחד הרעיונות שהגה פון-נוימן הוא שאותו זיכרון עשוי להחזיק שני סוגים של 'יצורים': תכניות ונתונים (שישמשו את התכניות).

כאשר אנו, כבני אדם המורגלים בשימוש במספרים עשרוניים, חושבים על התא הבסיסי ממנו נרכיב מערכת לשמירת מספרים, סביר שנחשוב על תא שעשוי להכיל ספרה עשרונית (כלומר ספרה שעשויה להכיל ערך שבין אפס לתשע). באמצעות שני תאים כאלה נוכל לשמור מספרים בתחום שבין אפס ל- 99, באמצעות שלושה תאים נוכל לשמור מספרים שבין אפס ל- 999 וכך הלאה. במחשב היחידה הבסיסית ממנה בנוי הזיכרון היא תא יחיד הקרוי **סיבית** (bit) ואשר מסוגל להכיל אחד משני

ערכים: אפס או אחד, במילים אחרות ערך **בינארי**. באמצעות שתי סיביות נוכל לשמור ארבעה ערכים שונים (00, 01, 10, 11), באמצעות שלוש סיביות נוכל לשמור שמונה ערכים שונים (שעשויים לייצג את המספרים אפס עד שבע, האותיות א' עד ח', או כל שמונה ערכים שנרצה). באמצעות שמונה סיביות נוכל לשמור 256 ערכים שונים. קבוצה של שמונה סיביות מכונה **בית** (byte). שני בתים (או לעתים ארבעה או שמונה בתים) מכונים **מילה** (word). מכיוון שסיבית בודדת היא יחידה קטנה מאוד, נהוג לציין את גודלו של הזיכרון ביחידות של בתים (גודלו של הזיכרון ביחידות של סיבית הוא, כמובן, פי שמונה מגודלו ביחידות של בית).

על הזיכרון ניתן לבצע אחת משתי פעולות: (א) **לאחסן** ערך בתא זיכרון כלשהו (במילים אחרות **לכתוב** ערך על התא), או (ב) **לשלוף** את הערך המצוי בתא זיכרון כלשהו (הקריאה\שליפה אינה מוחקת את הערך המצוי בתא). זיכרון המחשב כולל מילארדי בתים. על-מנת שניתן יהיה לפנות לבית רצוי כלשהו בזיכרון, (כדי לכתוב עליו או על מספר בתים רצופים המתחילים בבית זה, או כדי לקרוא את הערך המצוי בתא הזיכרון), נותנים לכל בית בזיכרון **כתובת**. לפיכך, לדוגמה, תכנית במחשב עשויה לכלול פקודה אשר מכניסה את הערך 17 לתא המצוי בכתובת 38789 בזיכרון.

במחשבים האישיים המיוצרים בעת כתיבת שורות אלה, גודלו של הזיכרון הראשי נע בין 512 mega byte (כלומר 512 מיליוני בתים) לבין 4 giga byte (כלומר ארבעה מיליארדי בתים) לערך.

1.2 המעבד

המעבד הוא המוח של המחשב, הוא זה שיודע לבצע (execute, לעיתים נאמר **להריץ** run) את התכניות השמורות בזיכרון. עת המעבד מריץ תכנית, ועל-ידי כך מבצע משימה כלשהי, אנו אומרים כי המחשב ביצע חישוב (computation) זה או אחר. בשפת יומיום חישוב משמש בהֶקְשר המתמטי, בעולם המחשבים חישוב מציין כל משימה שתורגמה לתכנית מחשב, (במילים אחרות שתוכנתה), ושמבוצעת על-ידי המחשב (דוגמה לחישוב: קרא מהמשתמש סדרה של מילים באנגלית, מיין את המילים על-פי סדר לקסיקוגרפי, כלומר על-פי הסדר בו הן מופיעות במילון, ואחר הצג את המילים הממוינות).

אמרנו שהמחשב (או ליתר דיוק המעבד שבו) מבצע תכנית, כלומר סדרה של פקודות. גדולתו של המחשב היא שהפקודות הבסיסיות אותן מכיר המעבד (ומסוגל לבצע) הן פשוטות ביותר, אך על-ידי שילובן יחד ניתן לבצע משימות מורכבות ביותר. בכך דומה המחשב ללגו. גם בלגו קיימות מספר קוביות קטנות ופשוטות אשר על-ידי הרכבתן יחד ניתן לבנות מבנים באופן שרק השמים (והתקציב) הם הגבול (תרתי משמע). בכך שונה המחשב ממכונת הכביסה: במכונת הכביסה קיימות מספר תכניות מוגדרות וקבועות מראש, ואתם אינכם יכולים לבנות לכם תכנית כבקשתכם אשר תכניס מים, תשטוף, תסחט בכל סדר שהוא וכמה פעמים שאתם רוצים.

את מהירות המעבד נהוג למדידות ביחידות של הרץ. מהירות המעבד במחשבים הביתיים המיוצרים כיום היא כ- 700 מגה הרץ (כלומר שעון המעבד מבצע 700 מיליון פעימות בשניה). קיים קשר *עקיף* בין מהירות זאת לבין מספר הפקודות בשפת מכונה שהמעבד מסוגל לבצע, ולכן יהיה זה מדויק אך בחלקו לומר שמעבד שמהירותו X מבצע תכנית כלשהי מהר יותר ממעבד שני שמהירותו Y (עבור $Y < X$).

1.3 שפת מכונה, שפת הרכבה, ושפה עילית

הפקודות אותן המעבד יודע לבצע נקראות **שפת המכונה** (machine language) של המחשב. תכונותיה של שפת המכונה הן: (א) היא מורכבת ממספר מוגבל של פקודות בסיסיות\פשוטות. (ב) היא ספציפית לכל מעבד ומעבד (שפת המכונה של מעבדים המיוצרים על-ידי חברה X (לדוגמה אינטל) שונה משפת המכונה של מעבדי חברת Y (לדוגמה: מוטורולה)). (ג) פקודותיה מקודדות כאפסים ואחדים (על-מנת שנוכל לשמרן בזיכרון).

פקודה אפשרית בשפת מכונה עשויה להראות באופן הבא: 00101 00001 101 (הרווחים הוכנסו רק כדי להקל עלינו כבני אדם את הקריאות), ומשמעותה עשויה להיות: שלוש הסיביות השמאליות (101) מציינות שיש לחסר את הנתון המופיע בכתובת שבחמש הסיביות הבאות (00001 כלומר כתובת מספר 1) מהנתון המופיע בחמש הסיביות שאחר-כך (00101 או הכתובת מספר 5).

כתיבת תכניות המורכבות מפקודות כנ"ל היא משימה המועדת לטעויות מרובות (לדוגמה: בהיסח הדעת יזין המתכנת 110 במקום 101 בתור מציין הפקודה שיש לבצע, והתוצאה תהיה שהמחשב יבצע פקודה שונה מפקודת החיסור אליה התכוון המתכנת). על-כן באו חכמים והגו את **שפת ההרכבה** (assembly language). פקודותיה של שפת ההרכבה זהות לאלה של שפת המכונה, אולם במקום לכתוב סְדרות של אפסים ואחדים (כפי שכותב המתכנת בשפת המכונה) כותב המתכנת בשפת ההרכבה פקודות בשפה מילולית. לדוגמה את הפקודה שתיארנו קודם לכן בשפת מכונה נכתוב בשפת הרכבה כ- SUB #1, #5 כלומר החסר (subtract) את הנתון בכתובת מספר 1, מהנתון בכתובת מספר 5. תכנית הכתובה בשפת הרכבה אין המעבד מסוגל לבצע (זו אינה תכנית בשפת מכונה). כדי שהמעבד יהיה מסוגל לבצע את התכנית יש ראשית לתרגמה משפת הרכבה לשפת מכונה. את פעולת התרגום עושה תכנית במחשב בשם **מרכיב** (assembler). שימו לב כי המרכיב מבצע עבודת תרגום די פשוטה, לדוגמה הוא מתרגם מילה כגון SUB לרצף של אפסים ואחדים כגון 101, את המספר העשרוני 5 עליו לתרגם למספר הבינארי 00101, וכך הלאה.

אין ספק כי תיכנות בשפת הרכבה נח הרבה יותר מאשר תכנות בשפת מכונה, אולם גם לשפת ההרכבה יש מספר חסרונות מהותיים: (א) היא מורכבת מפקודות שאינן די תואמות את דרך החשיבה של מתכנת אנושי, ועל כן התכנות בה פחות נח מכפי שהינו מעוניינים. (ב) מכיוון שפקודותיה זהות לאלה של שפת המכונה היא תלויית מעבד; משְמַע איננו יכולים להעביר תכנית שכתבנו בדִי עמל בשפת הרכבה של מחשב אחד, למחשב אחר בו מותקן מעבד שונה ('הדובר' שפת מכונה, ולכן גם שפת הרכבה שונה), וזו כמובן מגבלה מאוד משמעותית בעולם כה דינמי כמו עולם המחשבים. על שתי המגבלות הללו מתגברות (חלקית) **השפות העיליות**.

שפה עילית היא שפה שפקודותיה נהגו בדמיונו הקודח של מפתח השפה. פקודות השפה אינן קשורות למחשב זה או אחר, והסיבה להכללתן בשפה היא שלדעת מפתח השפה כתיבת תכניות תוך שימוש בפקודות אלה תהיה משימה יחסית נוחה, באשר הפקודות תואמות את צורת החשיבה של מתכנת אנושי עת האחרון מעוניין לכתוב תכנית מחשב. קריטריון נוסף אותו מפתח שפה ישקול הוא המידה בה התכנית תהיה קריאה לאדם אחר מזה שכתב אותה. על כן פקודות בשפה עילית נראות דומות במידת מה למשפטים בשפה האנגלית. לדוגמה הפקודה: if (x > y) then write(x) משמעה שאם תא הזיכרון המכונה בשם x מכיל ערך גדול מהערך המוחזק בתא הזיכרון המכונה y, אזי הצג את ערכו של תא הזיכרון x. מתכנן שפה

ישקול גם את המידה בה ניתן יהיה לתרגם בקלות וביעילות תכנית הכתובה בַשפה אותה הוא מפתח לשפת מכונה. לכאורה, אפשר היה להציע שהמתכנת יכתוב את תכניתו באנגלית ציחה, אולם אז יש קרוב לודאי שתרגום התכנית לשפת מכונה היה משימה בלתי אפשרית. על כן מפתח של שפה עלית צריך לשמור על מתח בין רצון לפתח שפה בה נקל יהיה לכתוב, לבין היכולת לתרגם תכנית בשפה אותה הוא מפתח לשפת מכונה (שהיא, ורק היא, השפה אותה 'דובר' המעבד).

כמעט כל התכניות הנכתבות בימינו מתוכנתות בשפות עליות כגון פסקל, C, C++, Java, בייסיק, קובול, ורבות אחרות. כמובן שתכנית הכתובה בשפה עילית אינה ניתנת לביצוע (להרצה) כמות שהיא על-ידי המעבד (שכן היא אינה כתובה בשפת מכונה). כדי שהמעבד יוכל לבצע את התכנית יש, ראשית, לתרגמה לשפת מכונה. פעולת התרגום משפה עילית לשפת מכונה נעשית על-ידי תכנה במחשב הנקראת **מהדר** או **קומפיילר** (compiler) ובת זוגה הנקראת **כורך** או **לינקר** (linker). הקומפיילר מתרגם את פקודות התכנית כפי שכתב המתכנת (בשפה עילית) לשפת מכונה. אנו אומרים כי הקומפיילר תירגם את **תכנית המקור** (source program) ל**קוד** (object code ולעיתים נדייק ונאמר code). בעגה אנו אומרים כי **קימפלנו** את התכנית (או שהקומפילר קימפל את התכנית). אולם הקוד המתקבל אינו שלם עדיין, ואינו ניתן להרצה (לביצוע) על-ידי המעבד. את השלמת התהליך לכדי תכנית ניתנת להרצה מבצעת תוכנת הלינקר. הלינקר מוסיף לקוד המתקבל נדבכים נוספים, שעל מהותם לא נעמוד בקורס זה, ויוצר **תכנית ניתנת להרצה** (executable code). את התכנית הניתנת להרצה יוכל המעבד לבצע. במקרים רבים איננו מדייקים בלשוננו ואנו משתמשים בביטוי לקמפל עבור התהליך המלא של קומפילציה ולינקינג.

ציינו כי שפת ההרכבה קשורה לשפת המכונה. שפה עילית אינה קשורה לשפת המכונה, רק הקומפיילר הוא שקושר בין התכנית הכתובה בשפה עילית למעבד זה או אחר. על-כן את אותה תכנית נוכל לתרגם באמצעות קומפיילר א' לשפת מכונה של מחשב #1, ובאמצעות קומפיילר ב' לשפת המכונה של מחשב #2. התוצאה היא שהתכנית הפכה להיות **נשיאה** (portable) במובן זה שניתן לשאת אותה, במילים אחרות להעביר אותה, ממחשב למחשב; ועת אנו עוברים ממחשב למחשב אין צורך לכתוב מחדש את כל התכניות שכתבנו על המחשב הישן. יש צורך רק בכתיבת תוכנות קומפיילר ולינקר חדשות, ובאמצעות תכנות אלה נוכל לתרגם את כל התכניות שכתבנו בשפה עילית למחשב החדש.

לקומפיילר תפקיד נוסף מֵעבר לתִרגום התכנית משפה עילית לשפת מכונה: במידה ובמהלך התרגום מאתר הקומפיילר שגיאות בתכנית המקור, הוא מודיע על-כך ואינו מבצע את התרגום. לדוגמה אם במקום לכתוב if יכתוב המתכנת בטעות iif, אזי הקומפיילר יתקל, מבחינתו, בפקודה שגויה בתכנית. הוא לא ינסה לנחש מה הייתה כוונת המתכנת, אלא הוא יודיע כי הוא איתר שגיאה בתכנית.

בשפת C עשוי הקומפיילר גם להזהיר אתכם על פקודות 'חשודות' בתכנית. אזהרות שכאלה לא תמנענה מהקומפיילר לתרגם את התכנית לשפת מכונה, ובכל אופן ראוי שהתכניות שתכתבו לא 'תזכנה' לאזהרות מצד הקומפיילר.

1.4 האדיטור, והדיבגר

הזכרנו כי המחשב מורכב מזיכרון ומעבד. התכניות שנכתוב במחשב שוכנות (בשלב ראשון) בזיכרון. התוכנה במחשב אשר מאפשרת לנו לכתוב תכניות, כלומר להזין את התכנית שכתבנו על דף, לזיכרון המחשב נקראת **עורך** או אדיטור (editor)

בלע"ז. האדיטור דומה למעבד תמלילים, אולם הוא מותאם במיוחד ורק לכתיבת תכניות, ככזה אין בו חלק מהכלים שקיימים במעבד תמלילים (באדיטור אין אפשרות להקליד אותיות דגושות למשל או לקבוע גופן כזה או אחר) ומנגד יש בו עזרים אשר מסייעים להקלדת התכנית באופן נח (לדוגמה: הוא צובע מילות מפתח שונות בצבעים שונים, מעמֵד את התכנית באופן המקובל). באמצעות האדיטור נקליד את התכנית שכתבנו (בשפה עילית), אחר באמצעות הקומפיילר והלינקר נתרגם את התכנית לשפת מכונה, ואז יוכל המעבד להריצה.

כלי נוסף שעומד לרשות המתכנת הוא ה**מנפה** (debugger). הדיבגר מסייע לכם לאתר שגיאות בתכנית. קודם ציינו כי במידה והקומפיילר מאתר שגיאות בתכנית הוא אינו מתרגם את התכנית לשפת מכונה. עתה נדייק ונאמר כי המהדר מאתר **שגיאות תחביריות** (כדוגמת החלפת if ב- iif, או העדרו של פסיק במקום בו הוא מצופה להופיע). לעיתים התכנית עשויה להיות חפה משגיאות תחביריות, על-כן הקומפיילר יוכל לתרגמה לשפת מכונה, אולם התכנית תכלול **שגיאות לוגיות**, כלומר שגיאות אשר גורמות לתכנית שלא לבצע את המשימה אותה היא אמורה לבצע. לדוגמה, נניח כי על התכנית לחבר את ערכם של תא הזיכרון המכונה בשם x עם תא הזיכרון המכונה y, אולם המתכנת, בהיסח הדעת, במקום לכתוב x+y כתב x-y. מבחינה תחבירית התכנית תקינה, ועל כן הקומפיילר יצליח לתרגמה לשפת מכונה, אולם עת התכנית תורץ היא לא תבצע את מה שהיה עליה לבצע. אנו אומרים כי התכנית כוללת שגיאה לוגית. סוג מיוחד של שגיאה לוגית היא שגיאה אשר גורמת למחשב להפסיק את ביצוע התכנית, שכן התכנית מנסה לבצע פעולה אסורה או בלתי אפשרית. לדוגמה נניח כי המתכנת כלל בתכניתו פקודה כגון x/y שמשמעה חלק את הערך המצוי בתא הזיכרון שכינויו x בערך המצוי בתא הזיכרון שכינויו y. עוד נניח כי מסיבה כלשהי תא הזיכרון y כולל את הערך אפס. עת המחשב (ליתר דיוק המעבד) ינסה לחשב את המנה (תוצאת החילוק), הוא ייכשל, שהרי לא ניתן לחלק באפס. המחשב יפסיק מיידית את ביצוע התכנית ויודיע כי ביצוע התכנית נקטע בשל שגיאה. אנו נוהגים לומר במקרה כזה כי התכנית **עפה** (aborted or terminated).

הדיבגר מאפשר לכם להריץ תכניות בצורה מודרכת, לבצע בכל פעם רק פקודה יחידה, להציג את ערכם של תאי זיכרון שונים, וכך לאתר את המקום בו התכנית שוגה.

במקרים רבים קיימת במחשב תכנה יחידה הנקראת **סביבת עבודה** ואשר כוללת את כל הכלים להם אנו נזקקים לשם כתיבת תכניות, כלומר אדיטור, קומפיילר, לינקר, ודיבגר. Microsoft Visual Studio, Eclipse, Anjuta, borland C, או Borland Project הן כמה דוגמות לסביבות עבודה.

1.5 מערכת ההפעלה

מחשב המורד מפס היצור נקרא מכונה עירומה, יש לו את כל הרכיבים הדרושים, אולם אין מי שיארגן את הרכיבים הללו לכדי מערכת אחת שלמה, אינטגרטיבית, בה ניתן לעשות שימוש מועיל. את הארגון והפיקוח על פעולת הרכיבים השונים מבצעת תכנה במחשב הקרויה **מערכת ההפעלה** (operating system). מערכת ההפעלה היא שמאפשרת לכם להשתמש במחשב באופן שתקבלו ממנו את שאתם צריכים, ולא תגרמו נזקים לנתונים ולתכניות השמורים בו. עם הדלקת המחשב פונֶה המעבד להרצת מערכת ההפעלה, ולכל אורך פעולת המחשב מערכת ההפעלה היא שמוֹרָה למעבד מה עליו לעשות בכל שלב, היא שמאפשרת לו להריץ תוכנה זו או אחרת אותה אתם מבקשים להפעיל. היא גם שמפקחת שהפעלתה של התכנה (במלים אחרות, שהרצתה על-ידי המעבד) לא תגרום לנזקים (למשל שהתכנה לא

תנסה לכתוב על תאים בזיכרון עליהם אין היא אמורה לכתוב). כל תכנית המורצת במחשב, בין אם זה מעבד תמלילים משוכלל, ובין אם תכנית קטנה שאתם כתבתם, מורצת בפיקוחה של מערכת ההפעלה.

מערכת ההפעלה כוללת מרכיבים רבים. לדוגמה עת תכנית נזקקת לזיכרון במחשב מערכת ההפעלה היא שמקצה את קטע הזיכרון שיעמוד לרשות אותה תכנית (וכך דואגת מערכת ההפעלה שתכניות שונות תקבלנה קטעי זיכרון שונים, ועל-כן לא תפרענה אחת לשניה). עת תכנית מעונינת להדפיס נתונים מערכת ההפעלה היא זו שבפועל מעבירה את הנתונים הדרושים למדפסת (מערכת ההפעלה תדאג לכך שאם שתי תכניות בקשו להדפיס בו זמנית אזי ראשית יודפסו נתוני התכנית האחת ורק אחר נתוני התכנית השניה). מערכת ההפעלה היא שאחראית לקרוא מהמקלדת את שמוקלד עליה (ולהעביר את המידע שמוקלד לַתכנית שזקוקה לו), והיא שמעבירה למסך את הנתונים שתכנית המורצת במחשב מעוניינת להציג על המסך. מערכת ההפעלה תדאג שהפלט של כל תכנית יופיע במסך במקום המתאים. בקיצור כל מלאכת ניהול המחשב והפיקוח על שקורה בו מסור בידיה הנאמנות(?) של מערכת ההפעלה.

1.6 ציוד המתחבר למחשב

עד כה תיארנו את המחשב כמורכב מזיכרון ומעבד. אלה הם המרכיבים הבסיסיים של המחשב כמכשיר אבסטרקטי; אך עת רוצים שהמחשב יהיה גם מכונה פִּרקטית יש לחבר לו מרכיבים נוספים, כפי שכולכם מכירים, מרכיבים הנקראים **ציוד קלט פלט** (**input output device** או בקיצור **i/o device**) או **ציוד היקפי**: המסך (ולעיתים רמקולים המחוברים למחשב) מאפשרים הצגת נתונים (במובן הרחב של המילה), המקלדת והעכבר מאפשרים הזנת נתונים.

מרכיב חשוב בציוד הקלט פלט הוא **הזיכרון המשני** המורכב במקרים רבים מ**הדיסק הקשיח (hard disk) , התקליטון\דיסקט (diskette),** או **זיכרון נייד** כדוגמת disk on key**.** ציינו כי הזיכרון (או ליתר דיוק הזיכרון הראשי) שומר את הנתונים הנכתבים עליו רק כל עוד המחשב דולק. במקרים רבים אנו מעונינים לשמור נתונים לאורך זמן, גם עת המחשב מכובה (לדוגמה: כתבנו תכנית מחשב חשובה או חיברנו יצירת מופת). מספר רכיבי ציוד מאפשרים שמירת נתונים גם עת המחשב כבוי; החשובים ביניהם הם הדיסק הקשיח והזיכרון הנייד. עקרונית דרך פעולתם של שני כלים אלה זהה. (דיסקטים, סרטים מגנטיים, הדומים לקלטות במכשירי הקלטה, ותקליטים אופטיים, compact disks, הם כלים נוספים המאפשרים שמירת נתונים גם עת המחשב אינו פועל).

מבחינתנו כמשתמשים, ההבדל המשמעותי בין הדיסק הקשיח לתקליטון או ל- disk on key הוא שהדיסק הקשיח מותקן בדרך כלל בַּמחשב באופן קבוע, ולא ניתן לשאתו ממקום למקום בנפרד מהמחשב בו הוא מותקן, לעומתו את התקליטון ניתן להתקין במחשב (להכניסו לכונן התקליטונים), לכתוב עליו או לקרוא ממנו את המידע הרצוי, ואחר לנתקו מהמחשב, ולשאתו עמכם לכל מקום אליו תפְנו. הבדל נוסף הוא שתהליך הכתיבה על גבי הדיסק הקשיח והקריאה ממנו מהירים יותר מאשר ביצוע אותן פעולות על גבי תקליטון. כמו כן נפחו של הדיסק הקשיח גדול מזה של תקליטון.

נפחו של תקליטון המיוצר כיום הוא כ- 1.5 מגה בית, בעוד נפחו של דיסק קשיח במחשב ביתי הוא בסדר גודל של מאתיים ג'יגה בית (כלומר מאתיים מיליארד בתים).

מושג חשוב אותו עלינו להכיר עת אנו דנים בזיכרון המשני הוא ה**קובץ (file).** נניח שכתבנו תכנית מחשב, או להבדיל פואמה, וברצוננו לשמור את יצירתנו לאורך זמן, ולכן על-גבי דיסק (או זיכרון נייד). כדי שהמחשב (או ליתר דיוק המעבד בפיקוחה של מערכת ההפעלה) יוכל לשמור את הנתונים עבורנו, הוא מקצה על-גבי הדיסק שטח או במילים אחרות קובץ, ועליו הוא שומר\כותב את הנתונים. אנו איננו יודעים היכן מצוי השטח שהוקצה על גבי הדיסק (כלומר, מה כתובתו), כל שאנו יודעים הוא מה השם שאנו בחרנו לתת לקובץ שנוצר. לדוגמה נניח שבחרנו לקרוא לקובץ my_poem. בהמשך נוכל להורות למחשב להדפיס את הקובץ ששמו my_poem, לטעון למעבד התמלילים קובץ זה, או אפילו למחוק את הקובץ הנ״ל. המחשב ידע לאיזה קובץ אנו מתכוונים שכן לכל קובץ יש שם ייחודי המזהה רק אותו (כמובן שניסיון לקמפל את הקובץ my_poem, המכיל פואמה היסטורית דידקטית, יגרום למהדר להודיע לנו על אינספור שגיאות שהוא מצא ב-׳תכנית׳ שביקשנו ממנו לקמפל).

עת אתם כותבים תכנית חדשה במחשב, באמצעות האדיטור, התכנית ראשית נשמרת בזיכרון. אחת הפעולות שהאדיטור מאפשר לכם לבצע היא לשמור את התכנית כקובץ בזיכרון המשני. עת תבקשו מהאדיטור לבצע פעולה זאת הוא ישאלכם לשמו של הקובץ שברצונכם ליצור ושיכיל את התכנית שכתבתם.

מושג נוסף שנרצה להציג בשלב זה הוא מושג **המחיצה (folder)** או **המדריך (directory).** נניח שאתם אנשים מסודרים. עוד נניח כי במסגרת עבודתכם אתם כותבים מכתבים, מחברים מנגינות ומציירים ציורים. על-כן סביר לצפות שבשולחן העבודה שלכם (הפיזי, הניצב בחדרכם) נמצא מגרת מכתבים, מגרה ובה מצויים תווים, ומגרה בה שוכנים ציוריכם. באופן כזה נקל יהיה עליכם לאתר ׳מסמך׳ (דף) רצוי זה או אחר. אם תוך כדי עבודתכם מתברר לכם שאתם כותבים מכתבים רבים ואיתור מכתב רצוי בין כלל המכתבים שכתבתם נעשה קשה, סביר שתפצלו את מגירת המכתבים למגרת מכתבים רשמיים לעומת מגירת מכתבים אישיים. גם במחשב קיים מנגנון דומה. האנלוג במחשב למגרה היא המחיצה או המדריך. על-כן במחשב האוניברסיטה סביר שתימצא מחיצה לכל תלמיד. כל תלמיד יוכל לשמור במחיצה שלו, ורק במחיצה שלו, חומר שהוא יצר. אם התלמיד לומד מספר קורסים יתכן שאת המחיצה שלו הוא יחלק למספר תת-מחיצות, אחת לכל קורס (ואולי כמה מסמכים\קבצים שאינם שייכים לשום קורס ישכנו במחיצה הראשית של התלמיד, לצד מחיצות המשנה של הקורסים השונים). יתכן שאת המחיצה של הקורס מבוא לתכנות יחלק התלמיד שוב למספר מחיצות אחת עבור כל תרגיל, כך שבמחיצה של כל תרגיל ישכון כל החומר השייך לתרגיל זה (כגון קובץ הכולל את תכנית המקור שהתלמיד יצר, קובץ הכולל את תרגום תכנית המקור לשפת מכונה).

1.7 שפת C

הקורס שלפנינו מתנהל בשפת C או ליתר דיוק בשפת C++. למעשה מרבית הנושאים שילמדו משותפים לשתי השפות גם יחד, ואת ההיבטים המרכזיים שמייחדים את C++ מ- C (קריא תכנות מונחה עצמים) לא נכיר במהלך קורס זה. (במקומות בהם חומר הלימוד נכון רק לשפת C++ ולא לשפת C נציין זאת).

ניתן לנהל דיונים ארוכים ומלומדים מדוע לבחור בשפה זו או אחרת לקורס ראשון במדעי המחשב. לצורך עניינו נאמר בקצרה כי C היא השפה הנפוצה ביותר ל**תכנות מודולרי (modular programming)** כפי שנלמד בקורס זה.

לשפת C מגבלה עבור מתכנתים מתחילים והיא שהשפה מאוד 'ליברלית', כלומר הקומפיילרים שלה נוטים להניח שהמתכנת יודע מה הוא עושה, ועל כן אם הוא כתב דבר מה יש להניח שיש לו סיבה טובה לעשות זאת. דא עקא עבור מתכנתים מתחילים הנחה זאת לעיתים קרובות אינה תקפה---מתכנת מתחיל כותב לעיתים שטויות חסרות פשר או תכלית. התוצאה היא שהשפה (או ליתר דיוק הקומפיילר) אינו די שומר עליכם, וחובת השמירה מועברת אליכם. עבור מתכנתים מתחילים זו לעיתים חובה כבדה (מדי). שנאמר "אלוהים מרחם על ילדי הגן...".

בקורס זה נלמד לתכנת בגישת **התכנות המודולרי**. קיימות גישות אחרות לתכנות (תכנות מונחה עצמים, object oriented programming, תכנות בלוגיקה, תכנות פונקציונלי). עבור גישות אחרות קיימות שפות אחרות שעשויות להתאים יותר או פחות משפת C (בפרט עבור תכנות מונחה עצים קיים ויכוח האם C++ או Java היא השפה המועדפת). עבור תכנות מודולרי, לכוחותינו, נראה כי C היא הבחירה המתאימה ביותר.

1.8 האלגוריתם

מושג אחרון אותו נכיר במסגרת ההקדמה הוא מושג **האלגוריתם** (algorithm). אלגוריתם הוא שיטה לפתרון בעיה. לדוגמה, ספר בישול או אפיה מכיל סדרת של אלגוריתמים (המכונים בדרך-כלל מתכונים) המסייעים בהכנת תבשילים ומאפים שונים (לדוגמה: מתכון להכנת ברווז בנוסח סצ'ואן). 'הבעיה' אותה פותר כל אלגוריתם בַּספר היא כיצד להכין מאכל כזה או אחר. עת אתם משתמשים במתכון כדי להכין את המאכל אתם *מבצעים* את האלגוריתם. בעולם המחשבים מקובל לעיתים לומר שאתם *מריצים* את האלגוריתם.

נציג שתי דוגמות נוספות שימחישו מהו אלגוריתם:

א. משוואה ריבועית היא ביטוי מהצורה: $y = a*x^2 + b*x + c$, עבור a,b,c שהינם מספרים ממשיים. לדוגמה: $y = 17*x^2 + 3879*x - 9$ או $y = -5*x^2 + 25$ הן משוואות ריבועיות. שורש המשוואה הריבועית הוא ערך ממשי שאם יוצב במקום x, אזי ערך ה- y המתאים לו יהיה אפס. רובכם ודאי זוכרים שהאלגוריתם, במילים אחרות הנוסחה, או השיטה, לאיתור זוג שורשי משוואה ריבועית היא:

$x_1, x_2 = [-b +- sqrt(b^2 - 4*a*c)]/[2*a]$

ב. עתה נציג אלגוריתם לבדיקה האם מספר טבעי כלשהו ראשוני. האלגוריתם הוא הבא:
   1. אם המספר הוא שתיים אזי הוא ראשוני.
   2. אם המספר הוא מספר זוגי השונה משתיים אזי הוא אינו ראשוני.
   3. אם המספר פרדי (במילים אחרות, אי זוגי) אזי בדוק את כל המספרים הפרטיים הקטנים משורש המספר, אם אף אחד מהם אינו מחלק את המספר אזי המספר ראשוני; אם לפחות אחד מהם מחלק את המספר אזי המספר פָּריק (במלים אחרות, לא ראשוני).

כל אלגוריתם מורכב מסדרה של פעולות בסיסיות אותן יש לבצע כדי להשיג את המטרה הרצויה, במילים אחרות כדי לפתור את הבעיה. במתכון לבישול או אפיה הפעולות הבסיסיות עשויות להיות: הקצף, טרוף, ערבב, הוסף. כמובן שההנחה היא שמבצע האלגוריתם יודע לבצע את הפעולות הבסיסיות (לדוגמה, הוא יודע כיצד מקציפים). בעולם החישובי הפעולות הבסיסיות הן הפקודות הקיימות בשפות

התכנות השונות. (במרבית שפות התכנות קיימות פקודות דומות). לדוגמה, באלגוריתם שהוצג מעל לשם בדיקת ראשוניות של מספר על מבצע האלגוריתם לדעת האם מספר הוא זוגי או פרדי, עליו להיות מסוגל לחשב שורש של מספר, כלומר לבצע פעולות שאינן לגמרי טריביאליות.

המילה אלגוריתם הינה שיבוש שמו של מתמטיקאי פרסי, בן המאה התשיעית: אבו ג׳עפר מחמד אל ח׳וארזמי (אל ח׳וארזמי שובש באנגלית לכדי אלגוריתם). לאורך למודיכם תכירו אלגוריתמים רבים הפותרים משימות מגוונות. בקורס זה נפתח צוהר לנושא, ע״י שנראה כיצד כותבים תכניות מחשב המשלימות משימות שונות (בשפה פורמאלית יותר אנו אומרים שהתכנית מממשת את האלגוריתם), כגון מיון של סדרת מילים או מספרים; חיפוש, במילים אחרות בדיקה, האם מילה\מספר כלשהו מצוי במאגר מילים\מספרים.

2. תכניות ראשונות

בפרק זה נציג מספר תכניות ראשונות, פשוטות. עבור כל תכנית נציג ראשית את הקוד שלה (כלומר, הפקודות שלה), ואחר נסבירה.

2.1 תכנית ראשונה: מבנה תכנית, ופקודת הפלט cout

נציג את התכנית עמה מקובל לפתוח את המסע בעולם התכנות:

```
#include <iostream>

int main()
{
     std::cout << "Hello World" ;

     return(0) ;
}
```

2.1.1 שורת ה- int main()

נדון ראשית בשורה השניה (`int main()`) שורה זאת מורה למהדר (ואחר גם למעבד) כי כאן מתחילה התכנית הראשית (main) שלנו; לצורך עניינו, בשלב זה, לא נדון בַּשאלה התכנית הראשית בניגוד לאיזה תכנית משנית? בשלב זה יש לנו רק תכנית ראשית, והשורה שכתבנו מציינת, כאמור, כי כאן היא מתחילה. עת המעבד יגש לבצוע התכנית (כמובן, רק אחרי שהיא תקומפל בהצלחה) הוא יתחיל את הביצוע מנקודה זאת. כל תכנית שנכתוב תכלול את השורה `int main()`, ותמיד ממנה יתחיל ביצוע התכנית.

בשורה השניה מופיעה בין היתר המילה `int` שהיא קיצור של integer או מספר שלם. משמעותה המדויקת של מילה זאת תובהר רק בהמשך, כעת נסתפק באמירה כי מילה זאת אומרת שהתכנית שלנו, עת היא מסיימת לרוץ, 'מחזירה' (במילים אחרות 'מודיעה') למערכת ההפעלה (שאפשרה לה לרוץ, כלומר שהתירה למעבד לבצעה) מספר שלם המציין באיזה אופן, או מאיזה סיבה, התכנית הסתיימה (האם אחרי שהתכנית השלימה את משימתה בהצלחה? או שמא אחרי שהתכנית גילתה כי היא אינה מסוגלת להשלים את פעולתה מסיבה זו או אחרת?). הפקודה האחרונה בתכנית (`return(0) ;`) היא הפקודה באמצעותה התכנית שלנו מחזירה\מודיעה למערכת ההפעלה את הערך הרצוי, במקרה שלנו את הערך אפס. גם משמעות המושג 'להחזיר ערך' תישאר עבורנו סתומה עוד זמן מה. סוגיה נוספת בה לא נדון בקורס זה היא: האם ומה עושה מערכת ההפעלה עם הערך המוחזר? כלומר, עת התכנית מסיימת, ומודיעה למערכת ההפעלה כי היא סיימה עם הקוד אפס (שכן היא מחזירה את הערך אפס), מה משמעות הדבר עבור מ.ה.. אעיר כי מי שעובד במערכת הפעלה ממשפחת יוניקס (כדוגת לינוקס) יכול אחרי הרצת התכנית להקליד את הפקודה: `echo $?` ויוצג לו הערך שהתכנית החזירה למערכת ההפעלה. עוד אציין כי הנוהג הוא שעת התכנית מסיימת בהצלחה, אחרי שהיא השלימה את משימתה, היא מחזירה את הערך אפס למערכת ההפעלה. אני נוהג לכתוב את הפקודה באופן: `return(0);` אולם למעשה הסוגריים אינם הכרחיים וניתן לכתבה גם באופן: `return 0 ;` .

למילים כגון int, main, return, מילים אשר מתארות פקודות של השפה אנו קוראים **מילים שמורות** (reserved words) או **מילות מפתח** (key words).

הסוגרים המסולסלים בשורה השלישית פותחים את גוף התכנית, ובני זוגם שבתחתית מורים שגוף התכנית נסגר. התכנית תופיע תמיד בין זוג סוגרים מסולסלים שכאלה. רבים נוהגים לכתוב את הסוגר הפותח בסוף השורה בה מופיע int main() (במקום להקצות לו שורה נפרדת).

לסיכום נאמר כי בשלב זה עליכם לזכור כי כל תכנית שלכם תיפתח בשורה: int main(), ותסתיים בשורה ; return (0). מהותן המדויקת של פקודות אלה תובהר רק בהמשך.

2.1.2 פקודת הפלט cout

השורה הרביעית בתכנית (std::cout << "Hello World" ;) כוללת את פקודת הפלט cout (הנהגית סי אאוט). הפקודה גורמת להצגת פלט רצוי על מסך המחשב. אחרי המילה cout מופיע זוג התווים >>. שימו לב לכיוונם (>> ולא <<), וכן הקפידו להצמידם זה לזה בלא רווח בניהם (>> ולא > >). אחרי שני תווים אלה מופיע: "Hello World". לרצף של תווים המופיעים בין גרשיים אנו קוראים **מחרוזת** או **סטרינג** (string). לפיכך גם: "xxx" וכן: "2X1" או: "@7%ללל יוסי !~{}" הן מחרוזות. עת המחשב פוגש במחרוזת בפקודת פלט הוא פולט את המחרוזת כמות שהיא (ללא הגרשיים שתוחמים אותה) למסך. בסיומה של פקודת ה- cout מופיע התו נקודה פסיק (;). תו זה מורה למחשב כי הסתיימה פקודה יחידה. במקרה שלנו הסתימה פקודת הפלט, בהמשך נכיר פקודות אחרות שתסתיימנה גם הן בנקודה פסיק. שימו לב כי בסוף השורה int main() לא מופיעה נקודה פסיק שכן בשורה זאת לא מופיעה פקודה.

חמשת התווים std:: המופיעים לפני המילה cout מורים לנו כי למעשה פקודת הפלט cout אינה חלק משפת C++ אלא הם חלק מ-'הספריה הסטנדרטית'. הספריה הסטנדרטית היא תוספות סטנדרטיות לשפה, אשר מגיעות יחד עם כל קומפיילר (std הוא קיצור של standard). מבחינתנו העובדה ש- cout אינה חלק מהשפה אינה משמעותית, שכן בכל סביבת עבודה בה נוכל לעבוד ב- C++, תמצא תמיד גם הספריה הסטנדרטית, ועל כן לנו יראה *כאילו* cout היא חלק משפת C++; אולם כדי להדגיש כי cout היא למעשה חלק מספריה זאת נדרשת הכתיבה: std::cout.

שימו לב כי בפקודת ה- cout כיוון החיצים הוא שמאלה: כביכול אנו 'מזרימים' את המידע *אל* הפלט, אל cout. בהמשך נכיר את פקודת הקלט, ובה כיוון החיצים יהיה הפוך שכן בה נזרים את המידע *מ*הקלט אל התכנית.

נניח שהתכנית שלנו היתה כוללת בנוסף לפקודה: std::cout << "Hello World" ; פקודה נוספת, שהיתה מופיעה מיד אחרי פקודה זאת והיתה: std::cout << "bye" ;
כיצד היה נראה הפלט שהיה מופיע על-גבי המסך? הפלט היה נראה כך: Hello Worldbye. כלומר המילים World ו- bye מוצגות זו אחר זו ללא רווח ביניהן. כדי להתגבר על תקלה זאת יכולנו לעשות אחד מכמה דברים: יכולנו להוסיף בתחילתו של הסטרינג הכולל את bye את התו רווח כך שהסטרינג היה נראה: " bye". מבחינתו של המחשב רווח הוא תו ככל תו אחר, ועת הוא נכלל בסטרינג הוא מוצג על המסך כמו כל תו אחר. במקרה שלנו הצגת תו זה היתה גורמת להפרדה הרצויה (לרווח) בין המילים World ו- bye. לעומת זאת, לו רצינו

שהסטרינג bye יופיע בשורה חדשה, מתחת ל- Hello World, היה עלינו לנקוט בפתרון שונה: את הפקודה ; "Hello World" << cout הינו משנים כך שהיא היתה נראית באופן הבא: ; cout << "Hello World" << std::endl נסביר: התווים >> שנוספו מורים שברצוננו לכלול מרכיב נוסף בפקודת הפלט, המילה std::endl (הוא קיצור של end line) מורה *שבעקבות* ביצוע פקודה זאת יש לעבור לשורה חדשה בפלט, כך שפלט נוסף, שיופיע בהמשך, יוצג בשורה חדשה. כלומר לַתוספת std::endl יש משמעות עבור פקודת הפלט הבאה (אם וכאשר זו תופיע). אתם רשאים לכלול בפקודת פלט מספר מרכיבי std::endl כפי רצונכם, כל מרכיב יגרום לקפיצת שורה בפלט. לדוגמה פקודת הפלט הבאה:

```
std::cout << std::endl << "Hello" << std::endl <<
"World" << std::endl << std::endl ;
```

תבצע: הפקודה תקפוץ שורה בפלט לפני הצגת המילה Hello, תקפוץ שורה בפלט אחרי הצגת Hello, (ולכן המילה World תופיע מתחת ל- Hello) ותקפוץ שתי שורות בפלט אחרי הצגת World, כך שפקודת הפלט הבאה תופיע לא בשורה הבאה במסך אלא עם שורה רווח.

כמובן שגם הכתיבה std::endl מציינת שהפקודה endl, הגורמת לקפיצת שורה, אינה חלק משפת C++, אלא היא חלק מהתוספת הקרויה הספריה הסטנדרטית.

ראינו כי עת תכניתנו כוללת כמה פקודות cout או כמה פקודות endl עלינו להקדים לכל פקודה את התחילית std:: . יש בכך משהו מוגיע, אותו נשמח לחסוך לעצמנו. אציג שתי דרכים לעשות זאת: אחת ראויה, והשניה נפוצה אך מאוד לא מומלצת. הדרך הראשונה היא להוסיף לתכנית, בין שורת ה- #include לשורת ה- int main() שורות נוספות:

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

בכך אנו מורים לקומפיילר כי בהמשך נשתמש בפקודות cout ו- endl הנכללות בספריה הסטנדרטית, ועליו להבין זאת. בגוף התכנית נכתוב: cout <<"hi" << endl << "bye" << endl ; כלומר כבר איננו מקדימים ל- cout ול- endl את התחילית std:: .

אדגיש כי עבור כל פקודה בספריה הסטנדרטית בה ברצוננו להשתמש עלינו להוסיף שורת using מעל שורת ה- int main() . (אך אל דאגה לא מדובר בכמות רבה מאוד של תוספות כאלה.)

פתרון שני שיחסוך לנו את הכתיבה std:: לפני כל פקודה, אולם שנחשב ע"י המהדרין לקלוקל, ועל-כן לא לגיטימי בקורס זה, הוא לכתוב מעל ה- int main() את השורה הבודדת: using namespace std; . בכך אנו מורים לקומפיילר כי נשתמש באופן חופשי בכל הפקודות הנכללות בספריה הסטנדרטית (במילים אחרות במרחב השמות הסטנדרטי); לא נצהיר על כל אחת ואחת מהפקודות בנפרד, ועליו להבין זאת. לצערי, בדוגמות רבות שתפגשו בספרים או ברשת תמצאו צורת כתיבה זאת. הסיבה היא שנוח מאוד להשתמש בה, והיא חוסכת עבודה, אולם כאמור היא נחשבת למאוד לא רצויה. ברמת הסיסמות אומר שהיא נחשבת בגדר 'זיהום של מרחב השמות'. מכיוון שמשפט זה לא באמת יכול להיות מובן לנו בשלב זה של חיינו, אזי אנו נוותר בגדר ה-'נעשה ונשמע' (בעיקר כדי שלא נולקה ע"י בודק התרגילים): במקומותינו לא משתמשים בשורה: using namespace std; נקודה!

מי מכם שיראה תכניות ישנות יותר בשפת C++ ימצא בהן את הפקודה:

```
#include <iostream.h>
```
במקום את הפקודה:
```
#include <iostream>
```
זוהי צורת כתיבה ישנה יותר, שכיום אינה מקובלת. עת השתמשו בצורת הכתיבה הישנה גם לא נדרשה ההתעסקות עם `std` כפי שתוארה מעל: לא היה צורך לכלול בתכנית לא פקודות `using`, ולא לציין `std::cout` . אולם, כאמור, צורת כתיבה זאת כבר עברה מין העולם.

פקודת הפלט `cout` קיימת רק ב- `C++`, בשפת `C` קיימת הפקודה `printf` אשר לא תוסבר כאן.

2.1.3 שורת ה- `#include`

נדון עתה בשורה הראשונה בתכנית: שורה זאת (`#include <iostream>`) תשאר עבורכם לאורך קורס זה כמעין מנְטְרה שאתם כותבים בתחילת כל תכנית בלי להבין למה. את מהותה המדויקת של הפקודה תלמדו רק בהמשך. באופן כללי נאמר כי לולא כללנו שורה זאת בתכנית לא היה הקומפילר מכיר את פקודת ה- `cout` והיה צועק עלינו שאנו כוללים בתכנית פקודה שגויה.

בשפת `C` (בניגוד ל- `C++`) היינו, כאמור, משתמשים בפקודת הפלט `printf`, ולצורך השימוש בה היינו מבצעים `include` לא ל- `iostream`, אלא ל- `stdio.h`. פקודת ה- `include` הייתה נכתבת באופן זהה.

2.1.4 עימוד

שימו לב כי את כל התכנית אנו כותבים באות קטנה (`lowercase`). שפת `C` מַבחינה בין אותיות קטנות לגדולות, ולכן לו כתבנו `INT MAIN()` או `Int main()` בְּמָקום `int main()` זו היתה שגיאה.

נושא נוסף עליו יש לתת את הדעת הוא ה**עימוד (הזחה, אינדנטציה** `indentation`**)**. שפת `C` מאפשרת לכם פורמט חופשי, כלומר אתם רשאים לכתוב את תכניתכם על הניירו\מסך באיזה מיקום שתחפצו, ולקומפיילר זה לא יפריע. לכן במקום לכתוב `int main()` יכולנו לכתוב:

```
int
    main
          ( )
```
אולם עימוד שכזה עלול להקשות מאוד על מתכנת אחר שיקרא את תכניתכם וינסה להבינה. אחת המטרות המרכזיות בכתיבת תכנית היא לכתוב תכנית קריאה. במקרים רבים הדבר חשוב יותר משאלת יעילותה של התכנית. מוסכמות (קונבנציות) רבות עוזרות להגדיל את הקריאות של תכניות. הראשונה ביניהן היא העימוד. כללי העימוד מורים למשל שאת השורה `int main()` כותבים כפי שכתבנו ולא באופן החלופי שהצגנו. כללי העימוד גם מורים כי פקודות התכנית המופיעות בין ה- `{` ל- `}` תופענה בהזזה ימינה כך שנקל יהיה להבחין בסוגריים, ולשים לב אילו פקודות תחומות ביניהם.

2.2 תכנית שניה: משתנים, טיפוסי משתנים ופעולות אריתמטיות

2.2.1 משתנים (variables)

באמצעות פקודות פלט כפי שהכרנו עד כה יוכל המתכנת לכתוב רק תכניות אשר עת תורצנה תצגנה על המסך את הגיגיו, וזה עלול להיות מעט מצומצם. כדי להרחיב את מגוון התכניות שביכולתנו לכתוב נרצה להכיר את ה**משתנים** (variables).

בפרק הקודם אמרנו כי זכרון המחשב עשוי לשמור נתונים ותכניות. עוד ציינו כי לכל תא בזיכרון יש כתובת באמצעותה אנו פונים לתא. כאשר אנו עובדים בשפה עילית, וברצוננו לשמור נתונים, איננו פונים לכתובות מפורשות בזיכרון. במקום זאת אנו נותנים לתאי הזיכרון בהם נרצה לעשות שימוש שמות. המחשב (או ליתר דיוק מערכת ההפעלה) מקצה לנו את תאי הזכרון בַּמַקום בו הוא בוחר, וכך שהוא ידע באיזה כתובת שוכן תא הזיכרון המכונה בתכנית בשם זה או אחר. אפשר לומר שתאי הזיכרון הם כמו 'תאי דואר' בהם ניתן לשים נתונים, ואשר מזוהים על-פי השם שאנו נותנים להם. נוכל ראשית לשים מידע בתוך תא כזה, אחר להתעניין במידע המצוי בתא, או לשנותו. לכן התאים גם נקראים משתנים—שכן ניתן לשנות את המידע המאוחסן בהם.

אנו נוהגים לקרוא לתאי הזיכרון בהם אנו שומרים נתונים **משתנים** שכן ערכם של תאים אלה עשוי להשתנות במהלך ריצת התכנית.

עת תכנית רוצה לעשות שימוש במשתנים עליה ראשית **להגדיר** (define) משתנים אלה, ולציין איזה סוג של נתונים המשתנים ידעו לשמור. בצורה פורמלית אנו אומרים שעל התכנית להגדיר את **טיפוס** (type) המשתנים.

נראה דוגמה: נניח כי ברצוננו לכתוב תכנית אשר מחברת את הערכים המצויים בזוג תאים המכילים מספרים שלמים (חיוביים ושליליים, אך ללא ספרות עשרוניות). על התכנית להכניס את הסכום לתוך תא זיכרון שלישי. כמו קודם, ראשית נציג את התכנית, ואחר נדון בה:

```
#include <iostream>

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

int main() {
   int num1, num2, sum ;

   num1 = 17 ;
   num2 = 3879 ;

   sum = num1 + num2 ;
   cout << "The sum of: " << num1 << " + " << num2
        << " = " << sum << endl ;

   return(0) ;
}
```

הפקודה ; int num1, num2, sum (בשורה החמישית) מורה למחשב כי בַהמשך התכנית שלנו מתעתדת להשתמש בשלושה תאי זיכרון (שלושה משתנים) אשר יהיו מסוגלים לשמור מספרים שלמים. השמות שאנו בוחרים לתת לשלושת המשתנים הם num1, num2, sum . המילה int לפני שמות המשתנים מורה כי טיפוס המשתנים שאנו מגדירים עתה הוא שלם, במילים אחרות המשתנים שלנו ידעו לשמור רק מספרים שלמים (חיוביים ושליליים). לו רצינו שתאי הזיכרון יוכלו לשמור גם מספרים שאינם שלמים, מספרים רציונאליים (כגון 0.78, 13.9, 0.007-) הינו מגדירים את המשתנים באופן הבא: double num1, num2, num3. המילה double מורה כי המשתנים שאנו מגדירים הם מטיפוס double , טיפוס המציין בשפת C מספרים ממשיים.

את שמות המשתנים אנו בוחרים על-פי רצוננו, ותחת הקפדה על מספר כללים: (א) שם של משתנה יתחיל תמיד באות או בקו התחתון (_) (underscore), (ב) שם של משתנה יכיל רק אותיות, ספרות וקווים תחתונים (ולכן השמות הבאים הם חוקיים: yosi, yosi13dan, yosi_cohen, _the_yosi, אך השמות הבאים אינם חוקיים: 13yosi, yosi!, yosi cohen (כאשר yosi cohen אמור להיות שם של משתנה יחיד הכולל בתוכו גם רווח)), (ג) שמו של המשתנה יעיד על תפקידו בתכנית (ולכן לא נשתמש בשמות משתנים כגון x, y אשר אינם מעידים על תפקידו של המשתנה).

הנקודה האחרונה מבין השלוש שמנינו היא מוסכמה נוספת אשר אמורה לסייע להגביר את הקריאות של תכניותינו. הקפידו עליה הקפדה יתרה. אני ממליץ שלא לתת למשתנים שמות בעברית. לדוגמה למשתנה אשר אמור להחזיק את שמו של המשתמש בתכנית קראו user_name ולא shem_hamishtamesh.

בשפת C, עת שם משתנה מורכב ממספר מילים *נהוג* להפריד בין המילים באמצעות הקו התחתון, לדוגמה: my_mother_name . בשפת ג'אווה נהוג לקרוא למשתנה באופן הבא: myMotherName, כלומר במקום להשתמש בקו התחתון מתחילים כל מילה המרכיבה את שם המשתנה, פרט למילה הראשונה, באות גדולה. יש המייבאים גם לשפת סי מוסכמה זאת. בשתי השפות, למרות שהדבר חוקי, לא נהוג ששם משתנה יתחיל בקו התחתון (כלומר השם: my_name_ אינו שם מקובל). קיימות גם מוסכמות אחרות לציון שמות משתנים, אולם אנו נתמקד בשתי אלה.

כאשר אנו מגדירים מספר משתנים מאותו טיפוס אנו מפרידים בין המשתנים השונים בפסיקים (,). בסוף הגדרת המשתנים מטיפוס כלשהו תופיע נקודה פסיק. ניתן להגדיר משתנים מטיפוסים שונים. אין מניעה מלחלק את הגדרת המשתנים המופרדים בפסיקים על-פני מספר שורות (ובהמשך נראה מדוע לעיתים הדבר גם שימושי), ואין מניעה מלהגדיר מספר פעמים משתנים מאותו טיפוס. נראה דוגמה שתמחיש את הנאמר:

```
int num1, num2 ;
double quot,
       prod,
       rem ;
int sum ;
```

2.2.2 פקודת ההשמה

עת אנו מגדירים משתנה, תא הזיכרון המתקבל מכיל ערך מקרי כלשהו, (לעתים נאמר שהמשתנה מכיל זבל). אין להניח כי טרם שהכנסנו ערך לתא יש בו את הערך אפס. כדי להכניס ערך למשתנה אנו משתמשים בפקודת **ההשמה** (מלשון לשים,

assignment): הפקודה: `num1 = 17;` היא דוגמה לפקודת השמה. בצד שמאל של הפקודה מופיע שם המשתנה לתוכו יש להכניס ערך, אחר-כך מופיע סימן ההשמה (התו =) ומימין לסימן ההשמה מופיע הביטוי (הערך) שיש להכניס למשתנה. לדוגמה הפקודה שהצגנו מכניסה למשתנה `num1` את הערך 17. באופן דומה הפקודה: `num2 = num1 + 9;` תכניס ל- `num2` את הערך 26 (בהנחה שב- `num1` יש 17). פקודת ההשמה מבוצעת על-ידי שראשית מחשבים את ערכו של הביטוי המופיע מימין ל- =, (לביטוי זה אנו קוראים ה- **rvalue** (קיצור של right value) של הפקודה, r= right), ורק אחר מאתרים את המשתנה לתוכו יש להכניס את הערך המחושב (כתובתו של משתנה זה בזיכרון נקראת ה- **lvalue** של הפקודה, l= left). מסיבה זאת הפקודה: `num1 = num1 + 1;` תתבצע כהלכה, וערכו של `num1` יגדל להיות 18. עת משתנה מקבל ערך חדש, הערך החדש מחליף את הערך הישן ואין דרך לשחזר את הערך הישן (אלא על-ידי ביצוע הפעולה ההפוכה).

2.2.3 פעולות אריתמטיות

ראינו את פעולת החיבור (המסומנת באמצעות הסימן +).
פעולות נוספות שקיימות על מספרים שלמים הן:

א. פעולת החיסור (המסומנת באמצעות התו -) לדוגמה: `num1 = num2 -9 ;`.

ב. פעולת הכפל (המסומנת על-ידי *) לדוגמה: `num2 = (num1 *3) +7 ;`.

ג. פעולת החילוק (מסומנת על-ידי /) לדוגמה: `num1 = num2 / 9`. שימו לב כי עת חילוק מבוצע על שני מספרים שלמים, הוא מתבצע כחילוק בשלמים עם שארית, ולכן קטע התכנית הבא:

```
num1 = 17;  num2 = 3;  cout << num1/num2 ;
```

ידפיס את הערך 5, שכן 3 נכנס 5 פעמים ב- 17 (ונותרת שארית 2). בהזדמנות זאת גם אעיר כי מותר, אך מאוד לא מומלץ להכניס מספר פקודות לשורה אחת בתכנית, כפי שעשיתי בדוגמה זאת; ראוי לכתוב כל פקודה בשורה נפרדת.

ד. פעולת השארית (מסומנת על-ידי %) היא המשלימה לפעולת החילוק, היא מחזירה את שארית החלוקה. לדוגמה: `num1 = 21; cout << num1 % 4 ;` יציג את הערך 1 (שכן 4 נכנס 5 פעמים ב- 21 והשארית היא 1). השארית תהיה אפס עת המחולק מתחלק בדיוק במחלק, כמו במקרה 4 % 16. פעולת השארית היא פעולה נפרדת מפעולת החילוק, היא מתעניינת אך ורק בשארית. אם ברצוננו לקבל הן את המנה, והן את השארית, יהיה עלינו לבצע שתי פעולות, כל אחת תחזיר את הערך שהיא מחשבת. פעולת השארית קרויה לעתים פקודת ה-'מודולו'. כמו המנה, כך גם השארית אינה מוגדרת עת המחלק הינו אפס.

על משתנים מטיפוס `double` קיימות הפעולות חיבור, חיסור, כפל, חילוק (רגיל), והן מסומנות באותו אופן כמו על שלמים. פעולת השארית לא קיימת על ממשיים. שימו לב כי אותו סימן: / , משמש אותנו לשתי פעולות דומות אך שונות: חילוק בשלמים, וחילוק בממשיים. המחשב קובע איזה פעולה תבוצע על-פי הנתונים אותם יש לחלק: אם שניהם שלמים יבוצע חילוק בשלמים; אם לפחות אחד מהם ממשי אזי יבוצע חילוק בממשיים.

אחדד את הנקודה שהוזכרה בפיסקה הקודמת. נניח שהגדרנו:

```
int num1, num2 ;
double x, y ;
```

והשמנו:

```
num1 = 23 ;
num2 = 4 ;
```

כבר אמרנו שהפקודה: `cout << num1/num2 ;` תדפיס את הערך 5, שכן 4 נכנס 5 פעמים ב- 23. באופן דומה הפקודה: `x = num1/num2;` תכניס ל- `x` את הערך 5 (ולא את הערך 5.75 כפי שכמה מכם היו עשויים לחשוב). הסיבה היא שכפי שאמרנו פעולת ההשמה ראשית מתעניינת באגף ימין של ההשמה, ושם היא מוצאת פעולת חילוק. מכיוון שפעולת החילוק מחלקת שני שלמים (`num1, num2` הם מטיפוס `int`), אזי היא מבוצעת כחילוק בשלמים, ולכן מחזירה את המנה חמש. רק עתה פקודת ההשמה שואלת את עצמה לאן יש להכניס את הערך חמש?, מגלה שיש להכניסו ל- `x`, ולכן `x` מקבל את הערך חמש. בסעיף הבא נראה כיצד ניתן לגרום לפעולת החילוק הנ"ל להחזיר את הערך `5.75` כך שהוא זה שיוכנס ל- `x`.

לפעולות השונות שאנו מבצעים (כגון חיבור, חיסור) אנו נוהגים לקרוא בשם **אופרטורים** (operators). הנתונים עליהם פועלות הפעולות נקראים **אופרנדים** (operands). לדומה בביטוי `num1 % 17` האופרטור % מופעל על האופרנדים `num1` ו- 17; בביטוי `num1 % (num2 – 3)` האופרטור % מופעל על האופרנדים `num1` ו- `(num2 – 3)`. אופרטור כדוגמת % אשר פועל על שני אופרנדים נקרא **אופרטור בינארי** (binary operator). לעומתו אופרטור המינוס, אשר פועל על אופרנד יחיד ומחזיר את הנגדי שלו נקרא **אופרטור אונארי** (unary operator). גם אופרטור הסינוס אשר מקבל זווית ומחזיר את סינוס הזווית הוא אופרטור אונארי).

לאופרטורים השונים מוגדרת **קדימות** (precedence) אשר קובעת באיזה סדר יוערך ביטוי הכולל מספר אופרטורים שונים. כולכם זוכרים, ודאי, כי ערכם של הביטויים 2*4+3 ו- 3+2*4 הוא 11 שכן קדימותו של הכפל גבוהה משל החיבור, ולכן ראשית תחושב המכפלה 2*4 ורק אחר יוסף למכפלה הערך 3. בהמשך נכיר אופרטורים נוספים. לטעמי עדיף שלא להסתמך על סדר הקדימויות הקבוע בשפה וראוי לשים סוגריים אשר יבהירו לכם, למחשב, ולמי שיקרא את תכניתכם כי יש לבצע את החישוב באופן זה או אחר. על כן אני נוטה להמליץ לכתוב את הביטויים שראינו קודם באופן: (2*4) + 3 או 3+ (2*4), וכך אין ספק איזה פעולה מבוצעת מתי.

שאלה נלווית לשאלת הקדימות היא שאלת **האסוציאטיביות** (associativity): בהינתן ביטוי כגון: 1-3-4 מה ערכו? האם ראשית יש לחשב את פעולת ההפרש השמאלית 3-4 וממנה להחסיר אחד? ולקבל את התוצאה אפס, או שמא ראשית יש לחשב את פעולת ההפרש הימנית 3-1 ואת תוצאתה להחסיר מ- 4, כלומר התוצאה היא שתיים. עבור אופרטור שהאסוציאטיביות שלו שמאלית (כמו אופרטור המינוס) יש לבצע ראשית את הפעולה השמאלית; עבור אופרטור שהאסוציאטיביות שלו ימנית יש לבצע ראשית את הפעולה שמימין. כמו במקרה הקודם, אני נוטה להמליץ על שימוש בסוגריים שיבהירו בדיוק את הרצוי, ויחסכו את הצורך לזכור מהי האסוציאטיביות, במילים אחרות באיזה סדר יבוצע החישוב.

2.2.4 המרת טיפוס

לעיתים אנו מעוניינים לבצע על משתנים שלמים חילוק כמו על מספרים ממשיים ולקבל את המנה המדויקת (בממשיים). לדוגמה: נניח שערכו של המשתנה השלם `num1` הוא 23, ואנו רוצים לחלקו ב- 4 ולקבל את המנה 5.75 (ולא 5 כפי שנקבל עת נבצע חילוק בשלמים שעה שנכתוב `num1/4`). כדי להשיג זאת נבצע את התעלול הבא: `cout << double(num1)/4`. נסביר: הפעולה `double (num1)` היא פעולת **המרת טיפוס** (type conversion). הפעולה *מחזירה* את הערך הממשי השקול לערך שהועבר לה. שימו לב כי *איננו* משנים בכך את טיפוסו של המשתנה `num1`, `num1` נולד שלם, חי שלם, וימות שלם, אנו רק מחשבים את הערך הממשי 23.0

השווה לערך השלם 23. את הערך שחושב אנו מעבירים לפעולת החילוק אשר עתה (מכיוון שאחד האופרנדים שלה הוא ממשי) מתבצעת כחילוק בממשיים.

לכל ביטוי במחשב יש טיפוס (במילים פשוטות יותר מִן או סוג), שהוא טיפוס הערך שהביטוי מחשב. לדוגמה: טיפוסו של הביטוי: `(num1 + num2) % (num1 – num2)` הוא שלם, שכן תוצאתו של ביטוי זה הוא מספר שלם. לעומת זאת הביטוי `double (num1)/4` הוא ביטוי ממשי שכן תוצאתו היא מספר ממשי. ניתן להכניס ערך מטיפוס א' לתוך משתנה מטיפוס ב', רק אם הטיפוס ב' מכיל בתוכו את הטיפוס א'.. על כן, לדוגמה, אם הגדרתם בתכנית כלשהי את המשתנים: `int int_var ; double double_var ;` אזי גם אם `double_var` מכיל כעת במקרה ערך שלם, ההשמה: `int_var = double_var;` נחשבת להשמה שגויה. הקומפיילר אומנם יתרגם את תכניתכם לשפת מכונה (ולא יִרְאה בכך שגיאה המונעת את התרגום) אך הוא יוציא לכם אזהרה (warning). מכיוון שאנו משתדלים לכתוב תכניות נקיות ככל האפשר אנו משתדלים להימנע מאזהרות קומפיילר. כדי להימנע מהאזהרה נשתמש בפעולת המרת הטיפוס ונחליף את ההשמה: `int_var = double_var;` בהשמה: `int_var = int(double_var);`. מה עשינו? ראשית המרנו את ערכו הממשי של הביטוי `double_var` בַּערך השלם המקביל לַערך הממשי של המשתנה; שנית, ערך שלם זה הכנסנו למשתנה `int_var`. במידה וערכו של הביטוי הממשי כולל שבר יבוצע **קיצוץ**, ולמשתנה השלם יוכנס הערך השלם הגדול ביותר שעדיין קטן מהערך הממשי. לדוגמה, נניח כי ערכו של `double_var` הוא שבע, אזי ההשמה: `int_var = int(double_var/4)` תכניס ל- `int_var` את הערך 1, שכן 7/4 = 1.75, ועת אנו ממירים את 1.75 לכדי ערך שלם אנו קוצצים אותו ומקבלים את הערך השלם 1 (שימו לב כי מכיוון שאחד האופרנדים של פעולת החילוק הוא ממשי החילוק המבוצע הוא חילוק בממשיים). באותו אופן: `int(-3.5)` הוא -3.

את פעולת המרת הטיפוס ניתן לכתוב כפי שראינו, באופן: `int(double_var)` או לחילופין באמצעות התחביר: `(int) double_var`. אם נדייק אזי התחביר הראשון מוכר רק ב- C++, בשפת C חובה להשתמש בכתיבה: `(int) x`.

אין הכרח לבצע המרת טיפוס עת מכניסים ערך מטיפוס צר יותר לתוך משתנה מטיפוס רחב יותר. על-כן לדוגמה עת מכניסים ערך שלם למשתנה ממשי אין צורך לבצע המרת טיפוס, ניתן לכתוב: `double_var = int_var;` ואין צורך לכתוב: `double_var = (double) int_var;`

2.2.5 טיפוסים מספריים נוספים

בשפת C קיימים טיפוסי משתנים רבים. את חלקם נכיר בהמשך. נמנה עתה כמה טיפוסים מספריים (הדומים על-כן ל- `int` ו- `double` שהכרנו). ניתן לחלק את הטיפוסים לשתי קבוצות: כאלה המיועדים לשמירת מספרים שלמים (חיוביים ושליליים), וכאלה המיועדים לשמירת מספרים רציונאליים (שברים, שלמים, ומספרים מעורבים).

א. הטיפוס `int` מתאים לשמירת מספרים שלמים, חיוביים ושליליים, בגודל 'רגיל', כלומר כאלה שאינם גדולים או קטנים באופן מיוחד בערכם המוחלט. טווח הערכים שניתן לשמור במשתנה מטיפוס `int` הוא תלוי קומפיילר, במקרים רבים מקצה הקומפיילר לכל משתנה שלם שנים או ארבעה בתים, כלומר 16 או 32 סיביות, במצב בו מוקצים למשתנה שני בתים ניתן לשמור בכל

משתנה שלם 2^{16} מספרים שונים, חציים (כמעט) חיוביים, וחציים שליליים. אם אין סיבה מיוחדת לרצות לשמור מספרים שלמים גדולים במיוחד או קטנים במיוחד אזי ראוי להשתמש בטיפוס זה אשר מותאם על-ידי הקומפיילר למעבד עליו מורצת התכנית—לגודל המספרים עליו פועל מעבד זה 'באופן טבעי'.

ב. במידה וברצונכם לשמור ערכים שלמים גדולים במיוחד בערכם המוחלט השתמשו בטיפוס `long int` (או בקיצור `long`). מנגד, במידה וברצונכם לשמור מספרים שלמים יחסית קטנים (בערכם המוחלט) השתמשו בטיפוס `short int` (או בקיצור `short`). מספר הבתים המוקצים לשמירת משתנה מטיפוס `long, short` הוא תלוי קומפיילר; במקרים רבים יוקצו למשתנה מטיפוס `short` שני בתים (כלומר כמו ל- `int`), ולמשתנה מטיפוס `long` ארבעה או שמונה בתים. כלומר הטיפוס short int מוכל בטיפוס int, והטיפוס long int מכיל את הטיפוס int.

ג. במידה ומשתנה שלם כלשהו (`int, short int` או `long int`) אמור לשמור רק ערכים טבעיים (כלומר חיוביים ואפס) ניתן להוסיף לפני שם הטיפוס את מילת המפתח `unsigned` ובכך להורות למחשב כי משתנה זה לא יאחסן מספרים שליליים, ועל כן יוכל להחזיק מספר כפול של מספרים חיוביים בהשוואה לעמיתו שאינו `unsigned`. על כן אתם יכולים להגדיר משתנים כדוגמת הבאים:

```
int a ;              // a normal signed int
unsigned int b;      // unsigned int i.e., natural number
unsigned c ;         // same as unsigned int c ;
long int d ;
long e ;
unsigned long f ;
```

ד. במידה וברצונכם לשמור מספרים רציונאליים 'רגילים' השתמשו בטיפוס double.

ה. על מנת לשמור מספרים רציונאליים גדולים במיוחד או קטנים במיוחד (כלומר קרובים לאפס) השתמשו בטיפוס `long double`.

ו. ניתן להגדיר גם משתנים ממשיים מטיפוס `float` להם מוקצה פחות זיכרון מאשר למשתנים מטיפוס `double`. כלומר הטיפוס `float` מוכל בטיפוס `double`, והטיפוס `long double` מכיל את הטיפוס `double`.

ז. בשפה מוגדר גם הטיפוס `char`. גם טיפוס זה מאפשר שמירה של מספרים שלמים קטנים בערכם המוחלט (יש מערכות בן הטיפוס מיועד לשמירה של מספרים טבעיים בלבד, ויש כאלה בהן באופן מחדלי הוא מיועד לשמירת מספרים חיוביים ושליליים. למשתנה מטיפוס `char` מוקצה בית יחיד בזיכרון). פרק תשע דן בהרחבה במשתנים מטיפוס `char` שכן ייעודם המרכזי הוא שמירה של תווים.

ח. בשפת C++, אך לא בשפת C, מוגדר הטיפוס bool. נדון בו בהמשך.

מישהו עלול לומר לעצמו: "מדוע עלי להסתבך עם שלל הטיפוסים השונים והמשונים? אגדיר את כל המשתנים בתכניתי מטיפוס double ואז אוכל להכניס לתוכם כל ערך שארצה!" מה נשיב לו כגמולו, לרשע זה? התשובה היא שטיפוסו של המשתנה צריך להתאים לסוג הערכים שהמשתנה אמור להכיל. על-כן אם משתנה אמור להכיל מספרים שלמים בלבד אין להגדירו מטיפוס `double` (גם אם הדבר אפשרי). וכל כך למה? שכן אם אנו סוברים שהמשתנה אמור להחזיק רק ערכים שלמים, ובכל זאת נגדירו מטיפוס double אזי אם בתכנית שלנו תחול שגיאה, ולמשתנה יוכנס בטעות ערך לא שלם; במצב זה הקומפיילר לא יעזור לנו לגלות את

השגיאה, הקומפיילר לא יתריע דבר, שכן מבחינתו, (בהנחה שהמשתנה מוגדר מטיפוס double) אין בעיה להכניס למשתנה ערך שאינו שלם. אך אם המשתנה יוגדר מטפוס int, ובטעות ננסה להכניס לתוכו ערך שאינו שלם, אזי הקומפיילר יתריע על כך, וייסיע לנו למצוא את השגיאה בתכנית. תלמידים, לא אחת, עובדים 'בראש פרנואידי'—התלמידים בהרגשה שמטרתם לקבור עמוק ככל האפשר את השגיאות בתכניתם על מנת שבודק התרגילים לא יגלה את השגיאות, וילקה אותם. זה מובן. אולם רוב העולם לא מעוניים להעלים את השגיאות שנופלות בתכניות שהוא כותב, אלא לגלותן (לפני שהתכניות יגיעו לידי הלקוח). לכן בד"כ אנו מעוניינים שהקומפיילר יגלה את השגיאות שלנו, ולכן, בפרט, נגדיר את המשתנים מהטיפוס המתאים להם.

מה יקרה אם ננסה להכניס למשתנה מטיפוס כלשהו ערך החורג מתחום הערכים שאותו משתנה עשוי להכיל, (לדוגמה: נניח כי מספר שלם יכול להכיל ערכים הקטנים או שווים 32000 ואנו מנסים להכניס למשתנה את הערך 50000)? מצב בו אנו מנסים להכניס למשתנה ערך גדול מכפי שהמשתנה יכול לאחסן נקרא **גלישה** (overflow), והוא, כמובן, מצב בלתי רצוי, אשר עלול לגרום לשגיאה בפלט שהתכנית תציג. על דרך השלילה: התכנית לא תעוף בשל גלישה.

בעיה אחרת בה אנו עשויים להיתקל בעת ביצוע חישוב כלשהו על-ידי תכניתנו היא בעיית הדיוק. נניח כי נכלול בתכניתנו את הפקודה הבאה:

```
cout << 1 - double(1)/3 - double (1)/3 - double (1)/3 ;
```

על-פי כללי האריתמטיקה אמורה הפקודה להציג את הפלט אפס; בפועל לא בהכרח זה יהיה הפלט שיתקבל. הסיבה לכך היא שבעת שעל התכנית לשמור שבר אינסופי, כדוגמת שליש, היא מאחסנת *קירוב* לערך האמיתי. הקירוב, כדרכו של קירוב, אינו שווה לערך המדויק, ועל-כן נוצרת שגיאה.

2.2.6 פקודת הפלט `cout`

לפני שאנו מסיימים סעיף זה נחזור לתכנית עמה פתחנו את הסעיף ונבחן את פקודת הפלט:

```
cout << "The sum of: "
     << num1 << " + " << num2 << " = "
     << sum << endl ;
```

פלט הפקודה יראה על המסך באופן הבא: `The sum of 17 + 3879 = 3896`.
שימו לב כי עת אנו כותבים `num1` שלא בין מרכאות מבין המחשב כי ברצוננו להציג את ערכו של *המשתנה* `num1` (ואם לא הגדרנו בתכנית משתנה בשם זה אנו מבצעים שגיאה עליה יעיר לנו הקומפיילר), לעומת זאת עת אנו כותבים " + " מציג המחשב את סימן ה- + עם רווח לפניו ואחריו (שכן הסטרינג שלנו כולל שלושה תווים בדיוק: רווח, סימן הפלוס ורווח נוסף). עד כאן ההסבר ברמה הטכנית. ברמה העקרונית הדוגמה שלפנינו מציגה את הכלל לפיו תמיד יש להסביר למשתמש מה הפלט המוצג לו. יכולנו להציג למשתמש גם את הפלט: `cout << sum;` אולם אז משתמש שאינו מכיר את תכניתנו היה עשוי לעמוד נבוך ולשאול את עצמו מה משמעותו של המספר המופיע לנגד עיניו על המסך. על כן תמיד, בכל תכנית לצד הנתון המוצג יופיע פישרו.

2.3 תכנית שלישית: פקודת הקלט `cin`, ותיעוד

התכנית האחרונה שכתבנו מציגה בפני המשתמש את סכומם של שני המספרים השלמים אותם הכניס *המתכנת* למשתנים `num1` ו- `num2`. בכל פעם שהמשתמש יריץ את התכנית יוצג בפניו, לפיכך, אותו פלט. המשתמש בתכנית לא יוכל להזין לה

שני מספרים כרצונו ולקבל מהתכנית את סכום המספרים שהוא הזין. זכרו כי כדי שהתכנית שכתבנו תציג סכום של שני מספרים אחרים (מ- 17 ו- 3879) יש לשנות את *התכנית*: יש להשים למשתנים שני ערכים אחרים, אחַר יש לקמפל שוב את התכנית, ורק אז עת היא תורץ היא תציג את סכום הערכים החדשים. סביר להניח שהמשתמש אינו מתכנת, הוא אינו יודע כיצד משנים את הערכים שהוכנסו למשתנים, וכיצד מקמפלים תכנית, על-כן הוא לא יוכל לבצע שנוי שכזה על-מנת שהתכנית תחשב לו סכום של שני מספרים אחרים. מעבר לכך, בד"כ המשתמש אינו מקבל את תכנית המקור (כלומר, את הקוד שכתבנו בשפת C), אלא את התכנית כשהיא כבר מתורגמת לשפת מכונה, ובה ודאי לא ניתן להכניס שינויים.

2.3.1 פקודת הקלט `cin`

כיצד אם כן נכתוב תכנית שתוכל לקלוט מהמשתמש שני מספרים שלמים כלשהם (שהמשתמש יזין), ואחר להדפיס את סכומם? לשם כך עלינו להכיר את פקודת הקלט `cin` (נהגית: סי אין). נציג תכנית הכוללת פקודה זאת ואחר נסבירה:

```
#include <iostream>

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

int main() {
   int num1, num2 ;

   cout << "Enter two integer numbers: " ;
   cin >> num1 >> num2 ;

   cout << "The sum of: " << num1 << " + " << num2
        << " = " << num1+num2 << endl ;

   return(0) ;
}
```

הפקודה בה נתרכז בתכנית זאת היא הפקודה: `cin >> num1 >> num2 ;` (שימו לב כי כיוון החִצים הפעם הוא ימינה: אנו 'מזרימים' נתונים מהקלט *לתוך* המשתנים; וזאת בניגוד לפקודת ה- `cout` בה כיוון החצים הוא שמאלה שכן היא 'מזרימה' נתונים *מתוך* המשתנים לפלט). עת המעבד מתחיל לבצע את התכנית הוא ראשית מבצע את פקודת הפלט (`cout`), הפקודה הבאה בה הוא נתקל היא פקודת ה- `cin`. עת המחשב נתקל בפקודה זאת הוא עוצר את ביצוע התכנית וממתין שהמשתמש יזין לתכנית קלט. על המשתמש להזין שני מספרים שלמים עם רווח או `Enter` ביניהם (וְלא עם פסיק או כל תו אחר ביניהם!), ואחר להקיש על מקש ה- enter. שני המספרים שהמשתמש יזין יכנסו למשתנים `num1`, ו- `num2` בהתאמה. עתה המחשב יוכל להמשיך בביצוע התכנית, בפְרט הוא יוכל להציג את סכומם של שני המספרים שהוזנו.

(המשתמש רשאי להקיש לפני, אחרי ובין המספרים גם רווחים, Enter-ים או tab-ים כרצונו, אך לא שום תו אחר פרט לשלושת אלה.)

שימו לב כי לפני פקודת הקלט מופיעה בתכנית פקודת פלט אשר מנחה את המשתמש מה הקלט שעליו להזין. כלל הזהב למתכנת המתחיל הוא: לפני כל פקודת קלט תופיע פקודת פלט המנחה את המשתמש מה הקלט המצופה ממנו. כלל זה משלים את הכלל: כל פקודת פלט תכלול גם הסבר למשתמש מה מהות הפלט המוצג בפניו.

נקודה נוספת אותה ניתן ללמוד מהתכנית האחרונה היא כי המחשב נבון דיו להתמודד עם פקודות פלט כגון: `cout << num1+num2;` , כלומר הוא ידע לחַשב ולהציג את ערכו של הביטוי האריתמטי הנכלל בפקודת הפלט.

שימו לב כי בעוד בפקודת הקלט החיצים פונים ימינה: מכיוון הקלט אל המשתנים לתוכם מוכנס המידע, בניגוד לכך, בפקודת הפלט החיצים פונים שמאלה: מכיוון המשתנים אל הפלט אליו מוזרם המידע.

במידה ואנו מבצעים: `cin >> num;` עבור משתנה `num` שטיפוסו שלם, והקלט המוזן על-ידי המשתמש אינו מספר שלם (אלא מספר רציונאלי, או מחרוזת אחרת) הקלט נכשל. בניגוד למה שאולי הייתם יכולים לצפות הדבר לא יגרום להפסקת ביצועה של התכנית (להעפתה), אלא לתופעה מוזרה אחרת בה נדון בעתיד הרחוק. לעת עתה נאמר זאת כך: בשלב הנוכחי של חיינו, תמיד נניח שאם המשתמש מצופה להזין נתון מטיפוס כלשהו הוא אכן עושה כן. המשתמש עשוי להזין *ערך שגוי*, ועם זאת עלינו להתמודד (למשל ציפינו למספר חיובי אך הוזן שלילי), אך תמיד נניח שאם אנו מצפים למספר שלם אזי זה גם יהיה טיפוס הקלט.

הזכרנו כי בשפת C (בניגוד ל- C++) פקודת הפלט המשמשת אותנו היא `printf`, באופן דומה פקודת הקלט מתאימה היא `scanf` (ולא `cin`).

2.3.2 תעופה של תכנית

מה יקרה אם במקום להזין שני מספרים שלמים יזין המשתמש קלט אחר (נניח מספרים ממשיים או תווים כלשהם שאינם ספרות)? במקרה זה המחשב יעצור את תהליכי הקלט של התכנית, ולא יקרא עוד נתונים לַמשתנים השונים (ביצוע התכנית ימשיך, באופן מאוד לא מוצלח, עם ערכי המשתנים הנוכחיים). תקלה דומה אך שונה תחול אם תכניתנו תנסה לבצע פעולה אסורה, כדוגמת חילוק באפס *בשלמים*. במקרה זה (בניגוד לקודמו) המחשב יעצור מיידית את ביצוע התכנית ויודיע כי **התכנית עפה** (aborted / terminated). מצב בו התכנית אינה מסוגלת עוד לקרוא נתונים מהמשתמש, קל וחומר מצב בו התכנית עפה נחשב למצב לא רצוי. כעיקרון, ניתן לומר שחובתנו לכתוב תכניות שתשלמנה את פעולתן כהלכה, וכמובן תצגנה את הפלט המתאים עבור כל קלט שהוא. אנו שואפים שגם אם המשתמש עושה שימוש לא ראוי בתכנית (כגון מזין אותיות במַקום בו הוא מצופה להזין מספרים) התכנית תתגבר על המצב, ותציג פלט מתאים. בפועל, בתכניות שנפגוש בקורס זה, לא נגן על התכנית הגנה מלאה בפני תקלות הנובעות משימוש לא נכון בתכנית. הסיבה לכך היא שנעדיף להשקיע את מרצנו באפיקים אחרים. יחד עם זאת כן נקפיד להקדים לכל פקודת קלט פקודת פלט אשר מסבירה למשתמש מה הקלט שעליו להזין (ונקווה שהוא יפעל על-פי ההנחיות).

אעיר כי חילוק באפס בממשיים מכניס לַמשתנה ערך המסומן כ: Inf (קיצור של infinite או אינסוף) ולא גורם להעפה של התכנית. לפי מיטב חוקי המתמטיקה, הגדלה או הקטנה של ערך זה **ב**שיעור כלשהו או **פי** שיעור כלשהו לא תשנה את ערכו של המשתנה, אך השמה חדשה לתוכו, שלא על סמך הערך הקיים, כמובן

תכניס למשתנה ערך חדש. מה שבעיקר חשוב לזכור הוא שחילוק באפס גורם להעפה של התכנית בשלמים, אך לא בממשיים.

2.3.4 תיעוד (documentation)

הזכרנו כי אחד ההיבטים החשובים ביותר בתכניות שנכתוב הוא נושא הקריאות של התכנית למתכנת אחר. כאשר אנו כותבים תכנית גדולה, המבצעת משימה מורכבת, הבנתה הופכת להיות משימה לא פשוטה אשר עלולה לצרוך זמן ומאמץ מרובים. כפי שודאי ידוע לכם זמן שווה כסף, ועל-כן נשאף לקצר את משך הזמן שיידרש ממתכנת אחר עד שהוא יכנס לרזי תכניתנו. אמרנו כי בחירת שמות משתנים משמעותיים, ועימוד התכנית על-פי המקובלות, הם שני כלים שבאים להגביר את הקריאות. הכלי המרכזי בשרות הגברת הקראות הוא **התיעוד** (documentation). תיעוד הוא הערות שאנו מוסיפים לתכניתנו, הערות שכל תפקידן הוא לסייע למתכנת אחר להבין את התכנית (עת הקומפיילר מתרגם את התכנית לשפת מכונה הוא מתעלם לחלוטין מהערות אלה). אנו מחלקים את התיעוד לשלושה מרכיבים:

א. <u>תיעוד ראשי</u>: יופיע לפני התכנית (מעליה), יכלול הסבר כללי אודות התכנית:
   1. כותרת המציינת במספר מילים מה התכנית עושה.
   2. פרטי המתכנת שכתב את התכנית.
   3. תיאור האלגוריתם על-פיו פועלת התכנית; במילים אחרות הסבר מה הצעדים שהתכנית שלכם מבצעת בכדי להשלים את המשימה לשמה היא נכתבה.

ב. <u>תיעוד משתנים</u>: לצד הגדרת *כל* משתנה ומשתנה יופיע הסבר מה תפקידו של המשתנה. אמרנו, אומנם, כי ניתן למשתנים שמות משמעותיים, אולם השמות המשמעותיים אינם פוטרים מהצורך גם לתעד את תפקידם של המשתנים.

ג. <u>תיעוד בגוף הקוד</u>: תיעוד זה יופיע כל כמה פקודות (כל בלוק) ויתאר במשפט *קצר* מה מבצע בלוק זה. בהנחה שהתיעוד הראשי נעשה כהלכה תיעוד זה יוכל להיות רק הרחבה מעטה של הקטע המתאים בתיעוד הראשי.

בשפת C קיימות שתי דרכים לכתוב תיעוד:

א. אם במַקום כלשהו בתכנית נכתוב /* (לוכסן נוטה ימינה ובצמוד לו כוכבית, בלי רווח ביניהם) יבין הקומפיילר כי אנו פותחים כאן הערת תיעוד, וכל מה שיופיע עד ל- */ הבא הוא הערה ממנה עליו להתעלם.

ב. אם במַקום כלשהו בתכנית נכתוב // (זוג לוכסנים נוטים ימינה צמודים זה לזה ללא רווח ביניהם) יבין הקומפיילר כי אנו פותחים כאן הערת תיעוד אשר משתרעת עד סוף השורה (ומה שמופיע בשורה הבאה כבר אינו הערה). הערה זאת אין צורך לסגור מפורשות, היא מסתיימת עת מסתיימת השורה.

כמה הערות אודות תיעוד:

א. הניסיון מראה שמתכנתים מתחילים נוטים לתעד יתר על המידה (בעוד עמיתיהם הותיקים מתעדים פחות מדי). הימנעו מתיעוד יתר. התיעוד הראשי שלכם לא אמור להשתרע על-פני יותר מעשר עד 15 שורות, והתיעוד בגוף הקוד לא ישתרע על-פני יותר משתיים שלוש שורות בכל פעם.

ב. זכרו כי מי שקורא את תכניתכם הוא מתכנת מיומן. אין צורך להסביר לו *מה* אתם עושים אלא *למה* אתם עושים, כלומר באיזה אופן קטע התכנית מקדם את השלמת המשימה.

ג. הפרידו תיעוד מקוד. את תיעוד המשתנים מקמו ככל האפשר בחלקו הימני של הדף\המסך כך שהתיעוד לא יתערבב בקוד ולא ימְנע מתכניתכם להראות ויזואלית כפי שתכנית מחשב נראית, כלומר שהתכנית שלכם לא תראה כמו טקסט רגיל אלא כמו תכנית מחשב. באופן דומה את התיעוד בגוף הקוד מקמו מעל הבלוק אותו אתם מתעדים ובמידת האפשר מוזז לימין כך שמי שיבחר לא לקרוא את התיעוד יוכל לאתר בקלות היכן מתחיל ונגמר כל הסבר.

ד. אל תתביישו להקיף את התיעוד שלכם במסגרת או לשים משמאלו קו אנכי של כוכביות (כפי שאדגים בהמשך). אסתטיקה היא דבר חשוב בחיים בכלל, ובתכניות מחשב בפרט; היא לא חזות הכל, אך אין להתעלם ממנה.

ה. תלמידים נוטים לעיתים לחשוב כי תכנית קצרה יותר (בסנטימטרים על הדף) היא גם תכנית טובה יותר. חלקית יש בכך מן האמת שכן תכנית מסורבלת יותר תהיה במרבית המקרים גם ארוכה יותר, וקצר עשוי להיות פשוט ואלגנטי; אולם קו מחשבה זה גורם לפעמים לתלמידים לכתוב את תכניתם בדחיסות נוראה. הישמרו מכך! אל תכתבו יותר מפקודה אחת בשורה (אפילו אם מדובר בפקודות קצרות). הכניסו שורות ריקות בין חלקים שונים של התכנית כדי לסייע לקורא להבחין ויזואלית שכאן נגמר קטע א' ומכאן מתחיל קטע ב'.

נציג כעת כיצד תראה התכנית האחרונה שכתבנו עם תיעוד. בהמשך אציג תכניות מתועדות נוספות על-מנת לסייע לכם ללמוד כיצד לתעד.

```
/*****************************************************
 *
 *              A program that sums two ints
 *              ============================
 *    Writen by: Yosi Cohen, id = 333444555, class: a1
 *
 * Algorithm: The program reads from the user two ints
 * ========== and prints their sum.
 *
 *
 *****************************************************/
#include <iostream>

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


int main() {
   int num1, num2;                        // the two inputs

   cout << "Enter two integer numbers: " ;
   cin >> num1 >> num2;

   cout << "The sum of: " << num1 << " + " << num2
        << " = " << num1+num2 << endl ;

   return(0) ;
}
```

לצד התיעוד, לעתים אנו מלווים את תכניתנו בקובץ נפרד לו נהוג לקרוא בשם README (באותיות גדולות). קובץ זה אינו מיועד למתכנת עמית, אלא לַמשתמש, על כן בו נכלול הסבר קצר מה התכנית לה נלווה הקובץ מבצעת, איזה קלט יש להזין לה, ומה היא פולטת, וכן כיצד יש לקמפל את התכנית (נושא שנעשה יותר מורכב עת תכניתנו מורכבת מכמה קבצים, נושא בו לא נדון בקורס זה).

2.3.5 פקודת הפלט cout

ננצל תכנית זאת גם כדי ללמוד שפקודת הפלט עשויה להשתרע על-פני מספר שורות *בתכנית*. לא ניתן לפצל את הפקודה *באמצע* סטרינג, אך כן ניתן לפצלה בין שני מרכיבים המוצגים על-ידה. על-כן תכנית שתכלול פקודה כגון:

```
cout << "A wrong way
          to break a string" ;
```

לא תתקמפל בהצלחה. אך תכנית שתכלול פקודה כגון הבאה לא תעורר מדנים:

```
cout << "This cout "
     << "command is broken "
     << num1 << " " << num2
     << "so what?"
     << endl << endl ;
```

למען הדיוק, אם את הפקודה העליונה אפשר לכתוב:

```
cout << "A correct way\
          to break a string" ;
```

ועתה היא תתקמפל בהצלחה. הלוכסן נוטה שמאלה בסוף השורה הראשונה מורה למהדר שהמחרוזת שנקטעת כאן נמשכת בשורה הבאה.

2.4 תרגילים

1. כתבו והריצו תכנית אשר מדפיסה את שמכם וכתובתכם.
2. כתבו והריצו תכנית אשר קוראת מהמשתמש את אורכן ורוחבן של צלעות מלבן. התכנית תדפיס את שטח והיקף המלבן.

<u>הקפידו על:</u>

1. תיעוד: כולל תיעוד ראשי, תיעוד תפקידי משתנים, ותיעוד **מינימלי** בגוף התכנית.
2. שמות משתנים משמעותיים.
3. עימוד\אינדנטציה\הזחה.
4. הנחיות למשתמש מה הקלט המצופה ממנו, והסבר מה הפלט שמוצג בפניו.

3. כתבו תכנית אשר קוראת ערכים לתוך זוג משתנים num1 ו- num2, ואחר מחליפה את הערכים כך שב- num1 יהיה הערך שנקרא לתוך num2, וב- num2 יהיה הערך שנקרא לתוך num1.
   א. כתבו את התכנית הנ"ל תוך שימוש במשתנה עזר נוסף.
   ב. כתבו את התכנית ללא שימוש במשתנה עזר נוסף (רק על-ידי הגדלת והקטנת כל-אחד מהמשתנים בשיעור משנהו).

   הסיקו משתי התכניות שכתבתם שלעיתים ניתן לכתוב תכנית אשר תשתמש בפחות משתנים אך תהא הרבה פחות קריאה, ולפיכך הרבה פחות רצויה.

4. כתבו תכנית אשר קוראת מהמשתמש שלושה מספרים חיוביים המציינים את אורכיהן של שלוש צלעות משולש. התכנית תציג למשתמש את שטח והיקף המשולש. כדי לחשב את שטח המשולש השתמשו בנוסחת Heron הקובעת כי

שטחו של משולש שווה לשורש הריבועי של: s*(s-a)*(s-b)*(s-c) עבור: a, b, c המציינים את אורך צלעות המשולש, ו- s המציין את מחצית היקף המשולש.
כדי לחשב שורש של ערך עליכם להוסיף לתכניתכם את המרכיבים הבאים:

א. הוסיפו לתכנית הוראת קומפיילר: `#include <cmath>` הוראה זאת תופיע לצד הוראות ה- `include` האחרות.

ב. כדי להכניס לתוך המשתנה `y` (מטיפוס `double`) את שורשו של `x` כתבו, בגוף התכנית את הפקודה: `y = sqrt(x);`.

3. פקודות תנאי

התכניות שכתבנו עד כה התקדמו סדרתית (כלומר פקודה אחר פקודה), מהפקודה הראשונה ועד האחרונה (או עד הסוף המר). במקרים רבים אנו זקוקים לתכניות שבהן פקודות מסוימות מתבצעות כתלות בערכם של משתנים שונים (למשל לא נרצה לבצע פעולת חילוק עת ערכו של המחלֵק הוא אפס). פקודות התנאי שנכיר בַּפרק הנוכחי, ופקודות הלולאה שנכיר בפרק הבא, הן שתאפשרנה לנו להתקדם בצורה יותר מתוחכמת מהאופנות הסדרתית, ולבצע פקודות שונות בתכנית רק במידת הצורך. אנו אומרים שפקודות התנאי והלולאה הן *פקודות בקרה* (control statements) במובן זה שהן מבקרות את התקדמות התכנית: הן קובעות לאן תתקדם התכנית בכל שלב (על-פי ערכי המשתנים).

3.1 פקודת `if` (ללא `else`)

פקודת ה- `if` היא פקודת התנאי הראשונה שנכיר. תחילה נכיר אותה בצורתה המצומצמת: ללא `else`, ואחר כך בצורתה הכללית: עם אפשרות ל- `else`. לצידה של פקודת ה- `if` קיימת פקודת תנאי נוספת, פקודת ה- `switch`, ואותה נכיר בַּהמשך.

3.1.1 מוטיבציה

נחזור לתכנית האחרונה שכתבנו. נניח כי ברצוננו להרחיבה כך שפלט התכנית יכלול לא רק את סכומם של שני המספרים שהוזנו, אלא גם את הפרשם, מכפלתם ומנתם. נציג את התכנית:

```
#include <iostream>

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


int main() {
 int num1, num2;                    // the two inputs

 cout << "Enter two integer numbers: " ;
 cin >> num1 >> num2;

 cout <<num1 << " + " << num2 <<" = "<< num1+num2<< endl;
 cout <<num1 << " - " << num2 <<" = "<< num1-num2<< endl;
 cout <<num1 << " * " << num2 <<" = "<< num1*num2<< endl;
 cout <<num1 << " / " << num2 <<" = "<< num1/num2<< endl;
 cout << "bye" ;

 return(0) ;
}
```

מה יקרה עת תכנית זאת תורץ על-ידי המשתמש? במידה והמשתמש יזין כקלט את המספרים 3 ו- 5 יוצג לו הפלט

```
8 -2 15 0 bye
```

כנדרש. אולם אם הקלט שמשתמש יזין יהיה 3 ו- 0 אזי עת התכנית תגיע לחשב את המנה num1/num2 היא תנסה לחלק באפס, מעשה שאינו חוקי, ועל-כן תועף (ביצוע התכנית יופסק טרם שהתכנית תשלים את פעולתה, בפרט התכנית לא תפרד מהמשתמש בידידות בברכת bye), וזה דבר שאינו ראוי. מצב בו תכניתנו מועפת הוא עבורנו בבחינת 'הגיהנום בהתגלמתו', רוצה לומר זה מצב חמור מאוד. אנו אומרים כי בתכנית יש שגיאה או bug. כיצד נתגבר על הבאג? כיצד נמנע מהמצב החמור בו התכנית מועפת? ניתן להחליט שבתחילת התכנית ננחה את המשתמש להזין שני מספרים שלמים, שהשני ביניהם שונה מאפס. זה פתרון אפשרי אך תבוסתני; אנו רוצים שתכניתנו תהיה כללית ככל האפשר, ותפעל על מגוון רחב של קלטים, ולא תמלט 'מלאחוז בשור בקרניו'. על-כן נידַרש לפתרון שונה, פתרון שיחייב אותנו להכיר את פקודת התנאי if.

3.1.2 דוגמה ראשונה לשימוש ב- if

נדגים את התכנית המתוקנת, זו הכוללת את פקודת התנאי, ואחר נסבירה.

```
#include <iostream>

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

int main() {
 int num1, num2;

  cout << "Enter two integer numbers: " ;
  cin >> num1 >> num2;

  cout<< num1 << " + " << num2 <<" = "<< num1+num2<< endl;
  cout<< num1 << " - " << num2 <<" = "<< num1-num2<< endl;
  cout<< num1 << " * " << num2 <<" = "<< num1*num2<< endl;
  if (num2 != 0)
      cout << num1 << " / " << num2 <<" = "
           << num1/num2<< endl;
  cout << "bye" ;

  return(0) ;
}
```

את פקודת הפלט, אשר מציגה את המנה, שילבנו בפקודת התנאי if, במלים אחרות כפפנו לפקודת if. עת המחשב מגיע לפקודת if הוא ראשית בודק האם התנאי המופיע בתוך הסוגריים, אחרי המילה if, מתקיים; במקרה שלנו: האם num2 שונה מאפס (הסימן =! משמעותו אינו שווה, או במילים אחרות שונה). במידה והתנאי מתקיים המחשב מבצע את הפקודה *היחידה* הכפופה לתנאי (במקרה שלנו פקודת הפלט); במידה והתנאי אינו מתקיים המחשב אינו מבצע את הפקודה הכפופה לתנאי. בשני המקרים המחשב ממשיך הלאה לפקודה הבאה בתכנית, במקרה שלנו הוא נפרד מהמשתמש בידידות ומסיים את התכנית כהלכה.

כיצד התכנית הנוכחית מתגברת על הבעיה אותה הצגנו קודם? במידה וערכו של num2 הוא אפס התנאי (num2 != 0) לא יתקיים, המחשב לא ינסה לחשב את המנה num1/num2 והתכנית לא תעוף. קנטרנים יטענו שהתכנית אכן לא תעוף

אולם היא גם לא תציג שום פלט, והמשתמש יוותר מופתע: מדוע הפעם לא מוצגת בפניו המנה. יש להודות על האמת שיש טעם בקנטור זה. פתרון אפשרי הוא להוסיף לתכנית פקודה נוספת, שתופיע מייד אחרי פקודת ה- if שראינו. בַּפקודה הנוספת נכתוב:

```
if (num2 == 0)
   cout << "Can not divide by 0" << endl ;
```

כלומר אם ערכו של num2 הוא אפס אזי הצג הודעה שאומרת כי לא ניתן לחלק באפס (הסימן == משמעותו: שווה ל).

באופן כללי פקודת התנאי if נכתבת באופן הבא: if (condition) statement נסביר: הפקודה מתחילה במילת המפתח if שמורה שזו פקודת תנאי. אחר-כך מופיע בתוך סוגריים תנאי ואחר כך פקודה שתתבצע במידה והתנאי מתקיים. הקפידו שלא לכתוב ; אחרי התנאי, כלומר **לא** לכתוב:

```
if (a>b); a = 0 ;
```

כתיבה זאת תחשב תקינה על-ידי המהדר, אשר לא יעיר לכם עליה דבר (גם לא יזהיר), אך היא ממש לא מה שברצוננו לכתוב!

3.1.3 תנאים

אילו תנאים ביכולתנו להציג?

א. ראינו את תנאי השוויון שנכתב לדוגמה: (num2 == (num3 %17)) הקפידו תמיד לכתוב *שני* סימני שווה *צמודים* זה לזה ללא רווח בניהם. שיכחת אחד משני ה- = -ים היא טעות נפוצה, היזהרו ממנה. הבחינו בין השמה: num2 = 0 אשר *מכניסה* ל- num2 את הערך אפס (*קובעת* שערכו יהיה אפס), לבין תנאי: num2 == 0 השואל *האם* ערכו של num2 הוא אפס (וכמובן שאינו משנה את ערכו של המשתנה).

ב. ראינו גם את תנאי האי-שוויון שנכתב לדוגמה: (num1 –num2 != num3) גם כאן יש להקפיד להצמיד את שני הסימנים ולמקם את סימן הקריאה משמאל לסימן השוויון.

ג. באופן דומה ניתן לכתוב גם < או >, לדוגמה: (num1 < (num2 + num3)).

ד. ניתן לדרוש גם => או =< גם כאן יש להקפיד להצמיד את שני הסימנים ולמקם את סימן השווה מימין.

ה. במידה וברצוננו לדרוש שמספר תנאים יתקיימו אנו יכולים לכתוב: (num1 > 0 && num1 < 10) כלומר num1 גדול מאפס *וגם* num1 קטן מעשר. שימו לב לכתוב *זוג* תווי ampercent (&) צמודים זה לזה עת ברצונכם לדרוש וגם. לא ניתן לנסח תנאי זה בַּאופן: (0 < num1 < 10) כפי שאתם אולי רגילים לכתוב במתמטיקה.

ו. במידה וברצונכם לדרוש שלפחות אחד מכמה תנאים יתקיים ניתן לכתוב: (num1 < num2 || num2 < num3 %21) (התו | מצוייר על המקלדת בדרך כלל כקו אנכי המחולק לשניים באמצעו, משמאל לתו ה- Enter).

ז. במידה וברצונכם לדרוש שתנאי לא יתקיים כתבו: (!(num1 > num2 –17)) כלומר לא נכון ש- num1 גדול מ- num2 פחות 17 (כמובן שתנאי זה ניתן לנסח בצורה פשוטה יותר כ- (num1 <= num2 –17)).

ח. ומכאן ניתן להמשיך ולבנות תנאים מורכבים יותר ויותר.

בצורה פורמלית אנו קוראים לתנאי **ביטוי בולאני** (boolean expression) ואומרים שאם ערכו של הביטוי הוא אמת (במילים אחרות: אם התנאי מתקיים) אזי הפקודה הכפופה תבוצע, ואם ערכו של הביטוי הוא שקר (במילים אחרות: אם התנאי אינו מתקיים) אזי הפקודה לא תבוצע.

3.1.4 גוש (block)

אמרנו כי בפקודת if ניתן לכפוף לתנאי רק פקודה *יחידה*. ומה אם ברצוננו לכפוף לתנאי מספר פקודות? במקרה זה 'נארוז' את הפקודות שברצוננו לכפוף לַתנאי בגוש\בלוק על-יד שנכניסן בין סוגריים מסולסלים. המחשב מתייחס לבלוק כאל **פקודה מורכבת** (compound statement) יחידה אשר כפופה לתנאי.
נראה דוגמה: ברצוננו לכתוב תכנית אשר מציגה את מנתם של שני המספרים המוזנים לה; אולם אם המחלק המוזן הוא אפס אזי התכנית נותנת למשתמש אפשרות לחזור ולהזינו.

לפני שנציג את התכנית נעיר כי מכאן ואילך לא נציג תכניות שלמות, אלא רק קטעי תכניות. בכל מקרה נניח כי קטע התכנית הוא חלק מתכנית שלמה, בפרט נניח כי כל המשתנים בהם אנו עושים שימוש הוגדרו קודם לכן. כמו כן מטעמי קיצור והתמקדות בעיקר נשמיט את ההנחיה למשתמש מה הקלט המצופה ממנו, ואת ההסבר אודות הפלט.

```
cin >> num1 >> num2 ;
if (num2 == 0)
{
    cout << "Can not divide by zero" << endl ;
    cin >> num2 ;
}
if (num2 != 0)
    cout << num1 / num2 ;
```

הסבר: במידה וערכו של num2 אפס ברצוננו לבצע שתי פקודות: (א) להודיע למשתמש כי לא ניתן לחלק באפס, (ב) לקרוא ממנו ערך חדש לתוך num2. כדי שנוכל לכפוף את זוג הפקודות לתנאי (num2 == 0) אנו אורזים אותן בין { }. לסדרה של פקודות התחומות בסוגריים מסולסלים אנו קוראים **גוש** (block).

עת אנו אורזים מספר פקודות בגוש, התנאי שבפקודת ה- if משמש רק כמעין שומר סף, אחרי שחלפנו על-פניו אין אנו מובטחים שהוא ממשיך להתקיים לכל אורך הגוש. נראה דוגמה: שוב נניח כי ברצוננו לקרוא שני מספרים ולהדפיס את מנתם, אך הפעם נניח כי במידה והדפסנו את המנה ברצוננו לקרוא זוג מספרים שני ולהדפיס גם את מנתם. נציג את התכנית:

```
cin >> num1 >> num2 ;
if (num2 != 0)
{
   cout << num1 / num2 ;
   cin >> num1 >> num2 ;
   cout << num1 / num2 ;
}
```

תכנית זאת שגויה במובן זה שהיא עלולה לעוף אם בזוג נתונים השני הערך השני יהיה אפס. העובדה שערכו של num2 היה שונה מאפס בעקבות הזנת הנתונים בפעם הראשונה, ועל-כן נכנסנו לבלוק, אינה מבטיחה שגם אחרי שקראנו את ערכו של num2 בשנית ערכו יישאר שונה מאפס. מה הפתרון? כיצד נתקן את הבאג בתכנית?

```
cin >> num1 >> num2 ;
if (num2 != 0)
{
```

```
   cout << num1/num2 ;
   cin >> num1 >> num2 ;
   if (num2 != 0)
      cout << num1 / num2 ;
}
```

שימו לב לעימוד: העיקרון דומה למקרה בו יצרנו עד כה את הגוש שכלל את כל פקודות התכנית כולן: כל הפקודות התחומות בגוש מופיעות בהזזה ימינה כך שבין ה- if לסוגר הסוגר (או בין הסוגר הפותח לסוגר הסוגר) לא תופיע כל פקודה.

3.2 תוספת `else`

נחזור לתכנית אשר מדפיסה את מנתם של זוג הנתונים המוזנים לה. ראינו את הגרסה המבצעת:

```
cin >> num1 >> num2 ;
if (num2 != 0)
   cout << num1/num2 ;
```

אמרנו כי מגבלתה של גרסה זאת היא שעת `num2 == 0` התכנית לא תדפיס כל פלט, ובכך יש טעם לפגם. ראינו את התיקון למגבלה זאת:

```
cin >> num1 >> num2 ;
if (num2 != 0)
   cout << num1/num2 ;
if (num2 == 0)
   cout << "Can not divide by zero" << endl ;
```

התיקון מוסיף פקודת `if` עם תנאי משלים (הפוך) לתנאי שבפקודת ה- `if` הראשונה.

במקום תיקון זה נרצה להציע פתרון אחר אשר השפה מעמידה לרשותנו:

```
cin >> num1 >> num2 ;
if (num2 != 0)
   cout << num1/num2 ;
else
   cout << "Can not divide by zero" << endl ;
```

נסביר: הוספנו לפקודת ה- `if` תוספת בדמות מרכיב `else`. משמעותו של מרכיב זה היא שאם התנאי שבפקודה אינו מתקיים יש לבצע את הפקודה (או את הבלוק) המופיע אחרי מילת המפתח `else`.

עת אנו מוסיפים לפקודה מרכיב `else` אנו מובטחים כי תמיד תתבצע או הפקודה הכפופה ל- `if` *או* הפקודה שאחרי ה- `else`, ורק אחת מן השתיים. נדגים את כוונתנו באמצעות קטע התכנית הבא:

```
if (num1 < num2) {
   cout << "The greater is: " << num2 << endl ;
   num2 = num1 - 1 ;
}
else
   cout << "The greater is: " << num1 << endl ;
```

נניח כי בעת שביצוע התכנית מגיע לקטע הקוד ערכו של `num2` גדול משל `num1`; לפיכך הבלוק שיתבצע יהיה זה שכפוף ל- `if`. בתום ביצוע בלוק זה התנאי ש- `num1 < num2` כבר אינו מתקיים, האם זה אומר שעתה הפקודה הצמודה ל-

else תתבצע? לא ולא, מכיוון שביצענו את הפקודה הכפופה ל- if לא תתבצע הפקודה הצמודה ל- else. בכך הקוד הנ"ל שונה מהקוד הבא:

```
if (num1 < num2) {
   cout << "The greater is: " << num2 << endl ;
   num2 = num1 – 1 ;
}
if (num1 >= num2)
   cout << "The greater is: " << num1 << endl ;
```

שכן בקטע הקוד האחרון גם אם התנאי הראשון יתקיים, ופקודת הפלט הכפופה לו תתבצע אין הדבר אומר שאנו מתעלמים ומדלגים על-פני פקודת הפלט השניה (שעתה כפופה ל- if שני שהינו בלתי תלוי בַּ- if הראשון); (למעשה כפי שכתבנו את הקוד אנו בטוחים שאם תופיע פקודת הפלט הראשונה, אזי גם השניה תופיע).

3.2.1 הערכה מקוצרת של ביטויים בולאניים

נניח כי בתכנית כלשהו מופיע התנאי (ליתר דיוק הביטוי הבולאני): (b!= 0 && a/b > 0) שפת C מבטיחה לנו כי בדיקתו (הערכתו) של ביטוי זה תעשה משמאל לימין, כלומר ראשית ייבדק התנאי b!=0 ורק אח"כ התנאי: a/b > 0. יותר מכך, מכיוון שהתנאים מופיעים בגימום (כלומר עם 'וגם' בניהם) אזי כדי שהתנאי בכללותו יסופק (יתקיים) יש צורך *ששני* מרכיביו יסופקו, ואם הראשון ביניהם אינו מתקיים אין כלל צורך לבדוק את המרכיב השני. נשים לב כי לתכונה זאת של השפה חשיבות רבה שכן אם ערכו של b הוא אפס אנו רוצים להימנע מבדיקת המרכיב a/b > 0 וזה מה שאכן יקרה. לכן גם יש חשיבות לכך שכתבנו את התנאי באופן הנ"ל, ולא באופן הדומה אך שונה: (a/b > 0 && b!=0) שכן אז ראשית היה נבדק התנאי a/b > 0 ואם במקרה ערכו של b היה אפס התכנית הייתה עפה (דבר שלא יקרה בניסוח המקורי שהצגנו).

מצב דומה חל בתנאי (b == 0 || a/b > 17). כאן על-מנת שהביטוי יסופק די בכך שאחד ממרכיביו יסופק; על-כן אם המרכיב b==0 מסופק אין צורך לבדוק גם את המרכיב a/b>17 וטוב שכך, שכן בדיקתו של תנאי זה (עת b == 0) הייתה גורמת לתעופתה של התכנית. בשל הדרך בה ניסחנו את התנאים, התנאי a/b >17 ייבדק רק אם b!= 0.

3.2.2 תנאים מקוננים

נניח כי אנו כותבים את הקוד הבא (במכוון נציג אותו עם עימוד קלוקל):

```
if (a > b) if (b > c) cout << "b>c" ; else cout << "???";
```

נשאלת השאלה לאיזה if מתייחס ה- else? האם ל- if הראשון או השני? התשובה היא שכל else מתייחס ל- if האחרון שטרם קיבל else. במקרה שלנו ה- else מתייחס, לכן, ל- if השני; לכן מבחינת העימוד מתאים היה לכתוב את הקוד הנ"ל באופן הבא:

```
if (a > b)
   if (b > c)
      cout << "b > c" ;
   else
      cout << "???" ;
```

באופן שהעימוד משקף את המשמעות, ומורה כי ה- else הוא 'בן-זוגו' של ה- if האחרון. נדגיש עם זאת שהעימוד *משקף* את המשמעות, אך אינו קובע אותה; כלומר גם לו ה- else היה מופיע מתחת ל- if הראשון הוא לא היה הופך להיות 'בן זוגו'. מה שהיה קורה הוא פשוט שהתכנית שלנו הייתה לקויה מבחינת העימוד שלה.

ומה אם אנו רוצים שה- else יתייחס ל- if הראשון דווקא (ול- if השני לא יתלווה מרכיב else)? לכך יש שני פתרונות. הראשון:

```
if (a > b)
{
   if (b > c)
      cout << "b > c" ;
}
else
   cout << "???" ;
```

כלומר תחמנו את ה- if הפנימי בתוך בלוק. עתה לא ייתכן שה- if נמצא בתוך גוש, ובן זוגו ה- else מחוץ לגוש; על כן ברור של- if הפנימי אין else, וה- else הקיים מתייחס ל- if החיצוני.

הפתרון השני:

```
if (a > b)
   if (b > c)
      cout << "b > c" ;
   else
     ;
else
   cout << "???" ;
```

מה עשינו הפעם? לַ- if הפנימי הצמדנו מרכיב else אשר הפקודה הנלווית אליו היא **הפקודה הריקה** (;). הפקודה הריקה היא פקודה לגיטימית ולעיתים אף שימושית (כמו בדוגמה זאת). בהמשך נראה מצבים נוספים בהם נשתמש לנו להנאתנו בַפקודה הריקה. עתה לכל if יש את ה- else שלו, ומתקיים הכלל שכל else מתייחס ל- if האחרון שטרם זכה ב- else.

באותו הקשר אציג טעות שכיחה למדי. נביט בקוד הבא:

```
if (a>b);
   a = 0 ;
```

מה יעשה קוד זה? אם a>b אזי מתבצעת הפקודה הריקה, (ואחרת היא אינה מתבצעת). מה שחשוב יותר: ההשמה: a = 0; *מתבצעת בכל מקרה*, היא אינה כפופה לתנאי! שכן לתנאי כפופה רק פקודה אחת, והיא הפקודה הריקה. למותר לציין שאינדנטציה בקוד זה שגויה: היא שגויה מפני שמין הסתם כותב הקוד התכוון לכפוף לַתנאי את ההשמה, אך הנקודה-פסיק שהוא כתב בטעות בסוף השורה שיבשה את הקוד.

3.2.3 מיון שלושה מספרים (גירסה א')

נציג עתה גרסה ראשונה של תכנית שלמה הקוראת מהמשתמש שלושה מספרים שלמים ומציגה אותם ממוינים מקטן לגדול.

```
/*********************************************************/
/*                                                       */
/*                  Sorting three integers               */
/*                  ======================               */
/*   Written by: Yosi Cohen, ID = 333444555, Class: A1   */
/*                                                       */
/*  This program reads from the user three ints and      */
/*  displays them sorted from smallest to largest.       */
/* The program examines the six combinations of size     */
```

```
/* (e.g., the first is smaller than the second that is */
/* smaller than the third).                            */
/*******************************************************/

#include <iostream>

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

int main()
{  int first, second, third ;     // the 3 inpt nums

   cout << "Enter three integer numbers: " ;
   cin >> first >> second >> third ;
   cout << "The sorted list is: " ;

   if (first <= second && second <= third)   // 1st smlst
      cout << first <<" "<< second <<" "<< third <<endl ;
   if (first <= third && third <= second)
      cout << first <<" "<< third <<" "<< second <<endl ;

   if (second <= first && first <= third)    // 2nd smlst
      cout << second <<" "<< first <<" "<< third <<endl ;
   if (second <= third && third <= first)
      cout << second <<" "<< third <<" "<< first <<endl ;

   if (third <= second && second <= first)   // 3rd smlst
      cout << third <<" "<< second <<" "<< first <<endl ;
   if (third <= first && first <= second)
      cout << third <<" "<< first <<" "<< second <<endl ;

   return(0) ;
}
```

התכנית שהצגנו פשוטה למדי (וזו תכונה רצויה) יחד עם זאת יש לה שתי מגבלות:

1. במידה והמשתמש יזין קלט הכולל אותו ערך מספר פעמים (כגון 3 5 3 או 3 3 3) יוצג לו הפלט מספר פעמים (ודאו שאתם מבינים מדוע. בדקו ראשית את המקרה 3 3 3). הפלט אומנם לא יהיה שגוי, אך יש טעם לפגם בכך שאנו מציגים אותו פלט מספר פעמים.
2. בתכנית זאת המחשב יבדוק בכל מקרה שישה תנאים, גם אם הקלט היה למשל 3 2 1 ולכן כבר התנאי הראשון התקיים, והפלט הדרוש הוצג, ועל-כן אין צורך לבדוק את יתר חמשת המקרים האחרים (ולגלות שהתנאים בהם אינם מתקיימים). כמו הפגם שציינו קודם גם האחרון אינו חמור אולם יש בו טעם לפגם, ואם אפשר לעשות משהו טוב יותר אזי עדיף.

לכן עתה נציג תכנית שניה אשר מבצעת אותה משימה באופן אחר.

3.2.4 מיון שלושה מספרים (גרסה ב')

בתכנית זאת אציג רק את 'לב' התכנית:

```
if (first <= second && second <= third)
      cout << first <<" "<< second <<" "<< third <<endl ;
else
{
   if (first <= third && third <= second)
      cout << first <<" "<< third <<" "<< second <<endl ;
   else
   {
      if (second <= first && first <= third)
          cout << second <<" "<< first <<" "
               << third <<endl ;
      else
      {
          if (second <= third && third <= first)
                cout << second <<" "
                     << third <<" "<< first <<endl ;
          else
          {
              if (third <= second && second <= first)
                  cout << third <<" "<< second <<" "
                       << first <<endl ;
              else
                  cout << third <<" "<< first <<" "
                       << second <<endl ;
           }
        }
    }
}
```

<u>נסביר</u>: אם מתקיים התנאי הראשון (`first <= second && second <= third`) אזי אנו מבצעים את פקודת הפלט הכפופה לו, ואיננו מבצעים את כל הגוש הכפוף ל- else, בפרט כמובן איננו שולחים עוד פלטים (ולכן אין חשש שהפלט יופיע פעמיים), ואיננו בודקים לחינם עוד תנאים. אם, לעומת זאת, התנאי הראשון אינו מתקיים, אזי אנו פונים לגוש ה- else ושם בודקים את התנאי השני. אם התנאי השני מתקיים אזי אנו שולחים את הפלט המתאים, ואיננו פונים לגוש ה- else הכפוף לתנאי השני. וכך הלאה. נשים גם לב שהתנאי השישי כבר לא מופיע בגרסה זו של התכנית שכן אם אף אחד מחמשת התנאים המופיעים לא מתקיים אנו 'מתדרדרים' ל- else האחרון (זה של ה- if החמישי) ושם כבר איננו צריכים לבדוק שהסידור הוא אכן : third, first, second שכן זו האפשרות היחידה שנותרה.

אם נבחן את התכנתי שכתבנו נוכל לשים לב שלמעשה כל הסוגריים המסולסלים המופיעים בקטע שמעל מיותרים: נבחן לדוגמה את האחרונים:

```
        else
        {
            if (third <= second && second <= first)
                cout << third <<" "<< second <<" "
                     << first <<endl ;
            else
                cout << third <<" "<< first <<" "
                     << second <<endl ;
         }
```

מוסיע בתוכם פקודת if-else אחת ויחידה (נזכור שהפקודה if (condition) steatemnet1 else statement2 זו פקודה אחת בלבד). הפקודה אולי קצת גדולה, ולכן לא מזיק לשים אותה בסוגריים, אך זה לא הכרחי. לכן על הסוגריים הפנימיים ביותר ניתן לוותר. באותו אופן, אם נבחן את קטע התכנית (את הסוגריים שסימכנו שניתן להסיר סגרתי בהערת תיעוד):

```
else
      {
        if (second <= third && third <= first)
              cout << second <<" "
                   << third <<" "<< first <<endl ;
        else
        // {
            if (third <= second && second <= first)
                cout << third <<" "<< second <<" "
                     << first <<endl ;
            else
                cout << third <<" "<< first <<" "
                     << second <<endl ;
         //}
      }
```

גם הסוגריים המופיעים בהם הינם מיותרים שכן בתוכם יש רק פקודת if-else אחת. אומנם בפקודה זו למרכיב ה- else כפופה פקודת if-else שלמה, אולם הפקודה המופיעה בתוך הסוגריים היא בסך הכל פקודת if-else אחת.

כך נוכל להמשיך ולהוריד את כל הסוגריים ונקבל את הקוד הבא:

```
if (first <= second && second <= third)
      cout << first <<" "<< second <<" "<< third <<endl ;
else
   if (first <= third && third <= second)
      cout << first <<" "<< third <<" "<< second <<endl ;
   else
      if (second <= first && first <= third)
          cout << second <<" "<< first <<" "
               << third <<endl ;
      else
          if (second <= third && third <= first)
                cout << second <<" "
                     << third <<" "<< first <<endl ;
          else
              if (third <= second && second <= first)
                  cout << third <<" "<< second <<" "
                       << first <<endl ;
              else
                  cout << third <<" "<< first <<" "
                       << second <<endl ;
```

בקוד זה נעשה, עתה, שינוי רק באינדנטציה, שינוי שכמובן לא ישנה את התנהגותו. השינוי הוא הבא: כל if המופיע אחרי else נעלה לשורה של ה- else שלפניו, וכל else נסיט שמאלה את מתחת ל- if הראשון:

```
if (first <= second && second <= third)
      cout << first <<" "<< second <<" "<< third <<endl ;
else if (first <= third && third <= second)
      cout << first <<" "<< third <<" "<< second <<endl ;
else if (second <= first && first <= third)
      cout << second <<" "<< first <<" "<< third <<endl ;
else if (second <= third && third <= first)
      cout << second <<" "<< third <<" "<< first <<endl ;
else if (third <= second && second <= first)
      cout << third <<" "<< second <<" "<< first <<endl ;
else
      cout << third <<" "<< first <<" "<< second <<endl ;
```

נבחן את הקוד שהתקבל: הקוד, כאמור, מבצע בדיוק אותה משימה, אולם אנו חושבים עליו מעט אחרת מכפי שחשבנו על הקוד הקודם. אנו חושבים על קוד זה כעל סדרה של תנאים (הנכתבים זה מתחת לזה), לכל אחד מהם כפופה פקודה (או גוש של פקודות). התנאים נבדקים בזה אחר זה, הראשון ביניהם שמתקיים גורם לנו לבצע את הפקודה שכפופה לו, ולהתעלם מכל יתר התנאים והפקודות להם. אם 'חלילה' אף לא אחד מין התנאים התקיים אזי אנו 'מתדרדרים' לפקודה הכפופה ל- else האחרון, אותה אני מכנה 'פח הזבל' ומבצעים אותה. אנו רשאים גם לא לכלול else אחרון עם פקודה הכפופה לו (בדוגמה שלנו משמעות הדבר היה השמטת שתי השורות האחרונות מהקוד) ואז במידה ואף לא אחד מהתנאים יתקיים לא יבוצע דבר.

3.2.5 מיון שלושה מספרים (גירסה ג')

אציג עתה תכנית הממיינת שלושה מספרים ביעילות. התכנית לא טריביאלית להבנה, וזו, כמובן, מגרעת שלה, אולם היא בודקת פחות תנאים, וזה היתרון שלה. אציג אותה לא כי אני חושב שיש לה יתרון מוחלט על קודמותיה (היות תכנית קשה להבנה הוא מגרעת לא קלת ערך), אלא כדי לחדד עוד את הבנתנו את נושא התנאים. (התכנית הנוכחית תוצג בצורה המקוצרת בה תוצגנה מרבית התכניות: ללא הסבר למשתמש איזה קלט הוא מצופה להזין, ללא הסבר אודות הפלט מוצג, ללא תיעוד. הסיבה לקיצור היא כדי לחסוך מקום ומאמץ ולהותיר רק את העיקר. אל תלמדו מתכניות כגון אלה כיצד יש לכתוב תכנית *שלמה* כהלכה!)

בתכנית זו שתלתי הערות תיעוד הרבה מעבר לנדרש, וזאת כדי לעזור לכם להבינה. עקבו אחר ההערות, אולם בתכניות שאתם כותבים אל תתעדו במידה שכזאת. זהו תיעוד יתר!

```
int main()
{  int first, second, third ;     // the 3 inpt nums

   cin >> first >> second >> third ;

   if (first <= second)
   {
      if (second <= third)
            cout << first << second << third ;
      else           // first < second && second >= third
                     // that is: second is the greatest
                     // we do not know, yet,
                     // who is the smallest
                     // so we need another if:
         if (first <= third)
            cout << first << third << second ;
         else
            cout << third << first << second ;
   }
   else                                  // first >= second
   {
      if (second >= third)
         cout << third << second << first ;
      else          // first >= second && third >= second
                    // that is: second is smallest.
                    // now, check who is largest:
         if (first >= third)
            cout << second << third << first ;
         else
            cout << second << first << third ;
   }

   return(0) ;
}
```

נעיר מספר הערות אודות תכנית זאת:

1. במידה מסוימת היא פחות קריאה, יותר מבלבלת, מקודמתה, ולכן הוספנו את התיעוד אשר מסביר בכל שלב מה התנאים שמתקיימים בנקודה בה מופיעה כל הערה.
2. בתכנית תמיד יוצג רק פלט יחיד (גם במקרה בו הקלט הוא 3 3 3, בדקו!).
3. בתכנית נבדקים לכל היותר שלושה תנאים (זכרו כי אם התנאי הצמוד ל- `if` אינו מתקיים פונים ישירות לגוש הצמוד ל- `else`, ובו בודקים תנאי נוסף רק אם באותו גוש מופיעה פקודת `if`).
4. הסוגריים התוחמים את הגוש שכולל את הפקודה המבוצעת במידה והתנאי הראשון מתקיים (כמו גם אלה שתוחמים את הגוש הצמוד ל- `else` של ה- `if` הראשון) מיותרים. אין בהם נזק, והוספנו אותם מטעמי קריאות, אך אין בהם הכרח, נסביר מדוע: ל- `if` הראשון כפופה פקודה יחידה שהינה פקודת `if-else`, ל- `else` של פקודה זאת שוב כפופה פקודה יחידה שהינה פקודת `if-else`.

3.2.6 שורשי משוואה ריבועית

משוואה ריבועית היא ביטוי מהצורה: $y= a*x^2 +b*x + c$, עבור a, b, c שהינם מספרים ממשיים כלשהם. לדוגמה: $y = 4*x^2 +3*x - 8$ או $y = -2*x^2 +18$ הן משוואות ריבועיות. **שורשי המשוואה הריבועית** הם מספרים אשר אם יוצבו במקום x יגרמו לכך שערכו של y יהיה אפס. לדוגמה שורשי המשוואה השניה מבין השתיים שהצגנו הם 3, ו- 3- . רובכם ודאי זוכרים כי הנוסחה למציאת שורשי משוואה ריבועית (המסומנים כ- x1, x2) היא:

$$x1, x2 = [-b +- sqrt(b^2 -4*a*c)] / [2*a]$$

כאשר הביטוי sqrt(…) מסמן את השורש של הערך המופיע בתוך הסוגריים. לביטוי: $b^2 -4*a*c$ נהוג לקרוא בשם **דיסקרימיננטה.**

נרצה עתה לכתוב תכנית אשר קולטת מהמשתמש שלושה מספרים ממשיים המהווים מקדמים של משוואה ריבועית (כדוגמת: 2- 0 ו- 18). התכנית תציג למשתמש את שורשי המשוואה.

לפני שנפנה להצגת התכנית נסביר כי בשפת C קיימת פונקציה אשר עת מעבירים לה ערך מחזירה את השורש החיובי שלו. הפונקציה נקראת sqrt, היא מקבלת פרמטר מטיפוס double (כלומר מספר ממשי שעשוי להיות גדול במיוחד), ומחזירה ערך מאותו טיפוס. כדי להשתמש בפונקציה זאת, כמו גם בפונקציות מתמטיות אחרות, יש להוסיף לתכנית שורת `include` נוספת: `#include <cmath>` (אין צורך בתוספת של פקודת `using`).

תכנית לפתרון משוואה ריבועית (גרסה ראשונה)

נציג את התכנית:

```
#include <iostream>
#include <cmath>

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

int main()
{
   double param_a, param_b, param_c,
          discriminant ;
```

```
   cin >> param_a >> param_b >> param_c ;

   discriminant = param_b * param_b
                  - (4 * param_a * param_c) ;
   if (discriminant < 0)
      cout << "No solution" << endl ;
   else
   {
      double x1, x2 ;

      discriminant = sqrt(discriminant);

      x1 = (-param_b + discriminant) / (2 * param_a) ;
      x2 = (-param_b - discriminant) / (2 * param_a) ;
      cout << x1 << " " << x2 ;
   }
   return(0) ;
}
```

נדון בתכנית שכתבנו:

1. ראשית אנו קוראים מהמשתמש את שלושת מקדמי המשוואה. אחר אנו משתמשים במשתנה עזר אשר מכיל את הדיסקרימיננטה. במידה והדיסקרימיננטה שלילית אנו מודיעים כי המשוואה אינה ניתנת לפתרון, ואחרת אנו מציגים את שורשי המשוואה. (אגב, לו היינו מוסיפים תיעוד ראשי לתכנית אזי הוא היה כולל הסבר דומה להערה זאת).

2. אחרי ה- `else` אנו פותחים גוש. מיד בתחילת הגוש אנו מגדירים שני משתנים `x1, x2`. משמעות הדבר היא כי משתנים אלה יוכרו רק בתוך הגוש בו הם הוגדרו. לו היינו מנסים מחוץ לגוש, למשל מייד לפני פקודת ה- `return`, לפנות לַמשתנים, היה הקומפיילר מודיע לנו כי אנו פונים למשתנים שלא הוגדרו. מכיוון שב- `x1, x2` אנו עושים שימוש רק בתוך הגוש אין לנו בעיה כזאת, ומסיבה זאת גם לגיטימי להגדיר את המשתנים בגוש זה ולא בתחילת התכנית (לצד הגדרת `param_a,…`). בשפת `C++` (אך לא בשפת `C`) מותר להגדיר משתנים *בכל מקום*, ולא רק בתחילת התכנית (מקום שהוא בעצם תחילת גוש), או בתחילת גוש פנימי כלשהו. בחוברת זאת נדבק בקונבנציה לפיה הגדרת משתנים תתבצע תמיד בתחילת הגוש (כפי שדורשת שפת `C`), אולם מוסכמה זאת אינה נשמרת על-ידי כל המתכנתים בשפה, והדבר אינו נחשב ללא לגיטימי להגדיר משתנים גם במקומות אחרים.

3. בתכנית הגדרנו את המשתנים, ומייד אחר-כך הכנסנו להם ערך. לתהליך בו אנו מכניסים למשתנים ערך אחרי הגדרתם אנו קוראים **איתחול** (initialization) של המשתנים (מלשון ערך התחלתי שמוכנס למשתנים). בשפת C ניתן לשלב את הגדרת המשתנים ואיתחולם. למשל ניתן לכתוב:

```
int a = 17,  b = 3, c;
```

ובכך להגדיר שלושה משתנים: `a` שמאותחל לערך **17**, `b` שמאותחל להיות **3**, ו- `c` שלא מאותחל (ועל-כן יהיה בו ערך מקרי כלשהו). בתכנית שלנו יכולנו לשלב את ההגדרה והאיתחול באופן הבא:

```
double x1 = (-param_b + sqrt(discriminant)) / (2 * param_a) ,
      x2 = (-param_b - sqrt(discriminant)) / (2 * param_a) ;
```

4. בתכנית המוצעת, במידה והדיסקרימיננטה שווה אפס (ולמשוואה על-כן שורש יחיד) יציג המחשב שורש זה פעמיים. כפי שכבר ציינו בעבר זה לא נורא, אבל זה

גם לא נעים; על-כן ראוי היה להכניס לתכנית בדיקה נוספת אשר תבדוק `if (discriminnat == 0)` ובמידה וכן תציג את השורש היחיד פעם יחידה.

5. עתה נשאל את עצמנו מה יקרה אם הקלט שיזין המשתמש יהיה: `-6 3 0` (כלומר אפס יוזן לתוך `param_a`)? התשובה היא שהתכנית שלנו תשגה (אך לא תעוף), שכן עת היא תנסה לחשב את ערכו של `x1` היא תנסה לחלק באפס (בממשיים). יש מי שיטען שהמשוואה $y = 0*x^2 +3*x -6$ כלל אינה משוואה ריבועית, זו משוואה ממעלה ראשונה, ועל-כן תכניתנו אינה אמורה לפתרה. על כך נשיב: בתיכנות אל תהיה צודק, הייה חכם. וחכם במקרה זה יעדיף שלא לשלוח את המשתמש לרכוש את תכניתו של עמיתו (אשר פותרת משוואה ממעלה ראשונה), אלא יכליל (בקלות יחסית) את תכניתו שלו כך שהיא תתמודד כהלכה גם עם משוואה כמו האחרונה שהצגנו, וגם עם משוואה כגון: $y = 0*x^2 +0*x -6$ (משוואה שאין לה כל שורש, שכן לא קיים ערך `x` שאם נציבו יתקבל אפס ל- `y`). לכן נפנה עתה להציג גרסה שניה, מלאה יותר, של התכנית לפתרון משוואה ריבועית:

<u>תכנית לפתרון משוואה ריבועית (גירסה שניה)</u>

נציג עתה גרסה שניה, נכונה יותר של התכנית לפתרון משוואה רבועית:

```
#include <iostream>
#include <cmath>

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

int main()
{
   double param_a, param_b, param_c,
          discriminant ;

   cin >> param_a >> param_b >> param_c ;

   if (param_a == 0)           // a linear equation
      if (param_b == 0)        // a constant function
         if (param_c == 0)
            cout << "Every x is a root" << endl ;
         else cout << "No roots" << endl ;
      else                      // a 'real' linear eq.
         cout << (-param_c)/param_b ;
   else                         // a 'real' quadratic eq.
   {
      discriminant = param_b * param_b
                   - (4 * param_a * param_c) ;
      if (discriminant < 0)
         cout << " No roots " << endl ;
      else if (discriminant == 0)
              cout << (-param_b)/ (2 * param_a) ;
      else
      {
         double x1, x2 ;

         discriminant = sqrt(discriminant);

         x1 = (-param_b + discriminant) / (2 * param_a) ;
```

```
            x2 = (-param_b - discriminant) / (2 * param_a) ;
            cout << x1 << " " << x2 ;
         }
      }
      return(0) ;
}
```

תכנית זאת מתגברת על מגבלותיה של קודמתה: היא מטפלת בכל צירוף שהוא של המקדמים, והיא אינה מציגה אותה תשובה פעמיים.

שימו לב לשרשרת ה- `if` וה- `else` בתחילת התכנית. כפי שהסברנו, כל `else` מתייחס ל- `if` האחרון שטרם 'נסגר'.

בַּגוש המטפל במשוואה ריבועית 'אמיתית' מופיעה שרשרת של `if … else if … else` בסעיף הבא נדון במבנה תכנותי זה.

למי מכם שתהה: בשפה לא קיימת פונקצית ההעלאה בריבוע ולכן כדי לחשב את x^2 יש לכתוב `x*x`. (הפונקציה `pow(x, y)` מחזירה את x^y עבור כל ערך `y` שהוא, ניתן להשתמש בה).

לפני שנסיים נעיר עוד הערה אודות שמות משתנים: לאורך הדרך אנו מדגשים את חשיבות התכנות הנכון, בפרט את חשיבות השימוש בשמות משתנים משמעותיים. לכן אנו מסתייגים משימוש בשמות משתנים כגון `a, b, c`. יחד עם זאת יש להודות על האמת כי *באופן יוצא מן הכלל* בתכנית זאת יש פשר לשמות כמו האחרונים, ועל-כן במקרה מאוד יוצא דופן זה, שימוש בשמות הללו לא היה בגדר הפסול.

3.2.7 שרשרת של `if` ו- `else`

תכנית זאת חוזרת לנושא שראינו בגירסה השניה של התכנית שמיינה שלושה מספרים. נניח כי ברצוננו לכתוב קטע תכנית אשר קורא מהמשתמש ציון מספרי בתחום שבין אפס למאה, ומציג את ערכו המילולי של הציון, לפי הכללים הבאים: עבור ציון שבין 95 ל- 100 יוצג: מעולה, עבור ציון שבין 85 ל- 94 יוצג: טוב-מאוד, עבור ציון שבין 75 ל- 84 יוצג: טוב, ועבור ציון נמוך מ- 75 יוצג: יש מקום לשיפור. התכנית:

```
cin >> grade ;
if (grade < 0 || grade > 100)
   cout << "Illegal grade\n" ;
else if (grade >= 95)
        cout << "excellent\n" ;
     else if (grade >= 85)
             cout << "very good\n" ;
          else if (grade >= 75)
                  cout "good\n" ;
               else cout << "a place for improvement\n" ;
```

עימדנו את התכנית באופן שכל `else` מופיע מתחת ל- `if` אליו הוא מתייחס. התוצאה היא שהתכנית גולשת יותר ויותר ימינה על המסך. אפשר לחשוב על קטע התכנית הנ"ל באופן מעט שונה: יש לנו כאן שרשרת של תנאים ופקודות הכפופות להם. בעת ביצוע התכנית, התנאים נבדקים בזה אחר זה, הראשון מביניהם שמסופק, הפקודה הצמודה אליו מתבצעת והמחשב מדלג מעבר ליתר התנאים והפקודות שבַּשרשרת. במידה ואף תנאי אינו מתקיים מתבצעת הפקודה הצמודה ל- `else` האחרון (ובמידה ולא היה בסוף השרשרת `else` שכזה, אזי במידה ואף

תנאי לא היה מסופק שום פקודה לא הייתה מתבצעת). מסיבה זאת נהוג לעמד קטע תכנית כנ"ל באופן הבא:

```
cin >> grade ;
if (grade < 0 || grade > 100)
   cout << "Illegal grade\n" ;
else if (grade >= 95)
        cout << "excellent\n" ;
else if (grade >= 85)
        cout << "very good\n" ;
else if (grade >= 75)
        cout << "good\n" ;
else cout << "a place for improvement\n" ;
```

כלומר עת ישנו שרשרת של: `if … else if … ekse if …` אנו כותבים את כולם באותה עמודה, כמו מעל, וחושבים על כך כעל סדרה של תנאים, הנבדקים בזה אחר זה (ועל-כן יש משמעות לסדר בו הם מופיעים), כך שהראשון ביניהם שמתקיים מביא לכך שהפקודה שכפופה לו, ורק היא, מבוצעת.

בקטע התכנית הנ"ל מופיעים בסוף כל סטרינג התווים `\n`, מה תפקידם? התו \ (backslash) עת מופיע בסטרינג משמש כ- escape character כלומר כתו המשנה את משמעותו (ה'מבריח' את משמעותו) של התו המופיע אחריו בסטרינג; במקרה שלנו השינוי הוא שבמקום שהתו n (המופיע אחרי ה- \) יציין את האות האנגלית en הוא מסמן קפיצת שורה בפלט. לכן הפקודה: `cout << "good\n" ;` שקולה לפקודה: `cout << "good" << endl ;`. מכיוון שהפורמט הראשון קצר יותר נהוג להשתמש גם בו. כמו עם `endl` ניתן לכלול בַסטרינג כמה `\n` וכל אחד מהם יגרום לקפיצת שורה. למשל בדקו למה תגרום הדפסת הסטרינג: `"\na broken\nstring\n\n" .`

3.3 משתנים בולאניים

בכל התכניות שכתבנו עד כה, השתמשנו רק במשתנים שהחזיקו ערכים מספריים. לעיתים אנו מעונינים לדעת האם 'אירוע' כלשהו התרחש בתכנית או לא, האם מצב מסוים מתקיים או לא; לשם כך די לנו במשתנה שיוכל להחזיק אחד משני ערכים 'כן' ו- 'לא' או אפס ואחד. שפת C++ (אך לא שפת C, בה לא קיים הטיפוס bool) מעמידה לרשותנו משתנים מטיפוס bool אשר נועדו לצרכים שכאלה. (הטיפוס bool נקרא על-שם מתמטיקאי אנגלי בשם ג'ורג' בול אשר במאה התשע-עשרה פיתח מתמטיקה של שני מספרים, ובכך ניבא במובנים מסוימים את המחשב, שזכרונו, כפי שאנו זוכרים, בנוי מאפסים ואחדים).

אם נגדיר בתכנית משתנים: `bool b1, b2;` אזי לתוך המשתנים נוכל להכניס אך ורק שני ערכים `true` ו- `false` (באות קטנה), ולא כל ערך אחר. באופן משלים נוכל גם לשאול: `if (b1 == true) …`. מישהו יכול לרצות להכניס למשתנים אלה את הערכים אפס ואחד, או לטעון כי הטיפוס בכלל מיותר. לא נתווכח על כך, רק נאמר שניתן להכניס למשתנה בולאני גם את הערכים אפס או אחד, אך *נהוג* להכניס להם דווקא את הערכים `true, false`. למה? כי ככה `Stroustrup`, מתכנן שפת C++ קבע.

נדגים היכן ומדוע משתנים בולאניים עשויים להיות שימושיים: נכתוב קטע תכנית אשר קורא מהמשתמש שלושה מספרים שלמים המציינים את אורך צלעותיו של משולש. התכנית מודיעה איזה מן משולש זה: שווה צלעות, שווה שוקיים, ישר זווית, 'סתם משולש', או שאורכי הצלעות שהוזנו לא יכולות להרכיב משולש. נזכור כי משולש עשוי להיות הן ישר זווית והן שווה שוקיים. כדי להתרשם עד כמה

השימוש במשתנים בולאניים יפשט את התכנית אני ממליץ לכם לנסות לכתבה בלי שימוש במשתנים אלה, ואחר להתבונן בתכנית שלפנינו:

```
cin >> edge1 >> edge2 >> edge3 ;

if (edge1 <= 0 || edge2 <= 0 || esge3 <= 0 ||
    edge1 >= edge2 + edge3 ||
    edge2 >= edge1 + edge3 ||
    edge3 >= edge1 + edge2 )
   cout << "Illegal input\n" ;
else if (edge1 == edge2 && edge2 == edge3)
      cout << "equilateral\n” ;
else
{
   bool isosceles = false,                // 2 sides equal
        right_angle = false ;

   if (edge1 == edge2 || edge2 == edge3 || edge1==edge3)
      isosceles = true ;

   if (edge1*edge1 == edge2*edge2 + edge3*edge3 ||
       edge2*edge2 == edge1*edge1 + edge3*edge3 ||
       edge3*edge3 == edge2*edge2 + edge1*edge1  )
           right_angle = true ;
   if (isosceles == true && right_angle == true)
      cout << “isosceles and right_angle\n” ;
   else if (isosceles == true)
      cout << “isosceles\n” ;
   else if (right_angle == true)
      cout << “right angle\n” ;
   else cout << “a simple triangle\n” ;
}
```

הסברים והערות:

1. אנו משתמשים כאן בשני משתנים בולאניים אשר אמורים לציין האם המשולש כן או לא שווה שוקיים, האם המשולש כן או לא ישר זווית. המשתנים מאותחלים בעת הגדרתם לערך `false` כדי לציין שאנו מניחים בתחילה שהמשולש אינו שווה שוקיים ואינו ישר זווית. אחר אנו בודקים האם המשולש שווה שוקיים. במידה ומתברר כי שתיים מצלעות המשולש שוות אנו מעדכנים את ערכו של המשתנה `isosceles`. באופן דומה אך בלתי תלוי אנו בודקים האם המשולש ישר זווית (על-פי משפט פיתגורס), ובמידה והוא כזה אנו מעדכנים את ערכו של `right_angle` להכיל את הערך `true`. עתה, על סמך ערכם של שני המשתנים אנו מציגים את הפלט הדרוש.

2. בתכנית זאת, בדומה לתכנית שתרגמה ציון מספרי למילים, אנו משתמשים בשרשרת של `if-else`: במידה והמשולש הוא ישר זווית וגם שווה שוקיים אנו מודיעים זאת (ולא בודקים את יתר האפשרויות), אחרת אנו בודקים האם המשולש שווה שוקיים, ואם כן מודיעים על כך; אם המשולש גם אינו שווה שוקיים אנו בודקים האם הוא ישר זווית, ולבסוף אם הוא לא קיים אף לא אחד מהתנאים שנבדקו אנו מסיקים שהוא 'סתם' משולש (במקרה שזה כלל אינו משולש חוקי כבר טיפלנו).

3. הקדמה: נניח כי num1 הוא משתנה מטיפוס int המכיל את הערך 17. אזי הביטוי: num1 הוא ביטוי מטיפוס int, וערכו של ביטוי זה הוא 17. באופן דומה: אם b הוא משתנה בולאני, אזי הביטוי: b, הוא ביטוי מטיפוס בולאני, וערכו הוא כערכו של המשתנה. ועתה לעיקר דְבָרֵנו: אמרנו כי בעת שהמחשב מגיע לפקודת if כגון: if (isosceles == true) הוא בודק האם התנאי מתקיים, במילים מדויקות יותר: האם ערכו של הביטוי הבולאני הוא true. בדוגמה שהצגנו ערכו של הביטוי הבולאני: isosceles == true הוא true בדיוק כאשר ערכו של המשתנה isosceles הוא true, כלומר כאשר ערכו של הביטוי הבולאני הכולל רק את שמו של המשתנה (הביטוי הבולאני: isosceles) הוא true. על-כן במקום לכתוב if (isosceles == true) ניתן, וגם ראוי, לכתוב if (isosceles) ושני התנאים יתקיימו בדיוק באותו מצב. באופן דומה, אם b הוא משתנה בולאני אזי במקום לכתוב if(b == false) ניתן, וגם ראוי, לכתוב if (!b), להזכירכם התו ! משמעו not כלומר הוא הופך את ערכו של הביטוי הבולאני עליו הוא מופעל. מניסיוני אני יודע כי סטודנטים מתקשים להבין נקודה זאת, על-כן נסכם ונאמר (בבחינת נעשה ונשמע) שבמקום לכתוב if (b == true) עדיף לכתוב if (b), ובמקום לכתוב if (b == false) עדיף לכתוב if (!b).

4. אמרנו כי משתנה בולאני משמש אותנו עת אנו רוצים לציין האם אירוע כלשהו התקיים, או האם מצב כלשהו מתקיים. על-כן יש הרואים במשתנה בולאני מעין דגל אשר עשוי להיות מונף (עת המצב מתקיים) או מורד (עת המצב אינו מתקיים). הסתכלות זאת היא לגיטימית; אולם היא גורמת לעיתים לתלמידים לשיים (לתת שם) משתנה בולאני בשם flag וזה לא טוב, שכן בכך אינכם מסבירים מה מהות הדגל? על מה הוא מאותת? שמו של המשתנה הבולאני צריך להעיד על המצב\אירוע עליו הוא מאותת.

5. בתכנית הכנסנו למשתנים הבולאניים ערכים תחיליים (שהיו false במקרה שלנו). לערך שמוכנס למשתנה מלכתחילה, לפני שנערכת בדיקה כלשהי האם זה אכן הערך המתאים, אנו קוראים בשם **ערך מחדלי** (default value), כלומר ערך שאם לא יקרה שום דבר אחר הוא הערך שהמשתנה יכיל.

6. לא ניתן לקרוא ערך לתוך משתנה בולאני, כלומר לא ניתן לבצע פקודה כגון: cin>> b;. כדי להדפיס ערך של משתנה בולאני המירו אותו לטיפוס int, כלומר הדפיסוהו באופן הבא: cout << int(b); הערך שיודפס יהיה 1 אם ערכו של b הוא true, ואפס אם ערכו של b הוא false.

7. אם b הוא משתנה מטיפוס bool אזי ניתן לכתוב השמה כגון: b = (num1 < 17) ; ולמשתנה b יוכנס הערך true אם ורק אם ערכו של המשתנה (המספרי) num1 גדול מ- 17. מאותה סיבה יכולנו גם לכתוב: isosceles = (edge1 == edge2 || edge2 == edge3 || edge1==edge3) ; ופקודת השמה זאת תכניס למשתנה isosceles את הערך true אם התנאי מתקיים, כלומר אם המשולש הינו שווה שוקיים, ואת הערך false אם התנאי אינו מתקיים, כלומר המשולש אינו שווה שוקיים. בדיוק מה שאנו רוצים.

3.3.1 הקשר בין ביטויים אריתמטיים לביטויים בולאניים

עד כה הכרנו ביטויים מספריים (הן שלמים והן ממשיים, כגון: (num1 + num2) %17) שערכם הוא *מספרי*, וביטויים בולאניים (כגון: num1 > num2 או isosceles != false) שערכם הוא *בולאני (כן/לא)*. בשפת C שני טיפוסי הביטויים אינם מנותקים אלה מאלה. בשפה מוגדר כי ביטוי מספרי שערכו שונה מאפס (הן ערך חיובי והן ערך שלילי) יחשב כביטוי בולאני שערכו הוא true, וביטוי מספרי שערכו אפס יובן כביטוי בולאני שערכו הוא false. על כן אם b הוא משתנה בולאני אזי ניתן לכתוב: b =(num1 + num2) %17 ול- b יכנס הערך false אם סכומם של num1 ו- num2 הוא כפולה של 17 (ועל כן שארית החלוקה של הסכום ב- 17 היא אפס), ויכנס ערך true אם סכומם של num1 ו- num2 אינו כפולה של 17. באופן דומה ניתן לכתוב: if (num1) והפקודה הכפופה לתנאי תתבצע אם ורק אם ערכו של num1 שונה מאפס.

באופן סימטרי כל ביטוי בולאני שערכו הוא true יוערך כבעל הערך המספרי אחד, וכל בטוי בולאני שערכו הוא false יוערך כביטוי מספרי שערכו הוא אפס. לכן אם נכתוב num1 = (num2 > 0) אזי ל- num1 ייכנס הערך אחד אם ערכו של num2 גדול מאפס, וייכנס הערך אפס אם ערכו של num2 קטן או שווה לאפס.

כדי לסבך את הדברים עוד יותר נאמר כי עת אנו מבצעים השמה, כגון num1 = 17, אזי ההשמה מחזירה ערך שהוא הערך שהוכנס למשתנה, במילים אחרות לביטוי num1 = 17 יש ערך, במקרה זה הערך 17, להשמה num2 = num1 + 3 יש את הערך 20, במילים אחרות היא מחזירה את הערך 20. עתה נניח כי מתכנת כלשהו רצה לכתוב בתכניתו את התנאי if (num2 == num1 +3) אולם הוא שגה, ובהיסח הדעת הוא כתב if (num2 = num1 +3). מבחינתו של המחשב זוהי פקודה לגיטימית לחלוטין: ראשית, למשתנה num2 *מושם* הערך num1+3, שנית, אם ערך זה אינו אפס (כלומר אם ערכן של num1 אינו 3-) אזי התנאי מתקיים (שכן כביטוי בולאני זהו ביטוי שערכו הוא true), ויש לבצע את הפקודה הכפופה לתנאי. כמובן שהמשמעות של שני התנאים, הנסיבות בהם הם מתקיימים, שונים לגמרי: בתנאי הראשון, אליו התכוון המשורר, ערכו של num2 אינו משתנה, והתנאי מתקיים רק עת num2 גדול ב- 3 מ- num1, בתנאי השני (אותו כתב פזור הדעת) אנו משנים את ערכו של num2 להיות num1+3, והתנאי מתקיים אם הערך שמושם ל- num2 אינו אפס. אנו אומרים כי לתנאי בניסוחו השגוי יש **תוצר לוואי** (side effect) שהוא שינוי ערכו של num2 (מעבר לבדיקת התנאי). יש, ובתוכם כותב שורות אלה, הסבורים כי תכנות עם תוצרי לוואי, גם אם הוא לגיטימי, הוא מאוד מבלבל ולא רצוי, ולכן את התנאי עם תוצר הלוואי ראוי להפריד להשמה, ואחריה פקודת if.

מכיוון שהשמה מחזירה את הערך שהוכנס למשתנה, אזי בשפת C ניתן לכתוב פקודה כגון: num2 = num1 = 17;. פקודה זאת ראשית תכניס ל- num1 את הערך 17, ואחר תכניס ל- num2 את הערך שמחזירה ההשמה num1 = 17, כלומר את הערך 17. לסיכום לשני המשתנים יוכנס אותו ערך, 17.

3.4 פקודת switch

פקודת ה- if-else היא פקודת התנאי הכללית ביותר שמעמידה לרשותנו השפה. כעיקרון ניתן לכתוב כל תכנית תוך שימוש בפקודה זאת בלבד. יחד עם זאת קיימים מצבים בהם נוח יותר, מתאים יותר, להשתמש בפקודת תנאי אחרת, פקודת ה- switch.

נתחיל בדוגמה: נניח כי ברצוננו לכתוב תכנית אשר קוראת מהמשתמש זוג מספרים שלמים, ואחר על-פי בקשתו מדפיסה את סכומם, או הפרשם, או מכפלתם, או מנתם של המספרים. המשתמש יקיש 1 כדי לקבל את הסכום, 2 כדי לקבל את ההפרש,3 כדי לקבל את המכפלה, ו- 4 כדי לקבל את המנה. אתם מוזמנים לכתוב את התכנית תוך שימוש בפקודת if-else. אנו נציג את התכנית תוך שימוש בפקודת ה- switch:

```
cin >> num1 >> num2 ;
cin >> request    // the operation to perform

switch (request)
{
   case 1: cout << num1 + num2 ;
           break ;
   case 2: cout << num1 – num2 ;
           break ;
   case 3: cout << num1 * num2 ;
           break ;
   case 4: if (num2 !=0)
              cout << num1 / num2 ;
           else
              cout << "Can not divize by zero\n" ;
           break ;
   default:
           cout << "a wrong request\n" ;
           break ;
}
```

נסביר: אנו קוראים את שני האופרנדים, ואחר את קוד הפעולה המבוקשת. עתה על-פי ערכו של המשתנה request (הפסוקית case(request) משמעה: על-פי ערכו של request) אנו מבצעים אחד מהבאים: אם ערכו של request אחד אזי אנו מציגים את הסכום, אם ערכו הוא שתיים אנו מציגים את ההפרש,..., אם ערכו הוא ארבע אזי אם num2!=0 אנו מציגים את המנה; ולבסוף (אם ערכו של request אינו בתחום 1..4) אנו מציגים הודעת שגיאה.

באופן כללי יותר:

1. הביטוי המבקר את פקודת ה- switch, זה המופיע בסוגריים מייד אחרי המילה switch, צריך להיות ביטוי **בן-מניה** (enumerated) כלומר ביטוי שניתן למנות בזה אחר זה את ערכיו האפשריים (כפי שניתן לעשות עבור מספרים שלמים, כדוגמת 17, 18, 19, ..., אך לא עבור מספרים ממשיים, שכן בממשיים אין עוקב, למשל לא ניתן לציין איזה מספר בא אחרי 17.0). במילים פשוטות: הביטוי המבקר יכול להיות מטיפוס מספר שלם, או ביטוי בולאני, הוא אינו יכול להיות ביטוי מטיפוס מספר ממשי. לכן ביטוי כגון (num1 + num2)%17 הוא ביטוי מבקר אפשרי (בהנחה שטיפוסם של num1, num2 הוא int), אך ביטוי כגון double(num1)/17 אינו ביטוי מבקר אפשרי.

2. עבור כל ערך צפוי של ביטוי הבקרה מופיעה שורת case מתאימה, אשר מתארת מה הפקודות שיש לבצע במידה וזה ערכו של ביטוי הבקרה. אין צורך לארוז את הפקודות שיש לבצע עבור כל מקרה ומקרה בבלוק.

3. ודאי שמתם לב כי את קבוצת הפקודות אותן מבצעים עבור כל ערך של ביטוי הבקרה אנו מסיימים בַּפקודה break. סיבת הדבר היא שבמידה ולא נכתוב את פקודת ה- break 'יתדרדר' הביצוע משורת ה- case המתאימה, והפקודות הנלוות לה, הלאה והלאה, והמחשב יבצע גם פקודות שמופיעות לצד שורות ה- case הבאות. נדגים באמצעות קטע מפקודת switch כלשהי:

```
case 1 : a = 0 ;
         break ;
case 2 : b = 17 ;
case 3 : c = 3879 ;
case 4 : d = 23;
         break ;
case 5 : cout "enough\n" ;
         break ;
```

במידה וערכו של ביטוי הבקרה יהיה 1 למשתנה a יוכנס הערך אפס, ואחר יבוצע break שיוציא אותנו מפקודת ה- switch (והביצוע יעבור לפקודה הבאה בתכנית). במידה וערכו של ביטוי הבקרה יהיה 2 יוכנסו ערכים למשתנים b, c, d ורק אז ייתקל המחשב בפקודת break שתסיים את ביצוע ה- if. במידה וערכו של ביטוי הבקרה יהיה 3 יוכנסו ערכים למשתנים c, d, ובמידה וערכו של ביטוי הבקרה יהיה 4 יוכנס ערך רק ל- d.

4. ההתנהגות הנ"ל של פקודת ה- break שימושית עת ברצונכם לבצע אותן פקודות עבור מספר ערכים שונים של ביטוי הבקרה, במקרה כזה תכתבו:

```
case 17:
case 18:
case 19: cout << "17 or 18 or 19\n";
         break ;
```

5. לערכים הכתובים לצד כל case אנו קוראים **תוויות** (labels, כדוגמת 17, 18, 19, בקטע התכנית האחרון). התוויות חייבות להיות *ערכים קבועים* שייכתבו על-ידי המתכנת, במילים אחרות *התוויות לא יכולות לכלול ערכי משתנים*. על-כן קטע התכנית הבא לא יתקמפל בהצלחה (בשל התווית num1):

```
case 17 : cout << "a correct label\n" ;
          break ;
case num1 : cout << "this label will cause an error\n";
            break ;
```

6. בסיומה של פקודת ה- switch הראשונה שהדגמנו כללנו מרכיב default. הפקודות הצמודות למרכיב זה תתבצענה אם ערכו של ביטוי הבקרה יהיה שונה מכל הערכים שהופיעו בכל תוויות ה- case השונות. מרכיב ה- default הוא מרכיב רשות, ניתן שלא לכלול אותו, ואז אם ערכו של ביטוי הבקרה יהיה שונה מכל ערכי התוויות פקודת ה- switch לא תבצע דבר.

נציג דוגמה נוספת לפקודת switch. גם הפעם נניח שהמשתמש מזין שני מספרים שלמים, וקוד פעולה שיש לבצע עליהם. קוד הפעולה עשוי להיות: 1=חיבור, 2= חיסור, 3= כפל, 4=חילוק, 5= שארית. מעבר לכך נניח כי אם המשתמש מבקש את קוד הפעולה 2 (=חיסור), אזי נציג לו הן את הסכום והן את ההפרש. אם הוא מבקש 4 (=חילוק) אזי נציג לו הן את המכפלה, והן את המנה, ואם הוא מבקש 5 (=שארית) אזי נציג לו את המכפלה, המנה והשארית. לשם כתיבת הקוד נשתמש בתכונת 'ההתדרדרות' של פקודת ה- switch עת לא מופיעה פקודת break. נציג את הקוד:

```
cin >> num1 >> num2 >> request ;
switch (request)
{
   case 2: cout << num1 – num2 << endl ;
   case 1: cout << num1 + num2 << endl ;
           break ;
   case 5: cout << num1 % num2 << endl ;
   case 4: cout << num1 / num2 << endl ;
   case 3: cout << num1 * num2 << endl ;
           break ;
   default: cout << "A wrong request\n" ;
            break ;
}
```

נסביר: במידה וקוד הפעולה הוא 1, מחושב הסכום, אחריו מתדרדר הביצוע לפקודת ה- break, אשר שוברת את ביצוע הפקודה, ובכך הודפס הסכום, כנדרש, ורק הוא. לעומת זאת, אם קוד הפעולה הוא 2, אזי ראשית מודפס ההפרש, מכאן מתדרדר הביצוע לשורה הבאה, ומודפס הסכום (כפי שהגדרנו שיש לבצע), עתה הוא מגיע לשורת ה- break, ולכן תוצאות נוספות אינן מודפסות. דבר דומה יקרה אם יוזנו קודי הפעולה 5, 4 או 3: אם יוזן קוד הפעולה 5 תוצג ראשית השארית, שנית המנה, ולבסוף המכפלה. אם, לעומת זאת יוזן קוד הפעולה 4, אזי תוצג המנה, ואחריה המכפלה. הזנת קוד הפעולה 5 תכניס אותנו בשורה המתאימה, תוצג המכפלה, ובזאת ישבר ביצוע הפקודה (כנדרש).

3.5 אופרטור ה- ?

אופרטור ה- ? הוא מעין if מקוצר. לעיתים אנו נזקקים לפקודת if כגון:

```
if (round % 2 == 0)
   a = 1 ;
else
   a = 2 ;
```

פקודה כזאת אנו יכולים להמיר בַּפקודה הקומפקטית יותר:

```
.a = (round % 2 == 0) ? 1 : 2 ;
```

נסביר: אופרטור ה- ? המופיע בַּפקודה החלופית לַ- if כולל שלושה מרכיבים:

א. מרכיב הנמצא משמאל ל- ? (כלומר בין סימן ההשמה, שאינו חלק מאופרטור ה- ?, אלא חלק מהפקודה בה נכלל אופרטור זה, לבין התו ?): זהו ביטוי בולאני (במילים פשוטות תנאי).

ב. מרכיב הנמצא בין ה- ? לַ- : , זהו הערך שאופרטור ה- ? יחזיר במידה והביטוי הבולאני מסופק (במילים פשוטות: במידה והתנאי מתקיים).

ג. מרכיב הנמצא מימין ל- : , זהו הערך שאופרטור ה- ? יחזיר במידה והביטוי אינו מסופק (במילים פשוטות: במידה והתנאי אינו מתקיים).

לכן בדוגמה שהצגנו איזה ערך יכנס ל- a? אם התנאי יתקיים, אופרטור ה- ? יחזיר את הערך 1, וזה הערך שיכנס ל- a, אחרת: אופרטור ה- ? יחזיר את הערך 2, ואז זה הערך שפקודת ההשמה תשים ל- a.

נציג דוגמה נוספת לשימוש באופרטור ה- ?. נניח כי ברצוננו להגדיל את a (שהינו משתנה שלם) כך שהוא יכיל את המספר הזוגי הקטן ביותר הגדול מערכו הנוכחי של a. נוכל להשתמש באופרטור ה- ? באופן הבא:

```
a = (a % 2 == 0) ? a+2 : a+1 ;
```

נסביר: אם ערכו של a זוגי (אם a%2 == 0) עלינו להגדיל את a בשתיים, ואם ערכו של a פרדי עלינו להגדילו באחד. אופרטור ה- ? עושה עבורנו את העבודה: במידה וערכו של a זוגי הוא מחזיר את a+2, ואחרת הוא מחזיר את a+1. הערך שהאופרטור מחזיר הוא שמושם ל- a, ולכן a מתעדכן כפי מבוקשנו.

שימוש אפשרי אחר לאופרטור ה- ? עשוי להיות בפקודת פלט:

```
cout << ( (b == true) ? "X" : "O" );
```

בדוגמה זאת אופרטור ה- ? בודק מה ערכו של b. במידה וערכו הוא true, מחזיר האופרטור את הסטרינג "X", ולכן סטרינג זה הוא שיוצג על-ידי פקודת ה- cout; במידה וערכו של b אינו true יחזיר האופרטור את הערך "O", ולכן זה יהיה הפלט שיוצג על-ידי פקודת ה- cout. אגב, הסוגריים החיצוניים (אלה שתוחמים את האופרטור על שלושת מרכיביו) חיוניים במקרה זה.

אעיר כי משני הצדדים של הנקודותיים צריך לעמוד ביטוי מאותו טיפוס, לכן הפקודה הבאה **לא** תתקמפל בהמלחה:

```
cout << ((num2 != 0) ? (num1/num2) : "cannot div") ;
```

בפקודה זאת מצדו השמאלי של הנקודותיים ניצב ביטוי מספרי, ומצדו הימני מחרוזת, וכאמור זה אסור.

3.6 תרגילים

3.6.1 תרגיל מספר אחד: הצגת יום בשבוע

כתבו תכנית הקוראת תאריך במאה העשרים ומציגה את היום בשבוע בו חל התאריך. התכנית תקרא מהמשתמש שלושה מספרים טבעיים: יום בחודש, חודש בשנה, שנה במאה העשרים (לדוגמה: כדי להזין את התאריך: 29/11/1947 יזין המשתמש את המספרים: 29 11 47). התכנית תחשב את היום בשבוע תוך שימוש באלגוריתם הבא:

א. נניח כי המשתנים day, month, year מחזיקים את הקלט.

ב. המשתנה year_div_4 יחזיק את המנה year/4 .

ג. המשתנה year_p_year_div_4 יחזיק את הסכום: year + year_div_4.

ד. המשתנה month_code יקבל ערך באופן הבא:

ערכו של month	ערכו של month_code
1	1
2	4
3	4
4	0
5	2
6	5
7	0
8	3

9	6
10	1
11	4
12	6

ה. המשתנה day_code יקבל את הסכום: day + year_p_year_div_4+month_code

ו. המשתנה day_in_week יקבל את הערך: day_code%7 כאשר שבת שקול לערך אפס, וימים א' עד ו' מתאימים לערכים אחד עד שש.

3.6.2 תרגיל מספר שתיים: הצגת תאריך במילים

כתבו תכנית הקוראת מהמשתמש תאריך בפורמט: day month year (כדוגמת: 29 11 1947). על התכנית להדפיס את התאריך בפורמט הבא:

א. היום יוצג בתוספת st, nd, rd, th כנדרש.

ב. החודש יוצג בשמו (January, February, March,…,December).

ג. השנה תוצג כפי שהיא נקראה.

כמו כן עליכם לבדוק שהקלט תקין:

א. היום בחדש הוא בין אחד ל- 31 בחדשים 1, 3, 5, 7, 8, 10, 12.

ב. היום בחדש הוא בין אחד ל- 30 בחדשים 4, 6, 9, 11.

ג. היום בחדש הוא בין אחד ל- 28 בחדש פברואר, פרט לשנה מעוברת בה היום בחדש הוא בין אחד ל- 29. שנה הינה מעוברת אם: מספר השנה הוא כפולה של ארבע, ואינו כפולה של מאה, או שמספר השנה הוא כפולה של ארבע מאות.

ד. השנה הינה מספר טבעי קטן או שווה מ- 3000.

3.6.3 תרגיל מספר שלוש

4 לולאות

שתי פקודות התנאי שראינו בפרק שעבר (if-else ו- switch), מהוות יחד עם פקודות הלולאה, שנכיר בפרק הנוכחי את אוסף **פקודות הבקרה** (control statements) הקיימות בשפה. פקודות בקרה, באופן כללי, מבקרות את מהלך התקדמות התכנית: אילו פקודות מתוך הנכללות בתכנית תבוצענה בהרצה כלשהי, וכמה פעמים הן תבוצענה. מתוכן פקודות התנאי גורמות לכך שפקודות כלשהן תבוצענה או לא תבוצענה, ופקודות הלולאה גורמות לכך שפקודות כלשהן תבוצענה כמה וכמה פעמים באופן מחזורי. שפת C מעמידה לרשותנו שלוש פקודות לולאה: while, for, do-while.

4.1 פקודת ה- while

פקודת התנאי הראשונה שנכיר היא פקודת ה- `while`. זוהי פקודת הלולאה הכללית ביותר; למעשה ניתן לכתוב כל תכנית תוך שימוש רק בפקודת לולאה זאת, יחד עם זאת ישנם מצבים בהם מתאים יותר להשתמש בפקודות הלולאה האחרות, ועל כן בפועל אנו עושים שימוש בכל שלוש פקודות הלולאה.

4.1.1 דוגמות ראשונות

נתחיל בדוגמה שתציג את השימוש בפקודה. נביט בשני קטעי קוד דומים אך שונים.

```
cin >> num1 >> num2 ;

if (num1 <= num2) {
   cout << num1 << " " ;
   num1++ ;
}
cout << "bye\n" ;
```

```
cin >> num1 >> num2 ;

while (num1 <= num2) {
   cout << num1 << " " ;
   num1++ ;
}
cout << "bye\n" ;
```

לפני שאנו מתעמקים בשני קטעי הקוד נסביר מה משמעות הפקודה `num1++;`: פקודה זאת מגדילה את ערכו של `num1` באחד, והיא שקולה על-כן לפקודה `num1 = num1 +1;` ניתן לכתוב את הפקודה גם באופן `++num1;`, בין שתי הדרכים השונות קיים הבדל דק עליו נעמוד בהמשך, לצורך ענייננו הנוכחי שתי צורות הכתיבה שקולות. הפקודה נקראת פקודת **הגדלה עצמית** (auto increment). בדומה לה קיימת בשפה גם פקודת ההקטנה העצמית (auto decrement) הנכתבת באופן: `num1--` או `--num1` פקודה זאת מקטינה את ערכו של המשתנה `num1` באחד.

נפנה עתה לבחון את קטע הקוד השמאלי. נניח כי המשתמש מזין לתכנית את הקלט: 5 3 (שלוש מוזן ראשון, חמש שני). התנאי `num1 <= num2` יסופק, על-כן התכנית תציג את ערכו של `num1`, כלומר תציג את הערך 3, וערכו של `num1` יגדל באחד להיות 4. עתה תמשיך התכנית לפקודה הבאה ותציג את הפלט `bye`.

עתה נבחן את קטע התכנית הימני: שוב נניח כי המשתמש מזין את הקלט: 5 3. גם כאן אחרי קריאת הקלט פונה התכנית לבדיקת התנאי `num1 <= num2`, מגלה

שהוא מסופק ועל-כן פונה לביצוע הגוש: היא מציגה את ערכו של num1 (כלומר את הערך 3), ומגדילה את ערכו של num1 באחד. אולם בתכנית זאת, מכיוון שפקודת הבקרה היא while (ולא if כפי שהיה בקטע הקוד השמאלי) התכנית אינה פונה לביצוע פקודת הפלט cout << "bye\n" אלא *חוזרת לכותרת הלולאה לבדיקת התנאי* *num1 <= num2*. מכיוון שהתנאי עדיין מתקיים התוכנית חוזרת ומבצעת את הגוש: היא מדפיסה את ערכו של num1 (כלומר את הערך 4) ומגדילה את ערכו של num1 להיות 5. בתום ביצוע גוף הלולאה חוזרת התכנית שוב לבדיקת התנאי שבכותרת הלולאה: התנאי עדיין מתקיים (num1 עדיין קטן אן שווה מ- num2) ולכן התכנית פונה לביצוע גוף הלולאה בשלישית: היא מדפיסה 5, ומגדילה את ערכו של num1 להיות 6. בתום ביצוע גוף הלולאה חוזר המחשב לבדיקת התנאי שבכותרת הלולאה: עתה התנאי כבר אינו מסופק ועל כן המחשב אינו מבצע את גוף הלולאה, הוא פונה אל מעבר ללולאה, במקרה שלנו לפקודה: cout << "bye\n".

באופן כללי פקודת ה- while נכתבת באופן הבא, במילים אחרות **התחביר** (syntax) של הפקודה הוא התחביר הבא: while (boolean exprssion) statement. **המשמעות** (semantic) של הפקודה היא המשמעות הבאה: חזור שוב ושוב כל עוד הביטוי הבולאני שבכותרת הלולאה מסופק: בצע את הפקודה שבגוף הלולאה, וחזור לבדיקת התנאי שבכותרת הלולאה.

שימו לב כי אם הקלט היה 3 5 אזי הלולאה שתיארנו לא הייתה מתבצעת אף לא פעם אחת, ואין בכך כל פגם.

נבחן עתה לולאה מעט שונה:

```
cin >> num1 >> num2 ;

while (num1 != num2) {
   cout << num1 << " " ;
   num1++ ;
}
cout << "bye\n" ;
```

נבדוק את התנהגות התכנית על הקלט: 3 5. עת הביצוע יגיע לכותרת הלולאה בפעם הראשונה התנאי שבכותרת יסופק, על-כן גוף הלולאה יבוצע: יוצג 5, ו- num1 יגדל להיות 6. הביצוע יחזור לכותרת הלולאה, שוב התנאי יסופק, על-כן גוף הלולאה יבוצע בשנית: יוצג 6, ו- num1 יגדל להיות 7. הביצוע יחזור לכותרת הלולאה, שוב התנאי יסופק... וכך עד בלי די. אנו אומרים כי התכנית נקלעה **ללולאה אינסופית** (infinite loop). בדרך כלל לולאה אינסופית הינה בגדר שגיאה בתכנית. במקרה שהוצג נקל יהיה לאתר את השגיאה שכן התכנית תמלא את המסך בפלט. אולם אם הלולאה אינה כוללת פקודות פלט אתם עלולים לחשוב שביצוע התכנית מתארך לו משום מה, ולא לחשוד שזו לולאה אינסופית. ככלל ניתן לומר שאם תכניתכם אינה מציגה פלט כהרף עין, סביר להניח שהיא כוללת לולאה אינסופית. בדרך כלל הפתרון הוא לעצור מיוזמתכם את ביצוע התכנית (על-ידי הקשת Ctrl + Break), ולאתר את השגיאה.

בסוגריים אעיר כי למעשה בתכנית זאת עת היא תורץ במחשב הלולאה לא תהיה אינסופית: בשלב מסוים התכנית תעצור. הסיבה לכך היא שערכו של num1 ילך ויגדל, עד אשר בשלב מסוים הוא יגיע לערך הגבוה ביותר שמשתנה מטיפוס int (בהנחה שזה טיפוסו) עשוי להכיל. ניסיון להגדיל את num1 מעבר לכך, בסיבוב הבא

בלולאה, יגרום לגלישה (overflow) ויכניס ל- num1 את הערך השלילי הקטן ביותר שניתן להכניס למשתנה שלם. מכאן ילך num1 ויגדל, עד שלבסוף ערכו ישתווה לזה של num2, ובכך הלולאה תיעצר.

4.1.2 הדפסת כפולותיו של מספר

נראה עתה דוגמה של קטע תכנית המשתמש בפקודת ה- while. קטע התכנית קורא מהמשתמש מספר שלם, ומציג את עשר כפולותיו הראשונות של המספר הנקרא. נציג את התכנית:

```
cin >> num ;

counter = 1 ;
while (counter <= 10) {
   cout << num * counter << endl ;
   counter++ ;
}
```

המשתנה counter (אשר אנו, כמובן, מניחים כי הוגדר קודם לכן) משמש כמשתנה הבקרה של הלולאה. המשתנה מונה כמה פעמים אנו מסתובבים בלולאה, שכן בכל סיבוב בלולאה (לעיתים נשתמש במונח בכל איטרציה iteration) ערכו גדל באחד. מכיוון שהמשתנה אותחל לפני הלולאה לערך אחד, והוא כאמור גדל בכל סיבוב באחד, הרי עת אנו מדפיסים את num*counter אנו מדפיסים את num*1, num*2, num*3,…, num*10 כלומר את עשר כפולותיו הראשונות של num כמבוקש.

אני מסב את תשומת לבכם לעימוד (הדומה לזה שפגשנו בפקודת ה- if).

4.1.3 הדפסת לוח הכפל

התכנית הקודמת שראינו הדפיסה את כפולותיו של מספר יחיד, עתה נרצה להכלילה לכדי תכנית אשר מדפיסה את לוח הכפל של 10 X 10. נוכל להתייחס ללולאה שכתבנו קודם לכן כמעין קופסה שחורה אשר מדפיסה כפולות של ערך נתון של num. בקופסה השחורה נוכל לעשות שימוש אם נכתוב פקודת לולאה חדשה אשר (במקום לקרוא את ערכו של num מהמשתמש) תריץ את num על הערכים 1, 2, 3,…10 ועבור כל ערך של num תקרא לקופסה השחורה. נציג את התכנית:

```
num = 1 ;
while (num <= 10) {
   counter = 1 ;
   while (counter <=10) {
      cout << num * counter ;
      counter++ ;
   }
}
```

נבחן את התכנית שכתבנו:

1. השורה השלישית עד השביעית הן קטע התכנית שראינו קודם לכן, ושעתה יכול להוות עבורנו כמעין קופסה שחורה אשר מדפיסה את כפולותיו של num נתון. קטע זה קיננו (מלשון קן nest) בתוך לולאה אחרת, חיצונית. לפני הלולאה החיצונית אִתחלנו את num1 להיות אחד, ועל הלולאה להריץ את num כך שהוא יתקדם בין הערכים 2, 3, …, 10.

2. בתכנית שכתבנו נפלה שגיאה (באג): אחרי ביצוע הקופסה השחורה שכחנו להגדיל את ערכו של num באחד. התוצאה היא שבסיבוב הראשון של הלולאה החיצונית (זו שהוספנו בתכנית האחרונה) תוצגנה כפולותיו של אחד, בתום ההצגה num לא יגדל להיות שתיים, ולכן בסיבוב השני של הלולאה החיצונית שוב תוצגנה כפולותיו של אחד, וכך עד בלי די, והתכנית שלנו תקלע ללולאה אינסופית. על כן נתקן את התכנית:

```
num = 1 ;
while (num <= 10) {
   counter = 1 ;
   while (counter <=10) {
      cout << num * counter ;
      counter++ ;
   }
   num++ ;
}
```

גרסה זאת של התכנית כבר תציג את עשר הכפולות של המספרים אחד עד עשר כמבוקש, אולם גם היא אינה מושלמת: בדרך-כלל אנו רגילים שלוח הכפל מוצג כך שכפולותיו של כל מספר מופיעות בשורה נפרדת; בתכנית שלנו כל הכפולות של כל המספרים מופיעות בשורה אחת ארוכה (הכוללת מאה נתונים). נרצה על-כן לשפר גם לקוי זה, ונעשה זאת על-ידי שאחרי שאנו מדפיסים את כפולותיו של מספר כלשהו (תוך שימוש בקופסה השחורה, במלים אחרות בלולאה הפנימית) נשבור שורה בפלט. נציג עתה את התכנית המושלמת:

```
num = 1 ;
while (num <= 10) {
   counter = 1 ;
   while (counter <=10) {
      cout << num * counter ;
      counter++ ;
   }
   cout << endl ;
   num++ ;
}
```

3. טעות נפוצה נוספת היא לאתחל את ערכו של counter לא במקום בו הוא מאותחל בתכניתנו (מעל הלולאה הפנימית), אלא מעל הלולאה החיצונית, (לצד איתחולו של num). בדקו לעצמכם מה הייתה עושה התכנית לו היא הייתה נכתבת באופן הנ"ל, בפרט איזה פלט היא הייתה מציגה.

4.1.4 זמן ריצה של תכנית

עת אנו בוחנים תכנית כלשהי, הפותרת בעיה כלשהי, בשיטה כלשהי, במילים אחרות תוך שימוש באלגוריתם כלשהו, אנו מעוניינים לדעת עד כמה ביצוע התכנית יחייב זמן יחסית רב או קצר, (יחסית לתכניות אחרות). כאשר אנו שואלים עד כמה ביצוע התכנית יהיה ממושך או קצר איננו מתכוונים לזמן הביצוע בשניות, שכן זמן זה תלוי במרכיבים רבים ומשתנים (כגון המחשב עליו מורצת התכנית). כאשר אנו דנים **במורכבות זמן הריצה** (time complexity) של תכנית אנו מודדים זאת במספר הפעולות שהתכנית מבצעת כפונקציה של גודל הנתונים עליהם היא פועלת (תוך שאנו מניחים שביצוע כל פעולה ופעולה דורש זמן יחסית אחיד). נבהיר את כוונתנו: התכנית האחרונה שלנו הדפיסה את לוח הכפל של עשרה מספרים, לכל אחד הוצגו עשר כפולותיו. גודל הנתונים במקרה זה מצוין על-ידי שני פרמטרים: מספר הנתונים שאת כפולותיהם יש להציג, ומספר הכפולות של כל מספר שיש להציג.

ברור שככל שיהיו יותר מספרים או יותר כפולות התכנית תצטרך לבצע יותר עבודה, ולכן השאלה אותה אנו שואלים היא: באיזה אופן גדל שיעור העבודה שעל התכנית לבצע כפונקציה של מספר הנתונים ושל מספר הכפולות. נציג דוגמה נוספת אשר תבהיר למה כוונתנו במונח 'מורכבות זמן ריצה כפונקציה של גודל הנתונים': נניח תכנית אשר ממיינת מספרים; גודל הנתונים עבור תכנית שכזאת יהיה מספר הנתונים שיש למיין, באשר ברור שככל שיש למיין יותר נתונים תידרש התכנית לבצע יותר פעולות. השאלה שנשאל את עצמנו תהיה שוב: באיזה אופן גדלה כמות העבודה שעל התכנית לבצע ככל שגדל מספר המספרים שיש למיין. לבסוף נחשוב על תכנית הבודקת האם מספר כלשהו הוא ראשוני; גודל הנתונים כאן הוא גודלו של המספר (ככל שהמספר יהיה גדול יותר יהיה על התכנית לבצע יותר עבודה); גם כאן נשאל באיזה מידה גדל שיעור העבודה שעל התכנית לבצע כפונקציה של גודלו של המספר שאת ראשוניותו יש לבדוק.

עת אנו דנים במורכבות זמן הריצה של תכנית (כפונקציה של גודל הנתונים) אנו מתעלמים מהבדלים שהינם בשיעור כפל במספר קבוע. כלומר מבחינת *הדיון התיאורטי* אם תכנית אחת רצה בזמן שהוא $17*n^3$ (עבור n שהוא גודל הנתונים), ותכנית שניה רצה בזמן שהוא $3879*n^3$, אזי אנו אומרים שמורכבות זמן הריצה של שתי התכניות זהה. הסיבה לכך היא שעת n הולך וגדל לכיוון האינסוף, כמות העבודה המבוצעת נקבעת בראש וראשונה ובעיקר על-פי n, והתרומה של הכפל בכל אחד משני הקבועים השונים הופכת להיות זניחה. כמובן שעבור משתמש היושב מול מסך המחשב יש משמעות רבה להבדל בין שני הערכים, אולם מנקודת ראותו של התיאורטיקאי הם הינו הך, (אך הם שונים מתכנית שרצה בזמן n^4 או בזמן 3^n). על-כן הדיון התיאורטי נעשה לצד (ולא בסתירה) לשאיפה לכתוב תכנית יחסית קצרה ויעילה (כזאת בה שיעורו של הקבוע יהיה קטן ככל האפשר).

כאמור, מבחינתו של התיאורטיקאי ההבדלים בזמני ריצה הם בין תכנית שרצה בזמן **לינארי** בגודל הנתונים, (כלומר בזמן שהוא יחסי ישר לגודל הנתונים), לבין תכנית שרצה בזמן שהוא **פולינומיאלי** בגודל הנתונים, כלומר שניתן לבטאו כפולינום של גודל הנתונים (כגון n^2 או n^{17}), שכן כאשר n ילך ויגדל *היחס בין* n לבין n^2 ילך ויגדל (בעוד *היחס* בין $17*n^9$ לבין $3879*n^9$ יישאר קבוע). באופן דומה נבדיל בין תכנית שרצה בזמן לוגריתמי בגודל הנתונים (זמן הריצה הוא lg(n)) לבין תכנית שרצה בזמן לינארי בגודל הנתונים. והסיבה היא שוב שככל ש- n הולך וגדל היחס בין n לבין lg(n) הולך וגדל.

מכיוון שברמה התיאורטית אנו מתעניינים בזמן הריצה בלי תלות בכפל בקבוע אזי אין לנו עניין בשאלה כמה פעולות מבוצעות בתוך כל לולאה, אלא רק בשאלה כמה פעמים מתבצעת כל לולאה (כפונקציה של גודל הנתונים). שכן נניח שלולאה אחת מתבצעת n^2 פעמים ויש בה 17 פקודות בגוף הלולאה, אזי כמות העבודה שאנו עושים היא $17*n^2$, לעומתה לולאה שניה מתבצעת אותו מספר פעמים, אך בגופה יש 3879 פעולות, כמות העבודה שנבצע עתה תהיה $3879*n^2$, וכבר אמרנו שמבחינתנו שני הערכים הללו שקולים. לכן עת אנו מתעניינים בזמן ריצה נתעניין רק במספר הפעמים שמתבצעות לולאות שונות. באופן פורמלי יותר אנו אומרים שאנו סופרים רק פעולות השוואה (שהרי לפני כל סיבוב בכל לולאה מתבצעת השוואה, במילים אחרות בדיקה של תנאי כלשהו).

נחזור עתה לתכנית שהציגה את לוח הכפל, ונסמן ב- m את מספר הכפולות שיש להציג, וב- n את מספר המספרים שאת כפולותיהם יש להציג, אזי הלולאה הפנימית רצה m פעמים (עבור כל מספר), והיא מורצת n פעמים על-ידי הלולאה החיצונית, לכן שיעור העבודה המבוצע על-ידי התכנית הוא m*n.

לאורך הדרך נעריך עבור תכניות שונות שנכתוב את זמן הריצה. אני ממליץ גם לכם להעריך בעצמכם את זמן הריצה של התכניות שאתם כותבים. זו מיומנות שחשוב לרכוש.

4.1.5 הדפסת מחלקי מספר

נציג עתה תכנית נוספת אשר עושה שימוש בלולאות. התכנית תקרא מהמשתמש מספר ותציג את מחלקיו (פרט לאחד ולמספר עצמו).

```
cin >> num ;

divider = 2 ;
while (divider < num) {
   if (num % divider == 0)
      cout << divider << " " ;
   divider++ ;
}
```

בתכנית המשתנה `divider` עובר בלולאה על הערכים `2, 3, …, num-1`, עבור כל ערך בודקים האם ערכו הנוכחי של `divider` מחלק את `num` (האם `num%divider == 0`) ואם כן מוצג ערכו הנוכחי של `divider`.

שימו לב כי את פעולת ההגדלה של `divider` איננו כופפים לתנאי `num % divider == 0`. שאלו את עצמכם האם התכנית הייתה פועלת כהלכה לו פקודת ההגדלה הייתה כפופה לתנאי הנ"ל, כלומר לו התכנית הייתה נראית באופן הבא:

```
cin >> num ;

divider = 2 ;
while (divider < num) {
   if (num % divider == 0)
   {
      cout << divider << " " ;
      divider++ ;
   }
}
```

מה הוא זמן הריצה של התכנית שהצגנו? גודלם של הנתונים כאן הוא גודלו של המספר שאת מחלקיו יש להציג, והלולאה היחידה רצה עבור ערכי `divider` הבאים: `2, 3, … num-1` כלומר לערך `num` פעמים. לכן זמן הריצה הוא לינארי ב- `n`, עבור `n` שמציין את גודל המספר שאת מחלקיו עלינו להציג.

האם ביכולתנו לכתוב תכנית יעילה יותר? חדי עין בניכם ודאי יאמרו מיד כי אין צורך לרוץ עם `divider` עד `num-1`, די להתקדם עד `num/2` שכן למספר לעולם לא יהיה מחלק הגדול מחציו. שיפור זה יקטין את זמן הריצה פי שניים, שיפור בהחלט משמעותי מההיבט המעשי, אך כזה המשאיר את מורכבות זמן הריצה בסדר גודל לינארי ב- `num` (כזכור, מבחינת מורכבות זמן הריצה אין הבדל בין זמן ריצה של `num`, `17*num` או `0.5*num`).

מתוחכמים יותר יוסיפו שאם `num` פרדי (אי זוגי) אזי אין טעם לבדוק ערכים זוגיים של `divider`, ובכך ניתן להקטין את זמן הריצה שוב פי שתיים. כמובן שגם שיפור זה לא משנה את מורכבות זמן הריצה.

למרות שהשיפור האחרון, גם עת בא לידי ביטוי (כלומר עת num פרדי), לא משנה את מורכבות זמן הריצה, אנצל אותו כדי לחדד נקודה נוספת בדיוננו אודות מורכבות זמן הריצה: כאשר אנו דנים במורכבות זמן הריצה אנו שואלים כמה עבודה מבצעת התכנית כפונקציה של גודל הקלט עבור *הקלט הגרוע ביותר האפשרי*, או לעיתים עבור *קלט ממוצע* (אנו מפרידים בין שני הדיונים: קלט גרוע ביותר וקלט ממוצע). השיפור האחרון שהצענו ישפר את זמן הריצה הממוצע, אך לא את זמן הריצה על הקלט הגרוע ביותר (שיהיה במקרה זה קלט הכולל מספר זוגי; וכאמור גם עת חל השיפור אין לו משמעות ברמה התיאורטית של מורכבות זמן הריצה).

אם כן, האם התכנית שהצגנו היא היעילה ביותר האפשרית לשם פתרון הבעיה? אם נקדיש לכך מעט מחשבה נגלה שאנו יכולים לשפר גם את מורכבות זמן הריצה: שכן די לרוץ עם divider עד שורש num. הסיבה לכך היא שאם ל- num יש מחלק הגדול משורשו, אזי לאותו מחלק יש 'בן זוג' הקטן משורש num, באופן שמכפלת 'בני הזוג' היא num. לדוגמה: למאה יש מחלק הגדול משורשו (מעשר) למשל 25, אולם למחלק זה יש 'בן זוג' הקטן מעשר: 4, באופן ש: 25. * 4 = 100. לכן נוכל לשפר את התכנית שהצגנו ולקבל את התכנית הבאה: :

```
cin >> num ;

divider = 2 ;
while (divider <= sqrt(num) {
   if (num % divider == 0)
     cout << divider << " " << num/divider << " " ;
   divider++ ;
}
```

זמן הריצה של תכנית זאת הוא $n^{1/2}$ וזה כבר שיפור גם מהבחינה התיאורטית, שכן ככל ש- n הולך וגדל היחס בין n לבין $n^{1/2}$ הולך וגדל.

לתכנית האחרונה שהצגנו יש מספר מגבלות:

1. במידה ושורש המספר הוא מספר שלם (כמו במקרה בו המספר שהוזן הוא 4 או 100) יוצג השורש פעמיים, וזה, כפי שכבר ציינו בעבר, לא מוצלח. אני מזמין אתכם לחשוב כיצד לפתור לקוי זה.
2. מגבלה שניה של התכנית היא שהמחולקים לא יופיעו בה ממוינים מקטן לגדול. על מגבלה זאת אין ביכולתנו להתגבר.
3. בכל סיבוב בלולאה אנו בודקים האם divider <= sqrt(double(num)) , ולשם כך מחשבים שוב, ושוב את שורשו של num. יש בכך משום הביזבוז. ניתן לחשב את שורש num פעם יחידה, לפני הלולאה; להכניסו למשתנה עזר (לדוגמה: nums_root), ובכל סיבוב בלולאה לשאול האם: divider <= nums_root. כך נסחוך, כמובן, עבודה וזמן ביצוע.

לסיום, אני מזכיר לכם כי על-מנת שתוכלו לזמן את הפונקציה sqrt() עליכם לכלול בתכניתכם את ההוראה #include <cmath> (אשר תיכתב לצד ה- #include <iostream>). אין צורך בתוספת פסוקית using.

4.1.6 בדיקה האם מספר ראשוני, ופקודת break מלולאה

נציג עתה תכנית נוספת אשר משתמשת בלולאות: ברצוננו לכתוב קטע תכנית אשר קורא מהמשתמש מספר ומודיע האם המספר ראשוני או פריק (לא ראשוני).

```
cin >> num ;
```

```
prime = true ;
divider = 2 ;
while( divider <= sqrt(num))
{
   if (num % divider == 0)
      prime = false ;
   divider++ ;
}
if (prime)
   cout << "prime\n" ;
else
   cout << "not prime\n" ;
```

כעיקרון, תכנית זאת דומה מאוד לקודמתה (גם זמן הריצה שלהן זהה): אנו עוברים על כל המספרים משתיים ועד שורש המספר הנקרא, ולכל אחד בודקים האם הוא מחלק את num.

אנו עושים כאן שוב שימוש במשתנה בולאני (prime). תפקידו של המשתנה לאותת לנו האם num ראשוני או פָּריק (לא ראשוני). המשתנה הבולאני מתופעל באופן הבא: בתחילה, לפני הלולאה, אנו מניחים באופן מחדלי כי המספר שקראנו ראשוני (prime = true). אחַר, בתוך הלולאה, אם אנו מגלים שערך כלשהו של divider מחלק את num אנו מעדכנים את ערכו של prime להיות false כדי לציין ש- num אינו ראשוני.

טעות נפוצה בקרב מתכנתים מתחילים היא לכתוב את הלולאה באופן הבא:

```
while( divider <= sqrt(num))
{
   if (num % divider == 0)
      prime = false ;
   else
      prime = true ;
   divider++ ;
}
```

בדקו לעצמכם מדוע גרסה זאת של התכנית היא שגויה (למשל הניחו כי ערכו של num הוא עשר, ועקבו אחר השתנות ערכי divider ו- prime).

גם לתכנית זאת אנו יכולים להכניס שיפור: במידה וגילינו כי המספר שקראנו פָּריק אין לנו צורך להמשיך ולהריץ את הלולאה, ניתן לצאת מהלולאה באופן מיידי ולהודיע כי המספר פריק. כיצד נעשה זאת? על-ידי שנוסיף לתנאי בלולאה מרכיב נוסף, כך שהוא יראה באופן הבא: while(divider<=sqrt(num)&& prime): עתה הלולאה תתבצע כל עוד: גם ערכו של divider קטן או שווה משורש num, וגם ערכו של prime הוא true (להזכירכם התנאי prime שקול לתנאי prime == true). במילים אחרות, אחרי שערכו של prime ישונה להיות false (דבר שיקרה אם וכאשר נגלה ערך של divider המחלק את num), התנאי שבכותרת הלולאה כבר לא יסופק, וביצוע הלולאה יופסק. (אני מסב את תשומת לבכם ששיפור זה ישנה את יעילות ריצתה של התכנית, אך לא את מורכבות זמן ריצתה, שכן במקרה הגרוע, בו הוזן לנו מספר ראשוני, זמן הריצה לא קטן).

לעיתים תלמידים מציעים שיטה אחרת להשיג אותה מטרה: יציאה מוקדמת מהלולאה עת גילינו שהמספר שהוזן פריק. השיטה הינה הבאה: אם גילינו שהמספר פריק אזי נכניס לבולאני את הערך false, וכן נעדכן את ערכו של div להיות num, ואז תנאי הלולאה כבר לא יתקיים. כלומר נכתוב את הלולאה באופן הבא:

```
while( divider <= sqrt(num))
{
   if (num % divider == 0)
   {
      prime = false ;
      divider = num ;
    }
   divider++ ;
}
```

לטעמי זו שיטת קידוד ברברית! הקפצת ערכו של משתנה הבקרה באופן פתאומי לערך שיגרום לתכנית לא להכנס ללולאה אכן תשיג את המטרה הרצויה, אולם היא בגדר סגנון תכנותי קלוקל.

עוד הצעת יעול של תלמידים שברצוני לדחות היא הבאה:

```
if (num > 2 && num %2 == 0)
   cout <<< "not prime\n" ;
else
{
    לולאת בדיקת הראשוניות כפי שהוצגה
}
```

כלומר הרעיון הוא שעבור זוגיים גדולים משתיים לא ניכנס כלל ללולאה. זו רק דוגמה אחת לתנאים נוספים שתלמידים מציעים ושלכאורה יחסכו עבודה במקרים מסויימים. אני פוסל רעיונות אלה שכן הם מסרבלים את הקוד, וכמו כן הקוד הכללי (והקצר) שניתן, לכל הפחות במקרה זה, מכסה גם את המקרים הללו ביעילות (כבר עת div יהיה שתיים נגלה שהמספר פריק ובכך נסיים את תהליך הבדיקה). המסקנה הכללית: *העדיפו לכסות את כל המקרים השונים באמצעות מקרה אחד*, והימנעו ככל האפשר מלהפריד את הטיפול לקלט מסוג א', קלט מסוג ב', וקלט מסוג ג'.

את השיפור שהזכרנו אנו יכולים להשיג גם בדרך חלופית. נציג את הקוד ואחר נסבירו:

```
while( divider <= sqrt(num))
{
   if (num % divider == 0)
   {
      prime = false ;
      break ;
   }
   divider++ ;
}
cout << ((prime) ? "prime\n" : "not prime\n") ;
```

הוספנו לַתכנית פקודת break. פגשנו כבר פקודה זאת בהקשר של פקודת switch, בהקשר הנוכחי יש לה תפקיד שונה אך דומה: פקודת break גורמת לעצירת ביצוע הלולאה *הפנימית ביותר* ולהעברת הביצוע לַפקודה שמייד אחרי

הלולאה הפנימית ביותר. במקרה שלנו, עת נגלה ש- num פריק נעשה שני דברים: ראשית נציין זאת לעצמנו במשתנה prime, ושנית נשבור את ביצוע הלולאה, מעשה שיפנה אותנו מיידית אל פקודת ה- cout המופיעה מייד אחרי הלולאה.

שימו לב גם לפלט הנשלח כאן באמצעות אופרטור ה- ? : אם ערכו של המשתנה prime הוא true אזי אופרטור ה- ? מחזיר את המחרוזת שמשמאל לנקודותיים ("prime\n"), ואחרת הוא מחזיר את המחרוזת שמימין לנקודותיים ("not prime\n"), והמחרוזת המוחזרת ע"י אופרטור ה- ? היא זו המודפסת ע"י פקודת ה- cout.

נדגיש כי break מוציא אותנו מהלולאה הפנימית ביותר (או מפקודת switch). נבחן עתה תכנית הקוראת סדרת מספרים, ולכל אחד מציגה האם הוא ראשוני או פריק:

```
cin >> num ;
while (num != 0)
{
   prime = true ;
   divider = 2 ;
   while( divider <= sqrt(num))
   {
      if (num % divider == 0)
      {
         prime = false ;
         break ;
      }
      divider++ ;
   }
   if (prime)
      cout << "prime\n" ;
   else
      cout << "not prime\n" ;
   cin >> num ;
}
```

בתכנית זאת תגרום פקודת ה- break ליציאה מהלולאה הבודקת האם המספר ראשוני. עתה יוצג הפלט המתאים, יקרא ערך חדש לתוך num, ונחזור (או לא, תלוי בערכו החדש של num) לסיבוב נוסף בלולאה החיצונית. כלומר פקודת ה- break 'זרקה' אותנו אל מחוץ ללולאה הפנימית ביותר, אך לא אל מחוץ לשתי הלולאות.

עתה, כשאנו שולטים ברזי פקודת ה- break אנו יכולים לכתוב את התכנית הבודקת האם מספר ראשוני בדרך מעט שונה, בלי להשתמש במשתנה הבולאני prime:

```
cin >> num ;

divider = 2 ;
while (divider <= sqrt(num)){
   if (num % divider == 0)
      break ;
   divider++ ;
}
if (divider <= sqrt(num))
```

```
   cout << "not prime\n" ;
else cout << "prime" ;
```

נסביר: במידה ואנו מגלים כי num פריק אנו שוברים את ביצוע הלולאה בטרם עת, *טרם* ש- divider חלף על-פני הערכים: 2, 3, … int(sqrt(num)), ופונים אל הבדיקה שאחרי הלולאה. ערכו של divider יהי על-כן קטן או שווה משורשו של num. לכן אם divider <= sqrt(num) אות הוא וסימן ש- num פריק, ואנו יכולים להודיע על-כך. מנגד, אם num ראשוני, אזי divider ימצה את הלולאה, יחלוף על-פני כל הערכים 2, 3, … int(sqrt(num)) תוך שאיננו מבצעים כל break; בסופו של דבר, עת divider יהיה גדול מ- sqrt(num), כבר לא נכנס יותר ללולאה, ונגיע לתנאי שמתחתה. בתנאי עת נשאל האם divider <= sqrt(num) התשובה תהיה לא, והפלט יהיה בהתאם.

אגב, שימו לב שאת הפקודה:

```
if (divider <= sqrt(num))
   cout << "not prime\n" ;
else cout << "prime" ;
```

אפשר לקצר תוך שמוש באופרטור סימן השאלה, שכן בכל תנאי אנו מבצעים כאן פקודת פלט, השאלה רק מה תהיה המחרוזת שנפלוט. על כן, את שלוש השורות שמעל נוכל להמיר בפקודה:

```
cout << ((divider <= sqrt(num)) ?
          "not prime\n" : "prime\n" ) ;
```

אופרטור ? בודק האם divider <= sqrt(num) . אם כן הוא (האופרטור) מחזיר את המחרוזת "not prime\n", והיא שמוצגת ע"י פקודת ה- cout, ואם לא הוא (האופרטור) מחזיר את המחרוזת "prime\n", והיא שנשלחת לפלט.

4.1.7 <u>הדפסת מחלקיו הראשוניים של מספר</u>

הפעם אנו רוצים לכתוב תכנית אשר קוראת מספר ומדפיסה את מחלקיו הראשוניים. תכנית זאת תדגים לנו באופן מלא את השימוש בלולאה כפולה, הנקראת גם לולאה מקוננת (מלשון קן): לולאה בתוך לולאה. הרעיון הוא שכמו קודם נעבור עם divider על המספרים הקטנים או שווים מ- num/2. עת גילינו כי ערך כלשהו של divider מחלק את num לא נחַפז להציגו, אלא ראשית נבדוק האם הוא ראשוני, ורק אם הוא יעמוד במבחן הראשוניות נציגו. כדי שהתכנית שלנו תהיה יחסית פשוטה, ומובנת גם למי שרזי התכנית האחרונה שכתבנו לא היו די נהירים לו, נכתוב את התכנית תוך שימוש במשתנה בולאני (וללא שימוש ב- break); אם כי ניתן, כמובן, לכתוב את התכנית גם ללא שימוש במשתנה בולאני (ועם break).

```
/*********************************************************
 *
 *      Prime dividers of a number
 *      ==========================
 *   Writen by: Yosi Cohen, ID: 333444555, Group: 17
 *
 * This program reads from the user a natural number and
 * prints its prime factors.
 * Algorithm: Run a loop that examines 2, 3, …number/2
 * ========== if a certain value, div, divides the
 * number examine if div is prime by checking if exists
 * a number in the range 2,…,sqrt(div) that divides div.
 *
```

```
 * Assumption: the input is a natural number.
 *
 ***************************************************/

#include <iostream>
#include <cmath>

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

int main()
{
   int num = 0,                    // the input number
       div = 2;          // examine if div divides num

   while (num < 2)
   {
      cout >> "Enter a natural number: " ;
      cin >>  num ;
   }
   cout << "The prime factors of " << num << "are:\n" ;

   while(div <= num/2)      //go over candidate factors
   {
      if (num % div == 0)
      {
         int div_div = 2 ;   // exmn if div_div dvdes div
         bool prime = true ;              // div is prime
         while(div_div <= sqrt(div) & prime)
         {                        // check if div is prime
            if(div % div_div == 0)
               prime = false ;
             div_div++ ;
         }
         if (prime)
            cout << div << " " ;
      }
      div++ ;
   }
   cout << endl ;
   return(0) ;
}
```

הסבר: אנו קוראים מהמשתמש את הקלט עד אשר הוא מזין מספר טבעי כנדרש (לשם כך אנו גם מאתחלים את num לכדי ערך אפס, כדי להבטיח שנכנס ללולאה לכל הפחות פעם יחידה). עתה אנו עוברים עם div על כל המספרים שבין 2 למחציתו של num, עבור כל מספר שכזה, אם הוא מחלק את num אנו בודקים האם הוא ראשוני. שימו לב כי המשתנים prime ו- div_div מוגדרים בתוך הבלוק של ה- if, וכמובן שרק בתוך גוש זה נעשה בהם שימוש.

4.1.8 ניחוש מספר שהמחשב הגריל, ומושג הקבוע (const)

עתה ברצוננו לכתוב תכנית בה המשתמש יצטרך לנחש מספר שהמחשב הגריל. כדי לכתוב את התכנית, ראשית, עלינו ללמוד כיצד תכנית מחשב יכולה להגריל מספר, במילים אחרות לייצר ערך אקראי.

יצור ערך אקראי

כדי שתכנית שאנו כותבים תוכל לייצר ערך אקראי, או סדרה של מספרים אקראיים, עלינו לבצע את הפעולות הבאות:

א. לפני הגרלת הערכים האקראיים עלינו לאתחל את מנגנון יצירת המספרים האקראיים. בשפה של דמויים היינו אומרים שעלינו 'להדליק' את מכונת יצור המספרים האקראיים. האתחול מבוצע על-ידי שאנו כותבים את הפקודה: `srand()`. פקודת ה- `srand()` מקבלת כפרמטר מספר טבעי (שלם אי שלילי) הנקרא **הזרע** (seed) של סדרת המספרים האקראיים. הזרע ש- `srand()` מקבלת קובע את סדרת המספרים האקראיים שהמחשב ייצר. במילים אחרות, אם בשתי הרצות שונות של התכנית נעביר ל- `srand()` את אותו ערך (את אותו זרע), ייצר המחשב בשתי ההרצות אותה סדרה של מספרים אקראיים; ולהפך: אם נעביר שני זרעים שונים, אזי תצמחנה מהן שתי סדרות שונות של מספרים אקראיים. שיטה מקובלת היא להעביר כזרע את ערכו של שעון המחשב עת התכנית מורצת; באופן זה אנו מבטיחים כי בכל הרצה תתקבל סדרה ייחודית של מספרים אקראיים.

ב. אחרי ש'הפעלנו' את מנגנון יצירת המספרים האקראיים אנו יכולים לקבל ערך אקראי בתחום `0..x` (עבור מספר טבעי `x`), על-ידי הפקודה: `rand() %(x +1)`. הסבר: `rand()` מחזירה לנו מספר אקראי שהיא בחרה. אם ניקח את שארית החלוקה של המספר שהוגרל ב- `x +1`, נקבל ערך בתחום שבין אפס ל- `x`.

ג. כדי שנוכל להשתמש ב- `srand()` וב- `rand()` עלינו לכלול בתכניתנו את ההוראה: `#include <cstdlib>`. כדי שנוכל לבדוק מה מורה שעון המחשב עלינו לכלול את ההוראה: `#include <ctime>` שתי הוראות אלה תכתבנה בראש התכנית, לצד הוראות ה- `include` האחרות.

ד. כדי לבדוק מה ערך שעון המחשב נשתמש בפקודה פקודה `time(NULL)`. במערכות יוניקסיות מחזירה פקודה זאת את מספר השניות שחלפו מאז ה: 1/1/1970, כלומר מספר טבעי שיהיה שונה בכל הרצה והרצה. מספר זה נעביר ל: `srand()`.

נראה לדוגמה תכנית שמדפיסה עשרה ציונים מקריים של עשרה תלמידים (ולשם כך מגרילה עשרה מספרים אקראיים בתחום שבין אפס למאה):

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main()
{
   int i = 0 ;

   srand((unsigned) time(NULL));// 'turn the machine on'

   while (i < 10) {
      std::cout << rand() % 101 << " " ;
                                    //gen a random val
```

```
      i++;
   }
   return 0;
}
```

לפני שאנו פונים לַתכנית אותה ברצוננו לכתוב נרצה להכיר מרכיב חשוב נוסף של השפה אשר משפר את נכונות וקריאות התכניות שאנו כותבים. מרכיב זה נקרא שימוש **בקבועים** (constants). בתכנית שראינו יצרנו עשרה ציונים על-ידי הגרלת עשרה מספרים אקראיים בתחום שבין אפס למאה. מה יקרה אם בעתיד נרצה לשנות את תכניתנו כך שהיא תדפיס עשרים ציונים אקראיים בתחום שבין אפס לעשר? נצטרך לעבור על התכנית ולשנות את המספרים המתאימים במקומות הרצויים (יהיה עלינו לשנות את כל המופעים של 'מאה' ל-'עשר'; אך להקפיד לעשות זאת רק במקומות בהם מאה מציין את מספר התלמידים בכתה, ולא במקומות אחרים בהם, אולי, יש לו משמעות אחרת). מעבר לכך מי שמביט בתכניתנו פוגש את המספרים עשר ומאה ועלול שלא להבין מה מהותם, מדוע דווקא הם שמופיעים בתכנית (הסיבה היא, להזכירכם, שיש לנו עשרה תלמידים, וציון הוא בדרך כלל בתחום 0..100).

נרצה על-כן לשפר את תכניתנו באופן שיקל לתקנה (למשל עת מספר התלמידים יגדל לעשרים, או אם נחליט שציון יכול להיות רק בתחום 0..10), ושתגבר קריאותה. נציג את האופן בו אנו עושים זאת, ואחר נסביר:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

const int NUM_OF_STUD = 10,      // num of stud in class
          MAX_GRADE = 100 ;      // highest possible grade

int main()
{
   int i = 0 ;

   srand((unsigned) time(NULL));

   while (i < NUM_OF_STUD) {
      std::cout << rand() % (MAX_GRADE +1) << " " ;
      i++;
   }
   return 0;
}
```

קבוע הוא למעשה 'משתנה' שלא ניתן לשנות את ערכו, מהערך התחילי שבהכרח הוכנס לו. בתכנית שלפנינו הוגדרו שני קבועים: NUM_OF_STUD, MAX_GRADE. שני הקבועים הם מטיפוס int (המילה int לפני הגדרתם מורה על-כך). הגדרת קבועים זהה לגמרי להגדרת משתנים, פרט לכך שלפני שם הטיפוס אנו מוסיפים את המילה const, וכן אנו חייבים לתת לקבועים ערך תחילי. אפשר לחשוב על הקבוע כמעין שם נרדף שאנו נותנים לערך המספרי, כלומר בכל מקום בו אנו כותבים בתכנית MAX_GRADE מבין המחשב כי למעשה בתכניתנו זהו שם נרדף למספר מאה, וכי אנו מתכוונים למספר זה.

באיזה אופן השימוש בקבועים משפר את התכנית (בהשוואה לתכנית ללא קבועים):

א. אם בעתיד מספר הסטודנטים שלומדים בכיתתנו יגדל מעשרה לעשרים כל מה שיהיה עלינו לעשות זה לשנות את ערכו של הקבוע (ולקמפל שוב את התכנית). לא נצטרך לחטט בקרבי התכנית ולגלות את כל המקומות בהם מופיע עשר, כמציין את מספר התלמידים בכתה.

ב. מי שיקרא עתה את תכניתנו כבר לא יתקל במספרים חסרי משמעות, אלא בקבועים בעלי שמות המעידים על משמעותם.

בשל העובדה ש: `NUM_OF_STUD, MAX_GRADE` הוגדרו כקבועים, ניסיון לשנותם בהמשך התכנית יגרום לשגיאת קומפילציה. כלומר אם אי-שם בהמשך התכנית ננסה לכתוב: `NUM_OF_STUD = 11` יודיע לנו הקומפיילר על שגיאה.

את הקבועים הגדרנו מחוץ לתכנית הראשית (מעל השורה `int main()`) בכך אנו הופכים אותם לקבועים **גלובליים** (global). על משמעות הדבר נדון בהמשך, כעת רק נציין שקבועים מקובל להגדיר במקום בו הגדרנו אותם כאן, כלומר כגלובליים.

מספר הערות נוספות אודות קבועים:

1. שמות הקבועים הופיעו בתכנית באות גדולה. זוהי מוסכמה המסייעת לקריאות. באופן כזה עת כתוב בתכנית x = Y ברור לקורא כי Y הוא קבוע (ולא משתנה אחר). מוסכמה חלופית היא לשיים קבועים בשמות המתחילים בתווים c_, כדוגמת: c_num_of_stud, c_max_grade.
2. אין מניעה כי לשני קבועים שונים, בעלי תפקידים שונים בתכנית, יהיה אותו ערך. לדוגמה: `const int NUM_OF_STUD = 100, MAX_GRADE = 100;` את המחשב הדבר אינו מבלבל: למאה נתנו שני שמות נרדפים שונים, בכל פעם שיופיע אחד מהם הכוונה היא למאה; גם מבחינתנו יש בכך הגיון: כיום, במקרה, גם מספר התלמידים הוא מאה, וגם הציון המקסימלי הוא מאה, אך לכל אחד משני המאה הללו יש משמעות שונה (ובעתיד הם גם עשויים להיות בעלי ערכים שונים).
3. שמו של קבוע, כמו של משתנה, צריך להעיד על *תפקידו*, ולא על ערכו. על-כן הימנעו מלתת לקבוע שם כגון: `const int TWO = 2`.
4. מתאים וגם ראוי להשתמש גם בקבועים בולאניים. לדוגמה, נניח משתנה אשר צריך להחזיק את מינו של התלמיד. סביר כי הגדרת המשתנה תהא: `bool sex`. עתה נניח כי כתבנו `sex = true;`, האם התכוונו לתלמידה אישה או לתלמיד גבר? זה לא ברור. אם לעומת זאת נגדיר: `const bool MALE = false, FEMALE = true;` ואחר נבצע את פקודת ההשמה: `sex = FEMALE;`, אזי ברור לכל בר דעת כי הכוונה לתלמידה ממין נקבה.
5. כלל הזהב למתכנת המתחיל: פרט לאפס ולאחד, שהינם קבועים בעלי תפקיד מיוחד, לא יופיע בתכניתכם *שום* מספר שהוא. כל ערך מספרי אחר יוגדר ראשית כקבוע; כך גם לגבי `true/false`.
6. בַּקובץ `cstdlib` מוגדרים קבועים בהם אתם יכולים לעשות שימוש לשם החזרת ערך בפקודת ה- `return` של התכנית. במידה והתכנית הסתימה אחרי שהיא השלימה את משימתה בהצלחה, החזירו את הקבוע `EXIT_SUCCESS` שערכו הוא אפס. במידה והתכנית הסתימה בלי שהיא השלימה את משימתה בהצלחה החזירו את הקבוע `EXIT_FAILURE` שערכו מוגדר להיות אחד.

נפנה עתה לתכנית אותה התעתדנו לכתוב בה על המשתמש לנחש מספר שהגריל המחשב:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using std::cin ;
using std::cout ;

const int MAX_VAL = 783, // largest possible num to pick
                // maximal num of gusses user is allowed
          MAX_NUM_OF_GUESSES = 21;

int main()
{
   int the_num,    // the num the computer pick by random
       current_guess,               // current user guess
       guesses_counter = 0 ;   // counter of user guesses


   srand((unsigned) time(NULL));
   the_num = rand() % MAX_VAL ;        // pick the number

                                          // guessing loop
   while (guesses_counter < MAX_NUM_OF_GUESSES) {
      cin >> current_guess ;
      if (current_guess == the_num)    // user guesses it
         break ;
      guesses_counter++;
   }
   if (current_guess == the_num)
       cout << "You did it, in " << guesses_counter +1
            << " trials\n" ;
   else cout << "Sorry, you failed. the number is: "
             << the_num << endl ;
   return EXIT_SUCCESS;
}
```

בתכנית השתמשנו בשני קבועים: האחד מציין מה יהיה ערכו המרבי של המספר אותו יגריל המחשב, והשני מורה כמה ניחושים נאפשר למשתמש לבצע.

אנו עושים שימוש בשלושה משתנים: האחד מחזיק את המספר שהמחשב הגריל, השני את הניחוש הנוכחי של המשתמש, והשלישי מונה אשר סופר כמה ניחושים המשתמש כבר הזין.

הלולאה שמריצה התכנית קוראת בכל איטרציה ניחוש חדש מהמשתמש, בודקת האם הניחוש נכון, ואם כן שוברת את ביצוע הלולאה. כמו כן אנו מונים בלולאה את מספר הניחושים שנדרשו למשתמש.

בעקבות היציאה מהלולאה עלינו לברר האם היציאה מהלולאה בוצעה עקב גילוי המספר בידי המשתמש (ואז `current_guess == the_num`), או מכיוון שהוא מיצה את מספר הניסיונות שהיתרנו לו. אנו, כמובן, משגרים הודעה מתאימה לכל אפשרות.

4.2 פקודת ה- for

פקודת ה- for היא פקודת הלולאה השניה ששפת C מעמידה לרשותנו. פקודת ה- for פחות כללית מפקודת ה- while, אך יותר קומפקטית, ובמקרים רבים נוחה יותר לשימוש. אחד האתגרים הניצבים בפני מתכנת מתחיל הוא ללמוד באיזה לולאה לעשות שימוש בכל עת.

נביט בקטע התכנית הבא אשר קורא זוג מספרים שלמים (קטן ואחַר גדול ממנו), ומציג את כל המספרים השלמים ביניהם (כולל זוג המספרים שנקראו):

```
cin >> num1 >> num2 ;
num = num1 ;
while (num <= num2) {
   cout << num << " " ;
   num++ ;
}
```

ולעומתו בקוד השקול לקודם אך דחוס ממנו:

```
cin >> num1 >> num2 ;
for (num = num1; num <= num2; num++)
   cout << num << " " ;
```

בקוד השני השתמשנו בפקודת ה- for, והתוצאה היא ש-'דחסנו' את הקוד וצמצמנו את נפחו משש שורות לכדי שלוש, תוך שבשורה של פקודת ה- for אנו מתארים שלושה היבטים של הלולאה:

א. *איתחולו* של משתנה הבקרה של הלולאה, המשתנה num, מתבצע במרכיב: num = num1.

ב. השיעור בו משתנה ערכו של משתנה הבקרה *בתום כל* איטרציה של הלולאה, מתואר במרכיב: num++.

ג. עד מתי יש לבצע את הלולאה: כל עוד num <= num2.

בפקודת ה- while שלוש הפעולות הללו מתוארות בנפרד, הדבר גורם לכך שמחד גיסא בפקודת ה- while ברור יותר איזה פעולה מתרחשת היכן ובאיזה שלב, ומאידך גיסא שלוש הפעלות, המתארות את התנהלות הלולאה, מפוזרות במקומות שונים, ועלול להיות קשה לאתרן.

מניסיוני אני יודע שמתכנתים מתחילים, עת רואים פקודת for, אינם בטוחים לעיתים איזה פעולה מתבצעת מתי; עצתי היא תרגמו את ה- for חזרה ל-while, ואז הדברים יתבהרו. כיצד יבוצע התרגום? נביט בלולאת for הבאה:

```
for(num = 17; num <= 3879; num += 3)
{ statement1 ;
  statement2 ;
}
```

כדי לתרגמה ללולאת while נבצע את התהליך הבא:

א. נאתחל: num = 17 לפני הלולאה.

ב. כותרת ה- while תכלול את התנאי (num <= 3879) while.

ג. גוף ה- while יהיה זהה לגוף ה- for: { statement1 ; statement2 ; }

ד. לגוף ה- while נוסיף *כפקודה אחרונה* את המרכיב num += 3 (שנלקח מכותרת ה- for).

נקבל את הלולאה:

```
num = 17 ;
while (num <= 3879) {
   statement1 ;
```

```
   statement2 ;
   a = a+3 ;
}
```

כמו לולאת while, גם לולאת for עשויה שלא להתבצע אף לא פעם אחת, ואין בכך כל פגם.

4.2.1 בדיקה האם זוג מספרים טבעיים הם חברים

שני מספרים num1 ו- num2 נקראים חברים אם סכום מחלקי num1 (כולל המספר אחד, אך לא כולל num1 עצמו) שווה ל- num2, וסכום מחלקי num2 שווה ל- num1. לדוגמה המספרים 220 ו- 284 הם חברים, וכן 6 הוא חבר של עצמו (בדקו).
ברצוננו לכתוב תכנית המציגה את כל זוגות החברים בין 1 ל- MAX (MAX יהיה קבוע של התכנית). התכנית תתנהל כלולאה כפולה: עבור כל זוג אפשרי של מספרים (num1 ו- num2) בין 1 ל- MAX, נסכום את מחלקי num1 ו- num2 (לתוך sum1 ו- sum2), ובמידה ו- num1 == sum2 && num2 == sum1 אזי נציג את num1 ו- num2. התכנית:

```
#include …

const int MAX = 1000 ;

int main()
{
   int num1, num2,
       sum1, sum2 ;

   for (num1 = 1; num1 <= MAX; num1++)
   {
      sum1 = 0 ;          // calculate dividers of num1
      for (int i = 1; i <= num1 /2; i++)
         if (num1 % i == 0)
            sum1 += i ;

      for (num2 = 0; num2 <= MAX; num2++)
      {
         sum2 = 0 ;
         for (i=1; i <= num2 /2; i++)
           if (num2 % i == 0)
              sum2 += i ;

         if (num1 == sum2 && num2 == sum1)
             cout << num1 << " " << num2 << endl ;
      }
   }
   return 0 ;
}
```

ראשית נסביר כמה פרטים יותר משניים ששולבו בתכנית זאת:

1. הפקודה sum += i היא כתיב מקוצר של הפקודה: sum = sum +i. באופן דומה במקום לכתוב a = a + (b*17) ניתן לכתוב: a += (b*17). גם שינויים אחרים בערכם של משתנים ניתן לכתוב בצורה דומה: לדוגמה, במקום

לכתוב `a = a /3` נוכל לכתוב `a /= 3`. יש להקפיד לכתוב את סימן הפעולה (כדוגמת + או /) משמאל ובצמוד לסימן ההשמה (=).

2. הכתיבה: `for (int i = …)` משמעותה שבכותרת לולאת ה- `for` אנו מגדירים משתנה בשם `i` (או בכל שם אחר שנחפוץ). כתיבה זאת לגיטימית בשפת `C++`, אך לא בשפת `C` (כפי שבשפת `C` ניתן להגדיר משתנים רק תחילת גוש, בעוד ב- `C++` ניתן להגדירם בכל מקום בתכנית. כלומר ב- `C` הלולאה הנ"ל לא מתקמפלת). בעיקרון, המשתנה המוגדר (`i` בדוגמה מעל) אמור להיות מוכר רק בגוף הלולאה, וכך גם המצב ב- `g++`. לצערי, ב- `Visual Studio` הגדרת המשתנה באופן הנ"ל שקולה לכך שנגדיר אותו *מייד לפני* ה- `for`; לפיכך המשתנה מוכר באותו גוש בו מופיע ה- `for` (ולא בתוך הגוש שמתבצע בכל איטרציה של לולאת ה- `for`!) כולל אחרי ה- `for`. בתכנית שלנו, לדוגמה, המשתנה `i` מוכר בגוש של ה- `for` החיצוני (זה שמריץ את `num1`), לפיכך אין לנו צורך להגדיר גם בפקודת ה- `for` השניה שמריצה את `i` את המשתנה שוב. אגב, לאינדקס של לולאה, אשר מונה את מספר הפעמים בה אנו מתגלגלים בלולאה, מקובל לקרוא בשם `i` (קיצור של `index`), אולם זה בגדר יוצא מן הכלל המעיד על הכלל מבחינת שיום משתנים, ראו הוזהרתם!

עתה נדון במהותה של התכנית: אנו מריצים בלולאה החיצונית את `num1` מאחד ועד `MAX`. עבור כל ערך של `num1` אנו מבצעים את הפעולות הבאות:

א. אנו סוכמים את מחלקי `num1` לתוך `sum1`.

ב. אחַר אנו רצים בלולאה (עם `num2`) על כל המספרים שבין אחד ל- `MAX`, ועבור כל אחד ואחד סוכמים את מחלקיו (לתוך `sum2`). ובמידה ומתברר ש- `num1` ו- `num2` חברים אנו מציגים אותם.

מה מורכבות זמן הריצה של התכנית?

1. בתכנית אנו מריצים לולאה חיצונית שרצה `MAX` פעמים. עבור כל ערך של `num1` אנו סוכמים את מחלקיו, דבר שמחייב לבצע לולאה שרצה עד מחציתו של `num1`. כמה עבודה אנו עושים בלולאה זאת בכל `MAX` הפעמים שהיא מורצת גם יחד? נניח שהלולאה הייתה מורצת עד `num1` בכל פעם (ולא עד מחצית `num1` כפי שהיא מורצת בפועל): אזי בריצתה הראשונה (עת `num1` שווה אחד) הייתה מתבצעת בה איטרציה יחידה, בריצתה השניה (עת `num1` שווה שתיים) היו מתבצעות בה שתי איטרציות, וכך הלאה, עד אשר בריצתה ה- `MAX`-ית היו מתבצעות בה `MAX` איטרציות. כלומר סה"כ בלולאת איתור סכום המחלקים היו מתבצעות: `1+2+…+MAX` איטרציות. נסכום את סכום הטור החשבוני המתקבל על פי הנוסחה `s = (a1 + an)*n/2` במקרה שלנו: `a1 = 1, an = MAX, n = MAX`, ולפיכך נקבל: `(1 +MAX)*MAX/2 = 0.5*MAX+0.5*MAX2` שהם סדר גודל של MAX^2 שכן כפל בקבוע אינו משנה לנו, ולכן ניתן להתעלם ממנו, וככל ש- `MAX` ילך ויגדל ערכו של `MAX` זניח יחסית ל- MAX^2 ולכן גם מהמחובר השמאלי בסכום ניתן להתעלם. נסכם את ממצאנו עד כה: לו לולאת איתור סכום המחלקים הייתה מתבצעת עד `num1` (ולא עד מחצית `num1`) אזי איתור כל סכומי המחלקים של `1,…,MAX` היה מחייב ביצוע של MAX^2 איטרציות של הלולאה המתאימה. בכל איטרציה בלולאה מתבצעת עבודה שהינה בשיעור מספר קבוע של פעולות, ולכן אנו יכולים להתעלם מכמות הפעולות המבוצעות בגוף הלולאה (ולהישאר עם המניה של מספר המעברים המתבצעים בלולאה). כמו כן מכיוון שאנו רצים רק עד `num1/2` (ולא עד `num1` עצמו) אנו עושים רק מחצית העבודה שמנינו, אולם גם גורם זה הינו חסר משמעות עת אנו דנים במורכבות זמן ריצה. נסכם אם כן שחישוב סכום המחלקים של `1,…,MAX` מחייב עבודה בשיעור MAX^2.

2. עבור כל ערך של num1 אנו מריצים את num2 בלולאה מאחד ועד MAX. בכל איטרציה של לולאה זאת (עבור ערך נתון של num2) אנו סוכמים את מחלקי num2 ואחר בודקים האם המספרים חברים. נעריך את כמות העבודה הנעשית: כפי שראינו בסעיף הקודם, חישוב סכום המחלקים של 1,…,MAX (עת num2 רץ מאחד ועד MAX עבור ערך קבוע של num1) מחייב עבודה בשיעור MAX^2. על עבודה זאת (חישוב סכום המחלקים של num2 עת num2 רץ מאחד עד MAX) אנו חוזרים MAX פעמים (עבור כל ערך של num1). לכן כמות העבודה הנעשית בחישוב סכום מחלקי num2 עבור כל MAX הערכים עליהם עובר num1 דורשת עבודה בשיעור $MAX^3 = MAX^2*MAX$.

בסיכומו של דבר כמות עבודה הנעשית בתכנית שלנו היא $MAX^3 + MAX^2$. כמו קודם, כאשר MAX הולך וגדל תרומתו של המחובר השמאלי הופכת להיות זניחה, ועל-כן ניתן להתעלם ממנה ולומר שמורכבות זמן הריצה של התכנית שכתבנו היא MAX^3.

האם יש ביכולתנו לכתוב תכנית יעילה יותר, אשר מורכבות זמן הריצה שלה תהא נמוכה יותר? התשובה היא כן. נסביר מדוע: אחרי שחישבנו ומצאנו שסכום מחלקי num1 הוא sum1 אין צורך לעבור עם num2 על כל המספרים מאחד ועד MAX כפי שעשינו בתכנית הקודמת. המספר היחיד שיש בכלל סיכוי שיהיה חבר של num1 הוא sum1 שכן תנאי הכרחי (אך לא מספיק) לכך שמספר כלשהו יהיה חבר של num1 הוא שערכו צריך להיות שווה לסכום מחלקי num1 שהוא בדיוק הערך השמור ב- sum1. על-כן אחרי שחישבנו את sum1 'נלהק' אותו בתפקיד num2, נסכום את סכום מחלקיו ואם יתברר שסכום מחלקיו שווה ל- num1 אזי נוכל להכריז על num1 ועל num2 (או sum1) כעל חברים. נציג את התכנית ואחר נדון בה (אנו מניחים כי המשתנים הדרושים הוגדרו כמו בתכנית הקודמת):

```
for (num1 = 1; num1 <= MAX; num1++)
{
   sum1 = 0 ;
   for (int i = 1; i<= num1 /2; i++)
      if (num1 % i == 0)
         sum1 += i ;

   num2 = sum1 ;

   sum2 = 0 ;
   for (i = 1; i<= num2 /2; i++)
      if (num2 % i == 0)
         sum2 += i ;

   if (sum2 == num1)
      cout << num1 << " " num2 << endl ;
}
```

נסביר את התכנית: אחרי שחישבנו את סכום מחלקי num1 לתוך sum1, אנו קובעים את num2, המועמד היחיד שעשוי להיות חברו של num1, להיות sum1. עתה אנו מחשבים את סכום מחלקי num2, ואם איתרע מזלנו ו- sum2 == num1 אזי איתרנו זוג חברים ואנו מציגים אותם.

מה מורכבות זמן הריצה של תכנית זאת? עבור כל ערך של num1 אנו מחשבים סכום מחלקיו לתוך sum1, ואחר את סכום מחלקי sum1, כלומר פעמיים אנו מחשבים את סכום המחלקים. כפי שראינו עלות חישוב כל סכום המחלקים של 1,…,MAX היא MAX^2. גם אם אנו חוזרים על חישוב המחלקים פעמיים (הן עבור num1 והן עבור sum1) עדיין מורכבות זמן הריצה נותרת בסדר גודל של MAX^2.

עוד הערה אחרונה לפני שנפרד מתכנית החברים: שתי התכניות שכתבנו תצגנה כל זוג חברים פעמיים (למשל עבור 220 ו- 284 פעם אחת הם יוצגו עת num1 יהיה 220, ו- num2 יהיה 284, ופעם אחת עת num1 יהיה 284 ו- num2 יהיה 220). כדי למנוע כפילות זאת ניתן להציג את הזוג רק אם ערכו של num2 גדול או שווה מערכו של num1. אני מזמין אתכם להוסיף את השיפור לתכניות שכתבנו.

4.2.2 הערות נוספות אודות פקודת ה- for

1. ראינו כי כותרת פקודת ה- for מכילה שלושה מרכיבים: ערכו התחילי של משתנה הבקרה של הלולאה, עד מתי יש להריץ את הלולאה, באיזה שיעור יש לשנות את ערכו של משתנה הבקרה בתום כל איטרציה. מותר, וגם נהוג, להשמיט חלק מהמרכיבים במידה ואין בהם צורך. נציג מספר דוגמות:

```
cin >> num1 >> num2 ;
for ( ; num1 <= num2; num1++)
   cout << num1 << " " ;
```

בדוגמה זאת לא היה צורך באתחול ערכו של משתנה הבקרה, (הוא אותחל, למעשה, בפקודת הקלט), ועל-כן מרכיב זה נשאר ריק.

בדוגמה הבאה אנו קוראים מספר, ומגדילים אותו שוב ושוב בשיעור המחלק הגדול ביותר שלו (שיהיה במקרה 'הגרוע' ביותר אחד), וזאת עד שערכו של המספר גדול מ- MAX (שהינו קבוע שאנו מניחים שהוגדר בתכנית):

```
cin >> num ;
for (  ; num <= MAX;  )
{
   for (divider = num1 /2 ; ; divider--)
      if (num % divider == 0)
         break ;
   num1 += divider ;
}
```

בדוגמה זאת בלולאה החיצונית לא נזקקנו לאתחל את ערכו של num, וכן לא נזקקנו להגדילו בתום כל סיבוב בלולאה (שכן הגדלתו נעשית במקום אחר בתכנית). בלולאה הפנימית, לעומת זאת, לא נזקקנו לתנאי סיום של הלולאה, שכן אנו יוצאים מהלולאה (הפנימית בלבד, כמובן) באמצעות פקודת break. אעיר כי מבחינת סגנון תכנותי, לכל הפחות את הלולאה החיצונית, סביר שהיה מתאים יותר לכתוב כלולאת while ולא כלולאת for, שהרי למעשה כל שנותר בכותרת הלולאה הוא תנאי סיום, וזה בדיוק מה שמופיע בלולאת while.

2. לעיתים אתם מעוניינים לכלול במרכיב האתחול או במרכיב השינוי שבכותרת הלולאה יותר מאשר משתנה יחיד. ניתן לבצע זאת אם מפרידים בין הפעולות השונות בפסיק:

```
for (a = 0, b = 17; a < b; a++, b--)
   cout << a << " " << b << endl ;
```

בדוגמה הנוכחית אתחלנו הן את ערכו של a והן את ערכו של b, וכן בתום כל סיבוב בלולאה הגדלנו את a באחד, והקטנו את b באחד.

3. מכיוון שכותרת לולאת ה- for כוללת בחובה כה הרבה מרכיבים אזי קורה לעיתים שבגוף הלולאה כבר אין צורך לבצע דבר, וגם זה לגיטימי. במקרה כזה תכללו בגוף הלולאה את הפקודה הריקה אשר כוללת נקודה-פסיק בלבד (;).
נראה דוגמה: קטע תכנית הסופר כמה ספרות נדרשות לייצוג מספר טבעי חיובי ממש (לדוגמה: 3879 הוא מספר המיוצג על-ידי ארבע ספרות): רעיון הפתרון: כל פעם נמחק מהמספר ספרה נוספת (על-ידי שנחלקו בעשר), ומנגד נגדיל את מונה הספרות במספר באחד.
הקוד:

```
cin >> num ;
for (temp_num = num, digit_counter = 0;

     temp_num != 0
     temp_num /= 10; digit_counter++)
                ;
cout << num << digit_counter;
```

הסבר: אנו מאתחלים את משנה העזר temp_num ממנו כל פעם נסיר ספרה כך שהוא יכיל את ערכו של num, ואת מונה הספרות אנו מאפסים. בלולאה נתגלגל עד אשר נסיר את כל הספרות מ- temp_num, כלומר הוא יתאפס. בכל סיבוב בלולאה נבצע את הפקודה הריקה, ולאחריה, במרכיב שינוי הערך המופיע בכותרת הלולאה נחלק את temp_num בעשר, ובכך נמחק ממנו ספרה נוספת, וכן נגדיל את מונה הספרות באחד. בתום הלולאה נציג את המספר שקראנו ואת מונה הספרות.

דוגמה אחרת: נניח לדוגמה כי ברצונכם לכתוב לולאה אשר מחשבת את המנה num1/num2 (לתוך המשתנה quot) וזאת בלי להשתמש באופרטור החילוק:

```
for (quot = 0, temp = num1;
     temp <= num2 ;
     quot++, temp += num1 )
          ;
```

אתם, ראשית, מוזמנים לבדוק את נכונות הקוד. הנקודה אותה רציתי להדגים כאן היא שגוף הלולאה כולל את הפקודה הריקה ותו לא, וזאת מכיוון שכל יתר הפעילות הדרושה להשלמת המשימה מבוצעת בכותרת הלולאה.

4. מתכנתים מתחילים מתקשים לעיתים להחליט באיזה לולאה מתאים להשתמש באיזה מצב. התשובה לכך לא תמיד חד משמעית, והיא גם תלויית סגנון תכנותי, ובכל אופן אציע מספר כללי אצבע מנחים:
   א. השימוש המובהק ביותר בלולאת for הוא עת יש משתנה בקרה ברור, על-פיו מתנהלת הלולאה, ערכו של המשתנה משתנה בתום כל סיבוב בלולאה באופן אחיד, וגם תנאי סיום הלולאה הוא על-פי אותו משתנה.
   ב. אם ערכו של משתנה הבקרה של הלולאה משתנה לא באופן קבוע ואחיד בכל איטרציה ואיטרציה (למשל הוא נקרא מהמשתמש, או מתעדכן על סמך חישוב כלשהו), אזי יתכן כי מתאים יותר שלא להשתמש בלולאת for.

4.3 פקודת ה- do-while

פקודת ה- do-while היא פקודת הלולאה הפחות שימושית. היא מתאימה למצבים בהם אתם יודעים שתדרשו להיכנס לַלולאה לפחות פעם אחת, ובמידת הצורך יותר מכך. אני מוצא אותה שימושית בעיקר עת עליכם לקרוא קלט מהמשתמש: במידה והקלט יהיה כמצופה די יהיה לקראו פעם יחידה, אחרת יהיה צורך להתגלגל בלולאה עד שמשתמש יאות להזין את שהוא מונחה להזין. נראה דוגמה:

```
do
{
   cout << "Enter a number in the range 1..” << MAX ;
   cin >> num ;
}
while (num < 1 || num > MAX) ;
```

נסביר: עת המחשב מגיע לביצוע הלולאה בפעם הראשונה, הוא 'גולש' לתוכה בלי לבדוק כל תנאי, הוא מבצע את שתי הפקודות הכלולות בגוף הלולאה, ורק עתה הוא ניגש לבדיקת התנאי: התנאי מורה לו שעליו לבצע את הלולאה כל עוד הוזן ערך קטן מאחד או גדול מ- MAX. במידה והתנאי מתקיים יחזור המחשב לסיבוב נוסף בלולאה.

יכולנו לנסח את הלולאה גם באופן הבא:

```
num = 0 ;
while (num < 1 || num > MAX)
{
   cout << "Enter a number in the range 1..” << MAX ;
   cin >> num ;
}
```

בנוסח זה היה עלינו לאתחל את num לערך שיבטיח שהמחשב יכנס לַלולאה לפחות פעם יחידה. בפקודת ה- do-while נחסך מאתנו הצורך באתחול זה.
יכולנו לנסח את הלולאה גם באופן:

```
for (num = 0; num < 1 || num > MAX ; )
{
   cout << "Enter a number in the range 1..” << MAX ;
   cin >> num ;
}
```

אני מוצא צורת כתיבה זאת כ- abuse של פקודת ה- for.

שימו לב כי בניגוד להרגלנו בפקודת do-while מופיעה נקודה-פסיק אחרי תנאי הלולאה.

4.4 continue

פקודת ה- continue היא 'בת זוגה הנזנחת' של פקודת ה- break באשר לפחות חלקית היא דומה לאחרונה, אך הרבה פחות שימושית ממנה.

לעיתים קורה שתוך כדי ביצוע איטרציה בלולאה אתם מגיעים למסקנה כי את המשך גוף הלולאה אין ברצונכם לבצע, אולם גם איננכם מעונינים לסיים את ביצוע הלולאה, כל שאתם רוצים הוא לפנות לסיבוב חדש בלולאה. במצבים כאלה באה לעזרתכם פקודת ה- continue. נדגים ואחר נסביר:

```
num = 1;
while (num != 0) {
   cin >> num ;
   if (num <= 0)
      contuinue ;
   cout << num << "<-->" << sqrt(num) ;
   for (int div = 1; div <= num/2; div++)
        if (num % div == 0)
             cout << div ;
   cout << "Another one please: " ;
}
```

בקטע התכנית הנ"ל אנו קוראים סדרת מספרים, עד קריאת אפס, ולכל מספר חיובי מדפיסים את שורשו ואת מחלקיו. במידה וקראנו מספר שלילי אנו לא רוצים להמשיך בביצוע גוף הלולאה, אלא לפנות לסיבוב חדש בלולאה, בו נקרא מספר חדש. אנו משתמשים בפקודת ה- `continue` אשר מקפיצה אותנו מהמיקום הנוכחי אל כותרת הלולאה, שם ייבדק שוב תנאי הלולאה, ובמידה והוא מסופק (במקרה שלנו אם `num != 0`) נכנס לסיבוב נוסף בלולאה. פקודת ה- `continue` חוסכת לנו למעשה פתיחה של גוש חדש. יכולנו לכתוב את קטע הקוד הנ"ל גם באופן הבא:

```
while (num != 0) {
   cin >> num ;
   if (num > 0) {
      cout << num << "<-->" << sqrt(num) ;
      for (int div = 1; div <= num/2; div++)
        if (num % div == 0)
             cout << div ;

      cout << "Another one please: " ;
   }
}
```

במקום להשתמש ב- `continue` ביצענו את כל יתר הפקודות בגוף הלולאה רק במידה והתקיים התנאי ההפוך לתנאי לו הכפפנו את ה- `continue` בקטע התכנית הקודם. המחיר ששילמנו הוא שהפקודות שהיו בקטע הקוד הראשון מתחת ל- `continue` נארזו בקטע הקוד השני בגוש חדש, התוצאה היא ששלמנו עוד רמה באינדנטציה: לדוגמה, הצגת המחלקים נעשית ברמת האינדנטציה השניה בלולאת ה- `do-while`, אך רק ברמת האינדנטציה השלישית בלולאת ה- `while`.

4.5 תרגילים

4.5.1 תרגיל מספר אחד: מציאת שורשים שלמים

עבור מספר טבעי m נגדיר את השורש ה- n-י שלו להיות המספר הטבעי k המקיים ש- k^n קרוב ל- m יותר מכל חזקה n -ית של כל מספר טבעי אחר 'k.

כתבו תכנית הקוראת מספרים שלמים עד שהיא קוראת את המספר אפס. עבור כל מספר טבעי שהתכנית תקרא עליה להציג את תשעת שורשיו הראשונים החל בשורש השני ואילך (השורשים השני, שלישי,..., העשירי).

אין להשתמש בפונקציות מתמטיות שונות (בפרט לא ב- sqrt, ln או כל פונקציה אחרת המוגדרת ב- cmath), יש לאתר כל שורש ושורש על-ידי לולאה מתאימה.

כמו בכל תכנית הקפידו על עימוד, תיעוד, שמות משתנים, פשטות, ושאר ירקות.

4.5.2 תרגיל מספר שתיים: חישוב סדרת שתיים שלוש

כתבו תכנית הקולטת מספר שלם חיובי ומבצעת עליו את הפעולה הבאה שוב ושוב: אם הוא זוגי היא מחלקת אותו בשתיים, אחרת היא מכפילה אותו בשלוש ומוסיפה אחד. על פעולה זו יש לחזור שוב ושוב עד אשר מתקבל המספר 1. בסוף התהליך יש להכריז כמה סיבובים הוא ארך ומה המספר הגבוה ביותר שיוצר במהלכו.

4.5.3 תרגיל מספר שלוש: הצגת מספרים ראשוניים עוקבים

הצגנו תכנית לבדיקת ראשוניות מספר שלם. השתמשו בתכנית זו על-מנת לכתוב תכנית המבקשת מן המשתמש שני מספרים טבעיים: גבול תחתון וגבול עליון לחיפוש. על התכנית להציג את זוג המספרים הראשוניים העוקבים הראשון שנמצא בתחום הנתון או להכריז כי לא נמצא זוג כזה בתחום המבוקש.

הגדרות:

- זוג ראשוניים ייקרא עוקב, אם ההפרש ביניהם הוא 2 בדיוק (לדוג': 3 ו 5, וגם 17 ו- 19).
- גבול חיפוש תחתון חוקי הוא מספר שלם גדול מ 2.
- גבול חיפוש עליון חוקי הוא מספר שלם הגדול לפחות ב 3 מהגבול התחתון הנוכחי.

4.5.4 תרגיל מספר ארבע: חנות השטיחים

בשאלה זו נדמה חנות שטיחים. את השטיחים תארגו במקום, בדגמים שונים, ובגדלים משתנים, לבקשת הלקוח – עד אשר יבקש הלה לעזוב את החנות. או אז תברכוהו לשלום.

דגמי השטיחים השונים, בגדלים לדוגמא:

<u>דגם 1:</u>

גודל 5	גודל 4
xoxox	xoxo
oxoxo	oxox
xoxox	xoxo
oxoxo	oxox
xoxox	

גדלים מותרים: 2-10.

<u>דגם 2:</u>

גודל 5	גודל 4
xxxxx	xxxx
oxxxx	oxxx
ooxxx	ooxx

oooxx	oooxx
oooo	oooox
	ooooo

גדלים מותרים: 2-10.

<u>דגם 3:</u>

גודל 4	גודל 6
xxoo	xxxooo
xxoo	xxxooo
ooxx	xxxooo
ooxx	oooxxx
	oooxxx
	oooxxx

גדלים מותרים: 2,4,6,8,10.

4.5.5 <u>תרגיל מספר חמש: חידת אותיות ומספרים</u>

לפניכם חידת "אותיות ומספרים" מן הגיליון האחרון של אחד מעיתוני הערב.
בפעולת החילוק הוחלפו הספרות באותיות. כל אות מייצגת ספרה שונה.
עליכם לשחזר את פעולת החשבון באמצעות תכנית מתאימה ולמצוא את הספרה שאיננה משתתפת בה.

		ה	ר	י	ם	נ	ס
			י	ס	ף	ז	ע
		ה	ר	ע			
		נ	נ	ע			
		ס	ף				

על מנת להרגילכם לאופן הכתיבה לעיל נציג לדוגמא את התרגיל 2537 : 92 = 27 (53)
בכתיב זה:

		2	5	3	7	9	2
		1	8	4		2	7
			6	9	7		
			6	4	4		
				5	3		

הנחיות לפתרון: מומלץ להגדיר משתנה עבור כל אות בחידה. כעת עליכם לקנן לולאות כך שכל ההשמות האפשריות לתשע האותיות ייבדקו. עבור כל השמה יש לבדוק האם תרגיל החילוק הארוך שמתקבל הוא המבוקש (וכן שכל האותיות שונות זו מזו!).
שימו לב: ניתן לייעל את הפתרון בדרכים רבות ושונות. אפשר למשל לחשוב מעט בעצמכם, ולהגיע למסקנות מקלות (למשל שהאות ג יכולה להיות רק ספרה בין 5 ל 9). צרופים מסוימים ניתנים לפסילה עוד בטרם השמתם את כל האותיות (למשל כבר ההשמה החלקית ב=1 ו א=3 לא מסתדרת עם החידה. אין צורך לכן לבדוק אף אחת מההרחבות שלה). וכו' וגו'

4.5.6 תרגיל מספר שש: הצגת סדרת מספרים בשורות

כתבו תכנית הקוראת סדרת מספרים המסתיימת בַּמספר בו היא מתחילה. על התכנית להדפיס את סדרת המספרים בשורות, כאשר מעבר שורה מתבצע עם שינוי סדר המספרים מסדרה עולה ליורדת ולהפך. איבר המתאים לשני קטעי הסדרה יודפס בכל אחת משתי השורות שאליהן ניתן לשייכו. אחרי קריאת הנתונים יש להציג גם סיכום שיציין כמה שורות עולות, וכמה שורות יורדות הודפסו.

הערות: (א) ניתן להניח כי אותו מספר לא יופיע פעמיים רצוף בקלט. (ב) ניתן להניח כי הקלט תקין, בפרט שהוא מסתיים באותו נתון בו הוא החל.

לדוגמה: עבור הקלט הבא (משמאל לימין): 3 1 3 4 5 6 -3 -1 2 4 8 6 3 5
יוצג הפלט:

```
3 5 6 8
8 4 2 –1 –3
-3 6
6 5 4 1
1 3

summary:
=======
There are 3 ascending lines.
There are 2 descending lines.
```

4.5.7 תרגיל מספר שבע: איתור מספר שהתכנית הגרילה

כתבו תכנית אשר פועלת באופן הבא:

א. התכנית קוראת מהמשתמש ערך טבעי X.

ב. התכנית מגרילה ערך שלם N בתחום 0..X.

ג. התכנית מזמינה את המשתמש לנחש את N. המשתמש מזין שוב ושוב את הניחוש שלו, ומקבל מהתכנית היזון חוזר האם המספר שהוא ניחש קטן או גדול מ- N.

ד. כאשר המשתמש הצליח לנחש את N תודיע לו התכנית בכמה צעדים הוא עשה זאת, וכמה צעדים לכל היותר היו נדרשים למשתמש רציונלי אך חסר מזל. (אתם מוזמנים לחשוב בעצמכם כיצד היה נוהג משתמש רציונלי חסר מזל, וכמה ניחושים היו נדרשים לו).

ה. התכנית תעניק למשתמש ציון שיחושב באופן הבא:

```
grade = (number of guesses a rational unlucky user would need)/
        (number of guesses our user needed)
```

ציונו של המשתמש לא יעלה על מאה (ולכן אם הערך הנ"ל גדול ממאה יקבל המשתמש את הציון מאה).

ו. התכנית שואלת את המשתמש אם ברצונו לחזור על התהליך הנ"ל, ובמידה והוא מעוניין תחזור למתואר בסעיף א'.

4.5.8 תרגיל מספר שמונה: מספר הספרות הנדרשות להצגת מספר

כתבו תכנית הקוראת מהמשתמש מספר טבעי n, ובסיס b. על התכנית לחשב כמה ספרות נדרשות לשם הצגת המספר n בבסיס b. לדוגמה: עבור המספר 17 והבסיס 10 יוצג הפלט 2, עבור המספר 17 והבסיס 2 יוצג הפלט 5, עבור המספר 3879 והבסיס 10 יוצג הפלט 4, עבור 3879 ו- 16 יוצג הפלט 4.

4.5.9 תרגיל מספר תשע: בדיקה האם מספר הינו פלינדרום

פלינדרום הוא מספר שתמונת המראה שלו זהה למספר עצמו. לדוגמה: 131, 1331, 1. כתבו תכנית הקוראת מספר ומודיעה האם הוא פלינדרום.

4.5.10 תרגיל מספר עשר: ייצור פלינדרום ממספר נתון

פלינדרום הוא מספר שתמונת המראה שלו זהה למספר עצמו. לדוגמה: 131, 1331, 1. נציג אלגוריתם לייצור פלינדרומים:

א. אם המספר שבידך הינו פלינדרום אזי הצג אותו וסיים.

ב. הפוך את סדר ספרותיו של המספר שבידך, והוסף את המספר המתקבל למספר שאת סדר ספרותיו הפכת.

ג. חזור לסעיף א'.

כתבו תכנית אשר מגרילה עשרה מספרים אקראיים, ומכל אחד מהם מייצרת פלינדרום. על התכנית להציג גם את תהליך הייצור, כלומר בכל שלב מה היה המספר המקורי, מהו היפוכו, ומה סכומם. כמו כן יש להציג כמה צעדים נדרשו לשם ייצור הפלינדרום.

4.5.11 תרגיל מספר אחת-עשר: חישוב סטטיסטים שונים על סדרת מספרים

כתבו תכנית הקוראת סדרת מספרים ממשיים עד קריאת הערך הקבוע END (שערכו מוצג למשתמש בתחילת התכנית). על התכנית להציג את המספרים בסדרה אשר ערכם שווה לממוצע המספרים שקדמו להם. עבור כל מספר כנ"ל יש להציג גם את מקומו בסדרה. כמו כן בתום תהליך הקריאה יש להציג כמה מספרים הוצגו (שכן הם היו שווים לממוצע הנתונים שהופיעו לפניהם).

4.5.12 תרגיל מספר שתיים-עשרה: חישוב רדיוס מעגל באופן איטרטיבי

כתבו תכנית אשר קוראת מהמשתמש:

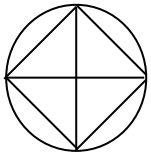
א. רדיוס של מעגל לתוך המשתנה `radius`.

ב. מספר טבעי לתוך המשתנה `figures`.

ג. מספר ממשי לתוך המשתנה `epsilon`.

התכנית תחשב ותציג את שטחם של המצולעים המשוכללים בני שלוש, ארבע, ... `figures` צלעות, החסומים במעגל שרדיוסו הוזן. במידה וההפרש בין שטחיהם של שני מצולעים בני `n` ו- `n+1` צלעות קטן מ- `epsilon` תודיע התכנית על-כך ותעצור (גם היא טרם חישבה את שטחיהם של `figures` מצולעים).

כיוון הפתרון: את שטחו של מצולע בן `n` צלעות החסום במעגל ניתן לחשב על-ידי חלוקת המצולע ל- `n` משולשים שווי שוקיים, שכל אחת משוקיהם אורכה כרדיוס המעגל, וזווית הראש של כל משולש היא בת `360/n` מעלות. לדוגמה, ריבוע החסום במעגל ניתן לחלק לארבעה משולשים באופן הבא:



שטחו של משולש ניתן לחישוב באמצעות הנוסחה: 0.5*a*b*sinus(alpha) עבור a, b המציינים את אורך צלעות המשולש, ו- alpha היא הזווית בין שתי הצלעות a, b.

הפונקציה sin מקבלת פרמטר מטיפוס double המציין זווית ברדיאנים, ומחזירה ערך מטיפוס double המציין את סינוס הזווית. להזכירכם, במעגל שלם יש 2*PI רדיאנים, לכן זווית הראש של המשולשים שווי השוקיים המופיעים בשרטוט הנ"ל היא 2*PI/4. את הקבוע PI עליכם להגדיר בתכניתכם. קבעו את ערכו להיות: 3.141592 . כדי להשתמש בפונקציה sin הוסיפו לתכניתכם את הוראת הקומפיילר: #include <cmath>

4.5.13 תרגיל מספר שלוש-עשרה: בדיקה כמה מספרותיו של מספר א' מופיעות במספר ב'

כתבו תכנית הקוראת זוגות של מספרים טבעיים, עד קליטת הזוג אפס, אפס. עבור כל זוג על התכנית להדפיס כמה מספרותיו של המספר הראשון בזוג מופיעות במספר השני בזוג. לדוגמה: עבור הזוג 123 ו- 135 יודפס 2, עבור הזוג 444 ו- 4 יודפס 3, עבור הזוג 123 ו- 456 יודפס 0.

5 מערכים חד-ממדיים

5.1 מהו מערך

המשתנים שהכרנו עד כה הכילו ערך יחיד בכל נקודת זמן. לעיתים אנו זקוקים למשתנים שיוכלו להכיל כמות רבה של נתונים בו זמנית. לדוגמה, נניח כי ברצוננו לשמור את ציוני ארבעים תלמידי הכתה בתנ"ך. כמו כן, נרצה אפשרות לטפל בכל הציונים בצורה שיטתית ונוחה: לקרוא את כל הציונים, להציגם, לחשב את הממוצע, להציג את מספרו של התלמיד המצטיין, ועוד. כדי שנוכל לבצע את כל המשימות הללו בנוחות אנו זקוקים למשתנה שיוכל להכיל ארבעים ציונים. המערך (array) הוא הכלי שמעמידה לרשותנו שפת C עבור משימות שכאלה.

נניח כי הגדרנו בתכנית משתנה: `int bible[40]`. מה קיבלנו? קיבלנו שורה של ארבעים משתנים, כל-אחד מהם מסוגל לשמור מספר שלם (טיפוסם הוגדר להיות `int`). המשתנים מכונים `bible[0], bible[1], … ,bible[39]` (כלומר מספרו של המשתנה הראשון, במילים אחרות של התא הראשון במערך, הוא אפס, ומספרו של התא האחרון במערך הוא אחד פחות מגודלו של המערך). אדגיש כי במערך יש *ארבעים* תאים שמספריהם הם בהכרח אפס עד 39. זה מאוד מבלבל ולא אינטואיטיבי למתכנתים מתחילים, עבורם טבעי יותר היה למספר את תאי המערך מאחד ועד ארבעים, אולם זה המצב, והוא אינו ניתן לשינוי. עת נלמד על מצביעים (`pointers`) נבין גם מדוע זה המצב, זו אינה קביעה שרירותית של מגדירי השפה.

בארבעים המשתנים שקיבלנו אנו יכולים לעשות בדיוק אותם שימושים שאנו יכולים לעשות בכל משתנה אחר מטיפוס `int`. על-כן אנו רשאים למשל לכתוב כל אחת מהפקודות הבאות:

```
bible[0] = 17;
cin >> bible[39];
bible[39]++;
cout << bible[0] % num1 ;
```

(בהנחה ש- `num1` הוא מטיפוס `int`).

לא אחת אני נשאל: 'האם ניתן לבצע על תא במערך כך, וכך וכך?' ותשובתי היא: האם על משתנה פרימיטיבי (פשוט, כפי שהכרנו עד כה) מותר לבצע פעולה זאת? אם כן אזי היא מותרת גם על תא במערך. תא בודד במערך אינו יותר ואינו פחות מכל משתנה שהכרנו עד היום.

מצד שני, נדגיש כי אין כל פעולה שניתן לבצע על המערך כשלם. לכן הפעולות הבאות *אינן* חוקיות:

```
cin >> bible;
cout << bible;
bible = 0.
```

אם עלינו לטפל בכל אחד ואחד מתאי המערך בנפרד אזי מה הועילו חכמים בתקנתם? במה מקדם אותנו השימוש במערך? התשובה היא שעל-ידי שימוש במערך ובלולאה ניתן לבצע בנוחות שורה של פעולות רצויות, כפי שנראה מייד. קודם לכן נכיר עוד היבט של שימוש במערך. נניח כי `index` הוא משתנה מטיפוס `int`, אזי אנו יכולים לכתוב:

```
index = 17;
```

ואחר:

```
cin >> bible[index];
```
מה עשינו? קראנו מהמשתמש ערך לתוך תא מספר 17 במערך. אם עתה נבצע:
```
index++ ;
```
ואחר נבצע:
```
bible[index] = 0 ;
```
אזי עתה אנו מאפסים את תא מספר 18 במערך. כלומר אנו יכולים להשתמש במשתנה מטיפוס `int` כאינדקס (מציין) של תאי מערך.

עת אנו משתמשים במשתנה כאינדקס של תאי המערך יש לדאוג שהמשתנה יכיל ערך שהוא מספר של תא הקיים במערך. לדוגמה אם נבצע:
```
index = 40 ;
```
ואחר:
```
bible[index] = 0 ;
```
אזי אנו מנסים לפנות לתא מספר 40 במערך, שעה שמספרי התאים במערך `bible` הם `0..39`, כלומר אין במערך תא מספר 40. במלים פורמאליות יותר: אנו חורגים מחוץ לתחומי המערך. לדאבוננו, בשפת `C` בדרך כלל המחשב לא יתריע עת תחרגו מגבולות המערך, וזאת למרות שאתם עלולים 'לדרוך' על משתנים אחרים בתכניתכם, או על מרכיבים אחרים בזיכרון שיפריעו להמשך עבודתכם. לא יגרם שום נזק למחשב כתוצאה מחריגה מגבולות מערך, אולם כדאי מאוד להיזהר ולהישמר מכך.

5.2 דוגמות ראשונות לשימוש במערך

כיצד אם כן נעשה שימוש במערך. נראה דוגמה ראשונה בה אנו מאפסים את המערך:
נניח כי בתכנית כבר הוגדרו:
```
int bible[40], index ;
```
עתה נוכל לכתוב:
```
for (index = 0; index < 40; index++)
   bible[index] = 0 ;
```
בלולאה זאת עת `index` יהיה אפס נפנה לתא מספר אפס במערך (התא הראשון) ונאפס אותו, עת `index` יהיה אחד נפנה לתא מספר אחד, ואפסו, וכך הלאה, עד אשר `index` יהיה 39 ואז נאפס את התא האחרון במערך. `index` יגדל להיות 40 בתום אותה איטרציה, אולם אז התנאי `index < 40` כבר לא יתקיים, לא נפַנס לסיבוב נוסף בלולאה, וטוב שכך. (בתכנית שלמה ראוי, כמובן, להמיר את המספרים 40 ו- 100 בקבועים).

באופן דומה, קריאת ארבעים ציונים לארבעים תאי המערך, תוך שמירה שלכל תא מוזן ציון סביר (בתחום אפס עד מאה) תיעשה באופן הבא:
```
for (index = 0; index < 40; index ++)
{
   do {
      cin >> bible[index] ;
   } while (bible[index] < 0 || bible[index] > 100) ;
}
```
בלולאה החיצונית אנו קוראים את ארבעים הציונים, בלולאה הפנימית אנו חוזרים וקוראים כל ציון וציון עד אשר הערך המוזן הינו בתחום `0..100`.

חישוב ממוצע הציונים יעשה באופן הבא:
```
sum = 0 ;
for (index = 0; index < 40; index ++)
```

```
   sum += bible[index] ;
cout << sum/40 ;
```
לתוך המשתנה sum אנו סוכמים הציונים במערך, לבסוף כדי להציג את הממוצע אנו מדפיסים את ערכו של המשתנה sum חלקי ארבעים הציונים שהוא מכיל. (הממוצע יתקבל, כמובן, כמספר שלם, נניח שלם הפשטות שדי לנו בכך.)

בעבר הזכרנו כי פרט לאפס ואחד לא יופיעו בתכנית מספרים אחרים. הדבר נכון גם לגבי גודלם של מערכים; במיוחד שגודלו של מערך עשוי להשתנות. בדוגמה שלנו אם גודל הכתה ישתנה יהיה עלינו לשנות את גודלו של מערך הציונים בתנ"ך. על-כן גודלו של מערך ייקבע תמיד על פי ערכו של קבוע שהוגדר (באזור הגדרת הקבועים, מתחת לאזור ה- include). מעתה ואילך נניח שגודלו של מערך ציוני התלמידים בתנ"ך הוגדר באופן הבא: ראשית הוגדר הקבוע

```
const int CLASS_SIZE = 40 ;
```

ושנית בתכנית הראשית הוגדר המשתנה:

```
int bible[CLASS_SIZE] ;
```

תלמידים רוצים לעיתים לכתוב קוד כגון הבא:

```
int size ;
cout << "Enter class size : " ;
cin >> size ;
int bible[size] ;
```

כלומר התלמידים רוצים לקבוע את גודלו של המערך bible על-פי ערך שיקרא מהמשתמש (לתוך המשתנה size). אנו אומרים שגודלו של המערך נקבע *דינאמית* בזמן ריצת התכנית, על סמך קלט שהזין המשתמש, ולא *סטאטית* בזמן כתיבת התכנית, על סמך ערך קבוע שהגדיר המתכנת. דא עקא (כלומר, למרבה הצער) בשפת C הדבר אינו חוקי! לכן תכנית כגון האחרונה לא תתקמפל בהצלחה. בהמשך נלמד לעבוד עם משתנים מסוג מצביעים, ועימם נוכל לממש רצון כגון זה שביטאנו כאן. אעיר כי ישנם מהדרים מתוחכמים אשר מתירים כתיבת קוד מסוג זה. הם עושים זאת על-ידי כך שהם, בלי ידיעתכם, ממירים את הגדרת המערך בהגדרה של מצביע. למרות שיש מהדרים שמתירים לכתוב באופן הנ"ל, אנו לא נכתוב קוד כנ"ל, בו גודלו של מערך נקבע על-פי ערכו של משתנה. אנו נקפיד להגדיר מערכים רק באמצעות קבועים.

שאלה נוספת שעולה לעיתים מצד תלמידים היא האם ניתן להגדיר מערך שיהיו בו נתונים מטיפוסים שונים (כלומר חלק מהנתונים שהמערך יחזיק יהיו, למשל, מטיפוס int, וחלק יהיו מטיפוס double)? התשובה היא לא. כל הנתונים במערך יהיו תמיד מאותו טיפוס.

5.3 מציאת האיבר המרבי במערך, והיכן הוא מופיע

עתה נרצה לכתוב תכנית אשר מציגה את מספרי תלמידים בכתה (ליתר דיוק מספרי התאים שלהם במערך, החל בערך אפס), כך שאותם תלמידים חולקים את הציון הגבוה ביותר בתנ"ך. נפתור את הבעיה בשתי דרכים שונות, האחת פשוטה יותר, ויעילה פחות, השנייה פשוטה פחות, יעילה יותר, אך בזבזנית בזיכרון

5.3.1 איתור האיבר המרבי ומופעיו בשיטת שני המעברים

על-פי שיטה זאת אנו סורקים את המערך פעמיים. בסריקה הראשונה נאתר את הערך המרבי במערך, בסריקה השנייה נציג את מספרי התאים בהם שוכן ערך זה. נציג את הקוד:

```
// find highest grade
max = -1 ;
for (stud = 0; stud < CLASS_SIZE; stud ++)
   if (bible[stud] > max)
      max = bible[stud] ;
cout << max << endl ;

// find the students that share the highest grade
for (stud = 0; stud < CLASS_SIZE; stud ++)
   if (bible[stud] == max)
      cout << stud << " " ;
```

<u>הסבר</u>: המשתנה `max` יכיל את הציון הגבוה ביותר בכיתה. אנו מאתחלים אותו לערך קטן מכל ערך אפשרי במערך: לערך `-1`. עתה אנו סורקים את המערך, בכל פעם שאנו נתקלים בתא שמכיל ערך גדול מהערך שמכיל `max` (`bible[stud] > max`) אנו מעדכנים את ערכו של `max` כך שהוא יכיל ערך זה. בתום הסריקה `max` מכיל את הציון המרבי בתנ"ך, ואנו מציגים ציון זה.

עתה אנו סורקים את המערך בשנית. בסריקה השנייה אנו מציגים את *מספרי התאים* בהם מצוי הערך המרבי. שימו לב שאם מתקיים התנאי: `bible[stud] == max` אזי אנו מציגים את *מספר* התא, כלומר את `stud`.

5.3.2 איתור האיבר המרבי ומופעיו בשיטת שני המערכים

השיטה השנייה לאיתור התלמידים שציונם גבוה ביותר תחייב אותנו להחזיק מערך עזר:

```
int best_stud_indexes[CLASS_SIZE] ;
```

מערך העזר יכיל את מספרי התאים במערך `bible` בהם מצוי הציון הגבוה ביותר *הידוע לנו עד כה*.

משתנה עזר נוסף:

```
int occupy = 0
```

יציין בכמה תאים במערך `best_stud_indexes` אנו עושים שימוש.

נתחיל בכך שנציג את הקוד, ואחר נסבירו ביתר פירוט:

```
max = -1
for (stud= 0; stud < CLASS_SIZE; stud ++)
{
   if (bible[stud] > max)     // a new best stud is found
   {
      max = bible[stud] ;
      best_stud_indexes[0] = stud; //only stud is best,
                                   //as far as we know
      occupy = 1 ;              //thr is only a sngl best
   }
   else if (bible[stud] == max)   //additional best stud
   {
      best_stud_indexes[occupy] = stud;
      occupy++ ;
   }
}

cout << max << endl ;
```

```
//display the best studs
for (stud = 0 ; stud < occupy; stud ++)
   cout << best_stud_indexes[stud] << " " ;
cout << endl ;
```

נסביר: אנו עוברים על המערך bible פעם יחידה בלולאת ה- for. אם אנו מגלים תלמיד שציונו גבוה מהציון המרבי הידוע לנו עד כה אנו עושים שלושה דברים: (א) אנו שומרים במשתנה max כי זהו הציון המרבי שפגשנו עד כה, (ב) במערך best_stud_indexes בתא הראשון (תא מספר אפס) אנו שומרים את מספרו של התלמיד, (ג) למשתנה occupy אשר מונה כמה תלמידים מצטיינים מצאנו עד כה, במילים אחרות כמה תאים במערך best_stud_indexes מצויים בשימוש, אנו מכניסים את הערך אחד. בזאת אנו מציינים שכעת יש תלמיד מצטיין יחיד, ולכן רק התא הראשון במערך best_stud_indexes מעניין אותנו, ואם יש ערכים בתאים השני ואילך אין להתייחס אליהם. לחילופין: אם עת אנו סורקים את המערך bible אנו מגלים תלמיד שציונו שווה לציון המרבי הידוע לנו עד כה (bible[stud] == max) עלינו לציין: (א) במערך best_stud_indexes את מספרו של התלמיד המצטיין הנוסף, (ב) במשתנה occupy כי תא נוסף במערך best_stud_indexes נכנס לשימוש.

אחרי שגמרנו לעבור על המערך bible אנו יכולים להציג את מספרי התלמידים המצטיינים כפי ששמורים ב- occupy תאיו הראשונים של המערך best_stud_indexes.

נעקוב אחר דוגמת הרצה: נניח כי תאי המערך bible מכילים את הערכים הבאים (משמאל לימין, השורה העליונה תציין את הערך בתא, והתחתונה את מספרו של התא:

5	3	5	5	7	5	7
0	1	2	3	4	5	6

בפרט נתעניין בהשתנות הערכים במערך best_stud_indexes.
התכנית כוללת לולאה יחידה:

א. בסיבוב הראשון בלולאה ערכו של משתנה הבקרה stud, הוא אפס, התנאי bible[stud] > max מתקיים. על-כן ערכו של max מעודכן להיות 5, לתא מספר אפס במערך best_stud_indexes מוכנס הערך 5, וערכו של occupy מעודכן להיות 1. בתום הסיבוב הראשון בלולאה studגדל להיות 1.

ב. תנאי הכניס ללולאה מתקיים שוב, ולכן אנו נכנסים לסיבוב שני בלולאה, התנאי bible[stud] > max אינו מתקיים, כך גם לגבי התנאי bible[stud] == max, לכן דבר לא מעודכן. בתום הסיבוב בלולאה stud גדל להיות 2.

ג. אנו נכנסים לסיבוב שלישי בלולאה: עתה מתקיים התנאי bible[stud] == max לפיכך לתא מספר 1 במערך best_stud_indexes מוכנס ערכו של stud, כלומר 2, ו- occupy גדל להיות 2 (כדי לציין ששני תאים במערך best_stud_indexes נמצאים עתה בשימוש). בתום הסיבוב בלולאה stud גדל להיות שלוש.

ד. בסיבוב הבא בלולאה קורה אותו מצב כמו בסיבוב הקודם: למערך best_stud_indexes לתא מספר 2 מוכנס הערך 3 (ערכו של stud), ו- occupy גדל להיות 3. בתום הסיבוב בלולאה studגדל להיות ארבע.

ה. בסיבוב הבא בלולאה מתקיים התנאי: `bible[stud] > max` לפיכך `max` מעודכן להיות 7 (כערכו של `bible[stud]`), לתוך התא מספר אפס במערך `best_stud_indexes` מוכנס הערך 4 (ערכו של `i`), ו- `occupy` מעודכן להיות אחד, כדי לציין שעתה אנו מתעניינים בתא הראשון במערך `best_stud_indexes` בלבד.

ו. בסיבוב הבא בלולאה `stud==5`, ואף אחד משני התנאים אינו מתקיים (`bible[stud] == 5 < max`).

ז. בסיבוב האחרון ערכו של `stud` הוא 6, והתנאי שמתקיים הוא השני מבין השניים, על-כן לתא מספר `1` (כערכו של `occupy`) במערך `best_stud_indexes` מוכנס הערך 6, ו- `occupy` גדל להיות 2. בזאת אנו מסיימים את התגלגלותנו בלולאה.

שימו לב כי בתא השלישי (התא מספר 2) במערך `best_stud_indexes` מצוי הערך שלוש, אולם בערך זה לא נתעניין, יען כי `occupy == 2` ולכן אנו מתעניינים רק בשני התאים הראשונים במערך `best_stud_indexes`. בלולאת הצגת הפלט אנו מציגים את ערכי שני התאים הראשונים במערך `best_stud_indexes` כלומר את ארבע ואת שש.

5.3.3 השוואת שני הפתרונות

אין ספק שהפתרון השני קשה יותר להבנה מהפתרון הראשון, וזה בהחלט חיסרון שלו. קוד פשוט ובהיר עדיף על-פני קוד מורכב וקשה להבנה. כמו כן, בעוד הפתרון הראשון מסתפק במערך הנתונים בלבד (ובעוד כמה משתנים פרימיטיביים), הפתרון השני מחייב מערך עזר נוסף (`best_stud_indexes`) שגודלו צריך להיות כגודלו של המערך המקורי (במקרה בו לכל התלמידים בכיתה אותו ציון, יהיו כולם מצטיינים). כלומר הפתרון השני מכפיל את כמות הזיכרון שהתכנית שלנו צורכת, וזו תכונה מאוד לא יפה שלו.

האם אל מול שני חסרונות אלה של הפתרון השני הוא לכל הפחות יעיל יותר מהפתרון הראשון? התשובה היא שבמקרה הסביר, בו כמות המצטיינים היא קטנה, הפתרון השני רץ פעם אחת על המערך המקורי, ופעם נוספת על תחילתו של מערך העזר, על מספר קטן של תאים (אלה המכילים את פרטי התלמידים המצטיינים), לעומת הפתרון הראשון שרץ פעמיים על המערך המקורי; כלומר במצב הסביר, הפתרון השני יעיל יותר בערך פי שתיים מהפתרון הראשון. במקרה הקיצוני, בו לכל התלמידים אותו ציון, ועל כן כולם יוכנסו למערך המצטיינים, יעשה הפתרון השני אותה עבודה כמו הפתרון הראשון: ירוץ פעמיים על מערך שגודלו כמספר התלמידים בכיתה.

שורה תחתונה: לא נראה שהפתרון השני עדיף על פני הראשון.

5.3.4 עיון חוזר בפקודת ההגדלה העצמית

לפני שאנו מסיימים את עניינו עם תכנית זאת נחזור לעיון נוסף **בפקודת ההגדלה העצמית.** בעבר הזכרנו כי ניתן לכתוב את הפקודה באופן `++num1` או באופן `num1++`. בשני המקרים ערכו של `num1` יגדל באחד. ההבדל בין שתי צורות הכתיבה בא לידי ביטוי עת אנו משתמשים בפקודת ההגדלה העצמית בשילוב עם פקודה נוספת. בשפת `C` אנו רשאים לכתוב פקודה כגון:

```
cout << num1++ ;
```

או לחילופין:

```
cout << ++num1 ;
```

נדון בשתי הפקודות הללו, תוך שאנו מניחים שלפני ביצוען ערכו של `num1` היה 17. בנוסח הכתיבה הראשון (`cout << num1++`) *ראשית* יוצג ערכו של `num1`, כלומר

יוצג הערך 17, *ורק אחר-כך* ערכו של num1 יגדל באחד להיות 18. בנוסח הכתיבה השני (cout << ++num1) *ראשית* ערכו של num1 יגדל להיות 18, *ורק אחר-כך* יוצג ערכו, ולכן הערך שיוצג יהיה 18.

באופן כללי יותר: עת אנו משלבים את פקודת ההגדלה (או ההקטנה) העצמית בפקודה אחרת, אזי אם האופרטור (++ או --) נכתב *משמאל* למשתנה הוא מבוצע *טרם* שמבוצעת הפעולה האחרת, ואם האופרטור מופיע *מימין* למשתנה אזי הוא מבוצע *אחרי* ביצוע הפעולה האחרת.

הֶקשר בו מקובל מאוד להשתמש בפעולת ההגדלה העצמית הוא טיפול במערכים. למשל בתכנית שכתבנו ביצענו את זוג הפקודות:

```
best_stud_indexes[occupy] = stud ;
occupy++ ;
```

במקום זאת אנו יכולים לכתוב את הפקודה היחידה:

```
best_stud_indexes[occupy++] = stud ;
```

נניח, לדוגמה, שטרם ביצוע הפקודה היה ערכו של occupy חמש, אזי מכיוון שאופרטור ה- ++ נכתב מימין למשתנה occupy תקרנה הפעולות הבאות: ראשית לתא מספר 5 במערך best_stud_indexes יוכנס ערכו של stud, ורק אחר-כך ערכו של occupy יגדל להיות שש.

5.4 חיפוש במערך

אחת הפעולות השכיחות במערכות מחשבים היא חיפוש האם נתון כלשהו מצוי בקרב סדרה של נתונים. לדוגמה, יתכן שאנו מחזיקים במערך כלשהו רשימה של תלמידים; ואנו נשאלים האם תלמיד שמספרו כזה וכזה רשום במוסדנו. כדי לענות על השאלה עלינו לסרוק את רשימת התלמידים ולבדוק האם קיים ברשימה תלמיד שזה מספרו.

לצורך המשך דיוננו נניח כי בתכנית הוגדר המשתנה:

```
int stud_list[LIST_SIZE];
```

ולמערך הוזנו מספרי התלמידים הרשומים במוסדנו. עוד נניח כי עת אנו מציירים את המערך (כפי שעשינו, למשל, בדוגמה הקודמת) תאי המערך מופיעים משמאל לימין, כלומר התאים שמספריהם נמוכים יותר מופיעים משמאל לתאים שמספריהם גבוהים יותר (בדומה לדרך בה ציירנו את המערך בדוגמה הקודמת שראינו).

5.4.1 חיפוש במערך לא ממוין

במידה ומספרי התלמידים מוחזקים לא ממוינים במערך (כלומר במידה ואין אנו רשאים להניח שהתלמיד שמספרו מינימאלי מופיע ראשון, זה שמספרו השני בגודלו מופיע אחריו, וכן הלאה) אזי כדי לבדוק האם תלמיד כלשהו מצוי ברשימותינו עלינו לסרוק את המערך **סדרתית** (serial): לעבור תא אחרי תא ולבדוק האם התלמיד מצוי בתא זה של המערך. נכתוב את קטע הקוד המתאים:

```
cin >> wanted ;      // read the number of the wanted stud
for (stud = 0; stud < LIST_SIZE; stud ++)
   if (stud_list[stud] == wanted)
      break ;
if (stud == LIST_SIZE)
   cout << "Not in list\n" ;
else
   cout << "In list\n" ;
```

בתכנית השתמשנו בטכניקה שראינו בעבר: במידה והתלמיד המבוקש נמצא אנו קוטעים את ביצוע הלולאה באמצעו (טרם ש- stud שווה בערכו ל- LIST_SIZE), ועל-כן אחרי הלולאה אנו יכולים להסיק על-סמך ערכו של stud האם התלמיד נמצא או לא.

חיפוש במערך לא ממוין ידרוש במקרה הגרוע (עת התלמיד אינו מופיע ברשימה) סדר גודל של LIST_SIZE צעדים, כלומר מורכבות זמן הריצה היא כגודל המערך בו אנו מחפשים.

5.4.2 חיפוש במערך ממוין

אם המערך בו אנו מחפשים ממוין, כלומר אם מספרי התלמידים מופיעים בו מסודרים מקטן לגדול (או להפך: מגדול לקטן), אזי אנו יכולים לחפש תלמיד מבוקש בצורה יעילה יותר: נתחיל בכך שאנו מחפשים בכל תאי המערך (שהרי אין לנו כל סיבה להניח שאם התלמיד מצוי במערך הוא מצוי בו באזור כזה או אחר), ונבדוק מה מספרו של התלמיד שנמצא בתא האמצעי במערך (זה שמספרו: (0 + LIST_SIZE-1)/2). במידה ובתא זה נמצא התלמיד המבוקש אזי נוכל לסיים את החיפוש תוך שאנו מודיעים כי התלמיד נמצא; אחרת: אם מספרו של התלמיד המצוי בתא האמצעי גדול ממספרו של התלמיד המבוקש אזי (מכיוון שהמערך ממוין מקטן לגדול) אנו יכולים להסיק כי אם התלמיד המבוקש מצוי בכלל במערך, הוא לבטח נמצא בחציו השמאלי של המערך; ובאופן סימטרי, במידה ומספרו של התלמיד המצוי בתא האמצעי במערך קטן ממספרו של התלמיד המבוקש אזי אנו יכולים להסיק כי אם התלמיד המבוקש מצוי בכלל במערך, הוא מצוי בחלקו הימני של המערך. בכל אחד משני המקרים האחרונים נמשיך בתהליך החיפוש עם חצי המערך המתאים (תוך שאנו בודקים את האיבר המצוי באמצעיתו, וחוזר חלילה).

נציג את התכנית המתאימה. בתכנית נשתמש במשתנים הבאים:

lo: יציין את מספרו של התא הנמוך ביותר הנכלל בקטע בו אנו מחפשים עתה.

hi: יציין את מספרו של התא הגדול ביותר הנכלל בקטע בו אנו מחפשים עתה.

mid: יציין את מספרו של התא המצוי באמצע הקטע הכולל את התאים: lo..hi.

wanted: יציין את מספרו של התלמיד המבוקש.

found: יציין האם התלמיד המבוקש נמצא.

```
found = false ;
lo = 0;
hi = LIST_SIZE - 1 ;

while (lo <= hi && !found) {
   mid = (lo + hi) /2 ;
   if (stud_list[mid] == wanted)
      found = true ;
   else if (stud_list[mid] < wanted)
      lo = mid + 1 ;
   else
       hi = mid -1 ;
}
if (found)
   cout << "In list\n" ;
else cout << "Not in list\n" ;
```

נסביר: בתחילה אנו מניחים באופן מחדלי כי התלמיד המבוקש אינו במערך (found = false;). כמו כן אנו קובעים את 'גבולות הגזרה' של קטע המערך בו

אנו מחפשים להיות כל המערך כולו (`lo = 0; hi = LIST_SIZE -1;`). עתה אנו נכנסים ללולאה המתנהלת כל עוד: קטע המערך בו ניתן לקוות למצוא את התלמיד המבוקש אינו ריק (`lo <= hi`), וכן לא אותתנו כי התלמיד נמצא (`!found` או במילים אחרות: `found == false`). בגוף הלולאה אנו מכניסים למשתנה `mid` את מספרו של התא המצוי באמצע הקטע בו אנו מחפשים, ואחַר אנו בודקים את היחס בין מספרו של התלמיד המבוקש לבין הערך `stud_list[mid]`, ועל פי היחס בניהם מסמנים לעצמנו שהמספר נמצא, או את 'גבולות הגזרה' החדשים.

נעקוב אחר ריצה לדוגמה של התכנית, ריצה שגם תשכנע אותנו (אני מקווה) שהתכנית מתנהגת כהלכה הן עת התלמיד הרצוי מופיע ברשימה, והן עת הוא אינו ברשימה. נניח כי: `LIST_SIZE = 11`, וכי באחד-עשר תאי המערך שוכנים הערכים הבאים:

5-	5	5	6	10	12	13	17	2000	2001	3879
0	1	2	3	4	5	6	7	8	9	10

נניח כי אנו מחפשים את 17.
בתחילה: `lo = 0, hi = 10, found = false`.

א. התנאי בכותרת הלולאה מתקיים, ולכן אנו נכנסים לסיבוב ראשון בלולאה: `mid = (lo + hi)/ 2 = 5` (במילים: כוונת הביטוי לאמר: `mid` מקבל את ערכו של `(lo + hi)/2` כלומר את הערך 5). בתא מספר 5 במערך שוכן הערך 12. ערך זה קטן מהערך המבוקש, על-כן אנו מעדכנים `lo = mid +1 = 6` (כלומר `lo` מקבל את הערך 6)., כדי לציין שעתה נתמקד בקטע המערך הכולל את התאים `6..10`, שכן אם 17 נמצא במערך אזי הוא נמצא בקטע זה.

ב. לקראת סיבוב נוסף בלולאה אנו שוב בודקים את התנאי שבכותרתה, והוא עדיין מתקיים, ולכן אנו נכנסים לסיבוב שני בלולאה. עתה `mid` מקבל את הערך: `(10 + 6)/2 = 8`. בתא מספר 8 במערך שוכן הערך 2000. ערך זה גדול מהערך המבוקש, לכן `hi = mid -1 = 7`. כלומר עתה אנו מחפשים את הערך בקטע המערך הכולל את זוג התאים `6..7`.

ג. לקראת סיבוב נוסף בלולאה אנו בודקים שוב את התנאי: `lo` שווה 6, `hi` שווה 7, ו- `foud` שווה `false`, ולכן אנו נכנסים לסיבוב שלישי בלולאה. `mid = (6 + 7)/2 = 6`. בתא מספר 6 במערך מצוי הערך 13, ערך זה קטן המערך המבוקש, לכן אנו מעדכנים: `lo = mid +1 = 7`.

ד. לקראת סיבוב רביעי בלולאה אנו בודקים שוב את התנאי, והוא עדיין מתקיים (ערכם של `lo` ושל- `hi` הוא שבע). לכן אנו נכנסים ללולאה: `mid = (lo + hi)/2 = 7`. בתא מספר 7 במערך מצוי הערך המבוקש ולכן אנו מעדכנים את `found` להיות `true`.

ה. לקראת כניסה לסיבוב נוסף בלולאה אנו בודקים שוב את התנאי, אולם עתה הוא כבר אינו מתקיים (לא נכון ש- `!found`), ולכן איננו נכנסים לביצוע סיבוב נוסף בלולאה. אנו פונים לפקודת ה- `if` שאחרי הלולאה, ומציגים את הפלט המתאים.

עתה נניח כי אנו מחפשים את 16.
בתחילה: `lo = 0, hi = 10, found = false`.

א. התנאי בכותרת הלולאה מתקיים, ולכן אנו נכנסים לסיבוב ראשון בלולאה: `mid = (lo + hi)/ 2 = 5`. בתא מספר 5 במערך שוכן הערך 12. ערך זה קטן מהערך המבוקש, על-כן אנו מעדכנים `lo = mid +1 = 6`.

ב. לקראת סיבוב נוסף בלולאה אנו שוב בודקים את התנאי שבכותרתה, והוא עדיין מתקיים, ולכן אנו נכנסים לסיבוב שני בלולאה. עתה `mid` מקבל את הערך

`mid = (10 + 6)/2 = 8`. בתא מספר 8 במערך שוכן הערך 2000. ערך זה גדול מהערך המבוקש, לכן `hi = mid -1 = 7`.

ג. לקראת סיבוב נוסף בלולאה אנו בודקים שוב את התנאי: `lo` שווה 6, `hi` שווה 7, ו- `foud` שווה `false`, ולכן אנו נכנסים לסיבוב שלישי בלולאה. `mid = (6 + 7)/2 = 6`. בתא מספר 6 במערך מצוי הערך 13, ערך זה קטן המערך המבוקש, לכן אנו מעדכנים: `lo = mid +1 = 7`.

ד. לקראת סיבוב רביעי בלולאה אנו בודקים שוב את התנאי, והוא עדיין מתקיים (ערכם של `lo` ושל- `hi` הוא שבע). לכן אנו נכנסים ללולאה: `mid = (lo + hi)/2 = 7`. בתא מספר 7 במערך מצוי הערך 17 הגדול מהערך המבוקש, ולכן אנו מעדכנים: `hi = mid -1 = 6`.

ה. לקראת כניסה לסיבוב נוסף בלולאה אנו בודקים שוב את התנאי, אולם עתה הוא כבר אינו מתקיים (לא נכון ש- `lo <= hi`), ולכן איננו נכנסים לביצוע סיבוב נוסף בלולאה. אנו פונים לפקודת ה- `if` שאחרי הלולאה, ומציגים את הפלט המתאים.

מהו זמן הריצה של התכנית שכתבנו? אם התחלנו עם מערך בן `N` תאים, אזי אחרי סיבוב אחד בלולאה יהיה גודלו של קטע המערך בו אנו מחפשים `N/2` תאים, אחרי שני סיבובים בלולאה יהיה גודלו של קטע המערך `(N/2) /2` ואחרי `i` סיבובים בלולאה יהיה גודלו של קטע המערך $N/2^i$. וכמה פעמים לכל היותר נסתובב בלולאה? במקרה הגרוע ביותר נסתובב בלולאה עד שקטע המערך יהיה בן איבר יחיד, כלומר עד אשר `i` יהיה כזה ש- $N/2^i = 1$. כדי לחלץ את ערכו של `i` (המצוי בביטוי הנוכחי שלנו במעריך שבמכנה) עלינו להוציא `lg` לשני אגפי הביטוי ולקבל: $\lg(N/2^i) = \lg(1)$. כזכור לנו (?):

א. $\lg(1) = 0$,

ב. $\lg(x/y) = \lg(x) - \lg(y)$,

ג. $\lg(x^y) = y*\lg(x)$.

אם נשתמש בשלושה כללים אלה, ונעביר אגפים נקבל ש- $\lg(N) = i*\lg(2)$. עד עתה לא ציינו את בסיס פונקצית ה- `lg`, עתה נקבע כי בסיס הלוג הוא שתיים, ועל-כן $\lg(2) = 1$, ולכן מספר הסיבובים המרבי בלולאה יהיה `i` כך ש- `i` שווה ל- $\lg_2(N)$.

לחילופין אפשר לומר כי מכיוון שאנו הולכים וחוצים את קטע המערך בכל סיבוב בלולאה, ואנו חוזרים על תהליך זה לכל היותר עד שקטע המערך הוא בגודל של תא יחיד, אזי מספר הסיבובים שנדרש לו יהיה לכל היותר $\lg_2(N)$ שכן על הפונקציה $\lg_2(x)$ אפשר לחשוב כעל פונקציה המתארת כמה פעמים אפשר לחצות את `x` עד שנגיע ל- 1. במילים אחרות $\lg_2(x) = y$ אם כדי להגיע ל- 1 מ- `x` יש לחצות את `x` בדיוק `y` פעמים. לדוגמה: $\lg_2(2) = 1$ שכן אם נחצה את 2 פעם 1 נקבל אחד; לעומת זאת: $\lg_2(16) = 4$ שכן אם נחצה את 16 נקבל 8, אם נחצה אותו נקבל 4, אם נחצה אותו נקבל 2, ואם נחצה את 2 נקבל 1, כלומר 4 פעמים חצינו את 16 כדי להגיע ל-1, ולכן $\lg_2(16)$ ערכו 4.

5.5 מיון בועות (Bubble sort)

בסעיף הקודם הנחנו כי אנו מחפשים נתון במערך ממוין. עתה נשאל: כיצד נגיע למצב המיוחל בו המערך ממוין? ניתן, כמובן, להניח כי המשתמש מזין לנו את הנתונים ממוינים; אולם המשתמש הוא משענת קנה רצוץ, ועל כן רצוי שלא להסתמך עליו, אלא להבטיח בכוחות עצמו את מיונו של המערך. תיאוריה ענפה במדעי-המחשב עוסקת בשאלה: כיצד נמיין נתונים, ואלגוריתמים רבים לביצוע מיון הוצעו. בהמשך דרכנו נכיר מספר אלגוריתמים מתוחכמים, יחסית, הממיינים מערך ביעילות. בשלב זה נרצה להכיר אלגוריתם הממיין מערך שלא ביעילות. (נוכל

להכיר בחוסר יעילותו של האלגוריתם שנכיר עתה רק אחרי שנשווה אותו לַאלגוריתמים היעילים שנכיר בהמשך).

האלגוריתם אותו נכיר עתה נקרא מיון בועות, והוא אחד מקבוצה של אלגוריתמי מיון לא יעילים (האלגוריתמים האחרים בקבוצה דומים לו למדי ונקראים מיון בחירה selection sort, ומיון הכנסה insertion sort).

כיצד מתנהל מיון בועות? נניח כי אנו ממיינים מקטן לגדול. המיון מתנהל כלולאה כפולה. הלולאה הפנימית, עת רצה בפעם הראשונה, דואגת לכך שהאיבר הגדול ביותר יגיע למקומו, כלומר לתא האחרון במערך. עת הלולאה הפנימית מורצת בפעם השנייה (על-ידי הלולאה החיצונית), היא דואגת למקֵם את האיבר השני בגודלו במקום המתאים לו, כלומר בתא האחד לפני אחרון במערך. וכך הלאה: עת הלולאה הפנימית מורצת בפעם ה- round -ית, היא דואגת למקם את האיבר ה- round -י בגודלו במקום המתאים לו, כלומר בתא מספר round-1 מסוף המערך. וכמה פעמים תורץ הלולאה הפנימית? אם במערך N איברים אזי יש להריצה N פעמים, או ליתר דיוק N -1 פעמים (שכן אחרי שהאיבר האחד לפני הכי קטן יוצב במקומו, האיבר הכי קטן כבר יידחק למקום הראוי לו: התא הראשון במערך).

כיצד תבצע הלולאה הפנימית את מלאכתה? היא תסרוק את המערך סדרתית (תא אחר תא) פעם יחידה, מתחילתו ועד סופו. בכל פעם שהיא נתקלת בזוג ערכים המצויים בזוג תאים סמוכים כך שהערך בתא השמאלי גדול מהערך בתא הימני, כלומר עת היא נתקלת בשני תאים סמוכים שמפרים את הסידור הרצוי (בו ערך שמשמאל קטן מערך שמימינו), הלולאה מחליפה בין ערכי התאים. בשלב מסוים בריצתה על פני המערך תיתקל הלולאה בַּערך הגדול ביותר במערך, ערך זה יהיה גדול מהערך שמימינו, ועל-כן הלולאה תחליף בינו ובין הערך שמימינו, אחרי ההחלפה שוב הערך יהיה גדול מהערך שמימינו, ועל כן שוב הוא יוחלף עם הערך שמימינו. וכך הלאה, הערך הגדול ביותר 'יפעפע' לקצה הימני של המערך (כמו בועה שמפעפעת מנוזל). בפעם הבאה שהלולאה הפנימית תרוץ יקרה מצב דומה: בשלב מסוים היא תתקל בערך השני בגודלו, ואז היא תפעפע אותו עד למקומו הרצוי (וכמובן שלא מעבר לכך, שכן מעבר לכך שוכנים ערכים גדולים יותר, ולכן היא לא תבצע החלפה).

נדגים ריצה יחידה של הלולאה הפנימית. נניח כי המערך כולל שמונה תאים עם הערכים הבאים:

5	3	6	5	7	2	2	4
0	1	2	3	5	5	6	7

כפי שניתן לראות המערך עדיין לא ממוין כלל, כלומר זו הריצה הראשונה של הלולאה הפנימית. הלולאה תשווה את התא הראשון (מספר 0) עם זה שלצידו (מספר 1), תגלה שהערך בתא השמאלי גדול מהערך בתא הימני ועל-כן תחליף בין ערכי התאים באופן שמצב המערך יהיה:

3	5	6	5	7	2	2	4
0	1	2	3	5	5	6	7

עתה הלולאה תתקדם לתא השני (תא מספר 1), היא תשווה אותו לזה שמימינו (תא מספר 2) ותגיע למסקנה כי אין צורך להחליף בניהם (5 < 6). עתה הלולאה תתקדם לתא השלישי ותחליף בינו לבין התא שמימינו (6 יוחלף עם 5) כך שנקבל:

3	5	5	6	7	2	2	4
0	1	2	3	5	5	6	7

עתה הלולאה תתקדם לתא הרביעי, ולא תחליף בינו לבין זה שמימינו. הלולאה תתקדם לתא החמישי, בשלב זה היא פוגשת באיבר הגדול ביותר במערך, ונשים לב לקורה מעתה וַאילך. לפי הכללים שתיארנו הלולאה תחליף בין הערך בתא החמישי לבין הערך בתא שמימינו, ומצב המערך יהיה:

3	5	5	6	2	7	2	4
0	1	2	3	5	5	6	7

הלולאה תתקדם לתא השישי ותחליף בינו לבין זה שמימינו. היא תתקדם לתא השביעי ושוב תחליף בינו לבין זה שמימינו. אנו רואים ששבע, הערך המרבי במערך, 'פעפע' לקצה המערך. מצב המערך עתה:

3	5	5	6	2	2	4	7
0	1	2	3	5	5	6	7

כיצד תראה הריצה השנייה של הלולאה הפנימית? שוב הלולאה תסרוק את המערך מתא מספר אפס ואילך. היא לא תחליף בין הערך בתא #0 לערך בתא שמימינו. היא לא תחליף בין הערך בתא #1 לערך בתא שמימינו (הם שווים בגודלם, ואין צורך על-כן להחליף בניהם). היא לא תחליף בין הערך בתא #2 לערך בתא שמימינו. היא כן תחליף בין הערך בתא #3 לערך בתא שמימינו, ואחר תשוב ותחליף בין הערך בתא #4 (הערך 6) לבין הערך שמימינו (הערך 2), ואחר תשוב ותחליף בין הערך בתא #5 לבין הערך שבתא מימינו (הערך 4), ואז היא לא תחליף בין הערך בתא #6 (הערך 6) לבין הערך בתא מימינו (הערך 7). מצב המערך בתום הריצה השנייה של הלולאה הפנימית יהיה:

3	5	5	2	2	4	6	7
0	1	2	3	5	5	6	7

נציג, ראשית, גרסה ראשונית של מיון בועות ואחר נשפרה:

```
for (round =0; round < LIST_SIZE -1; round++)
{
   for(i= 0; i< LIST_SIZE -1; i++)
     if (stud_list[i] > stud_list[i +1])
      {   int temp = stud_list[i] ;
          stud_list[i] = stud_list[i +1] ;
          stud_list[i +1] = temp ;
      }
}
```

כפי שהסברנו התכנית כוללת לולאות מקוננות. בכל סיבוב של הלולאה הפנימית אנו בודקים האם האיבר שמשמאל (זה שבתא מספר `i`) גדול מהאיבר שמימין לו (זה שבתא מספר `i+1`), ואם כן מחליפים בין ערכי התאים.

מספר הערות נוספות אודות התכנית:

1. שימו לב כי הלולאה הפנימית רצה כל עוד `i < LIST_SIZE -1`, כלומר כל עוד `i` אינו מגיע לתא האחרון במערך. הסיבה לכך היא שבכל סיבוב בלולאה אנו משווים את התא מספר `i` עם התא מספר `i+1`. אם היינו רצים עד התא האחרון במערך, כלומר עד אשר `i < LIST_SIZE` הינו מנסים בסיבוב האחרון בלולאה (בו ערכו של `i` היה `LIST_SIZE -1`) להשוות בין התא `stud_list[LIST_SIZE -1]` לבין התא `stud_list[LIST_SIZE];` אולם במערך שלנו אין תא שמספרו `LIST_SIZE`. כלומר הינו חורגים מגבולות המערך, וזו כמובן שגיאה חמורה. הימנעות משגיאות שכאלה מחייבת לעיתים טרחה מרובה ובדיקות מדוקדקות, אולם אין מנוס מכך, ראו הוזהרתם!

בתכניות שנכתוב בהמשך נמַנע בדרך כלל מעיסוק בפרטים שוליים אך קריטיים שכאלה שכן אין בהם עניין עקרוני. דא עקא יש בהם עניין מעשי רב, ולכן בתכניות שתכתבו הקפידו מאוד על ניסוח תנאי העצירה של הלולאות השונות.

2. בתכנית הגדרנו משתנה עזר בשם `temp`. `temp` הוא קיצור מקובל ל-`temporary`. משתנה בשם זה משמש לצורך שמירה *רגעית* של ערכים. הקפידו שלא לעשות במשתנה בשם שכזה שימוש מעבר למתואר כאן, בפרט הקפידו שתכניתכם לא תכלול מספר משתנים: `temp1, temp2, temp3, …`. שימו לב שהמשתנה `temp` מוגדר בתוך הגוש\בלוק של פקודת ה-`if`, לכן הוא גם מוכר רק בתוך גוש זה. ניסיון לפנות למשתנה מחוץ לגוש זה יגרום לקומפיילר להודיע לנו כי הוא אינו מכיר משתנה בשם זה.

3. בתכנית שכתבנו הלולאה הפנימית רצה בכל המקרים עד התא מספר `LIST_SIZE -2`. למעשה אין בכך צורך, ונסביר מדוע. בפעם הראשונה שהלולאה הפנימית מורצת, עליה למקם את האיבר הגדול ביותר במקומו בקצה המערך, ולשם כך עליה אכן לרוץ עד התא מספר `LIST_SIZE -2`. אולם בפעם השנייה שהלולאה הפנימית מורצת, עת עליה למקם את האיבר השני בגודלו במקומו, אין צורך שהיא תתעניין בתא האחרון במערך, ותשווה את הערך שבתא מספר `LIST_SIZE -2` לערך שבתא האחרון, שהרי בתא האחרון ניצב כבר האיבר הגדול ביותר ולכן ברור שההשוואה מיותרת (גם אם לא מזיקה). באופן דומה עת הלולאה הפנימית מורצת בפעם השלישית, עליה למקם את האיבר השלישי בגודלו במקומו. בעת שהיא מבצעת משימה זאת אין כל צורך שהיא תתעניין בערכים המצויים בשני תאי המערך האחרונים (בהם אנו כבר יודעים שמצויים שני הערכים הגדולים ביותר). וכך הלאה: עת הלולאה הפנימית מורצת בפעם ה-`round` -ית (עבור `round = 0, 1, …`), די שהיא תרוץ עד *ולא כולל* התא מספר `LIST_SIZE - round -1` ותשווה את הערך שבתא זה עם הערך שבתא שמימינו. על כן את הלולאה הפנימית אנו יכולים לכתוב באופן הבא:

```
for (i=0; i < LIST_SIZE - round -1; i++) …
```

4. שיפור נוסף שניתן להכניס לתכנית הוא השיפור הבא: בכל פעם שהלולאה הפנימית רצה, נבדוק האם היא ערכה שינוי כלשהו במערך, במילים אחרות, האם היא החליפה בין ערכי שני תאים כלשהם. במידה ובשלב כלשהו יתברר לנו כי הלולאה הפנימית לא עשתה דבר, נוכל להסיק כי המערך כבר ממוין, וניתן לעצור את תהליך המיון, כלומר לצאת מהלולאה החיצונית. את השיפור נממש באופן הבא:
   א. נוסיף לתכנית משתנה בולאני בשם `change`.
   ב. לפני כל כניסה ללולאה הפנימית נבצע: `change = false`.
   ג. בלולאה הפנימית, עת אנו מחליפים בין שני תאים סמוכים, גם נבצע: `change = true`.
   ד. מיד אחרי היציאה מהלולאה הפנימית נבדוק האם `change == false`, ואם כן נבצע `break` מהלולאה החיצונית.

   אני משאיר לכם כתרגיל קל להכניס את התוספות הדרושות לתכנית שהצגנו.

5. מה הוא זמן הריצה של מיון בועות במערך הכולל `N` מספרים? עת הלולאה הפנימית מורצת בפעם הראשונה, היא מבצעת `N-1` איטרציות. עת הלולאה הפנימית מורצת בפעם השנייה, היא מבצעת `N-2` איטרציות. וכן הלאה, עד אשר עת הלולאה הפנימית מורצת בפעם האחרונה היא מבצעת איטרציה יחידה. אם נסכום את מספר האיטרציות המתבצעות בלולאה הפנימית בכל

הפעמים בהן היא מורצת גם יחד על-ידי הלולאה החיצונית נקבל את סכום הטור החשבוני: 1+… + (N-2) + (N-1) בעבר כבר ראינו כי סכומו של טור זה ניתן לחישוב באמצעות הנוסחה: $S_{N-1} = (a_1 + a_{N-1})*(N-1)/2$ ובמקרה שלנו: $a_1 = N-1, a_{N-1} = 1$. אם נציב את הערכים הדרושים, נפתח סוגריים, ונתעלם מכפל בקבועים, נגלה כי מורכבות זמן הריצה היא בסדר גודל של N^2. (אני מזכיר לכם כי הסכימה שלנו סכמה את מספר האיטרציות שבוצעו בלולאה הפנימית בכל הפעמים שהיא מורצת גם יחד; אולם מכיוון שבכל איטרציה בלולאה אנו מבצעים מספר קבוע של פעולות אזי הערכה זאת מתארת את מורכבות זמן הריצה של התכנית). זמן הריצה שחישבנו נכון הן לגרסה הראשונית שהצגנו בתחילה, והן לתכנית אשר תכלול את שני השיפורים שתיארנו בהמשך, וזאת משום שעת אנו דנים בזמן הריצה אנו מתייחסים לזמן הריצה *במקרה הגרוע ביותר*. במקרה הגרוע ביותר השיפור שהוצע בסעיף הקודם לא יקטין כלל את זמן הריצה (שאלה קטנה: מהו המקרה הגרוע ביותר עבור מיון בועות? כלומר מה צריך להיות סידור הערכים התחילי במערך על-מנת שזמן הריצה יהיה המרבי?)

5.6 איתור מספרים ראשוניים בשיטת הכברה

כברה היא נפה גסה (במילים אחרות מסננת גסה). שיטת הכברה לאיתור מספרים ראשוניים הייתה מוכרת כבר ליוונים הקדמונים. היא הוצעה על-ידי המתמטיקאי היווני ארתוסטנס אשר חי במאה השלישית לני הספירה (ואשר, אגב, חישב בדיוק די רב את היקפו של כדור הארץ, מידע שאבד בהמשך, אך לו היה ידוע, היה גורם לקולומבוס, עת הוא הגיע לאמריקה, להבין שלא ייתכן שהוא כבר בהודו). עתה נכיר גם אנו את שיטת הכברה.

אלגוריתם הכברה פועל באופן הבא:

א. סמן את כל המספרים בתור מספרים שעשויים להיות ראשוניים (כלומר כמספרים שאנו מניחים מחדלית שהם ראשוניים עד שלא יתברר אחרת).

ב. יהי שתיים המספר הראשוני הנוכחי המוכר לך.

ג. חזור על התהליך הבא:

1. סמן את כל כפולותיו של המספר הראשוני הנוכחי המוכר לך כמספרים פריקים.
2. התקדם מהמספר הראשוני הנוכחי המוכר לך, עד אשר אתה מוצא מספר שלא צוין כלא ראשוני. בחר אותו כמספר הראשוני הנוכחי המוכר לך.

כדי לממש את אלגוריתם הכברה כתכנית מחשב עלינו להגדיר מערך:

```
bool primes[N] ;
```

עבור N שהוגדר כקבוע בתכנית. התכנית תציג את המספרים הראשוניים שבין שתיים ל: N-1 (מסיבות מתמטיות שונות, אחד לא נחשב למספר ראשוני, למשל מספר ראשוני צריך להתחלק בשני מספרים *שונים* בדיוק: באחד ובעצמו).

עוד נניח כי בתכנית הוגדרו הקבועים הבולאניים:

```
const bool IS_PRIME = true,
           NOT_PRIME = false ;
```

נציג עתה את קטע הקוד הדרוש:

```
//assume all nums are primes
for (num = 0; num < N; num ++)
   primes[num] = IS_PRIME ;
```

```
// go over the primes[] array
for (num = 2; num < N/2; num++)
{
   if (primes[num]== IS_PRIME)// if current num is prime
                              // mark all its multiples
as
                              // not prime
   {
      for (int mul = 2; num * mul < N; mul++)
         primes[num * mul] = NOT_PRIME ;
   }
}

for (num = 2; num < N; num ++)
   if (primes[num] == IS_PRIME)
      cout << num << " " ;
```

הסבר התכנית: לולאת ה- `for` הראשונה מסמנת את כל המספרים כמועמדים להיות ראשוניים. במילים אחרות כמספרים שאנו מניחים מחדלית בחיוב, כל עוד לא התברר לנו אחרת שהם ראשוניים.

הלולאה הכפולה המרכזית של האלגוריתם עוברת על כל המספרים החל בשתיים ועד `N/2` (לא כולל `N/2`). עבור כל ערך של `num`, אם `primes[num] == IS_PRIME` , כלומר אם `num` הוא ראשוני אזי אנו משתמשים ב- `num` כדי לציין שכל כפולותיו אינן ראשוניות, וזאת עושה הלולאה הפנימית המריצה את המשתנה `mul` כל עוד `num*mul < N`, כלומר כל עוד איננו חורגים מחוץ לגבולות המערך, בכל סיבוב בלולאה הפנימית אנו מסמנים עבור הכפולה הנוכחית של `num` שמספר זה אינו ראשוני.

ניתן דעתנו לכך שהתנאי: `if (primes[num]== IS_PRIME)` אינו הכרחי במובן זה שגם אם היינו משמיטים אותו התכנית שלנו לא הייתה שגויה. מדוע אם כן אנו כוללים אותו בקוד? לו היינו משמיטים את התנאי היינו משתמשים גם במספרים פריקים כגון 4, 6, 8, 9, ... כדי לסמן שכפולותיהם אינן ראשוניות. זה לא היה בגדר שגיאה אך זה סתם מיותר. את העובדה שכפולותיו של ארבע הינן פריקות ציינו כבר עת ציינו שארבע פריק, כלומר עת ערכו של `num` היה שתיים. באופן דומה את העובדה שכפולותיו של שש הן פריקות סימנו עת השתמשנו בשתיים וכן עת השתמשנו בשלוש. כלומר לו היינו משמיטים את התנאי הנ"ל התכנית שלנו הייתה עושה עבודה מיותרת (גם אם לא מזיקה).

הערה נוספת בעניין היעילות: בקוד שלנו, בלולאה הפנימית, אנו מרצים את `mul` החל בערך שתיים ואילך. למעשה ניתן להריצו החל בערך `mul` ואילך. נסביר מדוע באמצעות דוגמה: עת ערכו של `num` הוא שבע, וערכו של `mul` הוא `2, 3, 4, 5, 6` אנו מסמנים כפריקים מספרים שכבר סומנו בעבר. למשל `2*7` סומן עת `num` היה שתיים ואזי סימנו את: `2*7`, וכך הלאה ל: `3*7`.

לבסוף, בלולאה האחרונה בקוד, אנו מציגים את המספרים הראשוניים על-ידי שאנו מציגים את *מספרי* התאים בהם קיים הערך `IS_PRIME`. שימו לב כי אנו עושים כאן היפוך מסוים בהשוואה לשימוש השכיח במערכים: בדרך-כלל מספרו של התא הוא יחסית הטפל, אשר רק משמש לאיתור התא אשר תוכנו הוא העיקר.

בתכנית הנוכחית שלנו תוכנו של התא (`true` או `false`) הוא יחסית הטפל אשר רק מעיד האם מספרו של התא, שהוא כאן העיקר, ראשוני או פריק.

באלגוריתם זה, באופן קצת יוצא דופן, לא נדון בשאלת זמן הריצה שכן הדיון בה מחייב שימוש בשיקולים מתמטיים מורכבים (מה אחוז המספרים הראשוניים בכלל הטבעיים). לבֵטח ניתן להגיד שזמן הריצה קטן מ: N^2, אולם באמצעות שיקולים מתמטיים מורכבים יותר ניתן לתת הערכה טובה מזאת של זמן הריצה.

5.7 תרגילים

5.7.1 תרגיל מספר אחד: תרגול פשוט של מערכים חד-ממדיים

כתבו תכנית (יחסית קלה ופשוטה שתכניסכם לנושא המערכים) אשר:

א. מגדירה מערך של מספרים שלמים.

ב. מאפשרת למשתמש לציין כמה נתונים ברצונו שמערך יכיל. **אחר כך**

ג. מאפשרת למשתמש להזין נתונים בכמות אותה הוא הזין בסעיף ב'. **או**

ד. מגרילה ערכים לתוך תאי המערך (בכמות הדרושה). **אחר כך עד שהמשתמש מבקש לסיים הוא יוכל שוב ושוב לבצע את הפעולות הבאות (ע"י שהוא בוחר אותן מתפריט שיוצג בפניו):**

ה. לבקש לבצע הסטה מעגלית של המערך מקום אחד ימינה. בעקבות הבקשה יועבר הנתון שבתא #0 לתא #1, הנתון בתא #1 יועבר לתא #2,..., הנתון בתא #(N – 2) יועבר לתא #(N – 1), והנתון בתא #(N – 1) יועבר לתא #0.

ו. לבקש לבצע הסטה מעגלית של המערך מקום אחד שמאלה. בעקבות הבקשה יועבר הנתון שבתא #1 לתא #0, הנתון בתא #2 יועבר לתא #1,..., הנתון בתא #(N – 1) יועבר לתא #(N – 2), והנתון בתא #0 יועבר לתא #(N – 1).

ז. לבקש לבצע שיקוף של המערך סביב האיבר האמצעי בו (אם מספר האיברים במערך פרדי), או סביב ציר דמיוני המצוי בין שני האברים האמצעיים (אם מספר האיברים במערך זוגי). (<u>לדוגמא</u>: שיקוף של המערך הכולל את 1, 2, 3, 7, 1 יביאו למצב 1, 7, 3, 2, 1. שיקוף של 1, 2, 3, 7 יביאו למצב: 7, 3, 2, 1).

ח. לסיים.

<u>הערות</u>:

א. יש לוודא שהקלט תקין במובן זה שכמות הנתונים שהמשתמש רוצה להכניס למערך, וכן שקודי הפעולות שהוא מזין הינם סבירים. יש לחזור על תהליך קריאת הנתונים עד שיוזן קלט שאינו שגוי.

ב. אחרי הכנסת הנתונים למערך, כמו גם בעקבות ביצוע כל פעולה מבוקשת תדפיס התכנית את תוכנם של תאי המערך שבשימוש.

5.7.2 תרגיל מספר שתיים: סדרת שתיים שלוש

נגדיר **סדרת שתיים שלוש** להיות סדרה אשר האיבר הראשון בה הוא מספר טבעי כלשהו. אם ערכו של האיבר מספר n בסדרה הוא x, אזי ערכו של האיבר ה- (n+1) -י בסדרה הוא: (x % 2 ==0) ? x/2 : x*3 +1).

עליכם לכתוב תכנית אשר מדפיסה סדרות שתיים שלוש המתחילות במספרים 1 עד מאה, אולם יצור כל סדרה ייעצר עת מיוצר ערך שכבר הופיע בסדרה קודמת (ולפיכך תת הסדרה שהייתה מיוצרת מערך זה והלאה כבר יוצרה). לדוגמה:

1

1 <- 2

3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 (וכאן ייצור הסדרה נעצר שכן 2 כבר הופיע בעבר).

4 (כנ"ל—4 כבר הופיע בעבר)

5

3 <- 6

<u>רמז</u>: החזיקו מערך occur בן מאה תאים בולאניים. ערכו של occur[i] יהיה true אם ורק אם i הופיע כבר בסדרה שהוצגה. שימו לב כי הסדרות המיוצרות עשויות להכיל ערכים > 100, אולם ערכים אלה לא יזכרו (למה?). לתכנית זאת אין קלט.

5.7.3 <u>תרגיל מספר שלוש: ניהול טבלה בליגת כדור-רגל</u>

1. בתכנית זאת עליכם לדמות טבלה בליגת כדורגל דמיונית.
התכנית תחזיק **מבנה נתונים** שיורכב מארבעה מערכים:

א. מערך שיחזיק את שמות הקבוצות (שמות הקבוצות יהיו מספרים שלמים ייחודיים לכל קבוצה וקבוצה שיבחרו אקראית ע"י המחשב).

ב. מערך שיחזיק את מספר הנקודות שצברה כל קבוצה.

ג. מערך שיחזיק את מספר השערים שהבקיעה כל קבוצה.

ד. מערך שיחזיק את מספר השערים שספגה כל קבוצה.

דוגמה למערכים אפשריים בתחילת העונה (בהנחה של ארבע קבוצות. כל מערך הינו עמודה בטבלה):

<u>שם קבוצה</u>	<u>נקודות שצברה</u>	<u>שערים שהבקיעה</u>	<u>שערים שספגה</u>
38	0	0	0
69	0	0	0
96	0	0	0
2	0	0	0

במהלך העונה יתקיימו מספר סיבובים של משחקים. בכל סיבוב תשחק כל קבוצה כנגד כל קבוצה אחרת פעם יחידה. תוצאת כל משחק תוגרל ע"י המחשב (בהינתן שמספר השערים המרבי שקבוצה עשויה להבקיע במשחק הוא שבעה שערים). בעקבות ביצוע כל סיבוב יש להציג את תוצאות המשחקים שנערכו באותו סיבוב (כמה משחקים יערכו בכל סיבוב?), יש לעדכן את הטבלה, ולהציגה כמקובל (כלומר על-פי דרוג הקבוצות בה). העדכון יתבצע באופן הבא:

א. קבוצה שניצחה במשחק תזכה בשלוש נקודות נוספות, קבוצה שהפסידה במשחק תאבד נקודה, ומשחק שהסתיים בתיקו יזכה כל-אחת משתי המתמודדות בנקודה אחת.

ב. לכל קבוצה יעודכן מספר השערים שהיא הבקיעה\ספגה.

ג. בעקבות עדכון הנתונים הללו עבור כל המשחקים שנערכו בסיבוב כלשהו יש למיין את הטבלה (את המערכים השונים) כך שקבוצה שלה יותר נקודות תופיע לפני קבוצה לה פחות נקודות, בין שתי קבוצות להן אותו מספר נקודות נעדיף את זו שהפרש השערים שלה (מספר השערים שהקבוצה הבקיעה פחות מספר השערים שהקבוצה ספגה) גבוה יותר. בין שתי קבוצות להן אותו מספר נקודות ואותו הפרש שערים יקבע הסדר באופן מקרי.

ד. אחרי מיון הטבלה יש להציגה למשתמש ולאפשר לו להחליט האם ברצונו לקיים סיבוב נוסף של משחקים או לסיים.

לדוגמה: נניח כי הליגה שלנו כוללת ארבע קבוצות ששמותיהן 38, 69, 96, 2. נניח כי בסיבוב הראשון היו תוצאות המשחקים כדלהלן:

קבוצה1-	קבוצה2-	שערים1-	שערים2-	נקודות1-	נקודות 2-
38	69	2	1	3	1-
38	96	0	0	1	1
38	2	0	5	1-	3
69	96	1	1	1	1
69	2	4	2	3	1-
96	2	3	3	1	1

אזי (אם לא טעיתי) הטבלה תראה:

מקום	הקבוצה	נקודות	שערי זכות	שערי חובה
1	2	3	10	7
2	69	3	6	5
3	96	3	4	4
4	38	3	2	6

הערות:

א. הקלט לתכנית מוזן ע"י המשתמש רק עת ברצונו לציין האם להריץ סיבוב נוסף בליגה או שיש לעצור. הפלט בתום כל סיבוב יראה כפי המודגם בשתי הטבלות שהוצגו.

ב. מיון המערכים יתבצע תוך שימוש באלגוריתם המיון 'מיון הכנסה'. מיון הכנסה מתבצע באופן הבא: בתום הסיבוב מספר round של התהליך (round = 0, 1, 2, …) יהיה קטע המערך שבין תא #0 לתא #round ממוין. הדבר יושג ע"י שהערך בתא מספר round יוכנס למקום המתאים בקטע המערך 0..round-1, תוך שהערכים בתאים בקטע בהם מצוי ערך הגדול מהערך בתא מספר round מוסטים מקום אחד ימינה כדי לפנות מקום פנוי לתוכו יוכנס הערך מהתא מספר round. ההסטה ימינה תביא (אולי) את הערכים המוסטים להתפרש על-פני תאים 0..round (כאשר באמצע הקטע נותר תא פנוי לתוכו יוכנס הערך שהיה בתא מספר round). לדוגמא בסיבוב מספר 2 של התהליך (שהינו הסיבוב השלישי) יוכנס הערך מהתא #2 למקום מתאים לו בקטע המערך (הממוין כבר) הכולל את תאים 0,1, תוך שהערכים הרצויים בתאים אלה מוסטים ימינה (עד התא #2), ומפנים לו מקום. הצעה: כתבו לעצמכם תכנית עזר קטנה אשר ממיינת מיון בחירה מערך יחיד. אחרי שתכנית זאת תבצע את פעולתה כהלכה שלבו את הקוד שלה בתכנית שעליכם להגיש (תוך שאתם גם מכלילים אותו על-פי דרישות התרגיל).

ג. שימו לב כי אחרי הגרלת תוצאת כל משחק ומשחק ניתן לעדכן את שערי הזכות והחובה ומספר הנקודות שיש לכל קבוצה במערכים המרכזיים של התכנית. אין צורך להשתמש במערכי עזר כלשהם. המערכים יוחזקו לפיכך לא ממוינים במהלך הסיבוב. הם ימוינו בתום הסיבוב לפני הצגתם למשתמש.

5.7.4 תרגיל מספר ארבע: מיקום ציר במערך

בתרגיל הנוכחי עליכם לכתוב תכנית שתועלתה וטעמה יוברר לכם רק בהמשך עת נכיר את אלגוריתם המיון quick sort.

התכנית שתכתבו תקרא מהמשתמש סדרה של נתונים לתוך מערך (int data[...]). אחר תקרא התכנית מהמשתמש שני מספרים (lo, hi) המהווים אינדקסים של שני תאים במערך, באופן זה יזין\יציין המשתמש לתכנית נתונים אודות קטע רצוי לו במערך; הקטע יתחיל בתא data[lo] ויסתיים בתאdata[hi] תוך שהוא כולל גם את שני התאים הללו (על תכניתכם גם לבדוק שהמשתמש הזין ערכים סבירים לשני המשתנים הנ"ל). עתה תפעל התכנית באופן הבא: הערך בתא data[lo] ישמש אותנו כציר (פיבוט) באופן הבא: את כל הערכים בקטע המערך המבוקש שקטנים מהערך שבתא מספר lo (מהציר) תעביר התכנית לתאים בקטע המצויים משמאל לתא אליו היא תעביר את הנתון data[lo], את כל הנתונים בקטע שערכיהם גדולים או שווים לנתון שבתא data[lo] תעביר התכנית לתאים בקטע המצויים מימין לתא אליו היא תעביר את הנתון המצוי מלכתחילה בתא data[lo] (כאשר אנו אומרים משמאל\מימין אנו מתכוונים לתאים שהאינדקסים שלהם קטנים\גדולים יותר). אין חשיבות לשאלה לאן בדיוק יועבר בקטע כל נתון כל עוד הוא מועבר ל'צדו' הנכון של הציר. הפונקציה שתבצע את המשימה הנ"ל תחזיר לתכנית הראשית ערך שיציין את המקום במערך בו הושם הפיבוט.

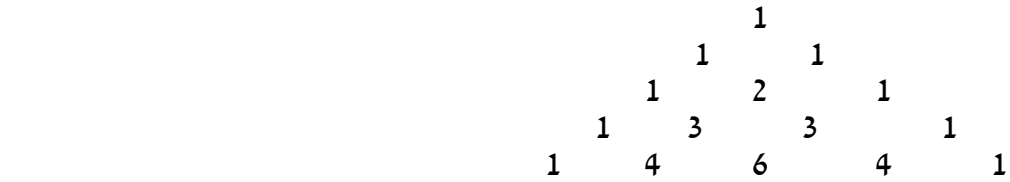
<u>לדוגמא</u>: במערך 3, 5, 0, 5, 7, 2, 4, 99, 99, 1 (הנתון 1 מצוי בתא #0, הנתון 3 מצוי בתא #9)
ואם המשתמש מזין את הערכים 0, 9 למשתנים lo ו- hi אזי את הנתון בתא data[0] שהינו 1 יש למקם במערך כך שכל הנתונים הקטנים ממנו (כלומר 0) ישכנו משמאלו, וכל היתר מימינו, וכך נקבל למשל את המערך: 3, 5, 5, 7, 2, 4, 99, 99, 1, 0 (כל סידור אחר של המערך בו 0 מצוי משמאל ל- 1 וכל יתר הנתונים מימין ל- 1 יחשב גם הוא כסידור קביל). מיקומו של הציר הוא עתה התא מספר 1 במערך, ולפיכך הערך 1 יוחזר על-ידי הפונקציה.
אם, לעומת זאת היה המשתמש מזין **במקום** 0 ו- 9 את הערכים 3 ו- 8, היה עלינו להזיז את data[3] כלומר את הנתון 4 כך שהנתונים הקטנים ממנו בקטע המתחיל בתא #3 ונגמר בתא #8 (כלומר הנתונים 2, 0) היו משמאלו, והנתונים הגדולים ממנו בקטע הנ"ל (כלומר 7, 5, 5) היו מימינו, (בכל יתר המערך איננו נוגעים), והיינו מקבלים את המערך עם הסידור: 3, 5, 5, 7, 4, 0, 2, 99, 99, 1 (או כל סידור אחר שהיה מקיים את הדרישות הנ"ל). הערך המוחזר: מיקומו של הציר = התא #5.

הקפידו על כללי התכנות, בפרט ובעיקר:
א. מודולריות: חלקו את תכניתכם לשגרות (שגרה לקריאת הנתונים למערך, שגרה לקריאת גבולות הקטע המבוקש, שגרה שתחליף בין ערכיהם של שני תאים במערך, ועוד שגרות כפי שתמצאו לנכון).
ב. תיעוד:
1. תיעוד ראשי של התכנית.
2. עבור כל שגרה: מה השגרה עושה, מה תפקיד כל פרמטר שלה (האם ואיזה מידע השגרה
מקבלת או מחזירה ממי שקרא לה באמצעות הפרמטר).
3. תיעוד משתנים: יש לתעד את תפקידו של כל משתנה בכל שגרה.
ג. שמות משמעותיים.
ד. שימוש בקבועים.
ה. אינדנטציה.

<u>הערה חשובה</u>: ניתן וגם ראוי לבצע את המשימה בצורה יעילה וללא שימוש במערך עזר. כיצד?

5.7.5 תרגיל מספר חמש: משולש פסקל

משולש פסקל הינו משולש מהצורה:



כלומר זהו משולש בו בשני קצות כל שורה ניצב הערך 1, וכל ערך אחר בכל שורה מחושב כסכום זוג הערכים בשורה מעל הניצבים מימינו ומשמאלו של הערך שיש לחשב (שימו לב שהמשולש מוצג כך שבין כל שתי שורות עוקבות [האחת שמספרה פרטי והשניה שמספרה זוגי] הנתונים אינם מוצגים בדיוק זה מעל זה, אלא בהסטה, ורק הנתונים שבשורות הזוגיות/הפרטיות מופיעים בדיוק אחד מעל השני).

עליכם לכתוב תכנית הקוראת מהמשתמש מספר טבעי (שיכול להיות חסום על-ידי קבוע של התכנית), ומציגה למשתמש משולש פסקל בעל מספר שורות כמבוקש.

<u>הערות</u>:

1. ניתן להניח שאורך כל שורה של המשולש אינו חורג מאורך שורה על המסך.
2. אין להשתמש בנוסחת הבינום (מי שכלל אינו מכיר נוסחה זאת יכול להתעלם מהערה זאת).
3. <u>הערה</u>: הפקודה `cout << setw(5) << i` תדפיס את ערכו של המשתנה השלם i על-פני חמישה מקומות, תוך שהיא מותירה משמאל לו רווחים כפי הצורך (לדוגמה אם ערכו של i הוא 17 יושארו שלושה רווחים לשמאלו). כדי שהמהדר יכיר את המרכיב setw() בפקודת הפלט יש להוסיף לתכנית את השורה: #include <iomanip.h>.

5.7.6 תרגיל מספר שש: חישוב מספר צירים בפרלמנט

באיי הרוח נערכים לבחירות לפרלמנט. עליכם לכתוב תכנית אשר תחשב ותציג כמה צירים בפרלמנט יהיו לכל מפלגה. בפרלמנט של איי הרוח P מקומות. בבחירות משתתפות N מפלגות ששמותיהן, לצורך עניינו, הם 0..N-1. תכניתכם תקרא מהמשתמש:

א. בכמה קולות זכתה כל מפלגה.

ב. מה אחוז החסימה הנהוג באיי הרוח.

ג. זוגות של מפלגות ביניהן קיימים הסכמי עודפים.

התכנית תציג כמה חברי פרלמנט יהיו מכל מפלגה. היא תחשב זאת על-פי הכללים הבאים:

א. קולותיהן של מפלגות שלא עברו את אחוז החסימה יועברו למאגר מרכזי, ויחולקו בין המפלגות שעברו את אחוז החסימה באופן יחסי למספר הקולות בהם זכתה כל מפלגה שעברה את אחוז החסימה. לדוגמה: נניח כי בבחירות השתתפו ארבע מפלגות אשר זכו במספרי הקולות הבאים: 6, 12, 12, 20 , וכי אחוז החסימה הוא חמשה-עשר אחוז. אזי המפלגה מספר אפס לא עברה את אחוז החסימה, ועל-כן קולותיה יחולקו בין יתר שלוש המפלגות באופן הבא: למפלגות מספר אחד ושתיים יוקצו: (6*12)/(12+12+20) קולות נוספים. למפלגה מספר שלוש יוקצו: (6*20)/(12+12+20) .

ב. יחושב מספר הקולות השקול למנדט יחיד. החישוב ייערך באופן הבא: (מספר צירי הפרלמנט) חלקי (סכום הקולות בהם זכו מפלגות שעברו את אחוז החסימה, אחרי הקצאה המחודשת).

ג. לכל מפלגה יוקצה מספר הצירים המגיע לה בסיבוב הראשון של הקצאת הצירים. מספר הצירים יהיה המספר השלם הגדול ביותר שקטן או שווה מהמנה: (מספר הקולות שקיבלה המפלגה) חלקי (מספר הקלות השקול למנדט יחיד).

ד. יחושב מספר הקולות העודפים שנותרו לכל מפלגה אחרי סיבוב ההקצאה הראשון.

ה. במידה ולמפלגה x יש הסכם עודפים אם מפלגה y (ובהנחה ששתיהן עברו את אחוז החסימה) יועבר מספר הקולות העודפים שיש למפלגה x למפלגה y או להפך, בהתאם לשאלה למי נותרו יותר קולות עודפים.

ו. יוקצו יתר המקומות בפרלמנט בשיטה הבאה: חזור על התהליך הבא כל עוד נותרו מקומות להקצות:

1. חשב את מספר הקולות לציר: סכום מספר הקולות שטרם תרמו להכנסת ציר חלקי מספר המקומות שיש להקצות.
2. כל מפלגה שב-'קופתה' נותר מספר קולות גדול או שווה לערך הנ"ל תקבל ציר נוסף, ומספר הקולות שבקופתה יקטן בשיעור המתאים.
3. במידה ואין מפלגה לה די קולות (ועדיין נותרו מקומות להקצאה), אזי הקצה את יתר המושבים למפלגות השונות באופן הבא: כל מפלגה תקבל ציר נוסף, תוך שההקצאה מתבצעת מהמפלגה בקופתה נותרו יותר קולות, ואילך, לפי סדר מספר הקולות שנותרו בידי המפלגות השונות.

לדוגמה: נניח כי אחרי הסכם העודפים נותרו ב-'קופתה' של מפלגה אחת חמישה קולות, בקופתה של מפלגה שניה שני קולות, ובקופתה של מפלגה שלישית קול יחיד. עוד נניח כי יש להקצות שני מקומות בפרלמנט. אזי המפתח למקום בפרלמנט הוא: $(5+2+1)/2=4$ המפלגה היחידה בקופתה נותר מספר גדול או שווה של קולות היא המפלגה הראשונה, ולכן מוקצה לה ציר נוסף, ומספר הקולות בקופתה קטן בארבע, לכדי קול יחיד. עתה המפתח למקום בפרלמנט הוא: $(1+2+1)/1=4$ אין אף מפלגה לה מספר כזה של קולות, על-כן המפלגה בידיה מספר מרבי של קולות (המפלגה השניה) תקבל את הציר הנוסף.

5.7.7 <u>תרגיל מספר שבע: מציאת חציון ושכיח</u>

במערך חד-ממדי של מספרים שלמים המכיל מספר פרדי N של נתונים מוגדר **הנתון החציוני** להיות זה ש- $(N-1)/2$ מיתר הנתונים (פרט לו) קטנים או שווים ממנו, ו- $(N-1)/2$ מיתר הנתונים גדולים או שווים ממנו. לדוגמה: בכל שלושת המערכים הבאים הנתון החציוני הוא שלוש:

5	3	1	2	4

3	1	1	3	3

3	3	3	3	3

3	3	5	1	1

במערך חד-ממדי של מספרים שלמים המכיל מספר זוגי M של נתונים מוגדר החציון להיות הממוצע בין האיבר המקיים שבדיוק (M-2)/2 מיתר הנתונים קטנים או שווים ממנו, לבין האיבר המקיים שבדיוק (M-2)/2 מיתר הנתונים גדולים או שווים ממנו. לדוגמה בשני מערכים הבאים ערכו של החציון הוא 3.5:

1	3	0	4	5	5

1	3	3	4	4	4

שכיח במערך חד-ממדי של מספרים שלמים הוא מספר שאין מספר המופיע במערך יותר פעמים ממנו. במערך ייתכנו מספר שכיחים. לדוגמה במערך האחד לפני אחרון מאלה שהוצגו השכיח הוא הנתון חמש, במערך האחרון שהוצג השכיח הוא ארבע, במערך הראשון שהוצג מופיע כל מספר פעם יחידה, ולכן כל מספר הינו השכיח. נחליט שרירותית כי במקרה כזה נבחר את קטן מבין השכיחים להיות ה-שכיח.

כתבו תכנית הקוראת מספר טבעי המציין מה אורכה של סדרת הנתונים שתוזן אחריו, ואחַר סדרת נתונים באורך המתאים. התכנית תציג את הממוצע, החציון והשכיח של סדרת הנתונים.

5.7.8 <u>תרגיל מספר שמונה: מציאת ערך מרבי במערך יוני-מודלי</u>

מערך יוני-מודלי הוא מערך המקיים שהערכים בו הולכים וגדלים (או גדלים שווים) מתא אפס במערך ועד תא מספר p כלשהו, ומהתא מספר p+1 ואילך הערכים הולכים וקטנים. לדוגמה המערך הבא הינו יוני-מודלי:

17-	2	3	12	17	17	3978	9

כתבו תוכנית אשר:

א. קוראת מהמשתמש נתונים לתוך מערך.

ב. בודקת שהמערך הינו יוני-מודלי. במידה והמערך אינו יוני-מודלי התכנית מציגה הודעת שגיאה ומסיימת.

ג. מאתרת ומציגה *בצורה יעילה* את האיבר המרבי במערך.

6 מערכים רב-ממדיים

המערכים שהכרנו עד כה היו מורכבים משורה אחת (ארוכה כרצוננו) של נתונים. לעיתים אנו זקוקים למערך שיכיל מספר שורות של נתונים. לדוגמה, בסעיף שעבר החזקנו את ציוני התלמידים בתנ"ך במערך חד-ממדי; עתה נניח כי הכתה בה אנו דנים לומדת מקצועות נוספים, ואנו רוצים להחזיק את ציוני התלמידים בכל המקצועות.

לכאורה נוכל להציע את הפתרון הבא: נחזיק מערך חד-ממדי עבור כל מקצוע ומקצוע:

```
int bible[40],
    hebrew[40],
    arabic[40],
    english[40] ;
```

אולם חסרונו של פתרון זה דומה לַחסרון אותו הצגנו עת התחלנו את דיוננו במערכים החד-ממדיים: אין לנו כלים נוחים לטפל *בכלל* המערכים בנוחות, למשל לקרוא ערכים *לכלל* המערכים בנוחות (ולא לכל אחד מהם בנפרד, באמצעות לולאה ייעודית), או לחשב ממוצע כולל של כלל התלמידים בכלל המקצועות.

על כן את הפתרון הנ"ל נפסול, ונציע פתרון אחר: מערך דו-ממדי. המערך הדו-ממדי הוא מעין טבלה או מטריצה. המערך כולל כך וכך שורות, וכך וכך עמודות. כל שורה במערך הדו-ממדי (כפי שאנו נגדיר אותו) תכיל ציוני תלמיד יחיד בכל המקצועות אותם לומדת הכתה. מספר השורות במערך יהיה כמספר התלמידים בכתה, ומספר עמודות יהיה כמספר המקצועות אותם לומדת הכיתה. כדי לקבל מערך דו-ממדי כנ"ל נזדקק להגדרות הבאות:

ראשית, נגדיר שני קבועים שיורו כמה תלמידים (לכל היותר) ילמדו בכתתנו, וכמה מקצועות (לכל היותר) תלמד הכתה:

```
const int MAX_STUD = 100,
          MAX_COURSE = 10 ;
```

שנית, נגדיר את המערך המבוקש:

```
int grades[MAX_STUD][MAX_COURSE] ;
```

במערך שהגדרנו האינדקס הראשון יעבור על התלמידים, שמספריהם (על-פי הקבועים שהגדרנו) יהיו בתחום `0..99`, האינדקס השני יעבור על המקצועות השונים, וערכיו יהיו תמיד בתחום `0..9`. בהמשך נפנה למערך באופנים שונים, במילים אחרות בחתכים שונים, אולם תמיד, תמיד נקפיד שהפניה תיעשה עת האינדקס הראשון המתייחס למספר התלמיד (והוא בתחום `0..99`), והאינדקס השני מייחס למספר הקורס (והוא בתחום `0..9`).

מערך דו-ממדי נקרא גם **מטריצה** (matrix).

6.1 דוגמות ראשונות

נתחיל בדוגמה המאפסת את המערך. נבחר לאפס את המערך באופן הבא: ראשית נאפס את ציוני התלמיד הראשון, אחר את ציוני התלמיד השני, וכן הלאה, עד שלבסוף נאפס את ציוני התלמיד האחרון. מבחינה תכנותית משמעות הדבר היא כי הלולאה החיצונית תעבור על התלמידים, ועת היא במקום ה-`stud`-י, כלומר עת היא מייחסת לתלמיד מספר `stud`, נריץ לולאה פנימית אשר תאפס את כל ציוניו של התלמיד מספר `stud`. נראה את קטע הקוד:

```
for (stud = 0; stud < MAX_STUD; stud++)
   for (course = 0; course < MAX_COURSE; course++)
      grades[stud][course] = 0 ;
```

הערות:

א. אנו מניחים, כמובן, כי המשתנים `stud, course` הוגדרו בתכנית קודם לקטע המתואר. עת אנו משתמשים בלולאות כפולות ראוי לתת גם לאינדקסים שמות משמעותיים, ולהימנע משמות כגון: `i, j, k` וכולי.

ב. בהתאמה לאופן בו הגדרנו את המערך, האינדקס הראשון רץ על הערכים `0..99`, והאינדקס השני על הערכים `0..9`.

נראה עתה קטע תכנית מעט שונה, בו אני קוראים את ציוני התלמידים מהמשתמש. אעיר כי אם בכוונתנו לקרוא את ציוני התלמידים מהמשתמש, כפי שנעשה מייד, אזי מיותר ראשית לאפס את המערך. על כן אין להתייחס לקטע הקוד הבא כאילו הוא מופיע בתכנית אחרי קטע הקוד הקודם. אלה שתי דוגמות נפרדות, כל אחת מציגה שימוש דומה אך שונה במערך דו-ממדי.

נכתוב את קטע התכנית בשני אופנים שונים. הגרסה הראשונה תניח כי ציוני התלמידים מוזנים לתכנית על-ידי המשתמש באופן הבא: ראשית מוזנים כל ציוני התלמיד הראשון, אחר מוזנים כל ציוני התלמיד השני, וכן הלאה, עד שלבסוף מוזנים ציוני התלמיד האחרון. קטע התכנית המתקבל כמעט זהה לקטע התכנית שראינו קודם לכן:

```
for (stud = 0; stud < MAX_STUD; stud++)
   for (course = 0; course < MAX_COURSE; course++)
      cin >> grades[stud][course] ;
```

הגרסה השנייה של קריאת הציונים תניח הנחה מעט שונה: ראשית מזין המורה את ציוני כל התלמידים במקצוע הראשון, אחר הוא מזין את ציוני כל התלמידים במקצוע השני, וכן הלאה, עד אשר לבסוף הוא מזין את ציוני כל התלמידים במקצוע האחרון. כדי לקרוא את הנתונים בחתך זה עלינו לשנות את סדר הלולאות: הלולאה החיצונית תעבור עתה על המקצועות השונים, ועת היא במקום ה-`course`-י, כלומר עת היא מייחסת למקצוע מספר `course`, נריץ לולאה פנימית אשר תקרא את ציוני כל התלמידים במקצוע מספר `course`. נראה את קטע הקוד:

```
for (course = 0; course < MAX_COURSE; course++)
   for (stud = 0; stud < MAX_STUD; stud ++)
      cin >> grades[stud][course] ;
```

הנקודה אותה ברצוני להדגיש היא כי בהשוואה לקטע הקוד הקודם, בקטע הקוד הנוכחי מפאת השינוי בחתך הזנת הציונים שינינו את סידור הלולאות (הלולאה החיצונית עוברת על הקורסים, בעוד הלולאה הפנימית עוברת על התלמידים), אך *לא שינינו את האינדקסים בעת הפניה למערך*, ולכן עדיין האינדקס הראשון (השמאלי), אשר מתייחס לתלמידים, עובר על הערכים `0..99`, בעוד האינדקס השני (הימני), אשר מתייחס למקצועות, עובר על הערכים `0..9`. מה היה קורה לו היינו הופכים את סדר האינדקסים, כלומר לו היינו כותבים `grades[course][stud]`? אזי כבר בסיבוב הראשון של הלולאה החיצונית (עת `course == 0`), עת הלולאה הפנימית הייתה מריצה את `stud` ומקדמת אותו לערך `10` (להזכירכם `stud` אמור לעבור על כל הערכים בין 0 ל- 99), היינו מנסים לקרוא ערך לתא `grades[0][10]`, אולם במערך שלנו אין תא כזה, (על-פי הדרך בה הגדרנו את המערך, טווח ערכיו של אינדקס הימני הוא `0..9`) ולכן אנו מבצעים

שגיאה! נבהיר: אם בעת הגדרת המערך קבענו שהאינדקס השמאלי (אינדקס השורות) יציין תלמידים, והאינדקס הימני (אינדקס העמודות) קורסים, אזי לא משנה כיצד אנו מתפעלים את הלולאות (איזו לולאה היא חיצונית ואיזו פנימית), בכל מקרה המציין השמאלי יציין תלמיד, והמציין הימני קורס.

נציג עתה אופנות הזנת נתונים שלישית: עתה נניח כי המורה מזין את הנתונים באופן הבא: בכל פעם הוא מזין שלושה נתונים הכוללים מספר תלמיד, מספר קורס, וציון רצוי (לדוגמה: אם ברצונו להזין לתלמיד מספר 17, במקצוע מספר 3, את הציון 88, יקליד המורה: 88 3 17 [17 ראשון, 88 אחרון]). כדי לאותת על סיום הזנת הנתונים יקליד המורה את הערך `-1` כמספר התלמיד. נכתוב את קטע התכנית:

```
while (true)
{
   cin >> stud ;
   if (stud == -1)
      break ;
   cin >> course >> grade ;
   if (stud >= 0 && stud < MAX_STUD &&
       course >= 0 && course < MAX_COURSE &&
       grade >= 0 && grade <= 100)
            grades[stud][course] = grade ;
   else
      cout << "Error in grade\n" ;
}
```

נסביר:

א. התכנית מתנהלת כלולאה אינסופית לכאורה, ממנה אנו יוצאים באמצעות `break` עת מוזן תלמיד שמספרו `-1`. הכתיבה `while (true)` גורמת לכך כי התנאי בלולאה יתקיים תמיד, ולכן זו, לכאורה, לולאה אינסופית.

ב. כמובן שאת `-1` ואת מאה ראוי להגדיר כקבועים.

ג. בגוף הלולאה אנו, ראשית, בודקים האם מספר התלמיד שהוזן מעיד על כך שיש לסיים את תהליך קריאת הנתונים, ואם אכן זה המצב אנו פועלים בהתאם (מבצעים `break` מהלולאה).

ד. אחרת (כלומר יש להמשיך בתהליך הקריאה), אנו קוראים את מספר הקורס המתאים, ואת הציון שיש להזין.

ה. *לעולם* איננו פונים למערך עם אינדקס שהינו ערך שהוזן על-ידי המשתמש, (שהוא, כידוע, טיפוס בלתי אמין בעליל), בלי לבדוק קודם לכן שהערך שהוזן מציין מספר של תא חוקי; לכן אנו כופפים את הפניה למערך ל- `if`.

ו. בתכנית שלמה יותר היה, כמובן, מקום להודיע למשתמש במידה והוא הזין קלט שגוי. בתכנית שכתבנו, מפאת הקיצור, ויתרנו על-כך.

ז. הנקודה המרכזית בקטע הקוד הנוכחי היא שכאן איננו עוברים על תאי המערך לא בחתך תלמידים, ולא בחתך מקצועות; בקטע הקוד האחרון אנו פונים בכל פעם לתא במערך עליו מורה לנו המשתמש (עת הוא מזין את מספר התלמיד, כלומר מספר שורה רצוי, ואת מספר הקורס, כלומר מספר עמודה רצוי).

עתה נכתוב קטע תכנית אשר מציג עבור כל תלמיד בַּכתה, בכמה מקצועות התלמיד נכשל. קטע התכנית יכלול לולאה כפולה: הלולאה החיצונית תעבור על תלמידי הכתה, הלולאה הפנימית תמנה עבור תלמיד נתון בכמה מקצועות הוא נכשל. אנו

מניחים כי בתכנית הוגדר הקבוע השלם PASS אשר מציין מהו ציון המעבר (לדוגמה 55), והוגדר המשתנה השלם counter. נציג את הקוד:

```
for (stud = 0; stud < MAX_STUD; stud++)
{
   counter = 0 ;
   for (course = 0; course < MAX_COURSE; course++)
      if (grades[stud][course] < PASS)
         counter++ ;
   cout << stud << " " << counter << endl ;
}
```

אעיר כי טעות שכיחה בלולאה כפולה כנ"ל היא למקם את ההשמה: counter = 0 ; מחוץ לשתי הלולאות (כלומר מעל הלולאה: for (stud = …). במצב זה עת אנו מתקדמים לתלמיד הבא (בלולאה החיצונית) איננו מאפסים את מונה הכישלונות, ולכן, לדוגמה, עבור התלמיד מספר 17 יוצג מספר הכישלונות של *כלל* התלמידים עד אליו, כלומר של התלמידים 0..17, וזו כמובן שגיאה.

עתה נכתוב קטע תכנית דומה, אשר מציג את המקצוע בו ממוצע הציונים הוא הגבוה ביותר (במידה וקיימים מספר מקצועות בהם ממוצע הציונים הוא מרבי, נציג את הראשון ביניהם). מכיוון שמספר התלמידים אשר לומדים כל מקצוע ומקצוע זהה, אזי המקצוע בו סכום הציונים הוא הגבוה ביותר, הוא גם המקצוע בו ממוצע הציונים הוא הגבוה ביותר. לכן קטע התכנית שנכתוב יכלול לולאה חיצונית אשר תעבור על המקצועות השונים. עבור כל מקצוע ומקצוע נבצע את התהליך הבא:

א. סכום את ציוני התלמידים באותו מקצוע (לתוך המשתנה sum),

ב. בדוק האם sum גדול מהסכום הגדול ביותר הידוע לך עד כה (ואשר שמור במשתנה max), ואם כן אזי: (1) שמור סכום זה בתור הסכום הגדול ביותר הידוע לך עד כה, וכן: (2) שמור (במשתנה best_course) את מספרו של אותו מקצוע כמקצוע בו הממוצע הוא המרבי.

נראה את קטע התכנית:

```
max = -1 ; // init max 2 a value less thn any pssble val

// go over all the courses
for (course = 0; course < MAX_COURSE; course++)
{
   // sum the grades of the current course
   sum = 0 ;
   for (stud = 0; stud < MAX_STUD; stud++)
      sum += grades[stud][course] ;

   if (sum > max)    // if cuurent sum is grtr thn
                     // the best sum known so far
   {   max = sum ;
       best_course = course ;
   }
}

cout << best_course ;
```

6.2 בדיקה האם מטריצה מהווה ריבוע קסם

ריבוע קסם הוא מערך דו-ממדי של מספרים אשר מספר השורות בו שווה למספר העמודות בו (כלומר הוא ריבוע), ואשר מקיים את התנאים הבאים:

א. סכום כל שורות המערך שווה.

ב. סכום כל העמודות במערך שווה, וזהה לסכום כל אחת משורות המערך.

ג. סכום האלכסון הראשי במערך (האלכסון המורכב מהתאים: `a[0][0], a[1][1],…,a[N-1][N-1]`) שווה לסכום כל אחת משורות במערך.

ד. סכום האלכסון המשני (זה המורכב מהתאים: `a[0][N-1], a[1][N-2],…,a[N-1][0]`) שווה לסכום כל אחת משורות המערך.

לדוגמה, המערך הבא הוא ריבוע קסם:

6	1	8
7	5	3
2	9	4

ברצוננו לכתוב קטע תכנית אשר בודק האם המערך `int matrix[N][N]`, לתוכו הוזנו כבר נתונים, הוא ריבוע קסם. התכנית תתנהל על-פי האלגוריתם הבא:

א. סכום את האלכסון הראשי, לתוך המשתנה `sum`.

ב. סכום את האלכסון המשני (לתוך המשתנה `curr_sum`), ובדוק האם הערך המתקבל שווה ל- `sum`. אם לא, ציין במשתנה בולאני כי המערך אינו ריבוע קסם, וסיים את תהליך הבדיקה.

ג. סכום כל אחת ואחת משורות המערך (לתוך `curr_sum`). אחרי סכימת כל שורה בצע אותה בדיקה כמו שתיארנו עבור האלכסון המשני.

ד. בצע עבור עמודות המערך את אותו תהליך כמו עבור שורות המערך.

```
magic = true ;  //assume, by default, the matrix is magic

sum = 0 ;
for (i = 0; i < N; i++)                 // sum main diagonal
   sum += matrix[i][i] ;

curr_sum = 0 ;
for (i = 0; i < N; i++)              // sum secondary diagonl
   curr_sum += matrix[i][N -1 -i] ;
if (curr_sum != sum)
   magic = false ;

for (row = 0; row < N && magic; row++)       // check rows
{
   curr_sum = 0 ;
   for (col = 0; col < N; col++)
      curr_sum += matrix[row][col] ;
   if (curr_sum != sum)
      magic = false ;
}
```

```
for (col = 0; col < N && magic; col++)      // check cols
{
   curr_sum = 0 ;
   for (row = 0; row < N; row++)
      curr_sum += matrix[row][col] ;
   if (curr_sum != sum)
      magic = false ;
}

if (magic)
   cout << "It is a magic square\n" ;
else cout << "It is not a magic square\n" ;
```

מספר הערות לתכנית:

א. השם `row` הוא שם מקובל למשתנה העובר על שורות מטריצה. השם `col`, קיצור של `column`, מציין, באופן דומה, עמודות.

ב. ערכו של `curr_sum` מאותחל לאפס *בתוך* כל לולאה כפולה, לפני הכניסה ללולאה הפנימית. האם זה הכרחי? במילים אחרות, האם יכולנו לאפס את `curr_sum` לפני כל לולאה כפולה (במקום בתוכה)?

ג. שימו לב כי כל אחת משתי הלולאות הכפולות (זו הבודקת את השורות, וזו הבודקת את העמודות) מתנהלת כל עוד `row < N && magic`. לכן אם יתברר (בעקבות בדיקת האלכסון המשני, בעקבות בדיקת שורה כלשהי, או בעקבות בדיקת עמודה כלשהי) כי הריבוע אינו ריבוע קסם, אזי ביצוע אותה לולאה ייעצר. יתכן אפילו שלא נכנס כלל לביצוע כל אחת משתי הלולאות הכפולות (מתי זה יקרה?).

ד. בתכנית הנוכחית קיים כפל קוד לא כל כך אסתטי: הקוד לבדיקת השורות, וזה לבדיקת העמודות מאוד דומים זה לזה. כפל קוד נחשב לפגם מאוד לא ראוי בתכנית; למרות זאת, בשלב זה של חיינו 'נבלע' זאת. יודעי ח"נ מוזמנים לחשוב כיצד ניתן למנוע את כפל הקוד, כלומר כיצד ניתן לכתוב לולאה כפולה יחידה אשר תבדוק הן את השורות והן את העמודות. <u>רמז</u>: במקום להשתמש בַּמשתנה היחיד `curr_sum` נשתמש בשני משתנים: `curr_sum_row`, `curr_sum_col`.

ה. אני מזמין אתכם לבדוק בעצמכם מה היא מורכבות זמן הריצה של התכנית.

6.3 בדיקה האם מערך קטן משוכן במערך גדול

נניח כי בתכנית כלשהי הוגדרו הקבועים:

```
const int ROWS1 = …,
          COLS1 = …,
          ROWS2 = …,
          COLS2 = … ;
```

והמשתנים:

```
int small[ROWS1][COLS1],
    big[ROWS2][COLS2] ;
```

עוד נניח כי למערכים הוזנו ערכים, ועתה ברצוננו לבדוק האם במערך `big` קיים תא (`big[row][col]`) כך שמתקיים שתא זה מהווה פינה שמאלית עליונה של תת-מערך במערך `big`, הזהה לגמרי למערך `small`. נדגים את כוונתנו באמצעות שני מערכים:

5	17	3879	9
2	1	***17***	***3879***
4	6	***1***	***7***

17	3879
1	7

נניח כי המערך העליון הוא big, והתחתון הוא small, וכי שורה מספר אפס במערך היא השורה העליונה, והעמודה מספר אפס במערך היא העמודה השמאלית ביותר. בהנחות אלה מתקיים כי התא big[1][2] מהווה פינה שמאלית עליונה של תת-מערך במערך big הזהה לגמרי למערך small. שימו לב כי התא big[0][1] מהווה פינה שמאלית עליונה של תת-מערך במערך big אשר שורתו הראשונה זהה למערך small, אך שורתו השנייה (הכוללת את הערכים 17, 1) אינה זהה לשורה השנייה במערך small.

נציג את קטע התכנית ואחר נסבירו (מעבר לתיעוד המפורט למדי ששתלנו בקוד):

```
nested = flase ;  // assume, by default it is not nested

// go over all possible cells of big[][], for each check
// if it is a top left corner of a sub matrix identical
// to small[][]

for (row_big = 0; row_big <= ROWS2-ROWS1 && !nested ;
     row_big++)
   for (col_big = 0; col_big <= COLS2-COLS1 && !nested;
        cols_big++)
   {
      identical = true ; // assume by default it is a
                         // corner of a desird sub-matrix

                         // check the last assumption
      for(row = 0; row < ROWS1 && identical; row++)
         for (col = 0; col < COLS1 && identical; col++)

                          // if it is false mark it as so
            if (big[row_big + row][col_big + col] !=
                small[ row ][ col ] )
                       identical = false ;

                    // if the last assption was not false
                    // mark that small is nested in big
      if (identical) nested = true ;
   }

if (nested)
   cout << "It is nested in position: "
        << row_big -1 << " " << col_big -1 ;
```

התכנית משתמשת בשני משתנים בולאניים:

א. `nested` המציין האם המערך הקטן משוכן אי-שם בתוך הגדול. אנו מאתחלים אותו לערך `false`, כדי לציין שהנחת המחדל שלנו היא שהמערך הקטן אינו משוכן במערך הגדול.
ב. `identical` המציין האם קטע כלשהו במערך הגדול זהה לגמרי למערך הקטן.

בתכנית קיימת לולאה כפולה חיצונית אשר עוברת על התאים המתאימים במערך הגדול. משתני הבקרה של זוג הלולאות החיצוניות הם: `row_big, col_big`. שימו לב במיוחד לתנאי הסיום של הלולאות, למשל לתנאי: `row_big <= !nested && ROWS2-ROWS1`. התנאי מורה כי:
א. עלינו להתקדם על שורות המערך הגדול עד לשורה מספר `ROWS2 - ROWS1` (כולל אותה שורה. אני מזמין אתכם לבדוק מדוע זו הדרך לנסח את תנאי הסיום. למשל מה יקרה אם `ROWS1 == ROWS2`, או אם `ROWS1 > ROWS2`? האם הלולאה שלנו תתנהל כהלכה? במילים אחרות האם יבַדקו כל התאים ורק התאים במערך `big` אשר יכולים בכלל להוות פינה שמאלית עליונה של תת-מערך הזהה למערך `small`? האם אין חשש שבזוג הלולאות הפנימיות נחרוג אל מחוץ לגבולות המערך `big`?)
ב. וכן נתנהל בזוג הלולאות החיצוניות כל עוד לא מצאנו תא במערך הגדול המהווה פינה שמאלית עליונה של תת-מערך הזהה למערך הקטן.

בתוך הלולאה הכפולה החיצונית (אשר עוברת על תאי המערך הגדול), קיימת לולאה כפולה פנימית, אשר עוברת על תאי המערך הקטן. לולאה כפולה זאת בודקת האם המערך הקטן זהה לתת-המערך הגדול שזו פינתו השמאלית העליונה. לפני הכניסה ללולאה הכפולה הפנימית אנו מניחים בחיוב כי התא הנוכחי במערך הגדול (התא `big[row_big][col_big]`) מהווה פינה שמאלית עליונה של קטע מערך הזהה למערך הקטן. (אנו עושים זאת באמצעות ההשמה: `identical = true`). עתה אנו סורקים את התאים המתאימים במערך הקטן ובמערך הגדול, ואם אנו מגלים כי הנחתנו הייתה שגויה אנו מעדכנים את ערכו של `identical`. אם אחרי סריקת המערך הקטן, והשוואת תאיו לתאים המתאימים במערך הגדול, ערכו של `identical` עדיין `true`, אזי אות הוא וסימן כי התא `big[row_big][col_big]` אכן מהווה פינה שמאלית עליונה של קטע מערך כנדרש, ואנו מעדכנים את ערכו של `nested`, ובהמשך יוצאים מזוג הלולאות החיצוניות (יען כי התנאי `nested!` כבר אינו מתקיים).

בפקודת ההדפסה אנו מדפיסים את ערכם של `row_big -1, col_big -1`, שכן נניח שגילינו תא כמבוקש עת ערכו של `col_big` היה 2. לפני כניסה לסיבוב נוסף בלולאה ערכו של `col_big` גדל להיות 3; עתה נבדק תנאי הכניסה, מתברר שאין להיכנס יותר ללולאה (לא נכון ש-`nested!`), ואנו פונים (אחרי יציאה באופן דומה גם מהלולאה שמריצה את `row_big`) לפקודת ה-`if`. בשלב זה ערכו של `col_big` כבר גדול באחד מהערך אותו ברצוננו להציג.

מה מורכבות זמן הריצה של התכנית שכתבנו? במקרה הגרוע (עת המערך הקטן אינו משוכן בגדול), יהיה עלינו לבדוק כל אחד ואחד מהתאים במערך `big` אשר עשויים להוות פינה שמאלית עליונה של תת-מערך כנדרש. במערך `big` קיימים `(ROWS2-ROWS1)*(COLS2-COLS1)` תאים שכאלה. עבור כל תא ותא שכזה (במקרה הגרוע) עלינו לסרוק את כל המערך הקטן, כלומר `ROW1*COLS1` תאים. כמות העבודה הכוללת תהיה לפיכך: `(ROW2-ROWS1)*(COLS2-COLS1)*ROWS1*COLS1`.

6.4 איקס-עיגול נגד המחשב

לסיום דיוננו במערכים דו-ממדיים נרצה לכתוב תכנית אשר משחקת איקס-עיגול (או איקס-דריקס-מיקס או tic-tac-toe) נגד המשתמש. המחשב בתכנית שלנו ישחק בחוסר תבונה גמור: הוא יגריל את הצעד שלו (תוך שהוא מקפיד, כמובן, לא להניח את הסימן שלו במשבצת בה קיים כבר סימן). כמו כן לשם הפשטות נניח כי תמיד המשתמש מתחיל במשחק (אני מזמין אתכם להכליל את התכנית לכזו בה כל אחד משני המתמודדים עשוי להתחיל).

```
// --------------------- include section --------------------------
-
#include <iostream>
#include <cstdlib>
#include <ctime>
// ---------------------  using section  --------------------------
-
using std::cin ;
using std::cout ;
using std::endl ;

// --------------------- const section   --------------------------
-
const int N = 3 , // board size
   EMPTY = 0,     // values borad[][] may hold
   COMP = 1,
   USER = -1 ;

// --------------------- main -----------------------------
int main()
{
   int board[N][N] ,          // the game board
       row, col,              // current move
       temp_row, temp_col,    // for printing the board
       round,                 //round in game
       i ;                    // index for loops
   bool win ;                 // did somebody win

   srand(time(NULL)) ;        // initialize the random number gen.

   // init the board to be empty
   for (row = 0; row < N; row++)
      for (col = 0; col < N; col++)
         board[row][col] = EMPTY ;

   // run the game (at most N*N rounds)
   for (round = 0; round < N*N; round++)
   {
      // get the next move
      do
      {
         if (round % 2 == 0)        // user's turn
            cin >> row >> col ;
         else                       // computer turn
         {
             row = rand() % N ;     // is picked by random
             col = rand() % N ;
         }
```

```
      } while (row < 0 || row >= N ||
               col < 0 || col >= N ||
               board[row][col] != EMPTY ) ;

      // mark the board
      board[row][col] = (round % 2 == 0) ? USER : COMP ;

      // draw current board
      cout << endl << "Board after round #" << round << endl ;
      for (temp_row = 0; temp_row < N; temp_row++)
      {
         for (temp_col = 0; temp_col < N; temp_col++)
            if (board[temp_row][temp_col] == EMPTY)
               cout << " - " ;
            else if (board[temp_row][temp_col] == USER)
               cout << " X " ;
            else cout << " O " ;
         cout << endl ;
      }

      // check if there is a winner
      win = true ;          // did someone win by completing line
      for (i = 0; i < N && win; i++)
         if (board[row][i] != board[row][col])
            win = false ;
      if (win)
         break ;

      win = true ;           // win by completind a column
      for (i = 0; i < N && win; i++)
         if (board[i][col] != board[row][col])
            win = false ;
      if (win)
         break ;

      win = true ;          //win by completing main diagonal
      for (i = 1; i < N && win; i++)
         if (board[i][i] != board[0][0] || board[i][i] == EMPTY)
            win = false ;
      if (win)
         break ;

      win = true ;          // win by cmpleting secondary diagonal
      for (i = 1; i < N && win; i++)
         if (board[i][N-1-i] != board[0][N-1] ||
             board[i][N-1-i] == EMPTY)
            win = false ;
      if (win)
         break ;
   }

   if (win)                       // if someone win
      if (round % 2 == 0)         // if it is in even round
         cout << "You won\n" ;    // it must be the user
      else cout << "Sorry, I won\n" ;
   else cout << "Tie\n" ;

   return EXIT_SUCCESS ;
}
```

6.5 משתנים ברי-מנייה, משתנים מטיפוס enum

בתכנית האחרונה שהצגנו הכיל כל תא במערך אחד משלושה ערכים אפשריים (שהוגדרו באמצעות קבועים): EMPTY, COMP, USER. כדי להפוך את תכניתנו ל-'נקייה' יותר נרצה להגדיר את המערך באופן שתאיו יוכלו להכיל רק את שלושת הערכים הללו (ולא כל מספר שלם שהוא). באופן כזה נגן על תכניתנו מפני שגיאות שעלולות להיגרם עת לתאי המערך יוזן ערך שגוי, ערך שאינו אחד משלושת הערכים הסבירים. כיצד נשיג את השיפור הרצוי? ראשית, במקום להגדיר את שלושת הקבועים באופן בו הגדרנו אותם נכתוב את השורה הבאה (אשר תופיע מתחת להגדרת הקבועים האחרים):

```
enum board_vals_t {EMPTY_BV, COMP_BV, USER_BV } ;
```

המילה השמורה enum מורה כי ברצוננו להגדיר טיפוס משתנים חדש, אשר את ערכיו אנו נמנה (enumerate) מפורשות. כלומר הערכים שניתן יהיה להכניס למשתנה שנגדיר בהמשך מטיפוס זה יהיו בדיוק אותם ערכים שאנו מונים עתה.

המילה שמופיעה אחר-כך, במקרה שלנו board_vals_t, מציינת את שמו של הטיפוס (היא אנלוגית לפיכך ל- int, float שגם הם שמות של טיפוסים). נהוג ששמו של טיפוס יסתיים בסופית _t (יש אדיטורים שהדבר גם מסייע להם לצבוע לכם את הערכים בצבעים מתאימים).

לבסוף, הערכים המופיעים בתוך הסוגריים הם הערכים שמשתנים מטיפוס board_vals יוכלו להכיל. במקרה שלנו משתנה מטיפוס board_vals_t יוכל להכיל בדיוק אחד משלושת הערכים: EMPTY_BV, COMP_BV, USER_BV ולא שום ערך אחר. כדי לעזור לקורא של התכנית לדעת ששלושה קבועים אלה שייכים לטיפוס enum יחיד (בניגוד אולי לקבוצת קבועים אחרת שתהיה שייכת לטיפוס enum אחר) נהוג שלכל הקבועים תהיה אותה סיומת, אותו 'שם משפחה', בדוגמה שלנו אלה התווים _BV .

על כן אם ב- main נגדיר enum board_vals_t bv ; אזי לתוך המשתנה bv (שטיפוסו הוא enum board_vals_t) נוכל להכניס אחד משלושת הערכים שמנינו, לדוגמה: bv = EMPTY_BV ;.

אפשר לחשוב על הטיפוס bool כאילו הוא הוגדר כבר באופן דומה:

```
enum bool {false, true} ;
```

ולכן עת אנו מגדירים משתנה שטיפוסו הוא bool אנו רשאים להכניס לתוכו בדיוק אחד משני הערכים שמופיעים בסוגריים.

אין אפשרות לקרוא מהמשתמש באמצעות פקודת cin ערך לתוך משתנה מטיפוס enum כלשהו. הדפסת ערכו של המשתנה תדפיס את המספר הטבעי אפס אם המשתנה מכיל את הערך הראשון ברשימת הערכים האפשריים, תדפיס את הערך אחד אם המשתנה מכיל את הערך השני ברשימת הערכים האפשריים, וכך הלאה.

באופן דומה לדרך בה הגדרנו את bv שהינו משתנה בודד מטיפוס board_vals, אנו יכולים להגדיר: enum board_vals_t board[N][N] ; ולקבל מערך דו-ממדי שכל תא בו יוכל להכיל אחד משלושת הערכים שתיארנו בהגדרת הטיפוס. המשך התכנית, כולל כל פעולות האיתחול, ועדכון הערכים במערך יראה בדיוק כפי שהוא נראה בתכנית שהצגנו.

למעשה, עת אנו מגדירים טיפוס כדוגמת board_vals_t כל ערך אפשרי בין ערכי הטיפוס שקול למספר שלם כלשהו. המחדל הוא שהערכים שקולים לערכים

השלמים אפס, אחד, ואילך. על-כן אם i הוא משתנה שלם אזי אנו רשאים לבצע את זוג הפקודות הבא: i = 2; bv = (enum board_vals_t)i ; נסביר: ראשית ל- i אנו מכניסים את הערך השלם שתיים. אחר אנו מכניסים ל- bv את הערך השקול לערכו של i, כלומר את הערך USER_BV. בשל שקילות זאת אנו רשאים גם להדפיס את ערכו של משתנה מטיפוס enum, והערך שיוצג יהיה הערך השלם השקול לערכו של המשתנה.

שימוש אפשרי אחר למשתנים מטיפוס enum עשוי להיות בתכנית הכוללת תפריטים, מתוכם צריך המשתמש לבחור אפשרות כלשהי. לדוגמה: נניח כי על המשתמש להזין אחד אם ברצונו שהתכנית תציג את סכומם של a, b, עליו להזין שתיים אם ברצונו לראות את הפרשם של המשתנים, וכו'. המשתנה אשר מכיל את הבקשה של המשתמש עשוי להיות מטיפוס enum. הקידוד יעשה באופן הבא:

א. באזור הגדרת הטיפוסים, מתחת לאזור הגדרת הקבועים, נגדיר את הטיפוס הרצוי:

```
enum requests_t {ADD_RQ, SUB_RQ, MUL_RQ, DIV_RQ } ;
```

ב. בגוף התכנית נגדיר משתנה מטיפוס requests:

```
enum requests_t the_request;
```

ג. לתוך משתנה זה לא נוכל לקרוא ערך ולכן נשתמש במשתנה עזר temp (שטיפוסו int) ונבצע:

```
do {
   cin >> temp ;
} while (temp < int(ADD_RQ) || temp > int(DIV_RQ)) ;

the_request = (enum requests_t) temp ;
```

נסביר: בלולאה אנו קוראים ערכך לתוך temp, עד אשר מוזן ערך שלם השקול לאחד הערכים מטיפוס enum requests_t (כלומר כל עוד הערך המוזן קטן מהערך השלם השקול לערך הקטן ביותר מטיפוס enum requests_t, או גדול מהערך השלם השקול לערך הגדול ביותר מטיפוס זה). אחרי שאנו בטוחים כי קראנו ערך שלם שניתן להמרה לערך מטיפוס enum requests_t, אנו מכניסים את הערך למשתנה the_request תוך שאנו מבצעים את ההמרה לטיפוס enum requests_t.

ד. עתה נוכל בהמשך התכנית לבצע פעולות שונות על-סמך ערכו של the_request.

נניח כי ברצוננו לכתוב קטע תכנית אשר מציג את הערכים השונים הקיימים בטיפוס requests. קטע התכנית יהיה כדלהלן:

```
for (enum requests_t r = ADD_RQ;
     r <= DIV_RQ;
     r = (enum requests_t )(int(r) +1))
   switch (r) {
      case ADD_RQ: cout << "ADD\n" ;
                   break ;
      case SUB_RQ: cout << "SUB\n" ;
                   break ;
      case MUL_RQ: cout << "MUL\n" ;
                   break ;
      case DIV_RQ: cout << "DIV\n" ;
                   break ;
   }
```

נסביר: פקודת ה- ++ מוגדרת על משתנים מטיפוס int, אך לא על משתנים מטיפוס enum requests_t (או מכל טיפוס בר מניה אחר), לכן בתום כל איטרציה בלולאה אנו מגדילים את r לערך הבא על-ידי קבלת הַערך השלם השקול לערכו, הגדלת אותו ערך באחד, המרת אותו ערך חזרה לערך מטיפוס enum requests_t ולבסוף הכנסת הערך הרצוי חזרה ל- r. בגוף הלולאה הדרך היחידה להדפיס את הסטרינג המציין את ערכו של r, היא לכתוב סטרינג זה מפורשות בעצמנו (וכמובן שבאחריותנו גם לעדכן את פקודת הפלט במידה וחל שינוי בהגדרת הטיפוס).

אנו רשאים להגדיר טיפוס enum גם באופן הבא:

```
enum e_t {E1 = 17, E2, E3, E4 = 18, E5 } ;
```

הטיפוס e_t כולל חמישה ערכים אפשריים: E1, E2, E3, E4, E5. הערך E1 שקול לערך השלם 17, לפיכך (מכיוון שלא נאמר אחרת) E2 שקול ל- 18, ו- E3 ל- 19. E4 שקול גם הוא ל- 18 (שכן כך נקבע מפורשות, ואין בכך כל פגם), ו- E5 שקול ל- 19.

טיפוסי enum קיימים הן ב- C++, והן בשפת C. בשפת C הם למעשה קבוצה של ערכים שלמים.

6.6 מערכים תלת-ממדיים

מערך דו-ממדי, כגון int a[5][10], הוא ליתר דיוק, מבחינת השפה, סדרה של מערכים חד-ממדיים. במקרה שלנו המערך הדו-ממדי הוא למעשה חמישה מערכים חד-ממדיים (כל-אחד בו עשרה תאים). חשיבותו של עקרון זה תודגש לנו עת נלמד פונקציות. בכל מקרה, בשפת C/C++ כל מערך הוא חד-ממדי, תמיד. אולם תא במערך החד-ממדי עשוי בעצמו להיות מערך, וכך מקבלים מערכים ממדים גבוהים יותר. לדוגמה המערך a[5][10] הוא מערך חד-ממדי בן חמישה תאים, כל אחד מהם הוא מערך (חד-ממדי) בן עשרה תאים.

כמו שעברנו ממערך חד-ממדי למערך דו-ממדי, אנו יכולים להמשיך ולהגדיל את מספר ממדי המערך כרצוננו. לדוגמה: נניח כי הכתה שלנו, אשר בתחילה למדה רק תנ"ך, ועל-כן נזקקה למערך חד-ממדי, ואחר עברה ללמוד מקצועות נוספים, ולכן נזקקה למערך דו-ממדי, היא למעשה רק כתה אחת מתוך מחזור הכולל מספר כיתות. ברצוננו לשמור את ציוני כל התלמידים במחזור. כיצד נעשה זאת? אנו זקוקים לסדרה של מערכים דו-ממדיים, או במילים אחרות: למערך תלת-ממדי. נגדיר את המערך באופן הבא:

```
int grades[MAX_CLASSES][MAX_STUD][MAX_COURSE] ;
```

קיבלנו מערך שהפניה אליו היא באמצעות שלושה אינדקסים: האינדקס הראשון מציין את הכתה, השני את התלמיד, והשלישי את המקצוע. לכן אם בהמשך נכתוב: grades[2][17][7] אנו פונים בכתה מספר שתיים, עבור התלמיד מספר 17, לציון במקצוע מספר 7. מבחינה ציורית ניתן לחשוב על המערך התלת-ממדי כמעין תיבה המורכבת מפרוסות פרוסות, כאשר כל פרוסה היא מערך דו-ממדי. הפרוסה הראשונה מתייחסת לכתה מספר אפס, הפרוסה השנייה מתייחסת לכתה מספר אחד, וכך הלאה.

כמו במקרה של מערך דו-ממדי, גם במערך תלת-ממדי עלינו להקפיד לפנות תמיד עם האינדקס המתאים במרכיב המתאים. נניח כי ברצוננו לכתוב לולאה אשר קוראת מהמורה ארבעה נתונים המציינים את: מספר הכתה, מספר התלמיד בכתה,

מספר המקצוע, וציון. התכנית תעדכן לתלמיד המבוקש, במקצוע המבוקש, את הציון שהוזן. (בעבר ראינו דוגמה דומה לזאת עבור מערך דו-ממדי):

```
while (1) {
   cin >> class ;
   if (class == -1) break ;
   cin >> stud >> course >> grade ;
   if (class >= 0 && class < MAX_CLASS &&
       stud >= 0 && stud < MAX_STUD &&
       course >= 0 && stud < MAX_COURSE &&
       grade >= 0 && grade <= 100)
            grades[class][stud][course] = grade ;
}
```

נציג דוגמה שנייה: נניח כי ברצוננו להציג את ממוצעי כל-אחת ואחת מהכיתות, בַמקצוע שאת מספרו נקרא מהמשתמש:

```
do {
   cin >> course ;
while (course < 0 || course >= MAX_COURSES) ;


// go over the classes
for (class = 0; class < MAX_CLASS; class++)
{
   sum = 0 ;
   // go over the students in the current class
   for (stud = 0; stud < MAX_STUD; stud++)
      sum += grades[class][stud][course] ;
   cout << "Average of class #" << class << " = "
        << sum/MAX_STUD << endl ;
}
```

נסביר: ראשית אנו קוראים את מספר המקצוע, עד אשר מוזן מספר מקצוע חוקי. עתה אנו נכנסים ללולאה כפולה: הלולאה החיצונית עוברת על הכיתות, הלולאה הפנימית סוכמת את הציונים באותה כיתה, במקצוע המבוקש, ואחר מציגה הלולאה החיצונית את ממוצע ציוני התלמידים באותה כיתה. שימו לב כי אנו מקפידים שהאינדקס הראשון יציין את הכתה, השני את התלמיד, והשלישי את המקצוע בהתאמה לדרך בה הגדרנו את המערך *ובלי קשר לשאלה איזו לולאה פנימית יותר, ואיזה חיצונית יותר*.

6.7 איתחול מערכים

בעבר ראינו כי משתנים פרימיטיביים ניתן לאתחל בהגדרה (לדוגמה: `int a = 17, b = 3879, c;`). גם מערכים ניתן לאתחל בהגדרה. אנו רשאים להגדיר:

```
int a[5] = {1, 1, 0, 2, 2 } ;
```

בכך הגדרנו מערך של חמישה תאים שלמים, תאים אפס ואחד במערך אותחלו לערך 1, תא מספר 2 אותחל לערך 0, ותאים מספר שלוש וארבע אותחלו לערך 2.

אנו רשאים לאתחל את המערך גם באופן הבא:

```
int a[5] = {1, 1 } ;
```

לשני התאים הראשונים במערך הכנסנו את הערך 1, שלושת התאים האחרונים במערך לא אותחלו. עת אנו מאתחלים רק כמה תאים ראשונים במערך ליתר התאים יוכנס הערך אפס. לכן אם אתה נדפיס את ערכו של `a[4]` יודפס הערך אפס.

ולבסוף אנו רשאים להגדיר מערך גם באופן הבא:

```
int a[] = {1, 1, 0, 2, 2} ;
```

מכיוון שלא נקבנו בגודלו של המערך במפורש (השארנו את הסוגריים המרובעים ריקים), אך איתחלנו חמישה תאים במערך, מסיק המחשב כי ברצוננו להגדיר מערך בן חמישה תאים, ולכן הגדרה הנוכחית שקולה להגדרה הראשונה שהצגנו.

כמובן שאיתחול תאי המערך לא מונע מאתנו בהמשך לשנות את ערכם של תאי המערך, הוא רק מכניס לתוכם ערך תחילי.

איתחול של מערך ניתן לבצע רק כחלק מהגדרתו. על-כן קטע הקוד הבא הוא שגוי, ולא יעבור קומפילציה:

```
int a[5] ;
a = {1, 1, 0, 2, 2} ;
```

מערך דו-ממדי `int a[2][4]` הוא מערך בן שתי שורות, וארבע עמודות. באופן מדויק יותר זהו מערך בן שני תאים, כל תא בו הוא מערך בן ארבעה תאים. איתחולו של מערך כזה יתבצע בהתאם לדרך המדויקת יותר בה תיארנו אותו:

```
int a[2][4] = { {2, 4, 6, 8}, {1, 3, 5, 7}} ;
```

לתאי השורה הראשונה (שורה מספר אפס) במערך הכנסנו את הערכים הזוגיים `2..8`, לתאי השורה השנייה הכנסנו את הערכים הפרטיים `1..7`, כלומר בתא `a[1][2]` נמצא הערך 5.

כמו עם מערך חד-ממדי גם מערך דו-ממדי ניתן להגדיר ללא ציון גודלו, ואז הגודל יוסק על-פי האיתחול:

```
int a[][] ={ {2, 4, 6, 8}, {1, 3, 5, 7}} ;
```

גם במערך דו-ממדי ניתן לאתחל רק את תחילת המערך, ואז ליתר התאים יוכנס הערך אפס. בפרט, הפקודה:

```
int arr[10][5] = { {0} } ;
```

תאפס את כלל תאי המערך. לתא #0 בתא #0 (שהינו מערך חד-ממדי) מוכנס הערך אפס מפורשות, ולכל יתר התאים מוכנס הערך אפס כתוצאה מכך.

באופן דומה הפקודה `memset` 'מורחת' ערך כלשהו על סדרה של בתים. גם בלי להבינה לעומר נקל להשתמש בה כדי לאפס מערך:

```
int arr[10][5] ;

memset(arr, 0, sizeof(arr)) ;
```

בקצרה: על שטח הזיכרון שמוקצה למערך `arr` , ושגודלו בבתים הוא `sizeof(arr)` בתים, אנו רוצים 'למרוח' את הערך אפס. על-מנת להשתמש בפקודה `memset` אין צורך בפקודות `include` או `using` כלשהן. אנו אומרים שזוהי *פונקציה מובנית* (`built in function`) של השפה.

6.8 <u>תרגילים</u>

התרגילים המופיעים להלן מתחלקים לשני סוגים:

א. חלקם (תרגילים מספר אחד ומספר ארבע) לקוחים מתוך בחינות שניתנו במוסדות להשכלה גבוהה שונים. הם תרגילים קטנים למדי אך לא קלים. בפרק שבע תמצאו תרגילים נוספים דומים להם.

ב. חלקם האחר (תרגילים מספר שתיים, שלוש וחמש) כוללים תכניות ארוכות למדי. למעשה ראוי שלא לכתוב תכניות כל-כך גדולות ללא שימוש בפונקציות (נושא שיילמד בפרק שבע), אלא לשם ההדגמה עד כמה כתיבת תכנית גדולה ללא שימוש בפונקציות היא רעיון קלוקל. אולם בצוק העיתים, וכדי להטמיע את נושא המערכים, ומכיוון שלשיטתי ראוי לדחות את העיסוק בפונקציות עד שהתלמידים יטמיעו נושאים פחות מעמיסים, אני ממליץ לכם בכל אופן להתמודד עם תכניות אלה. יש לקחת בחשבון שעֵת תכנית כזאת ניתנת כתרגיל במוסד להשכלה גבוהה, מוקצה לכתיבתה משך של כשבוע ימים.

דין דומה חל עם התרגילים המופיעים בפרקים הבאים.

6.8.1 תרגיל מספר אחד: איתור מסלול במערך

בתכנית כלשהי מוגדרים:

```
const int M= …, N = … ;
int matrix[M][N] ;
```

נאמר שתא [row1][col1] (במערך matrix) **<u>סמוך</u>** לתא אחר [row2, col2]
אם $|row1 - row2| \le 1$ וכן $|col1 - col2| \le 1$.
נתון הכלל הבא: במערך matrix ניתן **<u>להתקדם</u>** מתא [row1, col1] לתא סמוך [row2, col2] אם: matrix[row2, col2] = matrix[row1, col1] + 1 .
<u>מסלול</u> במערך מתחיל מתא כלשהו ומתקדם מתא לתא סמוך לפי הכלל הנ"ל.

<u>לדוגמא</u> במערך: 5 4 1 קיים מסלול אחד הכולל את המספרים: 3, 4, 5
3 7 8 ומסלול שני הכולל את הערכים: 7, 8, 9, 10.
17 10 9

כתבו תכנית הקוראת מהמשתמש נתונים לתוך מערך כדוגמת matrix ואחר:

א. בודקת שכל נתון מופיע במערך פעם יחידה (כלומר שלא קיים נתון שמופיע במערך יותר מפעם אחת). במידה והבדיקה לא עברה בשלום הודיעו באילו תאים נמצא נתון זהה, ועֲצרו את ביצוע התכנית.

ב. במידה והבדיקה מסעיף א' עברה בהצלחה, אתרו והציגו את המסלול הארוך ביותר הקיים במערך. במידה וקיימים מספר מסלולים ארוכים ביותר יש להציג את המסלול שמתחיל בתא שמספרו נמוך ביותר, כלומר את המסלול המתחיל בשורה שמספרה מזערי ובעמודה שמספרה מזערי.

6.8.2 תרגיל מספר שתיים: משחק אווירונים וצוללות

כתבו תכנית אשר תאפשר למשתמש לשחק את המשחק 'אווירונים וצוללות'.

התכנית תחזיק מערך דו-ממדי בגודל NxN. ותמקם בו את הצוללות הבאות:

א. ארבע צוללות שצורתן X, כלומר כאלה שתופסות תא יחיד.

ב. שלוש צוללות שצורתן XX, כלומר כאלה שתופסות זוג תאים סמוכים (זה לצד זה או זה מעל זה).

X

ג. שלוש צוללות שצורתן XXXX כלומר צלב במורכב מ-'גוף' באורך ארבעה תאים, ו-'כנף' הניצבת לגוף, X באורך של שלושה תאים.

ד. שתי צוללות שצורתן XXXX (כלומר כאלה שתופסות ארבעה תאים אופקיים או ניצבים).
גודלן של הצוללות (במילים אחרות אורכן של הצוללות מסעיפים א׳, ב׳, ד׳, ואורך כל ישר המרכיב את הצלב בצוללות בצורת צלב) ייקבע באמצעות קבועים. לגבי הצוללת בצורת צלב ניתן להניח כי הגוף הוא בן לפחות שני תאים, הכנף היא באורך פרדי, וכוללת לפחות שלושה תאים. הכנף תמיד תבלוט במידה סימטרית משני צדי הגוף, ומקום המפגש בין הכנף לגוף יהיה בתא השני מתחילת הגוף (בכיוון הציור של הגוף).

בין כל שתי צוללות יפריד לפחות תא ריק אחד מעלה, מטה, ימינה ושמאלה, אך לא בהכרח באלכסון. כלומר לא תתכנה שתי צוללות באורך אחד הניצבות זו צמודה לזו בצורה כזו: XX, אך כן תיתכנה שתי צוללות כנ״ל הניצבות זו בסמוך לזו בצורה:
– X
X -

התכנית תגריל את מיקומן של הצוללות באופן הבא: עבור הצוללות מסעיפים א׳, ב׳, ו-ד׳ תוגרל נקודת התחלה וכיוון הציור (הכיוון עשוי להיות מעלה, מטה, ימינה או שמאלה), עבור הצוללות המתוארות בסעיף ג׳ תוגרל נקודת המפגש בין הישרים, והכיוון (שעשוי להיות מעלה, מטה, ימינה או שמאלה). במידה והדבר אפשרי אזי הצוללת המתאימה תצויר; במידה וציור הצוללת יגרום לחריגה מהמערך, או למפגש עם צוללת אחרת אזי הצוללת לא תצויר, והמחשב ינסה להגריל שוב מיקום וכיוון לצוללת. במידה והמחשב ניסה TRIALS פעמים להציב צוללת ונכשל בכך, אזי הוא יודיע על כך למשתמש וביצוע התכנית ייעצר.

בהנחה שהמחשב הצליח להציב את הצוללות על הלוח כנדרש הוא יתקדם לשלב הניחושים. בשלב זה יזין המשתמש שוב ושוב קואורדינטות של תא במערך, והמחשב יודיע לו האם הוא פגע (או לא פגע) בצוללת כלשהי. במידה והמשתמש פגע בצוללת, ובכך סיים להשמידה, יודיע המחשב על כך בנפרד. כמו כן בתום כל סיבוב יציג מחשב למשתמש את מצב הלוח. תאי הלוח יתוארו באופן הבא:

א. תא שטרם נוחש יוצג באמצעות סימן שאלה.

ב. תא שנוחש ולא היה בו דבר יוצג באמצעות מקף.

ג. תא שנוחש ושהינו חלק מכלי שטרם הושמד לגמרי יוצג באמצעות התו פלוס.

ד. תא שנוחש ושהינו חלק מכלי שהושמד לגמרי יוצג באמצעות התו כוכבית.

כמו כן יוצג כמה כלים מכל סוג הושמדו לגמרי, וכמה טרם הושמדו לגמרי (כלומר הם לא נפגעו כלל, או שהם נפגעו חלקית אך טרם הושמדו).

בעת שהמשתמש השמיד את כל הכלים שהיו על הלוח הוא יקבל הודעה כי המשחק תם; כי על הלוח היו M תאים בהם היו צוללות, מתוך N תאים מהם מורכב הלוח; וכי למשתמש נדרשו K ניחושים כדי להשמיד את כל צי הצוללות שהיה על-גבי הלוח.

6.8.3 תרגיל מספר שלוש: תכנית CAD-CAM פרימיטיבית

בתרגיל זה עליכם לכתוב תכנת CAD-CAM פרימיטיבית.
התכנית תחזיק מערך דו-ממדי של משתנים בולאניים בגודל MxN. המערך יאותחל לכדי מטריצה ריקה.
התכנית תאפשר למשתמש לבצע את הפעולות הבאות:

א. שרטוט קו ישר במערך מהתא `x0, y0` לתא `x1, y1`.

ב. מחיקת קו ישר במערך מהתא x0, y0 לתא x1, y1.
ג. שרטוט מלבן מלא במערך. מלבן שפינתו השמאלית העליונה מצויה בתא x0, y0 ופינתו הימנית התחתונה מצויה בתא x1, y1.
ד. מחיקת מלבן מלא כנ"ל.
ה. שרטוט קווי המתאר של מלבן כנ"ל (ללא מילוי הפנים).
ו. מחיקת קווי המתאר של מלבן כנ"ל (ללא הפנים).
ז. שרטוט מעגל מלא שמרכזו בתא x0, y0 ורדיוסו הוא r. כל התאים שמרחקם מהתא x0, y0 קטן או שווה מ- r יצבעו.
ח. מחיקת מעגל כנ"ל.
ט. הצגת מצב המערך.
י. סיום.

6.8.4 תרגיל מספר ארבע: איתור תת-מערך רצוי בתוך מערך דו-ממדי

כתבו תכנית הקוראת נתונים לתוך מערך דו-ממדי של מספרים שלמים int matrix[M][N], ואחַר מאתרת תת-מערך גדול ביותר המצוי ב- matrix ומקיים שאברי תת-המערך הינם בסדר עולה; כלומר, אם הפינה השמאלית העליונה של תת-המערך מצויה בתא [row][col] במערך matrix, ואם גודלו של תת-המערך הוא m x n , אזי לכל תא שאינו אחרון בשורה בתת-המערך מתקיים a[i][j]<a[i][j+1] ולתא האחרון בכל שורה מתקיים שהוא קטן מהתא הראשון בשורה שאחריו בתת-המערך, כלומר: a[i][col+n-1]<a[i+1][col]). יש להציג את הפינה השמאלית העליונה של תת-המערך, את מספר השורות ומספר העמודות שהוא כולל. במידה וקיימים כמה תתי-מערך מקסימאליים בגודלם יש להציג את הראשון ביניהם.

לדוגמה עבור המערך:

7	4	2	0	1
8	***1***	***2***	***3***	5
2	***7***	***9***	***11***	13
1	***12***	***14***	***28***	30

יוצג כי התא [1][1] מהווה פינה שמאלית עליונה של תת-מערך מבוקש בגודל 3x3.

6.8.5 תרגיל מספר חמש: פולינומים

פולינום הוא ביטוי מהצורה: $y = a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x^1+a_0$,
לדוגמה: $y = 2x^5 +0x^4+0x^3+(-2)x^2+0x^1+17x^1$. ביטוי כדוגמת a_ix^i נקרא מונום (לדוגמה: $17x^1$). בעת שאנו כותבים פולינום נהוג להשמיט מונומים שערכו של המקדם בהם הוא אפס. על-כן את הפולינום שהצגנו נכתב בדרך-כלל בצורה הבאה: $y = 2x^5 -2x^2+17x^1$.

על פולינומים מוגדרות מספר פעולות:
א. הצבה של ערך ממשי במקום המשתנה x, וחישוב ערך y המתאים לאותו ערך x.
ב. גזירה של הפולינום, וקבלת פולינום הנגזרת. הנגזרת של פולינום: $y = a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x^1+a_0$ הוא פולינום: $y = na_nx^{n-1}+(n-1)a_{n-1}x^{n-2}+\ldots+a_1x^0$.
לדוגמה, הנגזרת של הפולינום: $y = 2x^5 +0x^4+0x^3+(-2)x^2+0x^1+17x^1$ הוא הפולינום: $y = 10x^4 -4x +17$.

ג. חישוב האינטגרל של פולינום p בקטע x0..x1. ראשית יש לחשב את הפולינום P המקיים שהנגזרת שלו היא p. שנית יש להחסיר את תוצאת ההצבה של x0 ב- P מתוצאת ההחסרה של x1 ב- P.

ד. חיבור זוג פולינומים וקבלת פולינום הסכום. סכומם של הפולינומים: $y = a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x^1+a_0$ ו: $y = b_nx^n+b_{n-1}x^{n-1}+\ldots+b_1x^1+b_0$ הוא הפולינום: $y = (a_n+b_n)x^n+(a_{n-1}+b_{n-1})xn^{-1}+\ldots+(a_1+b_1)x^1+(a_{0+}b_0)$ לדוגמה: סכומם של הפולינומים: $y = 2x^5 -2x^2+17x^1$ ו: $y = 17x^7+2x^2+5x^1+3$ הוא הפולינום: $y = 17x^7+2x^5+22x^1+3$.

ה. כפל פולינום במונום, וקבלת המכפלה. מכפלתו של הפולינום: $y = a_nx^n+a_{n-1}x^{n-}$ $^1+\ldots+a_1x^1+a_0$ במונום: b_mx^m הוא הפולינום: $y = (a_n*b_m)x^{n+m}+(a_{n-1}*b_m)x^{n-}$ $^{1+m}+\ldots+(a_1*b_m)x^{1+m}+(a_0*b_m)x^m$. לדוגמה: מכפלתו של הפולינום: $y = 2x^5 - 2x^2+17x^1$ במונום $3x^4$ הוא הפולינום: $y = 6x^9 -6x^6+51x^5$.

ו. חיסור שני פולינומים וקבלת פולינום ההפרש. חישוב ההפרש p2-p1 מתבצע באופן הבא:

1. כפול את הפולינום p2 במונום $-1x^0$ (כלומר כפול כל מקדם של p2 ב: -1).
2. חבר את הפולינום שהתקבל בסעיף 1' עם p1.

ז. כפל זוג פולינומים, וקבלת המכפלה. מכפלת הפולינום p1 שהינו: $y = a_nx^n+a_{n-1}x^{n-}$ $^1+\ldots+a_1x^1+a_0$ בפולינום: $y = b_mx^m+b_{m-1}x^{m-1}+\ldots+b_1x^1+b_0$. מחושבת באופן הבא:

1. חשב את סדרת המכפלות: $p1*b_ix^i$ (עבור i= m..0).
2. סכום את הפולינומים שחושבו בסעיף 1.

ח. חילוק זוג פולינומים וקבלת המנה: מנת החלוקה של הפולינום p1 בפולינום p2 היא הפולינום p3 עם שארית שהינה הפולינום p4 אם p1 = p3*p2 +p4 (כפי שמנת החלוקה של 11 ב- 2 היא 5 עם שארית 1 שכן: 11 = 1+ 2*5). אלגוריתם החלוקה של פולינומים דומה לחלוקה בשלמים, ומתנהל באופן הבא:

1 נניח כי p1 הוא: $y = a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x^1+a_0$ ו- p2 הוא: $y = b_mx^m+b_{m-1}x^{m-}$ $^1+\ldots+b_1x^1+b_0$ אזי המנה פולינום המנה שיסומן qout ופולינום השארית שיסומן rem יאותחלו להיות הפולינום 0 (במילים אחרות: $0x^0$).

2 כל עוד דרגתו של p1 גדולה מדרגתו של p2 בצע:

א. יהי a_ix^i המונום שדרגתו גבוהה ביותר ב- p1.

ב. חשב: $curr\_mon = a_i/b_i\ x^{i-m}$.

ג. חבר את curr_mon ל- quot. (הדבר שקול להוספת ספרה נוספת למנה בעת חילוק מספרים שלמים).

ד. החסר מ- p1 את p2*curr_mon. (הדבר שקול להקטנת המחולק בשיעור: הספרה הנוספת למנה כפול המחַלק, בחילוק בשלמים).

3. בתום הלולאה מסעיף 2' הנ"ל מכיל הפולינום quot את המנה, ו- p1 את השארית.

ט. חישוב הפולינום p ששורשיו הם $c_1,\ldots,c_n$. הפולינום p יחושב באופן הבא:

1. אתחל את p להיות: $x-c_1$.
2. חזור n-1 פעמים: עדכן את p להיות המכפלה של p בפולינום $x-c_i$ עבור i=2..n.

ניתן לייצג פולינום באמצעות מערך חד-ממדי באופן הבא: התא מספר i במערך ישמור את מקדם של x^i. לדוגמה: הפולינום $y = 2x^5 -2x^2+17x^1$ ייוצג באופן הבא:

0	17	2-	0	0	2
0	1	2	3	4	5

כתבו תכנית אשר מאפשרת למשתמש לבצע את הפעולות הבאות:

א. הזנת פולינום למאגר הפולינומים שהתכנית מחזיקה (המשתמש יזין את דרגת הפולינום, ואחר את מקדמי הפולינום מגבוה לנמוך, כולל מקדמים שערכם אפס).

ב. הצגת הפולינומים השמורים במאגר הפולינומים.

ג. הצגת תוצאת ההצבה של ערך c כלשהו בפולינום רצוי כלשהו המצוי במערך.

ד. גזירת פולינום רצוי כלשהו.

ה. חישוב האינטגרל של פולינום רצוי כלשהו בקטע x0..x1.

ו. הצגת הסכום, ההפרש, המכפלה, המנה+השארית של זוג פולינומים כלשהו (המשתמש יוכל לבקש כל אחת מארבי הפעולות בנפרד).

ז. הצגת הפולינום ששורשיו הם $c_1,\ldots,c_n$. יוזן n ואחר השורשים הרצויים.

ח. סיום.

מבנה הנתונים המרכזי שהתכנית תנהל הוא מערך דו-ממדי של ממשיים, אשר יאפשר לשמור קבוצה של פולינומים.

תנו דעתכם גם להצגה נאה של הפלט, כולל החזקות x^0, x^1, ומקדמים שליליים, או כאלה שערכם אפס.

6.8.6 תרגיל מספר חמש: מציאת ערך המצוי בכל שורות מערך

כתבו תכנית הקוראת ראשית נתונים לתוך מערך דו-ממדי. על הנתונים המוזנים לקיים כי הנתונים המופיעים בכל שורה מסודרים מקטן לגדול. במידה והנתונים המוזנים אינם כנדרש תוצג הודעת שגיאה, וביצוע התכנית יופסק.

במידה והנתונים שהוזנו הינם כנדרש תאתר התכנית *באופן יעיל* נתונים המופיעים בכל שורות המערך, ותציגם.

לדוגמה, עבור המערך:

3879	3879	17	5	5-
3879	16	15	14	5-
3879	3879	5-	10-	10-
6000	6000	6000	3879	5-

יוצגו: 3879 ו- 5-.

<u>רמז</u>: החזיקו מערך עזר חד-ממדי שישמור באיזה עמודה אתם מצויים בכל שורה של המערך הדו-ממדי. אתחלו את כל תאי מערך העזר לערך אפס, וקדמו את הערכים בתאי מערך העזר במידה ובשורה המתאימה מצוי ערך שהינו קטן מערך המצוי בשורה אחרת במערך במקום בו אתם ניצבים באותה שורה.

7 פונקציות

על-פי ניסיוני, פונקציות הוא נושא שגורם לתלמידים לקושי רב עד אשר הם מטמיעים את שיטפון המושגים החדשים הניחתים על ראשם. מסיבה זאת אני נוטה לדחות את הוראת הנושא, וללמד ראשית פקודות בקרה, ומערכים. להתרשמותי הקושי הנגרם בהטמעת נושא זה אינו תוצאה של האתגר האינטלקטואלי הרב שמציבות הפונקציות, אלא נגרם מהמונחים הרבים הכרוכים בתכנות עם פונקציות. הנחמה היא שבסופו של דבר מרבית הכוח משתלט על היעד.

לפונקציות יש שמות רבים ושונים: פונקציות, רוטינות (routine), סברוטינות (subroutine), שגרות, פרוצידורות (procedure), שיטות (methods).

נציין כי אם מרכיבי השפה שלמדנו עד עתה אפשרו לנו לכתוב תכניות *כהלכה*. על כן לא ניתן להפריז בחשיבות הנושא. ראשית נכיר אותו: נראה מה הכלי מעמיד לרשותנו ברמה הטכנית; רק אחר-כך נדון בחשיבותו, ונסביר כיצד יש לתכנת עם פונקציות. בכמה מלים, רק כדי לתת לכם על קצה המזלג את הרעיון הבסיסי אומר שעת אנו עובדים עם פונ' אנו משתדלים להקצות לכל תת-משימה של התכנית פונ' נפרדת שתהיה אמונה על השלמתה. באופן כזה התכנית הופכת להיות מודולארית, כלומר מורכבת מאוסף של מודולים, כל אחד, כאמור, מבצע תת-משימה של התכנית. באופן זה, ראשית, נקל יותר להבין מה התכנית, או מה כל קטע שבה, מבצע; ושנית, אם בשתי תכניות עלינו להשלים אותה תת-משימה נוכל להשתמש באותה פונ' אשר תיכלל בשתי התכניות, כך לא נצטרך בכל פעם 'להמציא את הגלגל מחדש'.

7.1 דוגמות ראשונות

נתחיל את היכרותנו עם נושא הפונקציות בסדרה של דוגמות פשוטות שיבהירו (?) את ההיבטים השונים של תכנות עם פונקציות. חלק מהדוגמות שנציג הן בבחינת שימוש קלוקל בפונקציות, ואחרות הן בגדר שימוש מיותר בפונקציות, באשר המשימה שהפונקציה מבצעת אינה מצדיקה פונקציה ייחודית; נציג אותן רק לצרכים דידקטיים.

7.1.1 דוגמה ראשונה לשימוש בפונקציה: זימון פונקציה, ביצועה, וחזרה ממנה

נציג דוגמה ראשונה לשימוש בפונקציה. כהרגלנו, נציג את הקוד ואחר נסבירו:

```
#include <iostream>
#include <stdlib>                  // for EXIT_SUCCESS

using std::cout ;

//-----------------------------------------
int main()
{
   cout << "A first output\n" ;
   f() ;
   cout << "A third output\n" ;
   f() ;
   cout << "A fifth output\n" ;
```

```
   return(EXIT_SUCCESS) ;
}
//---------------------------------------
void f()
{
   cout << "An output from f\n" ;
}
```

תכניתנו כוללת הפעם שתי פונקציות: הראשונה מוכרת לנו מימים ימימה: main, ממנה מתחיל ביצוע התכנית. השנייה היא הפונקציה f, אשר על אופן שילובה בתכנית נעמוד מייד. בכל תכנית בשפת C/C++ תהיה הפונ' main, ותמיד המעבד יתחיל את הרצת התכנית מפונקציה זאת. פונ' אחרות תכללנה בתכנית על-פי שיקול דעתנו. עת המעבד מתחיל בביצוע התכנית (כאמור, תמיד מה- main) הוא פוגש בפקודת הפלט: cout << "A first output\n" ; ומציג את הפלט המתאים. אחר הוא עובר לפקודה: f(); פקודה זאת **קוראת** לפונקציה f (במילים אחרות: **מזמנת** את הפונקציה f). משמעות הדבר היא כי עתה המעבד פונה לביצוע הפקודות הנכללות בפונ' f. בדוגמה שלנו f כוללת רק פקודה אחת. לכן עתה המעבד מבצע את פקודת הפלט: cout << "An output from f\n" ; לו היו ב- f פקודות נוספות הן היו מבוצעות בזו אחר זו, עד שהמעבד מגיע לסיומה של הפונקציה f. עת המעבד מסיים לבצע את f, הוא זוכר כי הוא נקרא ל- f מהתכנית הראשית, מהפקודה השניה ב- main, ולכן עת הוא מסיים את ביצועה של f עליו לחזור לתכנית הראשית, לפקודה שמיד אחרי זימונה של f, במקרה שלנו לַפקודה: cout << "A third output\n" ;. עתה מתקדם המחשב לזימון נוסף של f. שוב קורה מצב דומה: המחשב מסתעף לביצוע הפונ', שולח ממנה את הפלט המתאים, ועת הוא מגיע לסוגריים שמסיימים את הפונ' הוא זוכר מהיכן הוא הסתעף לפונ', ועל כן הוא חוזר לפקודת הפלט cout << "A fifth output\n" ;

הערות נוספות אודות התכנית שכתבנו:

א. קטע הקוד:

```
void f()
{
   cout << "A second output\n" ;
}
```

נקרא **הגדרת הפונקציה** f (function definition), שכן בקטע זה אנו מגדירים (במילים אחרות קובעים) מה הפונקציה מבצעת. בניגוד לכך הפקודה: f(); בתכנית הראשית, מהווה **קריאה** (או **זימון**) לפונקציה f.

ב. בהגדרת הפונקציה, לפני שמה, מופיעה המילה השמורה void (שמשמעה 'כלום'). נשים לה שה- void כאן, מקביל ל- int המופיע לפני שם הפונ' main. על פישרה של מילה זאת נעמוד בהמשך. לעת עתה נתייחס אליה כחלק מהגדרת הפונקציה.

ג. על-מנת שהתכנית שכתבנו תהיה שלמה, וניתנת לקומפילציה ולהרצה, עלינו להוסיף לפני התכנית הראשית (אחרי הגדרת הקבועים וטיפוסי enum) את השורה:

```
void f() ;
```

שורה זאת נקראת **ההצהרה על הפרוטוטיפ** (prototype) של f (יש הקוראים לכך: **החתימה** (signature) של f. כְּלל, הפרוטוטיפ כולל את השורה הראשונה בפונקציה בתוספת נקודה-פסיק (;).

ד. למעשה כבר בעבר השתמשתם בפונקציות בלי לתת על-כך את הדעת. למשל עת כתבתם את הפקודה: `srand(…)` זימנתם את הפונקציה `srand`. פונקציה זאת לא נכתבה על-ידנו, והיא גם אינה מציגה כל פלט, לכן אין לנו כל מושג על הנעשה בה, אולם כעיקרון היא אינה שונה מכל פונקציה שאנו כותבים. כמו בפונקציות שאנו כותבים, עת אנו קוראים לפונקציה `srand()` המעבד פונה לביצועה של אותה פונקציה, ועת ביצוע הפונקציה מסתיים המעבד חוזר לבצע את הפקודה המופיעה בתכנית אחרי הפקודה `srand()`.

7.1.2 דוגמה שניה לשימוש בפונקציה: פונקציה הכוללת משתנים לוקליים

עת אנו כותבים תכנית המשתמשת בפונקציות, נייעד לכל פונקציה תת-משימה מתוך המשימה הכללית שעל התכנית לבצע. לכן נניח עתה (מטעמים דידקטיים) כי חיבור שני מספרים, והצגת סכומם היא תת-משימה ראויה, אשר זכאית לפונקציה ייעודית משלה. נציג תכנית הכוללת פונקציה כמבוקש:

```
//-----------------------------------------------
#include <iostream>
#include <cstdlib>

//-----------------------------------------------
using std::cout ;

//-----------------------------------------------
void sum1() ;                          // prototype of sum1()
//-----------------------------------------------
int main()
{
   cout << "program execution starts here\n" ;
   sum1() ;
   cout << " program execution ends here\n" ;

   return(EXIT_SUCCESS) ;
}
//-----------------------------------------------
void sum1() {
   int num1, num2 ;

   cin >> num1 >> num2 ;
   cout << num1 + num2 << endl ;
}
```

נסביר: כמו בדוגמה הקודמת, ביצוע התכנית מתחיל בפקודת פלט (המופיעה בתכנית הראשית). עתה התכנית הראשית מזמנת את הפונקציה `sum1`, ולכן המעבד פונה לביצועה של פונקציה זאת, (תוך שהוא זוכר שאחרי ש- `sum1` תסתיים עליו לחזור לַתכנית הראשית לַפקודה שמייד אחרי הקריאה ל- `sum1`). בַּפונקציה `sum1` מוגדרים זוג משתנים: `num1, num2`. משתנים אלה מוכרים *רק* לפונקציה זאת. אנו אומרים כי הם **לוקליים** (local variables) לפונקציה. במילים אחרות לו בתכנית הראשית היינו כותבים: `num1 = 17;` אזי התכנית שלנו לא הייתה עוברת קומפילציה, תוך שהמהדר היה מודיע כי המשתנה `num1` אינו מוגדר בתכנית

הראשית. בשפת C משתנים לוקליים כדוגמת num1, num2 נקראים גם **משתנים אוטומטיים**. ביצועה של sum1 מתחיל בקריאת ערכים לזוג המשתנים, ואחר המחשב מציג את סכום הערכים שנקראו. בזאת מסתיים ביצועה של sum1, המעבד חוזר לתכנית הראשית, ומציג את פקודת הפלט המסיימת את ה- main.

אוסיף כי לו גם ב- main היו מוגדרים משתנים: int num1, num2 ; אזי ראשית לא היה בכך כל פְּסול (גם לא מבחינת הסגנון התכנותי), ושנית לא היה כל קשר בין משתני ה- main לאלה של הפונ' sum1, כלומר לו היינו מאתחלים את המשתנה num1 של התכנית הראשית לערך 17 אזי עת היינו נכנסים לפונ' המשתנה (השונה לגמרי) של הפונ' num1 היה עדיין מכיל ערך זבל. באופן דומה, לו בפונ' היינו מכניסים למשתנה num2 את הערך אפס, הדבר לא היה משפיע על המשתנה num2 של ה- main.

7.1.3 דוגמה שלישית: משתנים גלובליים (כאמצעי תקשורת בין התכנית הראשית לפונקציה)

עתה נניח כי אנו רוצים לכתוב תכנית בה התכנית הראשית היא שקוראת את הערכים לתוך זוג המשתנים num1, num2. על הפונקציה שנכתוב (שתקרא sum2) רק לבצע את משימת הצגת סכומם של המשתנים (לצורך הדיון נניח שחישוב והצגת סכום של שני משתנים הוא תת-משימה מוגדרת וראויה, אשר לכן זוכה לפונ' ייעודית אשר תבצע אותה).

אם נגדיר את המתשנים num1, num2 בתכנית הראשית, כפי שעשינו בכל התכניות שכתבנו עד לפרק זה, הם יהיו משתנים לוקליים ל- main, כלומר הם לא יהיו מוכרים לפונקציה sum2, ועל-כן היא לא תוכל לחברם. כמובן שגם הגדרת המשתנים בתוך sum2 לא תענה על צרכינו. אז מה עושים? ראשית נכיר פתרון קלוקל לסוגיה, ואחר גם פתרון ראוי. נתחיל בהצגת הפתרון הקלוקל. מסיבות שתובהרנה בהמשך נשתמש בשלושה משתנים num1, num2, num3:

```
#include <iostream>
#include <cstdlib>

using std::cout ;

//-----------------------------------------------
void sum2() ;
//-----------------------------------------------

int num1, num2, num3 ;
//-----------------------------------------------

int main() {
   cin >> num1 >> num2 >> num3 ;
   sum2() ;
   cout << "The End\n" ;
   return(EXIT_SUCCESS) ;
}
//-----------------------------------------------
```

```
void sum2() {
   cout << num1 + num2 << endl ;
}
```

מה עשינו? הגדרנו את המשתנים *מחוץ* לשתי הפונקציות, מעל התכנית הראשית. בכך הפכנו את המשתנים **לגלובליים** (global, כפי שבעבר עשינו עם הקבועים שהגדרנו). משמעות הדבר היא כי כל הפונקציות (כולל התכנית הראשית, וכל פונקציה אחרת) מכירות את המשתנים. ערכם של המשתנים נקבע על-פי הערך האחרון שהושם להם (בלי תלות בזהות המשים).

באיזה אופן מסייעים המשתנים הגלובליים להשגת המטרה שהצבנו לעצמנו? ביצוע התכנית מתחיל בתכנית הראשית אשר קוראת ערך לתוך המשתנים. עתה מזמנת התכנית הראשית את sum2. עת sum2 מתבצעת, בפרט עת היא מציגה את הסכום num1+num2 היא מוצאת במשתנים את הערך האחרון שהושם בהם (במקרה שלנו על-ידי התכנית הראשית), ולכן sum2 מציגה את הערך הרצוי.

אם כל-כך טוב אז מה כל-כך רע? מה שרע הוא השימוש במשתנים גלובליים. כעיקרון (שאת הסייגים לו נכיר בהמשך), שימוש במשתנים גלובליים נחשב לתכנות קלוקל! הסיבה לכך היא שעת אנו מתכנתים עם פונ' אנו מעוניינים לכתוב תכנית **מודולארית**, כלומר תכנית המורכבת מחתיכות חתיכות, באופן שאם נרצה בכך נוכל לשלוף כל חתיכה מהתכנית, ולהעבירה לתכנית אחרת, בה עלינו לבצע משימה דומה. שימוש במשתנים גלובליים פוגע במודולאריות שכן אנו כבר לא יכולים לחתוך את הפונקציה בלבד, ולשתלה בתכנית אחרת, עלינו להעביר עמה גם את המשתנים הגלובליים בהם הפונקציה עושה שימוש, וזה לא טוב.

אותו דין חל גם עם קבועים גלובליים (גם הם פוגעים במודולאריות). מדוע אם כן קבועים גלובליים אנו כן מתירים? הסיבה היא שקבועים יש יחסית מעט, ערכם נקבע בעת הגדרתם (ולא בכל מיני מקומות שונים ומשונים בתכנית), ועל כן מידת הפגיעה שלהם במודולאריות יחסית מצומצמת.

הפונקציה sum2 שכתבנו מציגה את הסכום num1+num2, ומה אם ברצוננו להציג גם את num1+num3 ואת num2+num3, האם עבור כל סכום כזה נזדקק לפונקציה נפרדת? כדאי מאוד שלא.

לכן הפתרון שהצגנו בתכנית האחרונה היה לקוי ביותר. נפנה עתה להצגת פתרון ראוי.

7.1.4 דוגמה רביעית: פרמטרים לפונקציה (באמצעותם התכנית הראשית והפונקציה יכולות לתקשר)

נפנה עתה להצגת פונקציה הכתובה כהלכה, ומציגה את הפרשם של שני משתנים, לתוכם קראה התכנית הראשית ערכים (מדוגמה זאת השמטתי את פקודות ה-include, using, prototype declaration מתוך הנחה שמרכיבים אלה כבר מוכרים לכם, ותדעו להוסיפם בכוחות עצמכם):

```
int main() {
   int num1, num2, num3 ;

   cin >> num1 >> num2 >> num3 ;
```

```
   dif(num1, num2) ;
   cout << "After first call to dif\n" ;
   dif(num1, num3) ;
   cout << "After second call to dif\n" ;
   dif(num3, num1) ;
   cout << "The End\n" ;

   return(EXIT_SUCCESS) ;
}
//-------------------------------------------
void dif(int op1, int op2) {
   cout << op1 - op2 << endl ;
}
```

נסביר את התכנית: ראשית נדון בפונקציה dif. הגדרת הפונקציה dif נפתחת בשורה: void dif(int op1, int op2). בַּסוגריים, המופיעים אחרי שם הפונקציה, ושעד כה היו ריקים, מופיעות עתה המילים: int op1, int op2 פסוקית זאת מורה לנו כי לפונקציה dif יש שני **פרמטרים** (parameters ויש המדייקים ואומרים **פרמטרים פורמאליים**). במילים אחרות לפונקציה יש שתי 'תיבות דואר'. התיבה הראשונה היא מטיפוס int והיא מכונה על-ידי הפונקציה בשם op1, התיבה השנייה גם היא מטיפוס int, והיא מכונה על-ידי הפונקציה בשם op2. באמצעות תיבות הדואר מתַקשרת הפונקציה עם מי שקורא לה. באיזה אופן מתבצעת הַתקשורת? מי שרוצה לקרוא לפונקציה dif *תייב* 'לשלשל' לה ראשית לתיבות *שני* ערכים מטיפוס int. עת הפונקציה 'מתעוררת' (כלומר עת היא מזומנת) היא עושה שימוש בערכים שהיא מוצאת בתיבות הדואר. בדוגמה שלנו היא מציגה את ההפרש בין הערך שהיא מוצאת בתיבה הראשונה, פחות הערך שהיא מוצאת בתיבה השנייה. נדגיש כי מי שקורא לפונקציה dif *תייב* לספק לה *זוג* ערכים שלמים. בשפת C אנו קוראים לערכים המסופקים לפונקציה בעת שקוראים לה **ארגומנטים** (arguments, או לעיתים **פרמטרים אקטואליים**). קריאה לפונקציה בַּאופן: dif(num1) תגרום לשגיאת קומפילציה. כמו כן יש חשיבות לסדר הארגומנטים המועברים בקריאה לפונקציה, שכן הארגומנט הראשון מתאים לפרמטר הראשון, והארגומנט השני לפרמטר השני. בדוגמה שלנו הקריאה dif(17, 3) תגרום לכך שבתיבת הדואר op1 תמצא הפונקציה את הערך 17, ובתיבה op2 את הערך 3. לפיכך הפלט שהפונקציה תציג יהיה 14. לעומת זאת הקריאה dif(3, 17) תשלשל את הערך 3 לתיבת הדואר הראשונה של הפונקציה (זו המכונה op1) ואת הערך 17 לתיבה השנייה (זו המכונה op2), ולכן הפלט יהיה 14-.

עד כאן הדיון בַּפונקציה dif. נפנה עתה לבחון את התכנית הראשית. שלושת המשתנים num1, num2, num3, הם עתה משתנים לוקליים של main. התכנית הראשית, ורק היא רשאית לפנות אליהם. אולם לא אלמן ישראל: מכיוון שלפונקציה dif יש פרמטרים, יכולה התכנית הראשית לקרוא ל- dif למשל באופן: dif(num1, num2) ועל-ידי כך להציג את ההפרש num1-num2. באופן דומה הקריאה dif(num1, num3) תציג את ההפרש num1-num3, וכך הלאה: dif הפכה להיות פונקציה כללית להצגת הפרש בין שני מספרים שלמים.

7.1.5 דוגמה חמישית: שימוש פרמטרי הפניה (reference parameters)

עתה נניח כי ברצוננו לכתוב תכנית אשר כוללת פונקציה אשר מחשבת את הסכום וההפרש של שני פרמטרים שהועברו לה. בניגוד לדוגמה הקודמת, אנו מעוניינים שתוצאת החישוב שמבצעת הפונקציה לא תוצג על-ידה, אלא תועבר לידי התכנית הראשית (אשר קראה לפונקציה), על-מנת שהתכנית הראשית תוכל לעשות שימוש בתוצאה כפי רצונה (בדוגמה שלנו השימוש יהיה שהתכנית הראשית תציג את הסכום וההפרש). מייד נציג את התכנית הדרושה, אך לפני כן נעיר כי תכנית זאת ניתן לכתוב בשפת C++, בה קיימים פרמטרי הפניה. בשפת C לא קיימים פרמטרי הפניה, ועל-כן בה לא ניתן לכתוב תכנית כפי שנציג. בשפת C כדי להשיג את האפקט הרצוי יש להשתמש במצביעים, כפי שנכיר בהמשך. ועתה להצגת התכנית:

```
int main() {
   int num1, num2, num3, num4 ;

   cin >> num1 >> num2 ;
   sum_dif(num1, num2, num3, num4) ;
   cout >> num3 >> num4 ;

   sum_dif(num1, num3, num4, num2) ;
   cout >> num4 >> num2 ;

   return(EXIT_SUCCESS) ;
}
//-------------------------------------------
void sum_dif(int op1, int op2,
             int &sum, int &dif) {
   sum = op1 + op2 ;
   dif = op1 - op2 ;
}
```

נתחיל בבחינת הפונקציה `sum_dif` . הפעם הפונקציה כוללת ארבעה פרמטרים: באמצעות שני הראשונים `op1, op2` מעבירה הפונקציה הקוראת ל- `sum_dif` (במקרה שלנו התכנית הראשית) את הערכים שאת סכומם והפרשם על הפונקציה לחשב. לתוך הפרמטר השלישי מכניסה הפונקציה את הסכום `op1+op2`, ולתוך הפרמטר הרביעי מכניסה הפונקציה את ההפרש `op1-op2`. מדוע כתבנו & (ampercent) לפני שמות הפרמטרים `sum, dif`? האמפרסנט מורה כי מדובר **בפרמטרי הניה** (reference parameters) וזאת בניגוד **לפרמטרי ערך** (value parameters) אותם הכרנו בסעיף הקודם (ולפני שמם לא כתבנו אמפרסנט). פרמטרי הפניה הם פרמטרים המקיימים שערך אשר הפונקציה תכניס לתוך הפרמטר יוותר אחרי ביצוע הפונקציה בארגומנט המתאים לאותו פרמטר. לעומתם פרמטרי ערך מקיימים שערך שתכניס הפונקציה לתוך הפרמטר לא יוותר אחרי ביצוע הפונקציה בארגומנט המתאים לאותו פרמטר. נסביר ביתר הרחבה תוך שאנו דנים בקריאה: `sum_dif(num1, num2, num3, num4)`. נניח כי לפני הקריאה ערכו של `num1` הוא 17, וערכו של `num2` הוא 3. בשל סידור הארגומנטים בקריאה, הארגומנט `num1` הוא המתאים לפרמטר `op1`, הארגומנט `num2` הוא המתאים לפרמטר `op2`, הארגומנט `num3` הוא המתאים לפרמטר `sum`, והארגומנט `num4` הוא המתאים לפרמטר `dif`. עת הפונקציה מבצעת `dif = num1 - num2` היא מכניסה בכך לפרמטר `dif` את הערך 14. מכיוון ש- `dif` הוא פרמטר משתנה אזי הערך שהפונקציה מכניסה לתוכו יוותר אחרי ביצוע הפונקציה בארגומנט המתאים, במקרה הנוכחי ב- `num4`. כדי לחדד את הדברים, ולהבהירם יותר נניח כי בפונקציה היו גם פקודות שלישית ורביעית באופן הבא: `op1 = 0; cout << op1;`. נזכור כי בקריאה לפונקציה בה אנו דנים הארגומנט המתאים ל- `op1` הוא `num1`. האם

בתום ביצוע הפונקציה ערכו של num1 יהיה אפס? לא ולא! מכיוון ש- op1 הוא פרמטר ערך, אזי שינויים שהפונקציה מכניסה לתוך op1 לא יוותרו ב- num1 בתום ביצוע הפונקציה. הערך שיודפס על-ידי הפונקציה יהיה אפס, שכן *בתוך* הפונקציה, אחרי שאיפסנו את op1 ערכו, באופן טבעי, אפס, אולם הערך שהכנסנו ל- op1 לא משנה את ערכו של num1.

כאמור, הנחנו כי טרם הקריאה הראשונה ל- sum_dif ערכו של num1 הוא 17, וערכו של num2 הוא 3. לכן בתום הקריאה הראשונה ערכו של num3 יהיה 20, וערכו של num4 יהיה 14, וזה הפלט הראשון שתציג התכנית הראשית. עתה התכנית הראשית מזמנת שוב את sum_dif. בקריאה השנייה הארגומנט המתאים ל- op1 הוא הפרמטר num1, הארגומנט המתאים ל- op2 הוא הפרמטר num3, הארגומנט המתאים ל- sum הוא הפרמטר num4, הארגומנט המתאים ל- dif הוא הפרמטר num2. לכן בתום הקריאה num4 יכיל את הסכום num1+num3 כלומר את 37, num2 יכיל את ההפרש num1-num3 כלומר את 3-.

אנו רואים אם כן כי פרמטרי ערך הם ערוצי תקשורת חד-כיווניים בין מי שקורא לפונקציה לבין הפונקציה: באמצעותם יכול הקורא להעביר נתונים לפונקציה (אך הפונקציה אינה יכולה להעביר חזרה מידע למי שקרא לה). האם פרמטרים משתנים הם ערוצי תקשורת חד-כיווניים בכיוון ההפוך? לא. פרמטרים משתנים יכולים לשמש אותנו להעברת נתונים בשני הכיוונים: גם בין הקורא לנקרא, וגם בין הנקרא לקורא. הפונקציה הבאה תדגים לנו זאת. הפונקציה שנציג swap, תקבל באמצעות זוג פרמטרים שני משתנים שבין ערכיהם עליה להחליף. לדוגמה אם לפני ביצוע הפונקציה ערכו של num1 הוא 11, וערכו של num2 הוא 22, אזי אחרי ביצוע הפונקציה, עבור הקריאה: swap(num1, num2) אנו רוצים ש- num1 יכיל את הערך 22, ו- num2 יכיל את הערך 11. כיצד תראה swap:

```
void swap(int &var1, int &var2)
{
   int temp ;
   temp = var1 ;
   var1 = var2 ;
   var2 = temp ;
}
```

swap כוללת שני פרמטרי הפניה באמצעותם מי שקורא לפונקציה מעביר לה מידע אודות ערכי המשתנים בניהם עליה להחליף; אולם swap גם משנה את ערכי המשתנים המועברים לה, וכך מחזירה לקורא מידע.

ננצל את swap כדי לעמוד על ההבדל בין תפקידם של הפרמטרים (במקרה שלנו פרמטרי הפניה כגון var1, var2) לבין תפקידם של המשתנים הלוקליים (במקרה הנוכחי temp). פרמטרים הם, כאמור, אמצעי תקשורת (בכיוון זה ו\או אחר) בין מי שקרא לפונקציה לבין הפונקציה. באמצעותם מועבר מידע בין הקורא לנקרא או להפך. לעומתם משתנים לוקליים הם כלים המסייעים לפונקציה הנקראת להשלים את פעולתה. הפונקציה אינה מקבלת מידע באמצעותם, ולא מחזירה דבר דרכם, היא זקוקה להם כדי לבצע את משימתה. בדוגמה שלנו כדי להחליף בין ערכי var1 ו- var2 זקוקה הפונקציה למשתנה העזר temp.

לסיכום הדיון על פרמטרי ערך לעומת פרמטרי הפניה נדגיש כי:

א. באמצעות פרמטר ערך אנו מעבירים לפונקציה *ערך*. על- כן אם הגדרנו פונקציה: void f(int n) ואם למשתנים num1, num2 הוכנס ערך, אזי אנו רשאים לקרוא ל- f באופנים הבאים: f(17); f(num1+num2);

`f(num1 %17);` או בכל אופן אחר שיעביר ל- `f` ערך שלם כלשהו. לעומת זאת אם למשתנה `num3` טרם הוכנס ערך, אזי הקריאה `f(num3)` היא בגדר שגיאה.

ב. לעומת זאת באמצעות פרמטר הפניה אנו מעבירים לפונקציה *משתנה* שאולי יש בו כבר ערך (ואז הפונקציה רשאית לעשות שימוש בערך הקיים במשתנה), ואולי לא (ואז על הפונקציה רק להכניס למשתנה ערך). לכן אם הגדרנו:

```
void g(int &m) { m= 3879; }
```

אזי אנו רשאים לקרוא ל- `g` באופן הבא: `g(num1);` , ואז `g` תכניס ל- `num1` את הערך 3879, אך איננו רשאים לקרוא ל- `g` באופנים הבאים: `g(17) ;` `g(num1+1) ;` שכן בשתי הקריאות הללו איננו מעבירים ל- `g` *משתנה* שלתוכו היא יכולה להכניס את הערך 3879.

בעבר אמרנו כי הפרוטוטיפ של הפונקציה (אשר ייכתב מעל ה- `main`) יכלול את השורה הראשונה של הפונקציה, בתוספת נקודה-פסיק. זה נכון, אולם החפצים בכך יכולים לקצר בכתיבת הפרוטוטיפ ולהשמיט ממנו את שמות הפרמטרים, תוך שהם מותירים רק את טיפוסם. לדוגמה את הפרוטוטיפ של `sum_dif` ניתן לכתוב באופן הבא:

```
void sum_dif(int, int, int &, int &);
```

7.1.6 דוגמה שישית: פונקציה המחזירה ערך

הפונקציה `sum_dif` שכתבנו קיבלה מהקורא לה זוג נתונים, והחזירה, באמצעות פרמטרי ההפניה שלה, זוג ערכים: סכומם והפרשם של הנתונים שהועברו לַפונקציה. במקרים רבים פונקציה צריכה להחזיר למי שקרא לה ערך יחיד שהוא תוצאת החישוב (במובן הרחב של המילה) שהפונקציה ביצעה. לשם הדוגמה נניח כי עתה ברצוננו לכתוב פונקציה אשר מחזירה למי שקרא לה רק את סכומם של הנתונים שהועברו לה. במקרה זה מתאים שהערך המוחזר לא יוחזר באמצעות פרמטר הפניה, אלא באמצעות מנגנון אחר אותו נכיר עתה. נכתוב את הפונקציה ואחר נסבירה:

```
int sum3(int op1, int op2)
{
   int temp = op1 + op2 ;
   return( temp ) ;
}
```

הפונקציה שכתבנו מקבלת שני פרמטרי ערך, באמצעותם מעביר לה הקורא את הערכים שעליה לחבר. הפונקציה מחשבת את הערך הרצוי לתוך המשתנה `temp`, ולבסוף מבצעת את הפקודה `return(temp)`. פקודה זאת גורמת לסיום ביצוע הפונקציה, תוך כדי 'זריקת' ערכו של `temp` לעבר מי שקרא לפונקציה. מה יעשה מי שקרא לפונקציה עם הערך ש-'נזרק' לעברו? 'יתפוס' אותו, ויכניסו לתוך משתנה. הדבר יקרה שכן הקריאה לַפונקציה תיעשה באופן הבא: `num3 = sum3(num1, num2%17);`. נסביר: הקריאה לפונקציה כבר אינה פקודה העומדת בפני עצמה, בודדה בשורה (כפי שכתבנו עת זימנו את `sum_dif(…);` למשל); הקריאה לַפונקציה משולבת בפקודת השמה. באופן כזה ראשית מבוצעת הקריאה לפונקציה, אחר הפונקציה מבצעת את פעולתה, ולבסוף עת הפונקציה מסיימת, ו-'זורקת' את תוצאת החישוב שלה לעברו של מי שקרא לה, הערך הנזרק מוכנס לתוך המשתנה הרצוי (`num3` בדוגמה שלנו).

פרט נוסף אליו יש לתת את הדעת הוא שטיפוס הפונקציה (כפי שמופיע לפני שם הפונקציה) כבר אינו `void`, יען כי הפונקציה שלנו כבר אינה מחזירה כלום, טיפוס הפונקציה עתה הוא `int`, שכן הערך שהפונקציה מחזירה ('זורקת') בתום פעולתה, הוא ערך מטיפוס `int`. זו שגיאת קומפילציה לסיים פונקציה אשר אמורה להחזיר

ערך שונה מ- void שלא באמצעות פקודת return אשר מחזירה ערך מהטיפוס המתאים.

אנו יכולים, אם כן, להבדיל בין פונקציה מטיפוס void (כזו המוגדרת כ- void vf(…) {…}) לבין פונקציה מטיפוס שאינו void (כדוגמת: int if(…) {…}): בעוד הקריאה לפונקציה מטיפוס void תופיע כפקודה בודדת (בשורה נפרדת בתכנית), הרי הקריאה לפונקציה מטיפוס שאינו void תופיע תמיד כחלק מפעולה נוספת אשר תעשה שימוש בערך המוחזר על-ידי הפונקציה. אילו שימושים אפשר לעשות בערך המוחזר מפונקציה?

א. ראינו כי ניתן להשימו לתוך משתנה בפקודת השמה (כמו במקרה: num1 = sum3(…) ;, עבור num1 שהינו משתנה מטיפוס int).

ב. אפשר להשתמש בערך המוחזר מהפונקציה לשם השוואה. לדוגמה: if (sum3(num1, num2%17) > 0) … במקרה זה הערך המוחזר אינו נשמר בשום מקום, נעשה בו שימוש לשם ההשוואה ותו לא. בהמשך לא נוכל להשתמש בערך שהוחזר שכן הוא אבד לנצח בתהום הנשייה.

ג. אפשר להשתמש בערך המוחזר מפונקציה אחת כדי להעבירו לפונקציה אחרת, או כדי להציגו בפלט. לדוגמה: cout << sum3(17, 3879) ; או sum_dif(sum3(num1, 17), 3879, num2, num3) ; (שימו לב כי את הערך המוחזר על-ידי הפונקציה sum3 אנו רשאים להעביר כארגומנט לפונקציה אחרת, רק אם הפונקציה האחרת מצפה לקבל פרמטר ערך).

פונקציה אינה יכולה להחזיר מערך בפקודת ה- return שלה.

הפונקציה sum3 דומה מהיבט טיפוס הפונקציה לתכנית הראשית (המוגדרת כ: int main()). ועל-כן ניתן לשאול למי מחזירה התכנית הראשית את הערך אפס (או את EXIT_SUCCESS) אותו היא מחזירה בפקודה: return(0)? התשובה היא שהתכנית הראשית נקראת, למשל, על-ידי מערכת ההפעלה, אשר עשויה להתעניין בערך שהתכנית הראשית מחזירה. יש הנוהגים להחזיר מהתכנית הראשית ערכים שונים אשר מציינים את הנסיבות בהם התכנית הראשית הסתיימה: בהצלחה, בעקבות הזנת קלט שגוי, בעקבות כישלון בהשלמת משימה זו או אחרת, ועוד.

הדמיון בין התכנית הראשית לכל פונקציה אחרת עשוי להעלות גם את השאלה: האם גם התכנית הראשית עשויה לקבל פרמטרים (ככל פונקציה אחרת)? התשובה העקרונית היא כן, אולם כדי שנוכל גם ללמוד אודות פרמטרים אלה, ואופן השימוש בהם, עלינו ראשית להכיר משתנים מטיפוס סטרינג. לכן נשוב לַשאלה אחרי שנלמד נושא זה.

נביט בדוגמה נוספת אשר תחדד עוד כמה נושאים הקשורים לפונ' המחזירה ערך. הפונ' שנציג מקבלת שלושה מספרים שלמים ומחזירה את הגדול ביניהם. זימונה יהיה בפקודה כגון: cout << max3(num1, 17, num2) ; והגדרתה:

```
int max3(int n1, int n2, int n3)
{
   if (n1 >= n2 && n1 >= n3)
      return n1 ;
   if (n2 >= n1 && n2 >= n3)
      return n12 ;
   if (n3 >= n1 && n3 >= n2)
      return n3 ;
   return 0 ;
```

```
}
```

ברצוני להסב את תשומת לבכם לשתי נקודות בקשר לפונ' זו:

א. בַּפונ' אין פקודת `else`, הסיבה היא שאם התנאי הראשון מתקיים (ו- `n1` הוא הגדול בין השלושה) אזי אנו מבצעים את פקודת ה- `return` *אשר מוציאה אותנו מיידית מהפונ'*, בפרט לא נגיע כלל לבדיקת התנאי השני. באותו אופן לגבי המצב בו `n2` הוא הגדול ביותר, ואז לא נגיע כלל לבדיקת התנאי השלישי.

ב. מדוע בסוף הפונ' מופיעה הפקודה הטיפשית: `return 0;` ? התשובה היא שהקומפיילר אינו חכם כמוכם. הוא אינו מבין שאחד משלושת התנאים לבטח ייתקיים, ועל כן אחד משלושת הערכים לבטח יוחזר. הקומפיילר חושש שמא חלילה אף תנאי לא יתקיים, ולכן הביצוע יתדרדר לסוגר שמסיים את הפונ' בלי שהפונ' תחזיר כל ערך שהוא. על כן לולא היינו כוללים בפונ' את הפקודה `return 0` היה הקומפיילר נותן לנו שגיאת קומפילציה. כדי להרגיע את הקומפיילר אנו כוללים את פקודת ה- `return 0` (או כל ערך אחר).

מהסיבות הנ"ל יכולנו לכתוב את הפונ' יותר בקצרה באופן הבא:

```
int max3(int n1, int n2, int n3)
{
   if (n1 >= n2 && n1 >= n3)
      return n1 ;
   if (n2 >= n1 && n2 >= n3)
      return n12 ;
   return n3 ;
}
```

<u>הסבר</u>: אם הגענו את הפקודה `return n3;` אות הוא וסימן ששני התנאים הראשונים לא התקיימו (שכן כל אחד מהם לו היה מתקיים היה מוציא אותנו מיידית מהפונ' ולא היינו מגיעים לפקודה זאת), ולכן הגדול מבין השלושה הוא `n3`, ואותו יש להחזיר.

<u>7.1.7 דוגמה שביעית: פונקציה עם פרמטר משתנה, המחזירה ערך</u>

מהדוגמה הקודמת שראינו ניתן היה להסיק, ובהצדקה מסוימת, שלְפונקציה שטיפוסה שונה מ- `void` לא יהיו פרמטרי הפניה: שהרי תפקידם של פרמטרי הפניה לאפשר לפונקציה להחזיר באמצעותם ערך, ופונקציה שטיפוסה שונה מ- `void` מחזירה את הערך *היחיד* שעליה להחזיר באמצעות פקודת `return`, (ולא באמצעות פרמטרי הפניה). ככלל המסקנה הנ"ל סבירה, אולם כמו בהרבה מקרים אחרים בעולמנו הקסום לכל כלל יש יוצא מין הכלל (ולעיתים אפילו הרבה יוצאים מין הכלל). לכן עתה נדגים מתי ומדוע אנו עשויים לכתוב פונקציה שטיפוסה יהיה `bool`, ויהיה לה גם פרמטר הפניה.

הפונקציה שנכתוב, `div`, היא פונקציה אשר מקבלת באמצעות שני פרמטרי ערך שני מספרים שלמים. במידה ומוגדרת מנה של שני הפרמטרים (במילים אחרות, במידה והמחַלק שונה מאפס) אזי על הפונ' להחזיר את מנתם הממשית של הפרמטרים; וזאת היא תעשה באמצעות פרמטר הפניה. הפונקציה תחזיר למי שקרא לה באמצעות פקודת ה- `return` איתות האם היא הצליחה או נכשלה במשימתה.

ראשית נדון באופן בו תקַרא הפונקציה (למשל מהתכנית הראשית). נניח כי בתכנית הוגדרו המשתנים השלמים `num1, num2` והושם להם ערך. כמו כן הוגדר

המשתנה הממשי result. בהנחות אלה נוכל לקרוא לפונקציה div, למשל, באופן הבא:

```
if (div(num1 +17, num2 %3879, result)==true)
   cout << result ;
else
   cout << "Can not divide by zero\n" ;
```

נסביר: אנו קוראים לפונקציה div ומעבירים לה שלושה ארגומנטים: (א) מחולק, שערכו בדוגמה שלנו num1+17, (ב) מחַלק, שערכו בדוגמה שלנו num2 %3879, (ג) משתנה לתוכו תכניס הפונקציה את המנה, בדוגמה שלנו זהו המשתנה result. הפונקציה תחזיר ערך שיהיה true או false. במידה והערך שהוחזר הוא true אות הוא וסימן כי בַּארגומנט השלישי מצויה מנתם של שני הארגומנטים הראשונים, ולכן אנו רשאים להציג את ערכו של ארגומנט זה. במידה והערך שהוחזר הוא false אנו מסיקים כי הפונקציה לא הצליחה לחשב את המנה, יען כי המחלק שהועבר לה (בארגומנט השני) ערכו אפס. אם זה המצב נציג הודעה מתאימה. (כמובן, שהמרכיב ==true בַּתנאי שבַּ- if הוא מיותר, כתבנו אותו לטובת אלה שעשויים להתקשות אם הוא ייעדר מהביטוי. כמובן גם שכמו כל פונקציה המחזירה ערך, גם div מופיעה כחלק מפקודה אחרת, בַּמקרה הנוכחי כחלק מתנאי בפקודת if).

עתה נפנה להגדרת הפונקציה div. מדרך הקריאה לה אנו יכולים להסיק פרטים רבים אודות הפרוטוטיפ שלה:

א. מכך שאנו משווים את הערך המוחזר על-ידי הפונקציה עם הערך true ניתן להסיק שטיפוס הפונקציה הוא bool.
ב. מכך שאנו קוראים לפונקציה בתור ארגומנט ראשון עם num1 +17 ניתן להסיק כי הפרמטר הראשון של הפונקציה הוא פרמטר ערך מטיפוס int.
ג. מכך שאנו קוראים לפונקציה בתור ארגומנט שני עם num2 %3879 ניתן להסיק כי הפרמטר השני של הפונקציה הוא פרמטר ערך מטיפוס int.
ד. מכך שאנו מציגים את ערכו של result אחרי חזרתה של הפונקציה ניתן לשער (אם כי לא להסיק בוודאות) כי הפרמטר השלישי של הפונקציה הוא פרמטר הפניה מטיפוס double.

ועתה הגדרת הפונקציה:

```
bool div(int op1, int op2, double &res) {
   if (op2 == 0)
      return(false) ;
   else
   { res = double(op1)/op2 ;
     return( true ) ;
   }
}
```

הסבר: הפונקציה בודקת האם הפרמטר השני שלה ערכו אפס, אם כן היא מסיימת את פעולתה באופן מיידי תוך שהיא מחזירה את הערך false. אחרת, ראשית לפרמטר ההפניה res מושמת המנה הממשית double(op1)/op2, ושנית מוחזר הערך true כדי לאותת שהחישוב הסתיים בהצלחה.

ננצל את הפונקציה שכתבנו כדי לחדד נקודה נוספת הקשורה לפקודת ה- return. פקודת ה- return עוצרת *מיידית* את ביצוע הפונקציה (במילים אחרות, מעיפה אותנו מיידית מהפונקציה), תוך החזרת הערך המופיע בפקודה. מסיבה זאת יכולנו לכתוב את הפונקציה div גם באופן הבא:

```
bool div(int op1, int op2, float &res) {
```

```
   if (op2 == 0)
      return(false) ;
   res = float(op1)/op2 ;
   return( true ) ;
}
```

נסביר: מהגרסה המתוקנת השמטנו את מרכיב ה- else. מדוע הדבר אפשרי? שכן אם ערכו של op2 הוא אפס, אזי פקודת ה- return תקטע באופן מיידי את ביצוע הפונקציה, תוך החזרת הערך false, ולא נגיע כלל להשמה שבשורה: res = …, וטוב שכך. מנגד, אם ערכו של op2 שונה מאפס, אזי התנאי שבפקודת ה- if לא יתקיים, לכן לא נבצע את הפקודה return(false), ונתקדם הלאה בביצוע הפונקציה לפקודת ההשמה res = …, הפקודה תושלם בהצלחה, ונמשיך לַפקודה return(true) אשר תחזיר את הערך המתאים.

ניתן להשתמש בפקודת return גם בפונקציה מטיפוס void. פקודת ה- return לא תכלול כל ערך מוחזר, היא תיכָּתב: return ;, והיא, כמובן, תקטע את ביצוע הפונקציה במקום בו היא מופיעה.

הפונקציה האחרונה שכתבנו גם מדגימה את הרעיון הדַי ברור שפונקציה עשויה לקבל מספר פרמטרים ממספר טיפוסים שונים, ולהחזיר ערך מטיפוס אחר, ואין בכך רבותא.

7.2 פונקציות הכוללות לולאות

הפונקציות שתופענה בסעיף זה אינן מציגות כל חידוש עקרוני; ולמעשה זה לא צריך להיות מפתיע שפונקציה עשויה לכלול בתוכה גם פקודות לולאה. אציג את הדוגמות הבאות רק על-מנת להעשיר את מגוון הדוגמות להן אנו נחשפים.

7.2.1 פונקציה המחזירה את סכום מחלקיו של מספר

נכתוב פונקציה אשר מקבלת כפרמטר ערך מספר טבעי. הפונקציה תחזיר באמצעות פקודת ה- return את סכום מחלקיו של הפרמטר. הקריאה לפונקציה (למשל מהתכנית הראשית) תיעשה בהקשר כדוגמת הבא:

```
cin >> num ;
sum = sum_dividers(num) ;
cout << "The sum of the dividers of: " << num << "="
     << sum ;
```

ועתה נפנה להגדרת הפונקציה sum_dividers:

```
int sum_dividers(int num) {
   int sum = 0 ;

   for (int i = 0; i <= num/2; i++)
      if (num % i == 0)
         sum += i ;
   return(sum) ;
}
```

בפונקציה שכתבנו אין כל חידוש עקרוני. נקודה קטנה אליה ניתן את הדעת היא כי בדוגמה זאת הן בתכנית הראשית יש משתנה בשם sum, והן לפונקציה יש משתנה בשם זה; באופן דומה, הן בתכנית הראשית מוגדר משתנה בשם num, והן לפונקציה יש פרמטר בשם זה. את המחשב הכפילות הזאת בשמות לא מבלבלת, ואם גם אתכם היא אינה מבלבלת אזי היא לגיטימית. חשוב לזכור שכשלעצמו אין כל קשר

בין num של התכנית הראשית לבין num של הפונקציה (או בין שני ה- sum –ים). בדוגמה המסוימת שהצגנו, במקרה או שלא במקרה, לפונקציה מועבר num כַּארגומנט המתאים לפרמטר num, ולכן בקריאה זאת אנו יצרנו קשר בין השניים.

7.2.2 פונקציה הבודקת האם מספר ראשוני או פריק

נציג עתה מספר גרסות של פונקציה המקבלת מספר טבעי ומחזירה את הקבוע הבולאני A_PRIME אם המספר שהועבר לה ראשוני, ואת הקבוע הבולאני NOT_PRIME אם המספר שהועבר לה פריק. הקריאה לפונקציה (בכל הגרסות שנראה) תעשה בהקשר כדוגמת הבא:

```
for (i=2; i <= N; i++)
   if (is_prime(i))
      cout << i << " is prime\n" ;
   else
      cout << i << "is not prime\n" ;
```

הגדרת הפונקציה בגרסתה הראשונה:

```
bool is_prime(int num) {
   bool prime = A_PRIME ;

   for (int i = 2; i<= sqrt(num) && prime; i++)
      if (num % i == 0)
         prime = NOT_PRIME ;

   return(prime) ;
}
```

הסבר: הפונקציה מחזיקה משתנה בולאני prime. המשתנה מאותחל לערך A_PRIME כדי לציין שאנו מניחים מחדלית שהמספר שהועבר לפונקציה באמצעות הפרמטר num ראשוני. עתה אנו בודקים את הנחתנו: אנו מריצים לולאה המתנהלת כל עוד: (א) לא בדקנו עבור כל המספרים הקטנים משורש num האם הם מחלקים את num, וכן: (ב) לא גילינו מספר המחלק את num. בגוף הלולאה אנו בודקים עבור כל ערך בַּתחום האם הוא מחלק את num, ואם כן מעדכנים את ערכו של prime להיות NOT_PRIME. בתום הלולאה אנו מחזירים את ערכו של prime.

כבר בעבר ראינו כי ניתן לכתוב קטע תכנית זה בצורה מעט שונה, תוך שימוש ב-break. נציג את צורת הכתיבה החלופית בהקשר של פונקציה:

```
bool is_prime(int num) {

   for (int i = 2; i<= sqrt(num); i++)
      if (num % i == 0)
         break ;

   if (i <= sqrt(num))
      return(NOT_PRIME) ;
   else
      return(A_PRIME) ;
}
```

הסבר: הפונקציה כוללת לולאה אשר בודקת עבור כל-אחד ואחד מהמספרים הקטנים או שווים משורש num האם הוא מחלק את num. במידה והתגלה מספר שכזה אנו קוטעים את ביצוע הלולאה באמצעות פקודת break. אחרי הלולאה אנו יכולים להסיק על-סמך ערכו של i האם num ראשוני או פריק: אם ערכו של i עדיין

קטן או שווה מ- sqrt(num) אות הוא וסימן שביצוע הלולאה נקטע בשל מציאת מחלק, ולכן על הפונקציה להחזיר את הערך NOT_PRIME, ואחרת (i מיצה את כל המספרים הקטנים או שווים מ- sqrt(num) ועבור אף אחד מהם לא בוצע break שכן אף אחד מהם אינו מחלק את num ולכן) יש להחזיר את הערך A_PRIME.

את קטע הקוד:

```
if (i <= sqrt(num))
   return(NOT_PRIME) ;
else
   return(A_PRIME) ;
```

ניתן לשפר במספר אופנים: כפי שכבר טענו בעבר, ה- else מיותר, וניתן לכתוב את הקוד באופן:

```
if (i <= sqrt(num))
   return(NOT_PRIME) ;
return(A_PRIME) ;
```

שכן אם i <= sqrt(num) תבוצע הפקודה: return(NOT_PRIME); , אשר תקטע את ביצוע הפונקציה, ולא נגיע כלל לפקודה: return(PRIME); , והערך שיוחזר יהיה על-כן כנדרש. מנגד אם i אינו קטן או שווה מ- sqrt(num) אזי לא תבוצע הפקודה: return(NOT_PRIME); , וביצוע הפונקציה ימשיך לַפקודה: return(A_PRIME); אשר תחזיר את הערך המתאים.

את קטע החזרת הערך ניתן לכתוב בצורה קומפקטית יותר תוך שימוש באופרטור ה-? באופן הבא:

```
return( i <= sqrt(num) ? NOT_PRIME : A_PRIME ) ;
```

נסביר: אם i <= sqrt(num) אזי אופרטור ה- ? מחזיר את הערך NOT_PRIME (הערך המופיע משמאל לנקודתיים), שמועבר לפקודת ה- return. ועל כן זה יהיה הערך שהפונקציה תחזיר באמצעות פקודת ה- return שלה. אם לעומת זאת התנאי i <= sqrt(num) אינו מתקיים, אזי אופרטור ה-? יחזיר את הערך A_PRIME (הערך שמופיע מימין לנקודתיים) לפקודת ה- return, אשר תחזיר, על-כן, ערך זה.

עתה נציג דרך שלישית לכתוב את הפונקציה:

```
bool is_prime(int num) {

   for (int i = 2; i<= sqrt(num); i++)
      if (num % i == 0)
         return(NOT_PRIME) ;

   return(A_PRIME) ;
}
```

הסבר: אם במהלך ריצת הלולאה אנו מגלים מחלק של num, אזי אנו יכולים לקטוע את ביצוע הפונקציה בו במקום תוך שאנו מחזירים את הערך NOT_PRIME. אם לעומת זאת מיצינו את הלולאה, והגענו לפקודת ה- return המצויה מתחת ללולאה, אזי אנו יכולים להסיק כי num ראשוני (לא מצאנו כל 'עד' שיורה שהוא פריק, ויגרום לנו לקטיעת ביצוע הפונקציה), ולהחזיר את הערך המתאים.

להזכירכם, גם פוני המחזירה void ניתן לקטוע באמצעה באמצעות פקודת return ריקה, כדוגמת:

```
if (i ==)
```

```
   return ;
```
<המשך הפונ'>

7.2.3 פונקציה הבודקת האם מספר הוא פלינדרום

מספר טבעי הוא פַלינדרום אם קריאתו במהופך נותנת אותו ערך כמו קריאתו הרגילה, במילים אחרות אם המספר שווה לתמונת הראי שלו. לדוגמה: 121, 12321 , 5775 הם פלינדרומים, אך: 122 או 1232 אינם פלינדרומים.

נרצה לכתוב פונקציה אשר מקבלת מספר טבעי ומחזירה את הערך `true` אם המספר הוא פלינדרום, ואת הערך `false` אחרת. לשם כתיבת הפונקציה נניח כי היפוכו של המספר המועבר לפונקציה כארגומנט הוא מספר שלם שיכול להיות מוכל במשתנה מטיפוס `int` (לדוגמה: אם המספר השלם הגדול ביותר שמשתנה מטיפוס `int` יכול להכיל הוא 32768 אזי היפוכו של 19999 הוא המספר 99991 והוא כבר אינו יכול להיות מוכל במשתנה מטיפוס `int`, אלא רק במשתנה מטיפוס `long int`).

הקריאה לפונקציה תיערך בהֶקְשר כדוגמת הבא:

```
do {
   cin >> num ;
} while (is_palin(num)) ;
```

כלומר אנו מנהלים לולאה הקוראת מספרים כל עוד המספרים המוזנים הם פלינדרומים.

הגדרת הפונקציה:

```
bool is_palin(int num) {
  int reverse = 0,
      save = num ;

   while (num != 0) {
      reverse = reverse *10 + num % 10 ;
      num /= 10 ;
   }
   return( reverse == save ) ;
}
```

<u>הסבר</u>: המשתנה `reverse` יכיל את היפוך ספרותיו של `num`. ערכו ילך ויבנה בהדרגה בַלולאה שהפונקציה מריצה. לדוגמה נניח שערכו של `num` בקריאה הוא 123. `reverse` מאותחל לערך אפס.

א. בסיבוב הראשון של הלולאה ערכו של `reverse` גדל להיות 3 (כערך הספרה האחרונה של `num`, שכן `num%10` ערכו 3, ו- `reverse*10` הוא אפס). מ- `num`, לעומת זאת אנו 'קוצצים' את הספרה האחרונה: `num /= 10`, מקטין את `num` מ- 123 ל- 12.

ב. בסיבוב השני בלולאה ערכו של `reverse` יגדל להיות 32 (כערך היפוך שתי הספרות האחרונות של `num`, שכן `num%10` ערכו 2, ו- `reverse*10` ערכו 30). בסיבוב זה בלולאה `num` קטן להיות 1 (12/10 =1).

ג. בסיבוב השלישי בלולאה ערכו של `reverse` גדל להיות 321, וערכו של `num` מתאפס.

בזאת מסתיימת הלולאה. התהליך בו אנו 'בונים' את `reverse` הולך ו-'אוכל' את `num` (שלבסוף מתאפס), ועל כן אנו שומרים את ערכו התחילי של `num` במשתנה

save. לבסוף אחרי שגמרנו לבנות את reverse אנו בודקים האם ערכו שווה לערכו של save.

מה התעלול המבוצע בפקודה: return(reverse == save)? הפקודה ראשית בודקת את ערכו של הביטוי הבולאני: reverse == save, (ערכו יהיה, כמובן, true אם ערכם של reverse ו- save שווה, ו- false אחרת), ואת הערך שהיא מחשבת היא מחזירה בפקודת ה- return.

7.3 פונקציות ומערכים

בסעיף זה נראה כיצד ניתן לשלב מערכים ופונקציות.

7.3.1 פונקציה המחזירה את הערך המרבי המצוי במערך

נניח כי בתכנית הוגדר המערך: int arr[N], והוזנו לתוכו נתונים. עתה ברצוננו לכתוב פונקציה אשר תחזיר (באמצעות פקודת return), את הערך המרבי במערך. הקריאה לפונקציה תהיה:

```
max = find_max(arr)
```

(עבור max שהינו משתנה שלם).

הגדרת הפונקציה find_max:

```
int find_max(int nums[])
{
   int max = nums[0] ;

   for (int i = 1; i < N; i++)
      if (nums[i] > max)
         max = num[i] ;

   return max ;
}
```

קוד הפונקציה פשוט למדי, ואין צורך להכביר עליו מילים. הפרט היחיד אותו יש להסביר הוא הגדרת הפרמטר: int nums[] . עת אנו מגדירים את הפרמטר של הפונקציה אנו מציינים שהוא מערך חד-ממדי של מספרים שלמים. בשפת C עת פונקציה מקבלת מערך חד-ממדי אין צורך לציין את גודלו של המערך, שכן הפונ' יכולה לקבל מערך חד-ממדי של מספרים שלמים בכל גודל שהוא. יכולנו להגדיר את הפרמטר גם באופן הבא: int nums[N], אולם זה לא היה משנה דבר, זה לא היה מעלה ולא מוריד. מישהו יכול לשאול אם כן: כיצד יודעת הפונ' לרוץ על N תאי המערך? ומהיכן מכירה הפונ' את N? התשובה לשאלה השנייה היא שהפונ' מכירה את N שכן הוא קבוע גלובאלי, ולכן מוכר בתכנית בכללותה: ב- main, כמו גם בכל הפונ'. התשובה לשאלה הראשונה היא שבפונ' הנוכחית המתכנת הוא שקבע שהפונ' תרוץ על N תאים של המערך, שכן כך הוא הגדיר את הלולאה.

נחדד את הנקודה: נניח שבתכנית הראשית היה מוגדר גם המערך: int arr2[2*N]; , ונניח שלמערך זה הוזנו נתונים. הפקודה:

```
cout << find_max(arr2) ;
```

הייתה גורמת להדפסת האיבר המרבי בין N תאי המערך הראשונים. הפונ' find_max מתעניינת רק ב- N תאים אלה. אם זה הפלט המתאים לנו אזי בקריאה זאת לפונ' אין כל בעיה.

באופן דומה: אם בתכנית מוגדר גם המערך: `int arr3[N/2];` וגם למערך זה הוזנו נתונים, אזי הקריאה:

```
cout << find_max(arr3) ;
```

היא בגדר תקלה חמורה, שכן הפונ' `find_max` תחרוג מחוץ לגבולות המערך, עת היא תנסה לרוץ על `N` תאים במערך שהועבר לה, שעה שבמערך יש רק `N/2` תאים. אני מזכיר לכם שבשפת סי, במילים אחרות במערב הפרוע, הדבר עלול שלא להסתיים בהעפת התכנית, שכן החריגה מחוץ לגבולות המערך לא תביא לחריגה מחוץ לגבולות הזיכרון שמוקצה לתכנית, וכך יצא שהשגיאה עלולה להיוותר בתכנית בלי שנשים לבנו לכך (עד שבודק התרגילים שלנו 'יכסח' אותנו עליה...).

כמו במקרים הקודמים שראינו הפרוטוטיפ של הפונקציה יהיה זהה לשורת הכותרת שלה (אולי בהשמטת שם הפרמטר). לכן במקרה הנוכחי נכתוב את הפרוטוטיפ באחד משלושת האופנים הבאים:

```
int find_max(int nums[N]) ;
int find_max(int nums[]) ;
int find_max(int []) ;
```

בתכנית מופיעה הפקודה: `return max ;` ללא סוגריים. זו אינה שגיאה. אני נוטה לכתוב את הערך המוחזר בתוך סוגריים כדי להדגיש כי זה הערך המוחזר באמצעות פקודת ה- `return`, אולם רבים מוותרים על הסוגריים.

אנו יכולים, כמובן, לכתוב גרסה משופרת של `find_max` אשר תקבל מעבר למערך גם את גודלו, וכך תדע עד לאן יש להריץ את הלולאה. קוד הפונ' יראה:

```
int find_max2(int arr[], int arr_size) {
   int max = arr[0] ;

   for (int i=0; i<arr_size; i++)
      if (arr[i] > max)
         max = arr[i] ;
   return max ;
}
```

זימונה של `find_max2` עם שלושת המערכים שתיארנו קודם יהיה:

```
cout << find_max2(arr, N)
     << find_max2(arr2, 2*N)
     << find_max2(arr3, N/2) ;
```

7.3.2 פונקציה המחזירה את הערך הזוגי המרבי המצוי במערך

הפונקציה שנציג עתה היא אחותה התאומה של הפונקציה שהחזירה את מנת שני המספרים השלמים שהועברו לה (במידה והמנה ניתנה לחישוב). הפונקציה שנכתוב מקבלת מערך המכיל מספרים שלמים. במידה ובמערך מצוי ערך זוגי, אזי הפונקציה תחזיר, באמצעות פרמטר משתנה, את הערך הזוגי המרבי המצוי במערך, ובאמצעות פקודת `return` איתות כי היא מצאה איבר מרבי. במידה ובמערך לא מצוי כל ערך זוגי תאותת הפונקציה כי היא נכשלה במשימתה. אופן הקריאה לפונקציה יהיה למשל:

```
if (find_max_even(nums, max) == true)
   cout << max ;
else
   cout << "No even value\n" ;
```

הגדרת הפונקציה:

```
bool find_max_even(int nums[], int &max_even) {
```

```
   bool even_found = false ;

   for (int i = 0; i < N; i++)
      if (nums[i] %2 == 0)
         if (!even_found) {
            even_found = true ;
            max_even = nums[i] ;
         }
         else if (nums[i] > max_even)
               max_even = nums[i] ;

   return even_found ;
}
```

<u>הסבר</u>: אנו סורקים את המערך. אם מצאנו איבר זוגי אזי אם זהו האיבר הזוגי הראשון שאנו פוגשים (ערכו של `even_found` עדיין `false`) אזי אנו מעדכנים כי נמצא איבר זוגי במערך (`even_found = true`), ולפרמטר `max_even` אנו מכניסים את ערכו של התא הזוגי הראשון שנמצא. אם לעומת זאת מצאנו איבר זוגי שאינו ראשון (לא נכון ש- `!even_found`) אנו בודקים האם ערכו של איבר זה גדול מהערך הזוגי המרבי שמצאנו עד כה, ואם כן מעדכנים את ערכו של `max_even`. בתום ביצוע הלולאה אנו מחזירים, באמצעות פקודת `return` את ערכו של `even_found`, אשר מעיד האם נמצא או לא איבר זוגי במערך. במידה ונמצא איבר כנ"ל אזי `max_even` מחזיק את ערכו של הזוגי המרבי שנמצא.

<u>7.3.3 פונקציה הקוראת נתונים לתוך מערך (גרסה א')</u>

עת טיפלנו במשתנים פרימיטיביים (במילים אחרות במשתנים שאינם מערכים), אזי אם רצינו להקנות לפונקציה את הכוח לשנות את ערכו של הארגומנט המועבר לה היה עלינו להגדיר את הפרמטר המתאים כפרמטר הפניה. לדוגמה: אם אנו רוצים שהפונקציה `zero` תוכל לאפס את ערכו של הפרמטר המועבר לה, ושהשינוי יוותר בארגומנט המתאים אחרי תום ביצוע הפונקציה, אזי עלינו להגדיר את הפונקציה באופן הבא: `void zero(int &var) {var = 0;}`. לו היינו משמיטים את האמפרסנט (&) מהגדרת הפרמטר אזי בתוך גוף הפונקציה יכולנו לכתוב `var = 0`, אולם בדוגמה הבאה הערך שהתכנית הייתה מדפיסה לא היה אפס אלא 17:

```
num1 = 17 ;
zero(num1) ;
cout << num1 ;
```

עם מערכים המצב שונה. אם פונקציה תשנה את ערכם של תאי מערך המועבר לה כפרמטר, אזי השינוי שהפונקציה תכניס לתאים יוותר בארגומנט המתאים גם אחרי ביצוע הפונקציה, וזאת גם אם לא כתבנו אמפרסנט לפני שם הפרמטר (ולמעשה בכל תנאי אסור לנו לכתוב אמפרסנט לפני שם הפרמטר). לדוגמה: נביט בפונקציה:

```
void kuku(int a[], int b) {
   a[0] = 17;
   b = 17 ;
}
```

ונניח את קטע הקוד הבא (בו `aa` הוא מערך של מספרים שלמים, `bb` הוא משתנה שלם):

```
aa[0] = 0 ; bb = 0 ;
kuku(aa, bb) ;
cout << aa[0] << " " << bb ;
```

מה יהיה הפלט שיוצג? השינוי שהפונקציה ביצעה לתא מספר אפס במערך a יוותר בתא מספר אפס במערך aa, ועל-כן הערך הראשון שהתכנית תציג יהיה 17, לעומת זאת השינוי שהפונקציה ביצעה לפרמטר (הערך) b, לא ישפיע על ערכו של הארגומנט bb וערכו יישאר אפס כפי שהוא היה לפני הקריאה לפונקציה.

עתה נניח כי בתכנית הוגדר המערך int nums[N]; וברצוננו לכתוב פונקציה אשר תקרא ערכים לתוך המערך. הקריאה לפונקציה תהיה: read_data(nums); הגדרת הפונקציה:

```
void read_data(int list[]) {
   for (int i = 0; i < N; i++)
      cin >> list[i] ;
}
```

כפי שהסברנו, שינויים שהפונקציה מבצעת לתאי הפרמטר שלה list, נותרים בתוך הארגומנט המתאים (במקרה זה nums) בתום ביצוע הפונקציה, וזאת למרות שלפני שם הפרמטר לא מופיע התו אמפרסנט. נחזור ונדגיש כי זו שגיאה לכתוב תו זה לפני שם פרמטר מסוג מערך.

האם יש ביכולתנו להשיג את אותו אפקט המושג באמצעות פרמטר ערך פרימיטיבי גם עבור מערך? (כלומר ששינויים שהפונ' מכניסה למערך לא יוותרו בו בתום ביצוע הפונ') התשובה היא: לא בדיוק אבל בערך. אם נגדיר את הפונקציה באופן הבא:

```
void kuku(const int list[]){ … }
```

אזי בגוף הפונקציה לא נוכל לשנות *כלל* את ערכם של תאי המערך; כלומר פקודה כגון: list[0] = 17 ; תגרום לשגיאת קומפילציה. שימו לב כי עבור פרמטר ערך פרימיטיבי לא זה המצב: עבור פרמטר ערך פרימיטיבי הפונקציה רשאית לשנות את ערכו; ושינויים שהפונקציה תכניס לפרמטר יוותרו בעינם *במהלך ביצוע הפונקציה*, הם רק לא יגרמו לשינוי ערכו של הארגומנט.

מטעמי סגנון תכנותי, עת פונ' אינה אמורה לשנות את ערכו של המערך המועבר לה ראוי להעביר את הפרמטר כקבוע.

נזכיר שוב כי פונקציה אינה יכולה להחזיר מערך באמצעות פקודת ה- return שלה.

7.3.4 פונקציה הקוראת נתונים לתוך מערך (גרסה ב')

עתה ברצוננו לכתוב גרסה שנייה של הפונקציה הקוראת נתונים לתוך מערך. בגרסה זאת לא נקרא בדיוק N נתונים לתוך המערך (עבור N המציין את גודלו של המערך), אלא נקרא לכל היותר N נתונים, ונחזיר לקורא בנוסף למערך המעודכן גם את מספר הנתונים שנקראו בפועל. את הנתונים נקרא למערך עד קליטת הערך 1– אשר יסמן את סוף הזנת הנתונים.

הקריאה לפונקציה תהיה: n = read_data2(nums); (עבור nums שהינו מערך של N מספרים שלמים, ו- n שהוא משתנה שלם פרימיטיבי).

הגדרת הפונקציה:

```
int read_data2(int nums[])
{
   for (int i= 0; i < N; i++)
   {
      int temp ;
```

```
      cin >> temp ;
      if (temp == -1)
         return(i) ;
      nums[i] = temp ;
   }
   return(N) ;
}
```

<u>הסבר</u>: הפונקציה מתנהלת כלולאה הקוראת את הנתונים בזה אחר זה. כל נתון נקרא ראשית לתוך משתנה העזר temp. במידה והוזן הערך -1, אנו קוטעים את ביצוע הפונקציה ומחזירים את ערכו של i (אתם מוזמנים לבדוק שזהו אכן הערך שיש להחזיר, ולא i-1 או i+1; למשל בחנו מה יוחזר אם כבר הערך הראשון שהמשתמש יזין יהיה -1). מנגד יש לזכור (ותלמידים רבים שוכחים זאת), כי ייתכן שהמשתמש יזין N ערכים שאינם -1 למערך. במקרה זה יש לעצור את תהליך הקריאה אחרי קריאת N הנתונים, ולא לגלוש מחוץ לתחומי המערך, וכן יש לזכור להחזיר את הערך N כמספר הנתונים שנקראו.

בעיה לא נדירה עם פונ' המחזירה ערך (שונה מ- void) היא הבעיה הבאה: נביט בפונ' הפשוטה הבאה:

```
bool even(int n) {
   if (n % 2 == 0)
      return true ;
   if (n % 2 != 0)
      return false ;
}
```

אתם ואני יודעים שהתנאים בפונ' מכסים כל ערך אפשרי של n, אולם הקומפיילר לא יודע זאת, והוא חושש שיהיה ערך של n עבורו לא יתקיים אף אחד משני התנאים, ועל כן עבור ערך זה הפונ' 'תדרדר' לסוגריים המסיימים אותה, בלי להחזיר כל ערך. לפיכך הקומפיילר, לשם שינוי לא בצדק, 'צועק' עליכם שהפונ' שלכם שגויה. הפתרון לבעיה: כדי 'להשקיט' את הקומפיילר הוסיפו בסוף הפונ' שלכם פקודה: return true; אנחנו יודעים שזו פקודה מיותרת, שלעולם לא תבוצע, אולם את הקומפיילר (הטיפש) היא מרגיעה, ובא לציון הגואל.

7.3.5 פונקציה המקבלת מערך דו-ממדי (רב ממדי)

מטרתה של פונקציה פשוטה זאת להדגים את השימוש במערך דו-ממדי בפונקציה. נניח כי בתכנית הוגדר המערך: int matrix[ROWS][COLS]. להזכירכם: מבחינת שפת C מערך דו-ממדי הוא למעשה סדרה של מערכים חד-ממדיים. בדוגמה שלנו המערך matrix הוא סדרה של ROWS מערכים חד-ממדיים (במילים אחרות של ROWS שורות), כל אחד מהם בן COLS תאים.

נניח כי למערך הנ"ל הוזנו ערכים ואנו קוראים לפונקציה:

```
cout << sum_matrix(matrix) ;
```

נציג עתה את הגדרת הפונקציה sum_matrix:

```
int sum_matrix(const int mat[][COLS]) {
   int sum = 0 ;

   for (int row = 0; row < ROWS; row++)
      for (int col = 0; col < COLS; col++)
```

```
            sum += mat[row][col] ;

    return sum ;
}
```
הנקודה היחידה אליה יש לתת את הדעת בהרחבה היא האופן בו אנו מגדירים את הפרמטר mat. ניזכר כי עבור מערך חד-ממדי אמרנו כי מבחינת השפה אין הכרח בציון מספר התאים במערך, ולכן, עבור מערך חד-ממדי, עת אנו מגדירים את הפרמטר, אנו נוהגים לכתבו int arr[] (ולא int arr[N]) . באופן דומה, מכיוון שמערך דו-ממדי הוא סדרה של מערכים חד-ממדיים אנו פטורים מלציין כמה מערכים חד-ממדיים מרכיבים את המערך הדו-ממדי. על כן במקום לכתוב int mat[ROWS][COLS] אנו נוהגים לכתוב int mat[][COLS] ובכך לציין כי אנו מעבירים מערך חד-ממדי בגודל כלשהו, שכל תא בו הוא מערך בן COLS מספרים שלמים. כפי שראינו שעת פונ' מקבלת מערך חד-ממדי ניתן לזמן אותה עם מערכים בגודל שונה, כך עת פונ' מקבלת מערך דו-ממדי כדוגמת: int mat[][COLS] ניתן לזמן אותה עם מערכים דו-ממדיים שונים הנבדלים זה מזה במספר שורותיהם, ובלבד שבכולם יש COLS עמודות. על דרך השלילה: לא נוכל להעביר לפונ' הנ"ל מערך: int matrix2[ROWS][COLS+1] עמודות. עבור מערך כזה יהיה עלינו לכתוב פונ' אחרת!

כמו עם מערך חד-ממדי, אנו רשאים לכתוב את הפרוטוטיפ גם באופן:
```
int sum_matrix(const int mat[ROWS][COLS]) {...}
```
אולם התוספת של ROWS לפרוטוטיפ היא חסרת משמעות עבור הקומפיילר.

באופן כללי, עת אנו מעבירים מערך לפונקציה אנו רשאים להשמיט מתיאור הפרמטר את המספר המציין את מספר הערכים בקואורדינאטה השמאלית ביותר. על כן, אם הגדרנו:
```
int grades[NUM_OF_CLASSES][NUM_OF_STUD][NUM_OF_COURSES] ;
```
אזי את הפונ' המקבלת מערך כנ"ל נוכל להגדיר באופן:
```
void f(const int grades[][NUM_OF_STUD][NUM_OF_COURSES])
{
    ...
}
```
נסביר: מערך תלת ממדי הוא למעשה מערך חד-ממדי שכל תא בו הוא מערך דו-ממדי (או ליתר דיוק כל תא בו הוא מערך חד-ממדי שכל תא בו הוא מערך חד-ממדי). את גודלו של המערך החד-ממדי אנו פטורים מלציין בתיאור הפרמטר, וכך עשינו.

7.3.6 העברת תא במערך כפרמטר לפונקציה

נניח כי כתבנו פונקציה המצפה לקבל כפרמטר משתנה פרימיטיבי כלשהו (מטיפוס int, float, bool וכדומה). אין כל מניעה להעביר לפונקציה כארגומנט תא כלשהו במערך. נציג מספר דוגמות פשוטות שידגימו זאת.

לדוגמה הפונקציה:
```
void sum(int num1, int num2) {
   cout << num1 + num2 ;
}
```
ונניח כי בתכנית הוגדרו: int a, b, arr[10]; ולמשתנים הללו הוכנסו ערכים. אזי כל הקריאות הבאות לפונקציה הן לגיטימיות: sum(17, 3879);
```
     sum(17, a);  sum(a, arr[1]);   sum(arr[0], arr[9]) ;
```

כפי שציינו בעבר, תא במערך של מספרים שלמים, הוא משתנה שלם, וכל פעול שניתן לבצע על משתנה שלם ניתן לבצע גם על תא במערך, בפרט העברתו כארגומנט לפונקציה.

נדגים את אותו עיקרון עבור פרמטרי הפניה. נציג את הפונקציה swap אשר מחליפה בין ערכיהם של שני משתנים:

```
void swap(int &num1, int &num2) {
   int temp = num1 ;
   num1 = num2 ;
   num2 = temp ;
}
```

ל- swap ניתן לקרוא בכל אחד מהאופנים הבאים:

```
swap(a, b);
                    swap(arr[0], arr[9]);  swap(a, arr[1]);.
```

7.4 משתנה סטטי בפונקציה

נניח כי בתכנית אשר אנו כותבים עלינו לבדוק מספר רב של פעמים האם מספרים טבעיים כלשהם (בתחום ידוע מראש 2..N) ראשוניים או פריקים. מכיוון שעלינו לבצע את התהליך פעמים רבות אנו מעדיפים שלא לשלם על כל בדיקה עבודה בשיעור שורש המספר שאת ראשוניותו עלינו לבדוק. איך נשיג זאת? נרצה לשמור מערך של בולאניים בשם primes, באופן ש- primes[i] יעיד האם i ראשוני או פריק. בעבר ראינו כבר את אלגוריתם הכברה אשר מאפשר בניית מערך כנדרש. דרישה נוספת שנציב בפני עצמנו היא שהבדיקה האם המספר ראשוני תיערך בפונקציה ייעודית is_prime, אשר תחזיר ערך בולאני. התכנית הראשית, למשל, תוכל לקרוא לפונקציה is_prime(num). נציג גרסה ראשונה (שגויה) של הפונקציה:

```
bool is_prime(int num)
{
   bool primes[N] ;

   for (int i = 2; i  < N; i++)
       primes[i] = true ;

   for (i = 2; i < N; i++)
   {
      if (primes[i])
      {
         for (mul = 2*i; mul < N; mul += i)
             primes[mul] = false ;
      }
   }

   if (num >= 2 && num < N)
      return( primes[num] ) ;
   return( false ) ;
}
```

<u>הסבר התכנית</u>: לא נעמוד על האלגוריתם שכן הוא מוכר לנו כבר מהפרק שדן במערכים חד-ממדיים. באופן כללי, לולאת ה- for מכניסה ערכים לַמערך כנדרש; אחַר תוך שימוש בַמערך אנו מחזירים את הערך הרצוי. אם כך האם התכנית

שכתבנו עונה על הדרישות שהצבנו בפני עצמנו? לא ולא! שכן *בכל* קריאה לַפונקציה ראשית נבנה את המערך, תוך השקעת עבודה רבה, שלא לומר מיותרת. אם הפונקציה תיקָרֵא X פעמים אזי X פעמים נבנה את המערך, וזה ממש לא מה שרצינו: כך אנו לא חוסכים בעבודה, נהפוך הוא אנו עושים עבודה מיותרת. אנחנו רצינו לבנות את המערך *פעם יחידה* (מן הסתם בקריאה הראשונה לפונקציה), כך שבקריאות השנייה ואילך לפונקציה נוכל כבר להשתמש בערכים השמורים במערך.
איפה, אם כך, טעינו? טעינו מכיוון שהמערך המוגדר בַּפונקציה הוא **משתנה אוטומטי** ולכן הוא אינו שומר על ערכיו מקריאה אחת לפונקציה לַקריאה השנייה, ולכן בכל קריאה לפונקציה עלינו לבנותו מחדש, ואין ביכולתנו להסתמך בקריאה השנייה לפונקציה על הערכים שהקריאה הראשונה לפונקציה הכניסה לתאי המערך. בשפת C המחדל הוא שכל משתנה לוקלי בפונקציה הוא משתנה אוטומטי, (אלא אם הוא הוגדר אחרת). מה שאנחנו זקוקים לו הוא **משתנה סטטי**: כזה השומר על ערכיו בין קריאה לקריאה. אם הפונקציה תכניס בקריאה הראשונה ערכים לתוך מערך סטטי, אזי בקריאה השנייה לפונקציה כבר נוכל להסתמך על הערכים המצויים במערך (ולא נאלץ לבנות את המערך מחדש). כדי להשלים את צרכינו עלינו גם לדעת, עת אנו נכנסים לביצוע הפונקציה, האם זו הקריאה הראשונה לפונקציה: ולכן בה עלינו ראשית לבנות את המערך, או שזו קריאה שאינה הראשונה, ואז אנו רשאים להשתמש בערכים המצויים כבר במערך. משתנה סטטי נוסף יסייע לנו לשמור גם מידע זה. לסיום נעיר כי אם אנו מגדירים משתנה סטטי, ומאתחלים אותו בהגדרה, אזי האיתחול מבוצע רק עת אנו נכנסים לביצוע הפונקציה בפעם הראשונה (ולא בפעמים הבאות בהן הפונקציה מתבצעת). נציג עתה את הפונקציה בגרסתה השנייה והנכונה:

```
bool is_prime(int num)
{
   static bool primes[N] ;
   static bool first_call = true ;

   if (first_call) // only in the first call build
   {               // the array
      for (int i = 2; i  < N; i++)
         primes[i] = true ;

      for (i = 2; i < N; i++)
      {
         if (primes[i])
         {
            for (mul = 2*i; mul < N; mul += i)
                primes[mul] = false ;
         }
      }
      first_call = false ;
   }

   if (num >= 2 && num < N)
      return( primes[num] ) ;
   return( false ) ;
}
```

<u>הסבר הפוני</u>: המשתנה הבולאני הסטטי first_call מעיד האם זו הקריאה הראשונה לפונקציה. ערכו מאותחל בהגדרה ל- true. לכן עת נכנס לביצוע

הפונקציה בפעם הראשונה התנאי `if (fist_call)` מתקיים, ואנו נכנסים לבלוק בו אנו בונים את המערך. בסיומו של בלוק זה אנו מבצעים את הפקודה: `first_call = false;`. מכיוון שהמשתנה הוא סטטי אזי בקריאות השנייה ואילך (בהן לא מבוצע האיתחול שבהגדרה) ערכו של המשתנה יהיה `false`, ועל-כן בקריאות השנייה ואילך כבר לא נבנה את המערך. הן בקריאה הראשונה, והן באלה שתבואנה אחריה, נשתמש בערכם של תאי המערך כדי להחזיר את הערך המבוקש. בקריאות השנייה ואילך ההחזרה תיעשה בלי שהרצנו לולאה כלשהי, ועל-כן במחיר קבוע, כפי ששאפנו מלכתחילה. מכיוון שגם המערך הוא סטטי אזי הערכים שהוכנסו לו בקריאה הראשונה לפונ' מחכים לנו בו עת אנו נכנסים לפונ' בפעם השנייה ואילך.

בְּמקום להשתמש במשתנים לוקליים סטטיים לַפונקציה `is_prime` יכולנו להשתמש במשתנים גלובליים, ולהשלים באמצעותם בהצלחה את המשימה שהצבנו בפנינו. לא עשינו זאת שכן, כפי שכבר ציינו, ככלַל שימוש במשתנים גלובליים נחשב לדבר בלתי רצוי.

המשתנים הסטטיים בהם עשינו שימוש בתכנית זאת היו לוקליים (לפונקציה `is_prime`). בשפת `C` קיימים גם משתנים סטטיים גלובליים, אולם משמעות המאפיין `static` עבור משתנים גלובליים, שונה *לחלוטין* ממשמעותו עבור משתנים לוקליים. אנו לא נעמוד כאן על משמעות המאפיין `static` עבור משתנים גלובליים. לכל היותר נביע את צערנו על כך שמפתחי שפת `C`, כנראה, שלא ניחנו בדמיון פורה, ועל-כן הם השתמשו באותה מילה לציון שתי תכונות שונות לגמרי, האחת של משתנים לוקליים (השומרים על ערכם בין קריאה לקריאה לפונקציה), והשנייה של משתנים גלובליים.

7.5 שימוש בפונקציות לכתיבת תכניות מודולאריות

שימוש מושכל ונכון בפונקציות הוא תנאי הכרחי לכתיבת תכניות טובות. המטרה אותה אנו שואפים להשיג היא כתיבת תכנית **מודולארית** המורכבת ממספר פונקציות (ממספר אבני בניין). כל פונקציה תבצע תת-משימה של המשימה הכללית המבוצעת על-ידי התכנית. כאשר התכנית גדולה דיה אזי גם תת-משימה שלה עשויה להיות מורכבת, ולכן תחולק גם היא בין מספר פונקציות נוספות, וכן הלאה.

השיטה באמצעותה נהוג לכתוב תכניות מודולאריות נקראת **עיצוב מעלה-מטה** (top-down design). על-פי גישה זאת, עת אנו ניגשים לכתוב את התכנית אנו אומרים לעצמנו: "לו הייתה עומדת לרשותי פונקציה המבצעת ..., וכן פונקציה המבצעת ..., ואולי גם פונקציה שלישית העושה ..., אזי כתיבת התכנית הייתה הופכת להיות משחק ילדים". אנו ממשיכים במשחק הנדמה-לי הזה, וכותבים את התכנית הראשית תוך הנחה שהפונקציות עליהן חלמנו אכן קיימות (או לכל הפחות שיהיה 'שוודי קטן' שיכתוב אותן עבורנו). עתה אנו מתפכחים מאשלייתנו (אפיה שוודים? ואיפה השוודיות?), ופונים לכתוב כל אחת ואחת מהפונקציות אותן שיווינו בעיני רוחנו. אם תוך כדי הכתיבה נראה לנו ש: "היה נחמד לו הייתה עומדת לרשותנו גם הפונקציה..."', אזי אנו מניחים גם את קיומה, וכך הלאה. באופן כזה אנו 'דוחפים כל פעם מתחת לשטיח' חלק מהמשימות אותן עלינו לתכנת, ועל-ידי כך משלימים (עם חורים), את המשימה הכללית יותר הניצבת בפנינו בכל שלב. hopefully עת אנו 'מרימים את השטיח', כדי להשלים את מה שדחפנו מתחתיו, מתברר לנו שמה שעלינו להשלים עתה הוא פשוט יותר (שכן הוא רק חלק מהמשימה שעמדה בפנינו עת דחפנו את אותו מרכיב מתחת לשטיח).

מניסיוני אני יודע שקל להגיד, קשה יותר לעשות, והמציאות מגלה שרכישת המיומנות של תכנות מודולארי, בשיטת עיצוב מעלה-מטה, היא אחת המשימות היותר מרכזיות הניצבות בפני מתכנתים מתחילים. למתכנתים מתחילים לעיתים טבעי יותר לעבוד דווקא בשיטת התכנון מטה-מעלה: ראשית הם מצליחים (בדי עמל) לכתוב תת-משימה כלשהי בתכנית אותה הם התבקשו לכתוב. עתה הם פונים לכתיבת תת-משימה שנייה. בשלב השלישי הם ניצבים בפני השאלה: כיצד לתפור את שתי תת-המשימות שהם השלימו לכדי חלק ראשון של התכנית השלמה? טלאי מפה, וטלאי משם מאפשר להם לבצע איכשהו את החיבור, ולהמשיך הלאה. באופן זה התכנית הולכת ונבנית, ועמה גם הטְלאים, ובסופו של יום התכנית נראית כמו כביש ממוצע בחבל ארץ זנוח במדינה יָם תיכונית שבשמה לא אנקוב. על-כן אני ממליץ לכם מאוד, למרות המאמץ הכרוך בכך, לרכוש את מיומנות התכנות בשיטת עיצוב מעלה-מטה.

התם, שלא לומר הרשע, עשוי לקום ולשאול מה כל-כך טוב בתכנית מודולארית? מדוע זה כל-כך חשוב לתכנת באופן זה? התשובה היא ש:

א. אם תכניתכם כתובה בצורה מודולארית אזי עת אתם, או עמיתכם, נאלצים לתקן או לשפר תכנית אותה אתם לא מכירים, אתם יכולים להתמקד בַּפונקציה המבצעת את תת-המשימה המתאימה, ולשנות רק אותה. אם בהמשך יתברר שהתיקון במקום לשפר רק קלקל, אזי הקלקול יהיה תַחום לפונקציה\ות מסוימות.

ב. במידה וכתבתם בעבר פונקציה המבצעת תת-משימה כלשהי, ועתה עליכם לכתוב תכנית המחייבת ביצוע של אותה תת-משימה, תוכלו לשתול את הפונקציה הדרושה בתכנית החדשה שאתם כותבים, ולקרוא לה מהמקום הדרוש בתכנית החדשה. באופן כללי יותר, אנו יכולים לבנות ספריות של פונקציות שימושיות, ולעשות בהן שימוש בכל עת שנזדקק למי מהן.

בהמשך נלמד כיצד אוסף פונ' המבצעות משימות דומות נארזות יחד לכדי ספריה, או מודול, אותו ניתן לכרוך (link) עם כלל התכנית, בדומה לאופן בו, לדוגמה, נארזו הפונ' המטפלות במחרוזות לכדי ספריה בה אנו עושים שימוש עת ברצוננו לטפל בתכניתנו במחרוזות.

מספר הערות טכניות אודות תכנות עם פונקציות:

א. כלל הזהב למתכנת המתחיל הוא כי גודלה של פונקציה לא יעלה על 'שיבר', במילים אחרות על חלון בינוני על מסך המחשב. פונקציה גדולה יותר תחולק לפונקציות משנה.

ב. תיעוד של פונקציה יכלול את המרכיבים הבאים:

1. הסבר אודות כל אחד ואחד מהפרמטרים של הפונקציה: מה הפונקציה מקבלת ומה הפונקציה מחזירה באמצעות אותו פרמטר.
2. הסבר קצר אודות המשימה שהפונקציה מבצעת, בפרט מה הערך שהיא מחזירה בפקודת ה- return שלה.

ג. *זוהי נקודה חשובה ביותר: על התכנית הראשית להוות מעין תיאור סכמטי של הפעולות המרכזיות המבוצעות על-ידי התכנית. לכן התכנית הראשית תכלול מספר פקודות תנאי ולולאה אשר מזמנות פונקציות המבצעות את תת-המשימות מרכזיות של המשימה אותה על התכנית להשלים*. התכנית הראשית לא תכלול פקודות קלט\פלט או פקודות השמה.

ד. צדו השני של המטבע שתואר בסעיף הקודם הוא שיש להקפיד שלא לַגְוון את התכנית הראשית ניוון יתר. לעיתים רואים תכניות בהן התכנית הראשית כוללת קריאה לפונקציה אחת או שתיים ותו לא. זה לא טוב, שכן באופן זה התכנית

הראשית אינה כוללת ביטוי לכל אחת מהפעולות המרכזיות המבוצעות בידי התכנית.

ה. מבחינת סידור הפונקציות בקובץ המקור: נהוג לכתוב ראשונה את התכנית הראשית, ואחר-כך בסדר חשיבות יורד את יתר הפונקציות. לפני כל פונקציה יבוא תיעוד קצר אודותיה. כדי להקל על הקורא להבחין היכן מתחילה ונגמרת כל פונקציה יש הנוהגים להפריד בין הפונקציות השונות בקו:

//----------

בשני הסעיפים הבאים נציג שתי תכניות הכוללות פונקציות וכתובות בצורה מודולארית.

7.5.1 תכנית להצגת שורשי משוואה ריבועית

בעבר כבר ראינו תכנית המבצעת משימה זאת. עתה נציגה תוך שימוש בפונקציות. נדוש כאן את נושא הפונקציות עד דק (באופן מוגזם) כדי להדגים את השימוש בהן לשם כתיבת תכנית שלמה.

```
enum sols_t {NO_RT, SINGLE_RT, TWO_RT, INFINITE_RT } ;
//-------------------------------------------------------------------------------------------
int main()
{
   duoble   a, b, c,
            x1, x2 ;
   enum sols_t num_of_sols ;              //how many roots were
found

   read_data(a, b, c) ;
   if (a == 0)
      num_of_sols = solve_linear(b, c, x1) ;
   else
       num_of_sols = solve_quadratic(a, b, c, x1, x2) ;

   display_sol(num_of_sols, x1, x2) ;

   return(EXIT_SUCCESS) ;
}
```

התכנית הראשית שלנו כוללת ביטוי לפעולות המרכזיות של התכנית:

א. קריאת הקלט באמצעות הפונקציה `read_data`.

ב. במידה והמשוואה אינה ממעלה שנייה קריאה לפונקציה `solve_linear` אשר פותרת משוואה ממעלה ראשונה. הפונקציה תחזיר כמה פתרונות היא מצאה (אפס, אחד, או אינסוף), במידה ונמצא פתרון יחיד אזי ערכו יוחזר בפרמטר המשתנה x1.

ג. במידה והמשוואה ממעלה שנייה אזי קריאה לפונקציה `solve_quadratic` אשר פותרת משוואה ממעלה שנייה. הפונקציה תחזיר כמה פתרונות היא מצאה (אפס, אחד או שניים). הפתרונות עצמם יוחזרו בפרמטרים המשתנים `x1, x2`.

ד. הצגת הפתרונות.

הטיפוס `sols_t` הוא טיפוס בן מניה (`enum`) שערכיו מציינים את מספר הפתרונות שנמצאו למשוואה ריבועית. הגדרתו:

```
enum sols_t {NO_RT, SINGLE_RT, TWO_RT, INFINITE_RT } ;
```

נפנה עתה לכתיבת הפונקציות השונות (שימו לב כי אנו עובדים בשיטת העיצוב מעלה מטה). הפונקציה read_data פשוטה למדי:

```
// this function reads the three params of the equation into its
// three params.
// the function returns through its params the 3 params of the
// equation.

void read_data(double &a, double &b, double &c)
{
   cin >> a >> b >> c ;
}
```

הפרמטרים של הפונקציה הם, כמובן, פרמטרי הפניה, שכן על הפונקציה לשנות את ערכם (עליה לקרוא לתוכם נתונים, שיישארו בארגומנטים המתאימים בתום ביצוע הפונקציה).

הפונקציה solve_liner:

```
// this function solves a linear equation of the form: y = a*x +b.
// it returns how many roots there are.
// in the param root it returns the value of the only root
// (if such exists)

enum sols_t solve_linear(double a, double b, double &root)
{
   if (a == 0)
   {
      if(b ==0)
         return(INFINITE_RT) ;
      return(NO_RT);
   }
   root = (-b)/a ;
   return(SINGLE_RT) ;
}
```

עתה נציג את solve_quadratic:

```
// this func solves a quadratic equation with param a != 0.
// a, b, c are the parameters of the equation.
// x1, x2 are its roots.
// the return statement returns how many roots were found.

enum sols_t solve_quadratic(double a, double b, double c,
                            double &x1, double &x2)
{
   double disc ;
   bool disc_not_neg = find_disc(a, b, c, disc) ;

   if (!disc_not_neg)
        return(NO_RT) ;

   if (disc == 0)
   {
      x1 = (-b)/(2*a) ;
      return(SINGLE_RT) ;
   }

   // if we reach this point then the equation has two roots
   x1 = ( (-b) + disc )/ (2*a) ;
   x2 = ( (-b) - disc )/ (2*a) ;
```

```
   return(TWO_RT) ;
}
```

בהתאם לעקרונות התכנון מעלה-מטה, הפונקציה solve_quadratic עושה שימוש בפונקציה find_disc. פונקציה זאת מקבלת את הפרמטרים a, b, c של המשוואה הריבועית ומחזירה ערך בולאני המורה האם הדיסקרימיננטה חיובית או שלילית. במידה והדיסקרימיננטה חיובית מחזירה הפונקציה בפרמטר disc את שורש הדיסקרימיננטה.

```
// the function receives the three params of the quadratic eq.
// if the discriminant is negative it returns false,
// else it returns true,
// and in the param disc it returns the root of the disc'

bool  find_disc(double a, double b, double c, double &disc)
{
   double temp = b*b -4*a*c ;

   if (temp < 0)
      return(false) ;

   disc = sqrt(temp) ;
   return(true) ;
}
```

כדי להשלים את המסכת נציג עתה את התכנית השלמה מהחל ועד כלה, כולל הכול (פרט להנחיות אודות הקלט, והסבר אודות הפלט):

```
//*****************************************************************
*
//                Ex #17: Solve a quadratic equation
//                ==================================
//                Writen by: Yosef Cohen, Id: 333444555
//
// This program solves a quadratic equation of the form:
// y = a*x*x +b*x + c,
// it reads from the user.
// If the equation is actually linear it is solved using the
formula:
// x = -c/b.
// else it is solved using the well known algorithm:
// x1, x2 = [ -b +- sqrt( b*b - 4*a*c) ] / (2*a)
//
//*****************************************************************
#include <iostream>
#include <cmath>
#include <cstdlib>
//----------------------- using section ----------------------------
using std::cin ;
using std::cout ;
using std::endl;
//----------------- constants and enum section --------------------

enum sols_t {NO_RT, SINGLE_RT, TWO_RT, INFINITE_RT } ;

//------------------ prototypes --------------------------
void read_data(double &a, double &b, double &c);
enum sols_t solve_linear(double a, double b, double &root);
```

```
enum sols_t solve_quadratic(double a, double b, double c,
                            double &x1, double &x2);
bool  find_disc(double a, double b, double c, double &disc);
void display_sols(enum sols_t num_of_sols,
                  double root1, double root2) ;

//-----------------------------------------------------------------
int main()
{
   double   a, b, c,
            x1, x2 ;
   enum sols_t num_of_sols ;              //how many roots were
found

   read_data(a, b, c) ;
   if (a == 0)
      num_of_sols = solve_linear(b, c, x1) ;
   else
       num_of_sols = solve_quadratic(a, b, c, x1, x2) ;

   display_sol(num_of_sols, x1, x2) ;

   return(EXIT_SUCCESS) ;
}
//-----------------------------------------------------------------
-
// this func solves a quadratic equation with param a != 0.
// a, b, c are the parameters of the equation.
// x1, x2 are its roots.
// the return statement returns how many roots were found.

enum sols_t solve_quadratic(double a, double b, double c,
                            double &x1, double &x2)
{
   double disc ;
   bool disc_not_neg = find_disc(a, b, c, disc) ;


   if (!disc_not_neg)
        return(NO_RT) ;


   if (disc == 0)
   {
      x1 = (-b)/(2*a) ;
      return(SINGLE_RT) ;
   }


   // if we reach this point then the equation has two roots
   x1 = ( (-b) + disc )/ (2*a) ;
   x2 = ( (-b) - disc )/ (2*a) ;
   return(TWO_RT) ;
}

//-----------------------------------------------------------------
-
// this function solves a linear equation of the form: y = a*x +b.
// it returns how many roots there are.
// in the param root it returns the value of the only root
// (if such exists)
```

```
enum sols_t solve_linear(double a, double b, double &root)
{
   if (a == 0)
   {
      if(b ==0)
         return(INFINITE_RT) ;
      return(NO_RT);
   }
   root = (-b)/a ;
   return(SINGLE_RT) ;
}

//-----------------------------------------------------------------
-
// the function receives the three params of the quadratic eq.
// if the discriminant is negative it returns false,
// else it returns true,
// and in the param disc it returns the root of the disc'

bool  find_disc(double a, double b, double c, double &disc)
{
   double temp = b*b -4*a*c ;

   if (temp < 0)
      return(false) ;

   disc = sqrt(temp) ;
   return(true) ;
}

//-----------------------------------------------------------------
-
// this function reads the three params of the equation into its
// three params.
// the function returns through its params the 3 params of the
// equation.

void read_data(double &a, double &b, double &c)
{
   cin >> a >> b >> c ;
}

//-----------------------------------------------------------------
-
void display_sols(enum sols_t num_of_sols,
                  double root1, double root2)
{
   if (num_of_sols == SINGLE_RT)
      cout << root1 << endl ;
   else if (num_of_sols == TWO_RT)
            cout << root1 << " " << root2 << endl ;
   else if (num_of_sols == INFINITE_RT)
          cout << "Every x solves\n" ;
   else
         cout << "No real root\n" ;
}
```

7.5.2 משחק החיים

לסיכום נושא הפונקציות נציג תכנית עם מעט יותר בשר, ואשר תדגים באופן מלא יותר את עקרונות התכנות המודולארי.
התכנית שנכתוב מוכרת בשם משחק החיים (Game of life), והיא מבצעת סימולציה פרימיטיבית של מושבת חיידקים פשוטה.

מושבת החיידקים מתוארת על-ידי מערך דו-ממדי; כל תא במערך מייצג מיקום אפשרי של חיידק במושבה והוא עשוי להכיל חיידק או להיות ריק. נגדיר את התאים הסמוכים לתא כלשהו במערך כשמונת התאים שמקיפים אותו מלמעלה, למטה, ימינה, שמאלה, ובאלכסון (מטבע הדברים לתא המצוי בשולי המערך יש פחות משמונה תאים סמוכים).

מושבת החיידקים מתנהלת מדור לדור; את מצב החיים במושבה בדור הראשון קובע המשתמש אשר מזין לתכנית את האינדקסים של התאים בהם יש חיים בדור זה.

מכאן ואילך מצב החיים במושבה נקבע על ידי התכנית באופן הבא:
א. חיידק לו ארבעה שכנים או יותר (בדור הנוכחי) ימות **בדור הבא** מצפיפות.
ב. חיידק לו שכן יחיד או פחות (בדור הנוכחי) ימות **בדור הבא** מבדידות.
ג. חיידק לו שניים או שלושה שכנים ימשיך לחיות.
ד. במקום ריק במושבה עם שלושה שכנים יינלד **בדור הבא** חיידק חדש.
ה. תא ריק לו מספר שכנים שונה משלושה יישאר ריק.

התכנית, תתנהל בלולאה. בכל איטרציה בלולאה תחשב התכנית על-פי הדור הנוכחי את הדור הבא, ותציגו בפני המשתמש, וחוזר חלילה. התכנית תתנהל מדור לדור כל עוד חל שינוי במושבה (במילים אחרות כל עוד הדור הבא שונה מהדור הנוכחי), וכן המשתמש, אחרי שהוצגה בפניו המושבה, מבקש להמשיך ולחשב דור נוסף.

שימו לב כי מכיוון שמצב החיים מחושב עבור הדור הבא על פי הדור הנוכחי, הרי אין משמעות לסדר בו אנו סורקים את המערך, שכן שכניו של כל תא נספרים בדור הנוכחי, ורק בהמשך, בדור הבא, יינצרו / יוכחדו החיים בתא או בשכניו.

נתחיל בהגדרת הקבועים להם נזדקק:

```
#include <iostream>
#include <cstdlib>
//-----------------------------------------------------------------
-
const int N = 10,          // Size of colony as user sees it.

   TOO_CROWDED = 4,         // Thresholds 4
   BORN_QUANTITY = 3,    creation/death
   TOO_LONELY = 1 ;

const bool EMPTY = false, // colony cells values
           LIFE = true;
//-----------------------------------------------------------------
-
```

עתה נציג את הפרוטוטיפים, שזה המקום המתאים להם בתכנית. הצגת הפרוטוטיפים בשלב זה היא, כמובן, בבחינת חכמה שלאחר מעשה. בדרך כלל רק

אחרי שאנו מסיימים את כתיבת התכנית אנו מסוגלים להציג את הפרוטוטיפים, ואז אנו יכולים לשַתלם במקום זה בתכנית.

```
void init(bool curr[][N+2]) ;
void read_data(bool colony[][N+2]) ;
void display(const bool colony[][N+2], int gen_num);
bool produce_next_gen(const bool curr[][N+2],
                      bool next[][N+2]) ;
bool want_to_continue() ;
```

אנו מתחילים את כתיבת התכנית בתכנון התכנית הראשית. בפרט נתכנן את מבני הנתונים, במילים אחרות את המשתנים המרכזיים באמצעותם תבצע התכנית את משימתה. מבנה הנתונים המרכזי לו נזדקק הוא מערך דו-ממדי של משתנים בולאניים שיקרא `curr` (קיצור של `current`). כל תא במערך יוכל להכיל חיידק, או להיות ריק. בעת הגדרת המערך נשתמש ב-'תעלול': המשתמש ידע כי במערך העומד לרשותו יש `N` שורות, מאחת ועד `N`, ו- `N` עמודות, מאחת ועד `N`. בפועל תחזיק התכנית מערך בן `N+2` שורות (משורה מספר אפס, ועד שורה `N+1`), ו- `N+2` עמודות. מה הועילו חכמים בתעלולם? התועלת היא שגם לתא שבשורה מספר 1, או לזה שבשורה מספר `N` יש שמונה תאים סמוכים. אנו נדאג לכך שבתאים שלקיומם המשתמש אינו מודע לא יהיו חיידקים, ולכן מספר השכנים של תא שישכון בשולי המושבה יהיה כמצופה. נעשה זאת על-ידי שנאתחל את התאים שבשוליים (אלה שלקיומם המשתמש אינו מודע) להיות ריקים, ובהמשך נדאג להשאירם ריקים על-ידי שלא נכניס לתוכם חיים.

פרט למבנה הנתונים המרכזי נזדקק למשתנה נוסף שיכיל את מצב החיים בדור הבא. המשתנה יקרא `next`, ויהיה זהה באופן הגדרתו ל- `curr`. עוד נשתמש במשתנה שימנה כמה דורות יוצרו במושבה, כאשר הקלט שמזין המשתמש ייחשב כדור מספר אפס.

```
int main()
{
   bool curr[N+2][N+2],   // current generation
        next[N+2][N+2],   // next generation

   int gen_counter = 0 ;                  // counter of generations
```

עתה נראה כיצד תיראנה הפקודות שבתכנית הראשית. כפי שאנו זוכרים, פקודות התכנית הראשית צריכות לשקף את תת-המשימות המרכזיות של התכנית. בתכנית שלנו אנו מבצעים:

א. איתחול של מבנה הנתונים `curr` לכדי מערך ריק מחיים. (כפי שנראה בהמשך אין צורך לאתחל גם את המערך `next`).

ב. קריאת מצב החיים התחילי מהמשתמש (לתוך המשתנה `curr`), והצגת מצב המושבה התחילי בפניו.

ג. אחר אנו נכנסים לַלולאה המרכזית של התכנית. בַלולאה אנו:

1. מחשבים את הדור הבא, על-סמך הדור הנוכחי. הפונקציה שמבצעת משימה זאת גם תחזיר לנו איתות האם מצב החיים בדור הבא שונה ממצב החיים בדור הנוכחי.
2. אם חל שינוי בין שני הדורות אנו מציגים את הדור הבא, והופכים אותו להיות הדור הנוכחי על-ידי העתקת תוכנו של המערך `next` על המערך `curr`.
3. אחרי הצגת הדור הבא אנו שואלים את המשתמש האם ברצונו לראות דור נוסף במושבה (או לסיים).

המשך התכנית הראשית יראה לפיכך:

```
   init(curr) ;                              // clear colony matrix
   read_data(curr) ;
   display(curr, gen_counter) ;              // display initial colony

   while (1)
   {
      bool change = produce_next_gen(curr, next) ;
      gen_counter++ ;
      if (change)                // if next generation != current
one
      {
                display(next, gen_counter) ;
                assign(curr, next) ;          // today's next gen'
is
                                              // tmmrw's current gen
          if (! want_to_continue())
             break ;                          // if usr wnts to
fnish
      }
      else
      {
           cout << "\nNo more changes in population.\n";
           break ;
      }
   }
   return(EXIT_SUCCESS) ;
}
//---------------------------------------------------------------
-
```

אחרי השלמת התכנית הראשית נפנה למימוש (כלומר לכתיבת) הפונקציות שאת קיומן הנחנו בתכנית הראשית. ראשונה נדון בפונקציה המרכזית, זו המחשבת את הדור הבא. הפונקציה מקבלת באמצעות הפרמטר `curr` את מצב המושבה בדור הנוכחי, ולפרמטר `next` היא מכניסה את מצב המושבה בדור הבא. (שימו לב כי לפני הפרמטר `curr` מופיע מילת הקוד `const` המציינת כי הפונקציה אינה רשאית להכניס שינוי לתאי המערך `curr`). תוך כדי חישוב הדור הבא בודקת הפונקציה גם האם יש הבדל בין שני הדורות, ובסיום פעולתה היא מחזירה ערך בולאני המציין זאת. הפונקציה משתמשת בפונקצית עזר אשר סופרת כמה שכנים יש לתא רצוי כלשהו במושבה (בדור הנוכחי). את הפרוטוטיפ של פונקצית העזר כתבנו בתוך הפונקציה, שכן רק פונקציה זאת משתמשת בפונקצית העזר, ולכן לגיטימי וגם מתאים למקם בתוכה את הפרוטוטיפ, ולהפכו בכך ללוקלי (כזה המוכר רק בפונקציה הנוכחית).

הפונקציה עוברת על תאי המערך, עבור כל תא ותא היא סופרת את מספר שכניו, ועל פי מספר השכנים קובעת את מצב החיים בדור הבא.

```
bool produce_next_gen(const bool curr[][N+2],bool next[][N+2])
{
   bool change = false,          // change between the 2 generations
   int count_neighbours(int row, int col,             // a prototype
                        const bool colony[][N+2]) ;

   for (int row = 1; row <= N; row++)
      for (int col = 1; col <= N; col++)
      {
               int neighbours = count_neighbours(row, col, curr) ;

               if (neighbours <= TOO_LONELY ||
```

```
                    neighbours >= TOO_CROWDED)
                         next[row][col] = EMPTY ;
               else if (neighbours == BORN_QUANTITY)
                         next[row][col] = LIFE ;
               else
                    next[row][col] = curr[row][col] ;

               if (next[row][col] != curr[row][col])
                  change = true ;
      }
   return(change) ;
}
//-----------------------------------------------------------------
-
```

עתה נציג את פונקצית העזר אשר סופרת כמה שכנים קיימים לתא שמקומו `row, col` במערך `colony`. הפונקציה מבצעת לולאה כפולה אשר סורקת קטע מערך בגודל 3x3, ולכן בן תשעה תאים, הכולל את התא `row, col` יחד עם שמונת שכניו. אם בתא יש חיים אזי יש הספירה שלנו תנפיק ערך גדול באחד מהערך האמיתי (שכן היא תספור גם את התא עצמו), ולכן אם יש בתא חיים אנו מאתחלים את מונה השכנים להיות `-1`, ועל-ידי כך מפצים על החיידק העודף שנִסְפר במהלך ביצוע הלולאה.

```
int count_neighbours(int row, int col, const bool colony[][N+2])
{
   int neighbours = (colony[row][col] == LIFE) ? -1 : 0 ;

   for (int i = row-1; i <= row +1; i++)
      for (int j = col -1; j <= col +1; j++)
          neighbours += int(colony[i][j]) ;
   return(neighbours) ;
}
```

עתה נציג את הפונקציה אשר קוראת את הנתונים התחיליים מהמשתמש. עליה לקבל את המערך לתוכו עליה לקרוא את הנתונים. הפונקציה תחזיק זוג משתנים לתוכם יזין המשתמש קואורדינאטות של תאים בהם יש לשתול חיים בתחילה. לדוגמה, כדי לציין שיש להכניס חיים לתא שבשורה 3, עמודה 7 יזין המשתמש את הערכים `3 7`. הפונקציה תבדוק שהערכים המוזנים חוקיים, ובמידה והם כאלה היא תעדכן את המערך. הפונקציה תתנהל כלולאה אינסופית אשר היציאה ממנה תתבצע באמצעות `break`, עת המשתמש יזין את זוג הערכים `0 0`, ובכך יסמן כי ברצונו לסיים את תהליך הזנת הקלט.

```
void read_data(bool colony[][N+2])
{
   int row, col ;

   while (1)
   {
      cout << "Enter a location of a germ:\n" ;
      cout << "Enter 0 0 to end input: " ;
      cin >> row >> col ;
      if (row == 0 && col == 0)                // end of input
         break ;
      if (row < 1 || row > N || col < 1 || col > N)
      {
                cout << "Index out of range: ("
```

```
                << row << "," << col << ")" << endl;
                continue ;
      }
      colony[row][col] = LIFE ;
   }
}
//-----------------------------------------------------------------
-
```

נפנה לפונקציה display. הפונקציה מציגה את מצב המושבה בַדור מספר gen_num. שימו לב כי אנו מגדירים בה שני קבועים שטיפוסם הוא char. לא דנו עד כה בטיפוס זה, בקצרה נגיד שמשתנה מטיפוס זה יכול להכיל תו (סימן, אות) יחיד. בדרך כלל מקובל להגדיר קבועים מעל ה- main, אולם מכיוון שהקבועים אותם אנו מגדירים כאן משמשים רק פונקציה זאת בחרנו הפעם להגדירם כלוקליים לפונקציה.

```
void display(cost bool colony[][N+2], int gen_num)
{
   int row, col ;
   const char LIFE_SIGN = '+',
              EMPTY_SIGN = ' ' ;

   cout << endl << endl << "Generation #" << gen_num << endl ;
   cout << "==================" << endl << endl ;      // 2 endl

   for (row = 1; row <= N; row++)
   {
      for (col = 1; col <= N; col++)
          cout << ((colony[row][col]== LIFE) ?
                     LIFE_SIGN : EMPTY_SIGN)) ;
      cout << endl ;
   }
}
 //-----------------------------------------------------------------
                                                                   -
```

הפונקציה init מאתחלת את מצב המושבה להיות מושבה ריקה. היא מאתחלת גם את השוליים להיות ריקים. מכיוון שאחר-כך, בפונקציה produce_next_gen איננו מטפלים בתאים שבשוליים אזי הם יישארו ריקים לנצח.

```
void init(bool curr[][N+2])
{
   for (int i = 0 ; i < N+2 ; i++)
      for (int  j = 0 ; j < N+2 ; j++)
          curr[i][j] = EMPTY ;
}
//-----------------------------------------------------------------
-
```

הפונקציה assign מקבלת באמצעות הפרמטר next את מצב המושבה בדור הבא, ומעתיקה אותו על הפרמטר curr. באופן כזה היא הופכת את מה שהיה הדור הבא להיות הדור הנוכחי. שימו לב לַשימוש במילת המפתח const: היא מופיע לפני הפרמטר next, שכן פרמטר זה לא אמור להשתנות על-ידי הפונקציה. כמובן שהיא אינה מופיעה לפני הפרמטר curr אשר אמור להיות מעודכן בידי הפונקציה.

```
void assign(bool curr[][N+2],const bool next[][N+2])
{
```

```
   for (int i = 1 ; i <= N ; i++)
      for (int  j = 1 ; j <= N ; j++)
          curr[i][j] = next[i][j] ;
}
//---------------------------------------------------------------------
-
```

פונקציה זאת קוראת מהמשתמש את הערך אחד אם ברצונו לראות דור נוסף, או את הערך אפס אם ברצונו לסיים. היא מחזירה את הערך הבולאני השקול לערך הנקרא, וכך מאותת לתכנית הראשית אודות רצונו של המשתמש.

```
bool want_to_continue()
{
   int cont ;

   cout << "Do u want to c another generation? (1/0): " ;
   cin >> cont ;
   return( bool(cont)) ;
}
//---------------------------------------------------------------------
-
```

7.6 מחסנית הזיכרון

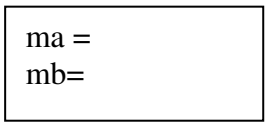
נניח שכתבנו תכנית בשפת C, במילים אחרות יצרנו קובץ מקור. בשלב השני קימפלנו את התכנית, ובכך יצרנו קובץ ניתן להרצה אשר שוכן, מין הסתם, על-גבי מצע האחסון (הדיסק). עתה ברצוננו להריץ את התכנית. לשם כך יש לטעון את התכנית לזיכרון הראשי. פקודות התכנית (בשפת מכונה) נטענות למקטע של הזיכרון הראשי הקרוי **קטע הקוד** (code segment) או **קטע הטקסט**. עת התכנית מתחילה להתבצע יש להקצות בזיכרון הראשי מקום בו יאוחסן תוכן המשתנים של התכנית. בתחילה יכיל המקום רק את משתני ה- `main`. בהמשך, עת תיקראנה פונקציות נוספות, יוחזקו גם הפרמטרים והמשתנים שלהן בזיכרון הנ״ל. האזור בזיכרון שיכיל את ערכי המשתנים, הפרמטרים (ומידע נוסף) יקרא, מסיבות שתובהרנה בהמשך, **קטע המחסנית** (stack segment). בסעיף זה נדון בדרך בה המחשב מנהל את קטע המחסנית של הזיכרון הראשי. נדגיש כי קטע המחסנית אינו שונה מבחינת החומרה מקטע הקוד, מקטע הערמה (אותו נכיר עת נדון במצביעים), או מקטע הנתונים (המכיל את המשתנים והקבועים הגלובאליים של התכנית). ההבדל בין הקטעים הוא רק בדרך בה מערכת ההפעלה מקצה אותם לתכניות (ואחר משחררת אותם). לכן יהיה זה חסר פשר לשאול: מה גודל קטע המחסנית של המחשב? למחשב יש זיכרון ראשי אחד, ומערכת ההפעלה מחלקת אותו לחלקים שונים, אותם היא מנהלת באופן שונה, על-פי צרכיה.

נניח כי התכנית שלנו נראית באופן הבא (השמטנו קטעים שאינם הכרחיים לצורך דיוננו כגון ה- `include` והפרוטוטיפים):

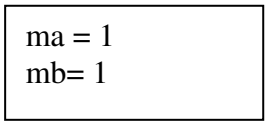
```
int main()
{
   int ma, mb ;

   ma = 1; mb = 1 ;
   f(ma, mb) ;
   cout << ma << mb ;                  // (+)
   mb = 1 ;
   ma = g(mb) ;
```

```
   cout << ma << mb ;                // (-)
   return 0 ;
}

void f(int fa, int &fb)
{
   int fc = 2 ;
   fa++ ; fb++ ;
   cout << fa << fb << fc ;
   ff(fa, fb, fc) ;
   cout << fa << fb << fc ;          // (*)
}

void ff(int ffa, int &ffb, int &ffc)
{

   ffa = ffb = ffc = 0 ;
}

int g(int ga)
{
   ga-- ;
   cout << ga ;
   return( ga -1) ;
}
```

נדון בדרך בה מתנהלת מחסנית הזיכרון לאורך ביצוע התכנית. ראשית מתחילה להתבצע התכנית הראשית. על-גבי המחסנית מוקצה מקום למשתניה של התכנית הראשית. מבחינה ציורית נציג זאת כך:



עת מוכנס ערך למשתנים (בפקודות ההשמה שמופיעות בתחילת ה- `main`) מתעדכנת המחסנית באופן הבא:



עתה התכנית הראשית קוראת לפונקציה `f`. פעולת הקריאה לפונקציה גורמת לכך שעל המחסנית נבנית **רשומת ההפעלה** (activation record) או **מסגרת המחסנית** (stack frame) של הפונקציה `f`. רשומת ההפעלה כוללת:

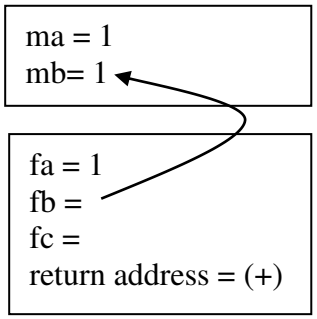
א. עבור כל פרמטר ערך מוקצה ברשומת ההפעלה מקום בו נשמר ערכו של הפרמטר.

ב. עבור כל פרמטר הפניה מוקצה ברשומת ההפעלה חץ. החץ נשלח לעבר הארגומנט המתאים לאותו פרמטר הפניה באותה קריאה. בעת ביצוע הפונקציה, עת התכנית פונה לַפרמטר, עוקב המחשב אחרי החץ המתאים, וניגש ישירות לתא הזיכרון בו נשמר הארגומנט.

ג. רשומת ההפעלה תכלול גם רישום מהי **כתובת החזרה** (return address) של הפונקציה בקריאה הנוכחית. כתובת החזרה היא הפקודה אותה יש לבצע אחרי שהפונקציה תסתיים (במילים אחרות, הפקודה שמופיעה מייד אחרי הקריאה לפונקציה); בדוגמה שלנו, בקריאה ל- `f` מהתכנית הראשית, כתובת החזרה היא פקודת הפלט לצידה כתבנו (+) בתיעוד.

ד. מידע נוסף אשר לא יעניין אותנו בקורס זה.

נציג את מצב המחסנית אחרי בניית רשומת ההפעלה של `f`:



נסביר: לפונקציה `f` שני פרמטרים: הפרמטר `fa` הוא פרמטר ערך, על כן ערכו של הארגומנט המתאים (במקרה שלנו `ma`) מועתק לשטח הזיכרון שהוקצה ל- `fa` על-גבי המחסנית. הפרמטר `fb` הוא פרמטר הפניה, על-כן משטח הזיכרון שהוקצה ל- `fb` על-גבי המחסנית נשלח חץ לעבר הארגומנט המתאים (במקרה שלנו `mb`). לפונקציה `f` משתנה לוקלי `fc`, וגם לו מוקצה מקום על-גבי המחסנית. כמו כן נשמרת ברשומת ההפעלה של `f` כתובת החזרה של הפונקציה.

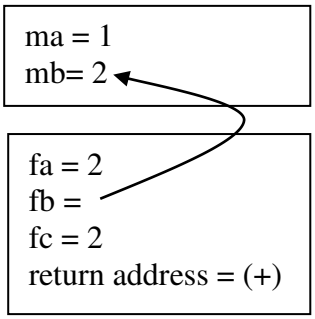
אחרי בניית רשומת ההפעלה של f מתחיל המחשב בביצוע הפונקציה:

א. עת fc מקבל את הערך 2 הדבר מעודכן ברשומת ההפעלה במקום שהוקצה לשמירת ערכו של fc.

ב. עת fa גדל באחד הדבר מעודכן ברשומת ההפעלה במקום שהוקצה לשמירת ערכו של fa.

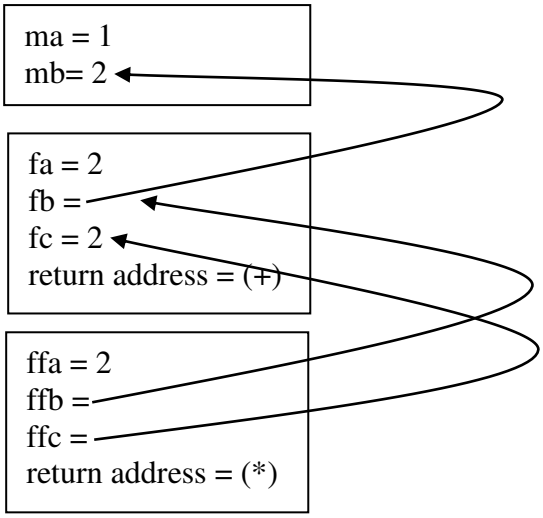
ג. עת fb גדל באחד המחשב הולך בעקבות החץ, ומעדכן את השינוי בשטח הזיכרון של הארגומנט המתאים ל- fb, כלומר בשטח הזיכרון של mb.

נציג את מצב המחסנית:



עתה המחשב מציג את ערכם של fa, fb, fc. את ערכם של fa, fc הוא שולף ישירות מרשומת ההפעלה של f. כדי להציג את ערכו של fb הוא שוב הולך בעקבות החץ ל- mb.

עתה מתבצעת קריאה ל- ff. נעקוב אחר מהלך הקריאה. ראשית נציג את מצב המחסנית אחרי הוספת רשומת ההפעלה של ff:



נסביר: לפונקציה ff שלושה פרמטרים: הפרמטר ffa הוא פרמטר ערך, לפיכך ערכו מועתק מערכן של הארגומנט המתאים (fa), הפרמטר ffb הוא פרמטר הפניה, על-כן נשלח חץ ממנו אל עבר הארגומנט המתאים fb (ומשם נשלח חץ נוסף אל mb), הפרמטר ffc גם הוא פרמטר הפניה, ולכן גם ממנו נשלח חץ אל עבר הארגומנט המתאים (fc). כתובת החזרה של הפונקציה ff היא פקודת הפלט המסומנת ב- (*) בפונקציה f.

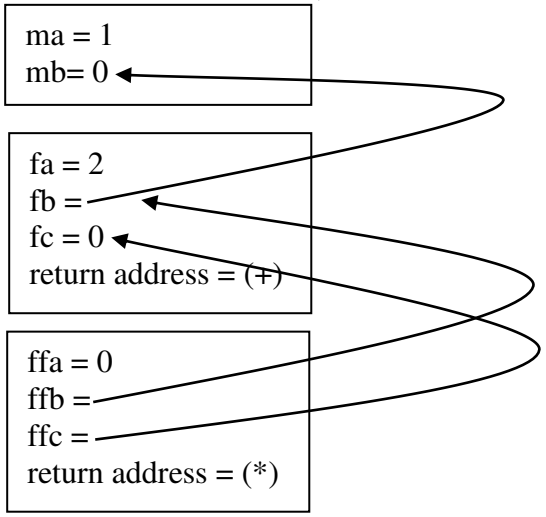
אחרי שגמרנו לבנות את רשומת הפעלה אנו פונים לביצוע הפונקציה:

א. עת הפונקציה מאפסת את ffc הולכים בעקבות החץ, ומאפסים את fc בפונקציה f.

ב. עת הפונקציה מאפסת את `ffb` הולכים פעמיים בעקבות החץ, ומאפסים את `mb` בתכנית הראשית.

ג. עת הפונקציה מאפסת את `ffa` הדבר נעשה בשטח הזיכרון שהוקצה לפרמטר ברשומת ההפעלה של `ffc` עצמה.

נציג את מצב המחסנית בעקבות ביצוע הפעולות הנ"ל:



עתה הפונקציה ffc מסתיימת, לפיכך רשומת ההפעלה שלה מוסרת מעל המחסנית. מכאן אנו גם למדים מדוע שטח הזיכרון נקרא מחסנית: הוא מתנהל בדומה למחסנית שבנשק: רשומת ההפעלה שהושמה עליו ראשונה (זו של main) היא שתוסר ממנו אחרונה, ורשומת ההפעלה שהושמה אחרונה (זו של ffc) היא שתוסר ממנו ראשונה. לעיתים נקראת דרך זאת לטיפול בנתונים בשם last in first out או בקיצור lifo. (בניגוד, למשל, לתור בו הראשון שהגיע ונכנס לתור הוא גם הראשון שיוצא מהתור, כלומר בתור הכלל הוא first in first out או בקיצור fifo).

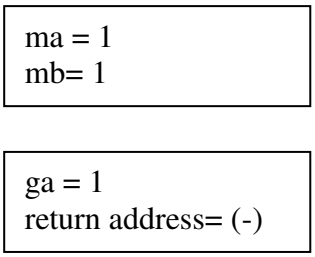
מדוגמה זאת אנו גם למדים כיצד בַּפוֹעל מושג האפקט המוכר לנו: שינויים שהפונקציה הכניסה לפרמטרי ערך לא נשארים בארגומנטים המתאימים אחרי תום ביצוע הפונקציה, וזאת משום שהם הוכנסו ל**העתק** של הארגומנט, ולא לשטח הזיכרון של הארגומנט עצמו. לעומת זאת שינויים שהפונקציה הכניסה לפרמטרי הפניה נותרים אחרי תום ביצוע הפונקציה בארגומנטים המתאימים, וזאת משום שמלכתחילה השינוי בוצע בשטח הזיכרון של הארגומנט, באמצעות הליכה בעקבות החץ.

אחרי תום ביצוע `ff` מתקדמת התכנית לפקודת הפלט המסיימת את `f`. הפקודה תציג את ערכו של `fa` שהוא 2, את ערכו של `fb` שהוא אפס (הוא אופס כזכור על-ידי `ff`), ואת ערכו של `fc` שגם הוא אפס.

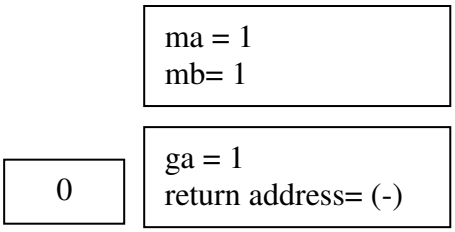
בזאת מסתיים ביצועה של `f`, ורשומת ההפעלה שלה מוסרת מעל המחסנית. שוב אנו רואים כיצד מושג המימוש של פרמטרי הפניה לעומת פרמטרי ערך: ערכו של `ma` לא שונה על-ידי `f` (שכן הפונקציה טיפלה בעותק שלו), אך ערכו של `mb` שונה (שכן הפונקציה שלחה חץ אליו עצמו).

הביצוע חוזר לתכנית הראשית, אשר בפקודת הפלט (+) תציג את ערכו של `ma` (שהוא אחד), ואת ערכו של `mb` (שהוא אפס). עתה התכנית הראשית תעדכן את `mb` להיות שוב אחד, ותקרא לפונקציה `g`. נעקוב אחרי מהלך ביצועה של `g`: לפונקציה

פרמטר ערך יחיד (אשר ישמר על המחסנית). ולכן תיאור המחסנית בעקבות הקריאה ל- g יהיה:



נסביר: ערכו של הארגומנט mb הועתק לשטח הזיכרון שהוקצה לפרמטר הערך ga. אחרי בניית רשומת ההפעלה מתחיל ביצועה של g. מוצג הפלט (1), ואחר הפונקציה מחזירה את ga-1 כלומר את הערך אפס, ובכך מסיימת. כיצד מוחזר הערך? לפני שהפונקציה מסיימת היא 'מניחה בצד' (ליתר דיוק בזיכרון המצוי בתוך המעבד ונקרא אוגרים, או במקום מוסכם על-גבי המחסנית) את הערך המוחזר על-ידה. מבחינה ציורית נתאר זאת כך:



אחרי 'הנחת' הערך, הפונקציה מסיימת, רשומת ההפעלה שלה מוסרת מעל המחסנית (או ליתר דיוק אנו מתייחסים לזיכרון שהוקצה לרשומת ההפעלה שלה כפנוי לשימוש חוזר). לבסוף, הערך ש-'הושם על-ידי הפונקציה בצד' מוכנס למשתנה ma של התכנית הראשית (שכן הקריאה ל- g מה- main הייתה: ma = g (...)). ומתקבל המצב:

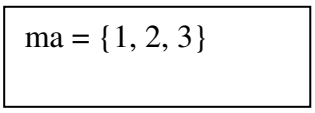
עתה התכנית הראשית מבצעת אף פקודת הפלט המסומנת כ- (-), מחזירה בעצמה את הערך 0, ובכך מסיימת (וגם שטח הזיכרון שהוקצה למשתניה נחשב למשוחרר, וניתן לשימוש חוזר על-ידי מערכת ההפעלה).

מה קורה עת מערך מועבר כארגומנט לפונקציה? את הפרטים המדויקים נוכל ללמוד רק אחרי שנדון במצביעים. בשלב זה נציין כי המערך אינו מועתק על-גבי המחסנית, אלא חץ נשלח מהפרמטר המתאים אל המערך (המועבר כארגומנט). כלומר הטיפול במערך זהה לטיפול בפרמטר הפניה (וזאת בהתאמה למה שכבר ידענו מבחינת האפקט המתקבל). נציג דוגמה קצרה (הכוללת רק את הפרטים ההכרחיים לענייננו):

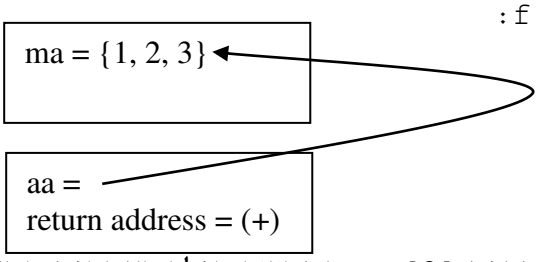
```
int main()
{
   int a[3]= {1, 2, 3} ;
```

```
   f(a) ;
   cout << a[0] << a[1] << a[2] ;        // (+)
   return 0 ;
}

void f(int aa[3])
{
   aa[2] = 17 ;
}
```

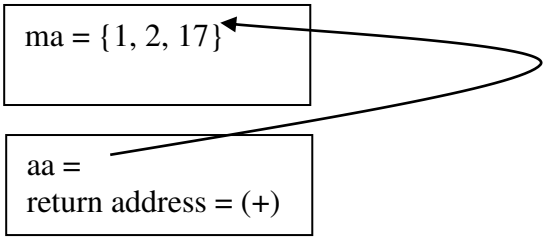
עת התכנית מתחילה לרוץ מוקצה על המחסנית מערך בשם a בן שלושה תאים, אשר גם מאותחל בהגדרה. ציורית נציג זאת כך:



עתה מתבצעת קריאה ל-f אשר מקבלת את המערך כפרמטר. נציג את רשומת ההפעלה של f:



עת f מעדכנת את aa[2], המחשב הולך בעקבות החץ מהפרמטר aa, לארגומנט a, ומעדכן שם את ערכו של התא מספר 2, ואנו מקבלים:



עתה ביצועה של f מסתיים. אנו מתעלמים משטח הזיכרון שהוקצה לרשומת ההפעלה שלה על-גבי המחסנית (במילים פחות מדויקות: רשומת ההפעלה של f מוסרת מהמחסנית). העדכון ש- f הכניסה ל- aa[2] נותר ב- a[2], שכן לשם הוא הוכנס מלכתחילה.

7.7 חוקי ה- scope

נבחן את התכנית הבאה (לא שאני ממליץ לכם לכתוב תכנית שכזאת, אך היא אינה שגויה מבחינת השפה):

```
int a ;              // a global variable named a

int main() {
   int a ;        // a local a of main. hides the global a

   a = 1 ;        // the local a of main gets the value 1
   cout << a ;
```

```
   f(3879) ;
   g(-3879) ;
   h() ;
   return 0 ;
}

void f(int a) {
   cout << a ;
   a = 2 ;        // the parameter a of f gets the value 2
}

void g(int b) {
{
   b = 3 ;
   a = 4 ;                // the global a gets the value 4
}

void h()
{
   int b ;

   cin >> b ;
   if (b % 2 ==0) {
     int a ;              // a local only to this block
     cin >> a ;           // the local a
     cout << a*a ;
   }
   cout << a*a ;      // the local a does not exist here
                      // so we refer to the global a (==4)
   a = 5 ;                // the global a gets the value 5
}
```

בתכנית מוגדרים מספר משתנים בשם a. השאלה שנשאל היא: עת פונקציה כלשהי פונה לְמשתנה בשם a, לאיזה a היא פונה? במילים אחרות היכן מוכר כל משתנה? היכן הוא חי? או מהו ה- scope של המשתנה?

א. נתחיל בתכנית הראשית, בה מוגדר משתנה לוקלי בשם a, ולכן עת התכנית מבצעת את ההשמה a = 1 היא מכניסה ערך למשתנה הלוקלי שלה.

ב. לפונקציה f יש פרמטר בשם a, ולכן עת היא פונה ל- a היא פונה לפרמטר שלה. הן במקרה הקודם, והן במקרה הנוכחי, הפרמטר 'מַסֵך' את a הגלובלי, העיקרון הכללי הוא שתמיד הפניה היא למשתנה הלוקלי ביותר.

ג. בפונקציה g אין לא פרמטר בשם a, ולא משתנה לוקלי בשם a; לכן עת g פונה ל- a, היא פונה ל- a הגלובלי. נשים לב כי זו הפעם הראשונה בתכנית בה מושם ערך ל- a הגלובלי. (אגב, בניגוד למשתנה לוקלי לפונקציה אשר מכיל ערך 'זבל' כל עוד לא הוכנס לו ערך אחר, משתנה גלובלי, כל עוד לא הוכנס לו ערך, מכיל את הערך אפס).

ד. בפונקציה h המצב מעט יותר עדין: בגוש של ה- if מוגדר משתנה בשם a, לכן פקודות הקלט והפלט בגוש זה פונות ל- a המוגדר בגוש. לעומת זאת פקודת הפלט שאחרי הגוש (cout << a*a) מצויה במקום בו a הלוקלי כבר אינו

קיים, ולכן היא מציגה את ריבוע ערכו של a הגלובלי (ולכן את הערך 16). גם עדכון ערכו של a מתבצע על ה- a הגלובלי.

ה. בשפת C++ קיים אופרטור המאפשר לפנות למשתנה גלובלי בשם a, גם אם באותו מקום מוכר משתנה לוקלי בשם a. הכתיבה היא a::‏ (כלומר כתיבת זוג נקודתיים לפני שמו של המשתנה). עצתי לכם אל תהיו כמו הפיקח שנחלץ ממצב אליו החכם לא היה נקלע מלכתחילה.

Overloading 7.8

נניח כי בתכנית כלשהי עלינו לכתוב את שלוש הפונקציות הבאות:

א. פונקציה המחליפה בין ערכי שני משתנים שלמים.

ב. פונקציה המחליפה בין ערכי שני משתנים ממשיים.

ג. פונקציה המחליפה בין ערכי שלושה משתנים שלמים באופן מעגלי: המשתנה השני יקבל את ערכו של הראשון, המשתנה השלישי יקבל את ערכו של המשתנה השני, המשתנה הראשון יקבל את ערכו של המשתנה השלישי.

בשפת C אנו רשאים לכתוב את שלוש הפונקציות באופן הבא:

```
void swap(int &num1, int &num2) {
   int temp = num1 ;
   num1 = num2 ;
   num2 = temp ;
}

void swap(double &num1, double &num2) {
   double temp = num1 ;
   num1 = num2 ;
   num2 = temp ;
}

void swap(int &num1, int &num2, int &num3) {
   int temp = num1 ;
   num1 = num2 ;
   num2 = num3 ;
   num3 = temp ;
}
```

הנקודה אותה ברצוני להדגיש היא כי לכל שלוש הפונקציות קראנו באותו שם, ואין בכך פְּסול. המחשב יודע להבדיל ביניהן, ולהחליט איזה פונקציה עליו להפעיל, שכן יש להן רשימות פרמטרים שונות. במילים פורמאליות יותר יש להן **חתימה** (signature) שונה. על כן אם הגדרתם בתכנית:

```
int i1, i2, i3 ;
double f1, f2 ;
```

ואתם מזַמנים את: swap(i1, i2); מבין המחשב כי יש להפעיל את הפונקציה הראשונה מבין השלוש שכתבנו. אם, לעומת זאת, אתם מזמנים את: swap(f1, f2) ; מבין המחשב כי יש להפעיל את הפונקציה השנייה מבין השלוש; ולבסוף הפקודה: swap(i1, i2, i3); תובן כזימונה של הפונקציה השלישית.

7.9 תרגילים

כתיבת חלק מהתרגילים שהוצגו בפרק שש ללא שימוש בפונקציות היא לכל הפחות איוולת, לכל היותר פשע כנגד האנושות, וללא ספק תכנות קלוקל ביותר. על-כן אתם מוזמנים לכתוב את התרגילים הללו מחדש באופן מודולארי, ועל-פי כללי התכנות הנאות.

התרגילים המופיעים להלן אינם בגדר תכניות שלמות ומלאות לשם תרגול והטמעת השימוש בפונקציות. התרגילים לקוחים מתוך בחינות שניתנו במוסדות להשכלה גבוהים שונים. אתם מתבקשים בהם לכתוב קטע מתכנית, קטע שיבדוק את שליטתכם במכלול הנושאים שנלמדו עד כה. אני ממליץ לכם לפתור תרגילים אלה בעיקר כהתכוננות לבחינה, או כדי *לבדוק* את שליטתכם בחומר. על-מנת *לשפר* את שליטתכם בחומר כדאי שתכתבו תרגילים גדולים יותר, כדוגמת אלה שהוצגו בפרק הקודם.

7.9.1 תרגיל מספר אחד: איתור גרם מדרגות מרבי במערך

יהי a[K][K] מערך דו-ממדי של מספרים שלמים. ההגדרות הבאות מתייחסות למערך זה.

<u>הגדרת מדרגה בגודל m (עבור m => 2) שמתחילה בתא [x][y]:</u>

סידרה "רציפה" של תאים ב- a:

[x][y], [x][y+1], … , [x][y+m-1],
[x+1][y+m-1], [x+2][y+m-1], … , [x+m-1][y+m-1]

כך שכל התאים מחזיקים את אותו הערך.

סדרת התאים נקראת מדרגה שכן התאים מסודרים במערך כמדרגה (בצורת האות 'ר').

<u>דוגמא:</u> נתון מערך 4x4:

	#0	#1	#2	#3
#0	**7**	**7**	**7**	7
#1	2	3	**7**	5
#2	**1**	**1**	**7**	6
#3	0	**1**	9	8

במערך זה קיימות המדרגות הבאות:

1. מדרגה בגודל 2: מתחילה בתא- [0][2] (ותאיה מחזיקים את הערך 1).
2. מדרגה בגודל 3: מתחילה בתא- [0][0] (ותאיה מחזיקים את הערך 7).

<u>הגדרת גרם בן n מדרגות-בגודל-m המתחיל בתא [x][y]:</u>

סידרה "רציפה" של n מדרגות, כל אחת בגודל m, כך שהראשונה בניהן מתחילה בתא [x][y] וכל מדרגה (פרט לראשונה) מתחילה בתא בו נגמרה קודמתה.

<u>דוגמא:</u> נתון מערך בגודל 5x5:

	#0	#1	#2	#3	#4	#5
#0	10	7	7	7	7	12
#1	9	9	9	7	7	13

#2	14	15	9	16	7	7
#3	6	6	9	9	9	7
#4	17	6	3	18	9	19
#5	20	6	6	21	9	22

במערך זה קיימים הגרמים הבאים:

1. גרם בן 3 מדרגות בגודל 2: מתחיל בתא [2][0] עם ערך 7.
2. גרם בן 1 מדרגות בגודל 3: מתחיל בתא [2][0] עם ערך 7.
3. גרם בן 1 מדרגות בגודל 2: מתחיל בתא [3][0] עם ערך 7.
4. גרם בן 2 מדרגות בגודל 3: מתחיל בתא [0][1] עם ערך 9.
5. גרם בן 1 מדרגות בגודל 2: מתחיל בתא [1][1] עם ערך 9.
6. גרם בן 2 מדרגות בגודל 2: מתחיל בתא [3][1] עם ערך 7.
7. גרם בן 1 מדרגות בגודל 2: מתחיל בתא [4][2] עם ערך 7.
8. גרם בן 1 מדרגות בגודל 2: מתחיל בתא [0][3] עם ערך 6.
9. גרם בן 1 מדרגות בגודל 3: מתחיל בתא [2][3] עם ערך 9.
10. גרם בן 1 מדרגות בגודל 2: מתחיל בתא [3][3] עם ערך 9.

- כתבו פונקציה שמקבלת מערך דו ממדי 6x6 ומחזירה (באמצעות return) את גודלו של גרם המדרגות הגדול ביותר במערך (מבחינת כמות המדרגות בגרם). באמצעות פרמטרים (reference) תחזיר הפונקציה מה גודל כל מדרגה בגרם ובאיזה תא הגרם מתחיל.
- במידה וקיימים כמה גרמי מדרגות מקסימליים בגודלם, יש להחזיר את נתוני הגרם שמתחיל בשורה הקטנה ביותר.
במידה וקיימים כמה גרמי מדרגות מקסימליים בגודלם באותה שורה, יש להחזיר את נתוני הגרם שמתחיל בעמודה הקטנה ביותר.
במידה וקיימים כמה גרמי מדרגות מקסימליים בגודלם באותה שורה + עמודה, יש להחזיר את נתוני הגרם שגודל מדרגותיו הוא הגדול ביותר.
(שימו לב שאין צורך "למיין" את הגרמים, התשובה תקבע לפי סדר הלולאות והפעולות שמתבצעות בהן).

7.9.2 תרגיל מספר שתיים: איתור מטוס מרבי במערך

יהי a[c_rows][c_cols] מערך דו-ממדי של מספרים שלמים. ההגדרות הבאות מתייחסות למערך זה. בתוכניתכם עליכם להגדיר את c_cols להיות 17 ואת c_rows להיות 7 .

נגדיר: מטוס במערך כקבוצה המרבית של תאים בעלי אותו ערך במערך כך ש:

א. חלק מהתאים מצויים בשורה/עמודה אחת (והם מכונים גוף המטוס).

ב. חלקם האחר של התאים מצויים בעמודה/שורה ניצבת לראשונה (והם יקראו כנפי המטוס).

ג. אורך גוף המטוס (מקצה לקצה) חייב להיות גדול מאורך כנפי המטוס (מקצה לקצה).

ד. שתי כנפי המטוס חייבות להיות זהות באורכן (בשיעור בו הן בולטות מהגוף) וכן על אורך כל כנף גדול מאפס (כלומר כל כנף בולטת מהגוף).

ה. הכנפיים אינן מתחברות לגוף בקצות הגוף.

דוגמה: נתון המערך

	0#	1#	2#	3#	4#	5#	6#	7#	8#	9#	10#	11#	12#	13#	14#	15#	16#
0#					**5**				4	3	3	3					
1#		6		**5**	**5**	**5**	**5**	**5**	4		3			2		**1**	
2#	6	6	6		**5**				4		3			2	**1**	**1**	**1**
3#		6									3	2	2	2	2	**1**	
4#														2		**1**	

הערה: נניח כי בתאים בהם לא צוין ערך, הערכים אינם בתחום 1..6 וכן שונים זה מזה.

במערך הנ"ל קיימים:

א. מטוס המורכב מ- 1 -ים ומ- 5 -ים.

ב. מספרי 2 אינם יוצרים מטוס: שתי כנפי המטוס אינן זהות באורכן (בין אם נסתכל על הרצף האופקי כגוף המטוס ובין אם נסתכל על הרצף האנכי כגוף המטוס).

ג. מספרי 3 אינם יוצרים מטוס: ה- "כנפיים" מתחברות לגוף בקצהו.

ד. מספרי 4 אינם יוצרים מטוס: ה- "כנפיים" באורך 0.

ה. מספרי 6 אינם יוצרים מטוס: גוף המטוס אינו ארוך מכנפיו.

כתבו פונקציה המקבלת מערך כנ"ל ומדפיסה תיאור מלא של המטוסים בו.

7.9.3 תרגיל מספר שלוש: איתור תולעת מרבית במערך

עבור מערך דו-ממדי של מספרים שלמים, נגדיר **תולעת** באורך k במערך להיות סדרה של k תאים, שכל אחד מהם סמוך לקודמו (כלומר הינו אחד משמונת שכניו במערך, שֶכן הוא מצוי מעליו, מתחתיו, מימינו, משמאלו או באלכסון לו) כך שהערכים בתאים מהווים סדרה עולה של מספרים שלמים עוקבים n, n+1, …, n+k-1 (n, האיבר הראשון בסדרה, עשוי להיות כלשהו). למשל במערך:

3	13	15	17	30
70	51	28	29	55
71	10	52	54	56
72	12	7	53	11

בתא [0][1] מתחילה התולעת באורך שלוש הבאה: 70,…,72 (הנגמרת בתא [0][3]).
בתא [0][1] מתחילה התולעת באורך שבע הבאה: 50,…,56 (הנגמרת בתא [4][2]).
בתא [2][1] מתחילה התולעת באורך שלוש הבאה: 28,…,30 (הנגמרת בתא [4][0]).

כתבו פונקציה המקבלת מערך דו-ממדי אשר ידוע עליו כי כל נתון בו מופיע פעם יחידה, ומחזירה את אורכה ואת נקודת התחלה של התולעת הארוכה ביותר במערך. במידה וקיימות מספר תולעים מקסימליות באורכן ניתן להחזיר אחת מהן כפי רצונכם.

7.9.4 תרגיל מספר ארבע: איתור ריבועי קסם במערך

כתבו פונקציה המקבלת מערך דו-ממדי, ומציגה את כל תת-המערכים מגודל שתיים או יותר המהווים ריבוע קסם. עבור כל תת-מערך יש להציג את הקואורדינטה במערך של הפינה השמאלית העליונה שלו, ואת גודלו.

7.9.5 תרגיל מספר חמש: איתור מסגרת במערך

נתון מערך דו-ממדי של מספרים שלמים בגודל NxN.
נגדיר: **היקף מדרגה k במערך** מורכב מתאי המערך המצויים בשורות k, N-1-k, ובעמודות k, N-1-k ואשר אינם נכללים בהיקפים מדרגה קטנה מ- k. במילים אחרות, אלה התאים המרכיבים את היקף הריבוע בגודל k x k שפינתו השמאלית העליונה היא התא [k][k].

בדוגמה הבאה מסומן היקף מדרגה אחד (באמצעות המספרים 3 ו- 5).

	3	3	3	3	
	5			3	
	3			3	
	3	5	3	3	

נגדיר: **מסגרת** במערך הינה סדרה של היקפים סמוכים זה לזה המקיימים שבכל אחד מהתאים המרכיבים כל אחד מההיקפים מצוי אותו ערך (אם כי היקפים שונים עשויים להיות מורכבים מערכים שונים).

נדגיש כי מסגרת עשויה להיות מורכבת, לדוגמה, מהיקפים מסדר שתיים, שלוש וארבע; אך לא תיתכן מסגרת המורכבת מהיקפים שתיים וארבע בלבד (ללא ההיקף מסדר שלוש). אם בַהיקף מסדר שלוש לא בכל התאים מצוי אותו ערך, אזי במערך יש שתי מסגרות נפרדות, כל אחת מורכבת מהיקף יחיד.

אם בדוגמה הנ"ל היינו משנים את זוג ה- 5 –ים המופיעים בציור להיות 3, וכן היינו קובעים כי בכל התאים בשורה הראשונה, בשורה האחרונה, בעמודה הראשונה, ובעמודה האחרונה מצוי הערך 17, אזי היינו מקבלים מסגרת המורכבת מההיקף מספר אפס, ומההיקף מספר אחד.

כתבו פונקציה המקבלת מערך דו-ממדי של מספרים שלמים המכיל נתונים. הפונקציה תאתר ותחזיר את המסגרת המרבית במערך, כלומר המסגרת המורכבת מכמה שיותר היקפים. בין מסגרות המורכבות מאותו מספר של היקפים העדיפו את זאת הכוללת יותר תאים. הפונקציה תחזיר באמצעות פקודת ה- return שלה כמה היקפים כוללת המסגרת, ובאמצעות זוג פרמטרים משתנים את הפינה השמאלית העליונה של המסגרת.

מה זמן הריצה של פונקציה שכתבתם?

7.9.6 תרגיל מספר שש: איתור סולם מרבי במערך

נגדיר: **סולם באורך k** במערך דו-ממדי בגודל M x N של מספרים שלמים להיות סדרה של תאים, $c_1,\ldots,c_k$ בכולם אותו ערך והם מקיימים כי אם c_i נמצא בשורה מספר row בעמודה מספר col אזי אם row < M-1 אזי c_{i+1} נמצא במערך בשורה מספר row+1 ואחרת (אם row = M-1) c_{i+1} נמצא במערך בשורה מספר אפס. באופן דומה: אם c_i נמצא בעמודה מספר col אזי אם col < N-1 אזי c_{i+1} נמצא במערך

בעמודה מספר col+1 ואחרת (אם col = N-1) אזי c_{i+1} נמצא במערך בעמודה מספר אפס.

לדוגמה במערך:

2	3	**5**	6	8
7	3	4	**5**	8
2	**7**	6	9	**5**
5	9	**7**	2	2
4	4	4	4	4
1	2	5	5	3

קיים סולם באורך ארבע המורכב מתאים שערכם חמש והמתחיל בתא [2][0], וקיים סולם באורך שלוש המורכב מתאים שערכם שבע והמתחיל בתא [0][1].

כתבו פונקציה המקבלת מערך דו-ממדי של מספרים שלמים. הפונקציה תחזיר בפקודת ה- `return` שלה את אורכו של הסולם הארוך ביותר המצוי במערך, ולפרמטרים משתנים תוכנס נקודת ההתחלה של הסולם. במידה והקיימים מספר סולמות מקסימליים באורכם ניתן להחזיר אחד מהם על-פי בחירתכם.

מהו זמן הריצה של הפונקציה שכתבתם?

7.9.7 תרגיל מספר שבע: איתור תת-מטריצות סימטריות במטריצה

מערך דו-ממדי נקרא סימטרי אם לכל תא `a[i][j]` בו מתקיים כי:
`a[i][j] ==a[j][i]`.

כתבו פונקציה המקבלת מערך דו-ממדי, ומחזירה את מקומו של תת-המערך הסימטרי הגדול ביותר המשוכן במערך.

8 רקורסיה

בניגוד לפרקים אחרים בהם אנו מכירים מרכיבים שונים של השפה, פרק זה אינו מלמד היבט חדש של השפה, אלא אוֹפנוּת כתיבת תוכניות. רקורסיה הינה דרך לפתרון בעיות. במילים אחרות זוהי תכונה אפשרית של אלגוריתם: אלגוריתם עשוי להיות רקורסיבי, ועשוי להיות לא רקורסיבי. ישנן בעיות שניתן לפתור בנוחות באמצעות אלגוריתמים רקורסיביים. כבר עתה נזהיר כי בעיות אותן ניתן לפתור בטבעיות גם לא באמצעות אלגוריתם רקורסיבי, עדיף לפתור שלא באמצעות אלגוריתם רקורסיבי, וזאת משום שאלגוריתמים רקורסיביים הינם ראשית קשים להבנה (כפי שמייד נחווה על בשרנו), ושנית יקרים לביצוע (כפי שנסביר בהמשך).

בעיה כלשהי מתאימה לפתרון רקורסיבי אם ניתן לנסח את פתרונה באופן הבא:

א. אם הבעיה מוצגת עבור 'נתונים פשוטים' אזי פתרונה הוא מיידי.

ב. אחרת (כלומר אם הבעיה מוצגת עבור נתונים 'שאינם פשוטים') פתור אותה באופן הבא:

1. פתור את אותה בעיה עבור 'נתונים מעט יותר פשוטים'.
2. בהינתן פתרון הבעיה עבור הנתונים ה-'מעט יותר פשוטים', בצע עוד 'צעד קטן נוסף' אשר ישלים את פתרון הבעיה עבור הנתונים המקוריים.

נדגים את כוונתנו ראשית באופן אינטואיטיבי. נניח שהבעיה בה אנו דנים היא: באיזה קווי אוטובוס עלי לנסוע כדי להגיע מישוב X לישוב Y ? המקרה הפשוט עבור בעיה זאת הוא עת קיים קו אוטובוס ישיר מספר L מהישוב X לַישוב Y. במקרה זה התשובה היא סע בקו מספר L. עתה נניח כי הבעיה מוצגת עבור נתונים שאינם פשוטים, לדוגמה כיצד אגיע מבאר-שבע למטולה? כדי לפתור את הבעיה, תוך שימוש בסכמה הרקורסיבית, נפתור ראשית את אותה בעיה עבור נתונים מעט יותר פשוטים: כיצד נגיע מבאר-שבע לקריית-שמונה. אחרי שנדע את הפתרון לבעיה זאת נוסיף עוד צעד קטן אחד: סע בקו 15 מקרית-שמונה למטולה, ובכך נשלים את פתרון הבעיה עבור נתונים המקוריים. עתה אנו ניצבים בפני הבעיה החדשה: כיצד נגיע מבאר-שבע לקריית-שמונה? גם במקרה זה הנתונים אינם פשוטים, ולכן נשוב ונפתור את הבעיה בשיטת שני השלבים: ראשית נפתור את אותה בעיה עבור נתונים מעט יותר פשוטים: כיצד נגיע מבאר-שבע לתל-אביב? אחרי שנשלים את פתרון הבעיה המעט פשוטה יותר נבצע עוד צעד אחד קטן: ניסע מתל-אביב לקריית-שמונה בקו 840. אם כן עתה אנו ניצבים בפני הבעיה המעט יותר פשוטה: כיצד נגיע מבאר-שבע לתל-אביב? גרסה זאת של הבעיה כבר מוצגת עבור נתונים פשוטים, והפתרון לה הוא מיידי: סע מבאר-שבע לתל-אביב בקו 3879. עתה, אחרי שפתרנו את הבעיה האחרונה (כיצד להגיע מבאר-שבע לתל-אביב) אנו יכולים להוסיף את הצעד הקטן הנוסף (כיצד להגיע מתל-אביב לקריית-שמונה) ובכך להשלים את פתרון הבעיה הפחות פשוטה: כיצד להגיע מבאר-שבע לקריית-שמונה. אחרי שפתרנו גם בעיה זאת אנו יכולים להוסיף את הצעד הקטן הנוסף (כיצד להגיע מקריית-שמונה למטולה), ובכך להשלים את פתרון הבעיה המקורית.

8.1 דוגמות ראשונות

בסעיף זה נציג מספר פתרונות רקורסיביים לבעיות פשוטות. הבעיות שנציג ניתנות לפתרון פשוט וקל גם באמצעות אלגוריתמים לא רקורסיביים, ועל-כן ראוי לפתרן ללא שימוש ברקורסיה. הסיבה להצגת הבעיות והפתרונות המופיעים בסעיף זה

היא לסייע לכם לרכוש את צורת החשיבה הרקורסיבית. כפי שתווכחו כל לולאה ניתן להמיר ברקורסיה, זה לא אומר שכל לולאה גם ראוי להמיר ברקורסיה. מניסיוני אני יודע שתלמידים שרכשו בדִי עמל את צורת החשיבה הרקורסיבית ממהרים לעיתים יתר על המידה להשתמש בה. אל תלקו בהרגל לא רצוי זה.

8.1.1 חישוב n!

הדוגמה בה מקובל לפתוח את הדיון באלגוריתמים רקורסיביים היא זו של n!. כפי שאתם אולי יודעים n! מוגדר להיות המכפלה: n*...*2*1. לדוגמה: 1! = 1, 4! = 1*2*3*4 = 24. ברצוננו לכתוב פונקציה אשר מקבלת מספר טבעי n ומחזירה את n!. כמובן שהדרך הטבעית והפשוטה לכתוב פונקציה זאת היא באמצעות לולאה:

```
unsigned int factorial(unsigned int n)
{
   unsigned int fact = 1 ;

   for (unsigned int i=1; i<= n; i++)
      fact *= i ;

   return fact ;
}
```

אולם אנו רוצים לפתור בעיה זאת באופן רקורסיבי. נפעל על-פי הסכמה שהוצגה בתחילת הפרק:

א. הנתונים הפשוטים, עבורם פתרון הבעיה הוא מיידי, הם עת n == 1, שכן אז n! = 1.

ב. כיצד נפתור את הבעיה עת היא מוגשת עבור נתונים שאינם פשוטים (כלומר עבור n>1)?

1. ראשית נפתור את הבעיה עבור נתונים מעט פחות מורכבים: נחשב את (n-1)!, ואת הפתרון נשמור במשתנה עזר בשם temp.
2. אחרי שנפתור את הבעיה עבור הנתונים המעט יותר פשוטים, נבצע עוד צעד קטן נוסף, והוא חישוב temp*n, ובכך נשלים את פתרון הבעיה עבור הנתונים המקוריים.

עתה נתרגם את האלגוריתם הנ"ל לתכנית מחשב (לַפונקציה שנציג נוסיף פקודת פלט אשר תסיע לנו בהמשך לעקוב אחר מהלך התקדמות התכנית. כמובן שכעיקרון פקודת הפלט מיותרת):

```
unsigned int factorial(unsigned int n) {
   unsigned int temp ;

   if (n == 1)
     return(1) ;
   else {
      temp = factorial(n-1) ;
      cout << "about to return " << temp*n << endl;
      return( temp * n) ;
   }
}
```

נסביר: אם n==1 אנו מחזירים מיידית את הערך 1. אחרת (וחלקכם ודאי שמו לב שה- else הוא מיותר): נחשב ראשית את (n-1)! על-ידי קריאה רקורסיבית לפונקציה. את הערך המוחזר נכניס למשתנה temp, ואז נשיב את temp*n.

לכאורה פשוט ביותר (ולמי שמורגל בחשיבה רקורסיבית לא רק לכאורה), למעשה מבלבל ביותר: כיצד יתכן שפונקציה תקרא לעצמה? התשובה היא שהדבר ייתכן וכפי שנראה, לעיתים יש בו גם טעם. הנקודה אותה יש להפנים היא שעת עותק א' של הפונקציה (כלומר קריאה א' לַפונקציה) קורא לעותק ב' של הפונקציה (כלומר לקריאה ב' לַפונקציה) הדבר אינו שונה מאשר עת פונקציה א' קוראת לפונקציה ב', כלומר:

א. ביצועו של עותק א' מושהה עד לסיום ביצועו של עותק ב'.

ב. רשומת ההפעלה של עותק ב' נבנית על-גבי המחסנית. ברשומת ההפעלה מוקצה מקום למשתנים ולפרמטרים. לעותק ב' יש משתנים ופרמטרים משלו, נפרדים מאלה של עותק א'; לכן אם גם בעותק א' וגם בעותק ב' מוגדר המשתנה `temp` אזי אין כל קשר בין ערך שמוכנס למשתנה ז בעותק אחד, לבין ערכו של המשתנה בַּעותק השני! רשומת ההפעלה של עותק ב' כוללת בין היתר את כתובת החזרה, שהיא כתובת של פקודה כלשהי בעותק א'.

ג. עותק ב' של הפונקציה מתבצע (במידה והוא כולל קריאות לפונקציות אחרות חל ביחס אליו הדין המוכר לנו היטב מהמצב בו פונקציה קוראת לפונקציה).

ד. עת עותק ב' מסתיים, רשומת ההפעלה שלו מוסרת מעל המחסנית, והמחשב חוזר לבצע את עותק א'. המחשב פונה בעותק א' לַפקודה שהינה כתובת החזרה של עותק ב'.

נדגים את מהלך ביצוע הפונקציה שכתבנו, תוך שאנו עוקבים אחר מצב המחסנית. נניח כי הקריאה הראשונה לפונקציה מהתכנית הראשית הייתה: `cout << factorial(4);`. נסמן פקודה זאת בציון (+). כדי להקל עלינו את המעקב נוסיף (ברישומים שלנו 'בכאילו', אך לא בקוד הכתוב 'באמת'!) לכל עותק של הפונקציה אינדקס. נדגיש כי תוספת האינדקס נעשית רק לצורך בהירות ההסבר; למעשה, כפי שראינו, קוד הפונקציה אינו כולל אינדקסים. את העותק הראשון של הפונקציה, זה הנקרא מהתכנית הראשית נציין כ- $\texttt{factorial}_4(4)$ כלומר נוסיף לו את האינדקס 4 שיסייע לנו להבדיל בין עותק זה לבין העותקים הבאים (שיווצרו בשל הקריאות הרקורסיביות). באופן כללי, לכל עותק של הפונקציה נוסיף אינדקס שהוא כערך הארגומנט המועבר לפונקציה בקריאה לה. נביט שוב על הקוד של הפונקציה לו הוספנו ('בכאילו') את האינדקסים המתאימים:

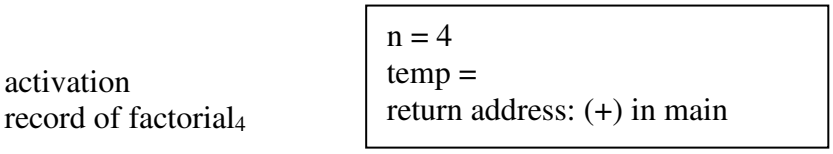
```
unsigned int factorial4(unsigned int n) {
   unsigned int temp ;

   if (n == 1)
     return(1) ;
   else {
      temp = factorial3(n-1) ;          // (a)
      cout << "about to return " << temp*n << endl;
      return( temp * n) ;
   }
}
```

נסביר: העותק אותו אנו בוחנים כרגע הוא $\texttt{factorial}_4$ אשר נקרא מהתכנית הראשית. עותק זה כולל קריאה *לפונקציה שונה לגמרי* בשם $\texttt{factorial}_3$ שאת הקוד שלה נציג מייד:

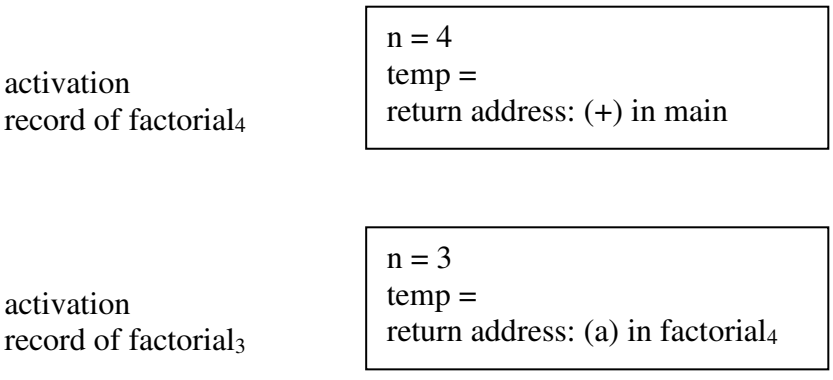
```
unsigned int factorial3(unsigned int n) {
   unsigned int temp ;

   if (n == 1)
```

```
      return(1) ;
   else {
      temp = factorial2(n-1) ;           // (b)
      cout << "about to return " << temp*n << endl;
      return( temp * n) ;
   }
}
```

נניח לעת עתה את $factorial_3$ בצד, ונחזור לעקוב אחר מהלך ביצועה של $factorial_4$. ראשית, נבנית רשומת ההפעלה של $factorial_4$:



אחרי בניית רשומת ההפעלה ניתן להתחיל בביצוע הפונקציה: ראשית נבדק התנאי (n==1) שאינו מתקיים (ערכו של n *של* $factorial_4$ הוא 4). לכן הביצוע פונה לגוש של ה- else, ובו יש להכניס לתוך temp את הערך שמחזירה קריאה לפונקציה כלשהי ששמה (במקרה או שלא במקרה) $factorial_3$. מה עושים עת יש להכניס למשתנה ערך המוחזר על-ידי פונקציה? קוראים לַפונקציה, משְמע ראשית בונים את רשומת ההפעלה שלה על-גבי המחסנית. מה ערכו של הארגומנט (המתאים לפרמטר n של $factorial_3$) המועבר לפונקציה? זהו n-1 של $factorial_4$, כלומר 3. מהי כתובת החזרה של $factorial_3$? במילים אחרות מהיכן קראו ל- $factorial_3$? כתובת החזרה היא ההשמה בַּפקודה המסומנת כ- (a) ב- $factorial_4$. נציג אם כך את מצב המחסנית אחרי הוספת רשומת ההפעלה של $factorial_3$:



אחרי בניית רשומת ההפעלה של $factorial_3$ על-גבי המחסנית ניתן להתחיל בביצוע הפונקציה. הפקודה הראשונה ב- $factorial_3$ (שהינה פונקציה שונה לגמרי מ- $factorial_4$, וביצועה אינו קשור לזה של $factorial_4$, פרט לכך שהיא נקראה על-ידי $factorial_4$) היא פקודת if. לכן נבדוק את התנאי (n==1). nשאת ערכו אנו בודקים הוא n של $factorial_3$ (שערכו 3), ולכן התנאי אינו מתקיים. אנו פונים, לפיכך, לגוש של ה- else, ובו עלינו להכניס למשתנה הלוקלי temp (של $factorial_3$) ערך שמחזירה קריאה לפונקציה כלשהי (שונה לגמרי מ- $factorial_3$) ששמה $factorial_2$. ראשית נציג את הקוד של אותה פונקציה שונה לגמרי:

```
unsigned int factorial2(unsigned int n) {
   unsigned int temp ;
```

```
    if (n == 1)
      return(1) ;
    else {
       temp = factorial1(n-1) ;          // (c)
       cout << "about to return " << temp*n << endl;
       return( temp * n) ;
    }
}
```

כיצד תראה רשומת ההפעלה של $factorial_2$ שתיבנה על-גבי המחסנית? הארגומנט המועבר לה הוא $n-1$ של $factorial_3$, כלומר הערך 2; ערך זה יועתק לפרמטר הערך n של $factorial_2$. מהי כתובת החזרה של $factorial_2$? במילים אחרות מי קרא ל- $factorial_2$? ל- $factorial_2$ קראה $factorial_3$, מפקודת ההשמה המסומנת כ- (b). נציג את מצב המחסנית בעקבות הוספת רשומת ההפעלה של $factorial_2$:

activation record of $factorial_4$

n = 4 temp = return address: (+) in main

activation record of $factorial_3$

n = 3 temp = return address: (a) in $factorial_4$

activation record of $factorial_2$

n = 2 temp = return address: (b) in $factorial_3$

אחרי בניית רשומת ההפעלה של $factorial_2$ על-גבי המחסנית ניתן להתחיל בביצוע הפונקציה. הפקודה הראשונה ב- $factorial_2$ (שהינה פונקציה שונה לגמרי מ- $factorial_3$ או $factorial_4$, וביצועה אינו קשור לזה של הפונקציות הללו, פרט לכך שהיא נקראה על-ידי $factorial_3$ שנקראה בעצמה על-ידי $factorial_4$) היא פקודת if. לכן נבדוק את התנאי (n==1). n שאת ערכו אנו בודקים הוא n של $factorial_2$ (שערכו 2), ולכן התנאי אינו מתקיים. אנו פונים, לפיכך, לגוש של ה- else, ובו עלינו להכניס למשתנה הלוקלי temp (של $factorial_2$) ערך שמחזירה קריאה לפונקציה כלשהי (שונה לגמרי מ-

(factorial$_2$) ששמה factorial$_1$. ראשית נציג את הקוד של אותה פונקציה שונה לגמרי:

```
unsigned int factorial1(unsigned int n) {
   unsigned int temp ;

   if (n == 1)
     return(1) ;
   else {
      temp = factorial0(n-1) ;          // (d)
      cout << "about to return " << temp*n << endl;
      return( temp * n) ;
   }
}
```

כיצד תראה רשומת ההפעלה של factorial$_1$ שתיבנה על-גבי המחסנית? הארגומנט המועבר לה הוא n-1 של factorial$_2$, כלומר הערך 1; ערך זה יועתק לפרמטר הערך n של factorial$_1$. מהי כתובת החזרה של factorial$_1$? במילים אחרות מי קרא ל- factorial$_1$? ל- factorial$_1$ קראה factorial$_2$, מפקודת ההשמה המסומנת כ- (c). נציג את מצב המחסנית בעקבות הוספת רשומת ההפעלה של factorial$_1$:

activation record of factorial$_4$

n = 4 temp = return address: (+) in main

activation record of factorial$_3$

n = 3 temp = return address: (a) in factorial$_4$

activation record of factorial$_2$

n = 2 temp = return address: (b) in factorial$_3$

activation record of factorial$_1$

n = 1 temp = return address: (c) in factorial$_2$

אחרי בניית רשומת ההפעלה של $factorial_1$ על-גבי המחסנית ניתן להתחיל בביצוע הפונקציה. הפקודה הראשונה ב- $factorial_1$ היא פקודת התנאי. נבחן את ערכו של n (כמובן, n של $factorial_1$). לשמחתנו ערכו של n הוא 1, כלומר התנאי מתקיים; לכן אנו יכולים לפנות בששון ובשמחה לביצוע הפקודה הכפופה לתנאי, כלומר לפקודה `return(1);`. בזאת מסתיים ביצוע של $factorial_1$. את הערך המוחזר על-ידי הפונקציה אנו מניחים בצד, ואת רשומת ההפעלה שלה אנו מסירים מהמחסנית. נציג את מצב העניינים:

activation record of $factorial_4$

n = 4 temp = return address: (+) in main

activation record of $factorial_3$

n = 3 temp = return address: (a) in $factorial_4$

activation record of $factorial_2$

n = 2 temp = return address: (b) in $factorial_3$

1

מה קורה עם הערך 1 שהוחזר על-ידי $factorial_1$? לאן הוא מוחזר? התשובה לשאלה זאת הייתה מצויה ברשומת ההפעלה של $factorial_1$: כתובת החזרה של $factorial_1$ היא פקודת ההשמה `(c)` ב- $factorial_2$, לכן הערך המוחזר על-ידי $factorial_1$ יוכנס למשתנה temp של $factorial_2$.

נעצור לרגע את דיוננו הפרטני לטובת הסתכלות מעט יותר גלובלית: התכנית הראשית קראה ל- $factorial_4$, זו קראה ל- $factorial_3$, אשר קראה ל- $factorial_2$, אשר קראה ל- $factorial_1$. בכל פעם שפונקציה כלשהי קראה לפונקציה אחרת ביצועה של הפונקציה הקוראת הוקפא עד אשר הפונקציה הנקראת תחזיר את הערך המבוקש, או אז תוכל הפונקציה הקוראת להמשיך בפעולתה. עתה $factorial_1$ הסתיימה, כפי שראינו הערך המוחזר על-ידה מוכנס למשתנה temp של $factorial_2$, ובכך $factorial_2$ מוצאת מההקפאה וממשיכה להתבצע. נבחן את מצב רשומת ההפעלה של $factorial_2$ עת היא חוזרת לחיים:

activation record of $factorial_2$

n = 2 temp = 1 return address: (b) in $factorial_3$

אחרי השלמת פעולת ההשמה שהכניסה ערך ל- temp, אנו ממשיכים לפקודת הפלט אשר מציגה את הפלט: `about to return 2` (שכן ערכו של הביטוי `temp*n` של

$factorial_2$ הוא 2). מכאן אנו מתקדמים לפקודת ה- return של $factorial_2$ אשר מחזירה את הערך 2. כפי שאנו זוכרים, עת פונקציה מסתיימת הערך המוחזר על-ידה 'מושאר בצד', ורשומת ההפעלה שלה מוסרת מעל המחסנית (או ליתר דיוק מתייחסים לקטע הזיכרון שהוקצה לרשומת ההפעלה שלה כאילו הוא פנוי לשימוש חוזר). רשומת ההפעלה גם כוללת את כתובת החזרה אשר מורה מה יעשה עם הערך המוחזר על-ידי הפונקציה: במקרה של $factorial_2$ כתובת החזרה היא ההשמה (b) ב- $factorial_3$, השמה אשר מכניסה ערך למשתנה temp של $factorial_3$. נציג את מצב המחסנית עם סיומה של $factorial_2$::

activation record of $factorial_4$

n = 4 temp = return address: (+) in main

activation record of $factorial_3$

n = 3 temp = return address: (a) in $factorial_4$

2

עתה הערך 2 אשר 'הושאר בצד' על-ידי $factorial_2$ מוכנס למשתנה temp של $factorial_3$, אשר מוצאת בזאת מקיפאונה. נציג את מצב רשומת ההפעלה של $factorial_3$ עת היא שבה לחיים:

activation record of $factorial_3$

n = 3 temp = 2 return address: (a) in $factorial_4$

שימו לב כי $factorial_2$ אומנם הגיעה לפקודת ה- return שלה (כלומר חלפה על-פני פקודת ההשמה שהכניסה ערך ל- temp, ועל פקודת הפלט) אולם הדבר אינו משפיע ולא כהוא זה על המקום בו נמצא ביצועה של $factorial_3$: $factorial_3$ הוקפאה בעת ביצוע פעולת ההשמה (b), ולכן עת היא מופשרת ביצועה ממשיך מאותה השמה. עם השלמת פעולת ההשמה (b) של $factorial_3$, אנו פונים לפקודת הפלט אשר מציגה: about to return 6. משם אנו מתקדמים לפקודת החזרת הערך של $factorial_3$. הפונקציה מחזירה את ערך 6, אשר 'מושם בצד', ואחר (כפי שמורה לנו כתובת החזרה של $factorial_3$) מוכנס בהשמה (a) למשתנה temp של $factorial_4$. בזאת יוצאת מקיפאונה $factorial_4$. כמו עם $factorial_3$ גם $factorial_4$ ממשיכה להתבצע *מהמקום בו היא הופסקה*, והעובדה ש- $factorial_2$, $factorial_3$ השלימו את ביצוען, בפרט את פקודת ה- return שלהן אינה משפיעה עליה. $factotial_4$ משלימה את ההשמה (a), ואחר ממשיכה לפקודת הפלט: about to return 24 (שכן לתוך המשתנה temp שלה הוכנס הערך 6 שהוחזר על-ידי $factorial_3$, וערכו של הפרמטר n שלה הוא 4). לבסוף $factorial_4$ מחזירה את הערך 24 לפקודת הפלט (+) של התכנית הראשית (אשר מציגה ערך זה למשתמש).

לסיכום הדוגמה ברצוני לציין כי המעקב אחר ביצוע התכנית תוך שימוש בַמחסנית הוא אומנם מוגיע ביותר, אולם להרגשתי עבור מתכנתים מתחילים אין לו חלופה.

על-כן אני ממליץ לכם לרכוש את המיומנות הדרושה, תוך ביצוע המעקב לאט לאט ובזהירות, ותוך רישום מדוקדק מאין באנו ולאן אנו הולכים בכל שלב ושלב.

8.1.2 חישוב סכומם של שני מספרים טבעיים

נמשיך עם דוגמה נוספת אשר תסייע לנו לרכוש את צורת החשיבה הרקורסיבית. נניח כי ברצוננו לכתוב פונקציה אשר מקבלת שני מספרים טבעיים num1 ו- num2. הפונקציה מחזירה את סכומם של שני המספרים, תוך שהיא עושה שימוש רק בפעולות +1 ו- -1 (אך לא בפעולות החבור או החיסור באופן כללי).

נציג את האלגוריתם תוך שימוש בסכמת הפתרון הרקורסיבי כפי שהוצגה בתחילת הפרק:

א. המקרה הפשוט, בו הפתרון הוא מיידי, הוא עת ערכו של אחד משני האופרנדים הוא אפס, שכן אז ערכו של הסכום הוא ערכו של האופרנד השני, ואין מה לחַשב. אנו נהיה מעט פחות מתוחכמים ונאמר כי אם ערכו של הפרמטר num2 הוא אפס אזי ערכו של הסכום הוא num1.

ב. במקרה הכללי (בו ערכו של num2 שונה מאפס) חשב את הסכום באופן הבא:

1. חשב את סכומם של num1 ו- (num2-1) והכנס אותו לתוך משתנה עזר temp.
2. החזר את temp +1.

אנו אומרים כי *הרקורסיה מתבצעת על num2* שכן ערכו של num2 הוא שמשתנה מקריאה לקריאה (בניגוד לערכו של num1 שאינו משתנה לאורך הקריאות הרקורסיביות השונות).

תרגומו של האלגוריתם הנ"ל לפונקציה בשפת C הוא כדלהלן:

```
unsigned int sum(unsigned int num1, unsigned int num2)
{
   unsigned int temp ;

   if (num2 == 0) terurn( num1 ) ;

   temp = sum(num1, num2-1) ;           // (-)
   return( temp +1) ;
}
```

הקריאה לפונקציה תיעשה למשל בהקשר הבא:

```
a = 5; b = 2 ;
c = sum(a, b) ;      // (+)
```

נעקוב אחר מהלך ביצוע הפונקציה. ראשית נבנית רשומת ההפעלה שלה על-גבי המחסנית. כמו קודם נשתמש באינדקס כדי להבחין בין העותקים השונים של הפונקציה. בדוגמה הנוכחית האינדקס יהיה כערכו של הפרמטר num2.

activation record of sum_2

num1 = 5
num2 = 2
temp =
return address = (+)

עתה הפונקציה מתחילה להתבצע. התנאי `(num2 == 0)` אינו מתקיים, ולכן למשתנה `temp` ברצוננו להכניס ערך אשר יוחזר על-ידי הפונקציה: `sum(num1, num2-1)`. כדי לקרוא לפונקציה זאת נבנה, ראשית, את רשומת ההפעלה שלה על-גבי המחסנית:

activation record of sum_2

num1 = 5
num2 = 2
temp =
return address = (+)

activation record of sum_1

num1 = 5
num2 = 1
temp =
return address = (-) in sum_2

נסביר: ערכו של `num2` בקריאה הנוכחית הוא `1`, שכן הארגומנט המועבר בקריאה ל- sum_1 הוא `num2-1` של sum_2. כתובת החזרה של sum_1 היא הפקודה `(-)` של sum_2 שכן במהלך ביצוע השמה זאת קראה sum_2 ל- sum_1.

sum_1 מתחילה להתבצע (כמובן מהפקודה הראשונה בה. וכמובן שהיא אינה מושפעת מכך ש- sum_2 כבר ביצעה פקודה זאת). sum_1 בודקת האם ערכו של num_2 שלה הוא אפס. התשובה היא לא, ולכן היא מתקדמת לפקודת ההשמה אשר גורמת לה לקרוא ל- `sum(num1, num2 -1)` כלומר ל- `sum(5, 0)`. כמו בכל קריאה לפונקציה נתחיל בבניית רשומת ההפעלה על-גבי המחסנית:

activation record of sum_2

num1 = 5
num2 = 2
temp =
return address = (+)

activation record of sum_1

num1 = 5
num2 = 1
temp =
return address = (-) in sum_2

activation record of sum_0

num1 = 5
num2 = 0
temp =
return address = (-) in sum_1

נסביר: ערכו של num2 בקריאה הנוכחית הוא 0, שכן הארגומנט המועבר בקריאה ל- sum_0 הוא num2-1 של sum_1. כתובת החזרה של sum_0 היא הפקודה (-) של sum_1 שכן במהלך ביצוע השמה זאת קראה sum_1 ל- sum_0.

sum_0 מתחילה להתבצע. היא שואלת האם ערכו של n (*שלה*) הוא אפס? התשובה היא כן, ולכן sum_0 מסיימת, תוך שהיא מחזירה את ערכו של num1, כלומר את הערך 5. הערך המוחזר על-ידי sum_0 'מושם בצד', ורשומת ההפעלה של sum_0 מוסרת מהמחסנית. כתובת החזה של sum_0 היא ההשמה (-) ב- sum_1, ולכן הערך 5 המוחזר על-ידי sum_0 מוכנס בהשמה (-) למשתנה temp של sum_1. בזאת מוצאת sum_1 מההקפאה. היא ממשיכה מפקודת ההשמה לפקודת ה- return אשר מחזירה את ערכו של temp+1, כלומר את הערך 6. לאן מוחזר ערך זה? כתובת החזרה של sum_1 מורה לנו כי הערך המוחזר מוכנס בהשמה (-) למשתנה temp של sum_2. עתה sum_2 מוצאת מהמקפיא. ההשמה (-) בה מושלמת (סו"ס), והיא יכולה להתקדם הלאה לעבר פקודת ה- return שלה בה היא מחזירה את ערכו של temp+1 כלומר את הערך 7 (שהוא אכן סכומם של 5 ו- 2). הערך 7 מוחזר למשתנה c כפי שמורה רשומת ההפעלה של sum_2.

אני מזמין אתכם לחשוב כיצד באופן דומה, תוכלו לכתוב את הפונקציה unsigned int mul(unsigned int num1, unsigned int num2). הפונקציה תקבל שני מספרים טבעיים, ותחזיר את מכפלתם, תוך שהיא עושה שימוש אך ורק בפעולות חיבור וחיסור (ללא שימוש בפעולת הכפל).

בַּפונקציות שכתבנו בשני הסעיפים האחרונים עשינו שימוש במשתנה עזר temp. אין בכך פסול, אולם הדבר אינו הכרחי, וניתן לכתוב כל אחת מהפונקציות גם ללא שימוש במשתנה עזר. נציג את הפתרון עבור sum, ואני משאיר לכם כתרגיל לשכתב גם את factorial:

```
unsigned int sum(unsigned int num1, unsigned int num2)
{
   if (num2 == 0) terurn( num1 ) ;

   return( sum(num1, num2-1) +1) ;
}
```

הסבר: הקריאה הנוכחית לפונקציה מחזירה ישירות את הערך המוחזר על-ידי הקריאה הרקורסיבית לפונקציה, ועוד אחד. במילים אחרות, ראשית נערכת קריאה רקורסיבית לפונקציה (עם num2-1), שנית לערך המוחזר על-ידי הקריאה הרקורסיבית מוסף 1, ושלישית הערך המתקבל מוחזר על-ידי הקריאה הנוכחית לַפונקציה.

8.1.3 <u>חישוב החזקה של שני מספרים טבעיים</u>

עתה ברצוננו לכתוב פונקציה אשר מחזירה את החזקה של שני מספרים טבעיים ללא שימוש בפקודת לולאה. הפונקציה: unsigned int power(unsigned int base, unsigned int exp) תחזיר את $base^{exp}$. בשלב זה של למודנו את נושא הרקורסיה יהיו ודאי תלמידים שיציעו את הפתרון הבא:

א. המקרה הפשוט הוא עת exp == 0, ובו החזר את הערך 1.

ב. במקרה המורכב:

1. הכנס לתוך משתנה עזר temp את תוצאת הקריאה הרקורסיבית ל- power(base, exp -1).
2. החזר את temp*base.

זה פתרון לגיטימי, אשר מחייב `exp` קריאות רקורסיביות. נציע עתה פתרון יעיל יותר, אשר יחייב פחות קריאות רקורסיביות. הפתרון יסתמך על חוקי החזקה הבאים: **(א)** $base^{2*exp} = base^{exp} * base^{exp}$ **(ב)** $base^{2*exp+1} = base^{exp} * base^{exp} * base$.

האלגוריתם הרקורסיבי לחישוב $base^{power}$ המסתמך על החוקים הללו הוא:

א. המקרה הפשוט: אם `exp == 0` החזר את הערך 1.

ב. המקרה מורכב:

1. הכנס למשתנה עזר `temp` את תוצאת הקריאה הרקורסיבית ל-`power(base, exp/2)`.
2. אם `exp` זוגי אזי החזר את `temp*temp`, ואחרת החזר את `temp*temp*base`.

נציג את הקוד המתאים:

```
unsigned int power(unsigned int base, unsigned int exp) {
   unsigned int temp ;

   if (exp == 0) return 1 ;

   temp = power(base, exp/2) ;

   if (exp % 2 == 0)
      return( temp *temp ) ;
   return(temp * temp * base) ;
}
```

<u>הערה</u>: את קטע הקוד:

```
   if (exp % 2 == 0)
      return( temp *temp ) ;
   return(temp * temp * base) ;
```

ניתן להמיר בקוד הבא:

```
return( (exp % 2 == 0) ? temp*temp : temp*temp*base ) ;
```

נדגים כיצד תתנהל ריצה של הפונקציה עבור הקריאה `cout << power(2, 5)`.

א. בקריאה הראשונה ערכו של `exp` הוא 5, לכן הקריאה הראשונה מייצרת קריאה רקורסיבית ל-`power(2, 5/2)` כלומר ל-`power(2, 2)`.

ב. בקריאה הרקורסיבית השניה ערכו של `exp` הוא 2, ולכן קריאה זו קוראת ל-`power(2, 1)`.

ג. בקריאה הרקורסיבית השלישית ערכו של `exp` הוא 1, לכן קריאה זאת קוראת ל-`power(2, 1/2)` כלומר ל-`power(2, 0)`.

ד. בקריאה הרקורסיבית הרביעית ערכו של `exp` הוא אפס, ולכן הקריאה הרביעית מסיימת, ומחזירה את הערך 1.

ה. הערך המוחזר (אחד) חוזר למשתנה `temp` של הקריאה השלישית. <u>ב</u>קריאה השלישית ערכו של `exp` הוא פרדי (אחד), ולכן הקריאה השלישית מחזירה את הערך `temp*temp*base = 1 * 1 * 2 = 2`. ערך זה מוחזר לקריאה הרקורסיבית השניה.

ו. בקריאה הרקורסיבית השניה ערכו של `exp` הוא שתיים. ולכן עת לקריאה זאת מוחזר הערך 2 היא עצמה מחזירה את `temp*temp = 2*2 = 4`. ערך זה חוזר לקריאה הרקורסיבית הראשונה.

ז. בקריאה הרקורסיבית הראשונה ערכו של `exp` הוא חמש. ולכן עת לקריאה זאת מוחזר הערך 4 היא עצמה מחזירה את `temp*temp*base = 4*4*2 = 32`. ערך זה מוצג בפקודת הפלט למשתמש.

עתה נשאל את עצמנו מדוע הגרסה של הפונקציה שמימשנו עדיפה על-פני הגרסה הנאיבית יותר שהוצעה בתחילת הסעיף (ושדומה לַפונקציות שכתבנו בַּסעיפים הקודמים)? נשאל את עצמנו כמה קריאות רקורסיביות היו נערכות לו התבקשנו לחשב את `power(2, 32)` בפונקציה שכתבנו, ובזו שדחינו ולא מימשנו?

א. בגרסה שלא כתבנו: היה מספר הקריאות הרקורסיביות כגודלו של `exp` (ליתר דיוק: כגודלו של `exp` ועוד אחד, שכן אנו עורכים קריאה גם עבור הערך אפס). כלומר זמן הריצה של הפונקציה שלא מימשנו הוא לינארי (במילים אחרות יחסי ישר) לגודלו של המעריך.

ב. בגרסה שמימשנו: הקריאה עם `exp` שערכו 32, הייתה מובילה לקריאה עם `exp` שערכו 16. הקריאה האחרונה הייתה מובילה לקריאה עם 8, שהייתה מובילה לקריאה עם 4, אחר עם 2, אחר עם 1, ולבסוף עם 0. כמה קריאות רקורסיביות היו נערכות לפיכך? כמספר הפעמים שניתן לחצות את ערכו התחילי של `exp` עד שמגיעים לאחד (או ליתר דיוק לאפס). במילים פורמליות יותר, כפי שכבר הסברנו בעבר (עת דנו בחיפוש בינארי) מספר הקריאות הרקורסיביות יהיה בסדר גודל של $\log_2(\texttt{exp})$.

כדי לסבר את האוזן, למשל עבור 2^{1024} היה מספר הקריאות הרקורסיביות בגרסה שלא מומשה 1025 =(1+ 1024), ובגרסה שמומשה 11 =(1+ $\log_2(1024)$).

8.1.4 <u>תרגום מספר טבעי מבסיס 10 לבסיס 2</u>

בסיס שתיים הוא בסיס בו קיימות שתי ספרות בלבד (אפס ואחת, במקום עשר ספרות שקיימות בבסיס עשר בו אנו משתמשים בדרך כלל). על כן המספרים הקיימים בבסיס שתיים הם: 0, 1, 10, 11, 100, 101, 110, 111, 1000,... . נסביר: נבחן לדוגמה את 11. מכיוון שכל הספרות שעומדות לרשותנו הן אפס ואחת הרי המספר הקטן ביותר שביכולתנו לייצר ושיהיה גדול מ- 11 הוא 100. אחרי 100 יבוא 101, ובבסיס שתיים העוקב ל- 101 הוא 110, שכן בעזרת אפס ואחד המספר הקטן ביותר שניתן לבנות ושיהיה גדול מ- 101 הוא 110. לכן ההתאמה בין בסיס שתיים לבסיס עשר היא כדלהלן: 1=1, 10=2, 11=3, 100=4, 101=5, 110=6, 111=7, 1000=8,... . אנו יכולים להצדיק התאמה זאת גם באופן אחר: בבסיס עשר משקלה של ספרת האחדות הוא 10^0, משקלה של ספרת העשרות הוא 10^1, משקלה של ספרת המאות הוא 10^2, וכן הלאה. בבסיס שתיים משקלה של ספרת האחדות הוא 2^0, משקלה של ספרת העשרות הוא 2^1, משקלה של ספרת המאות הוא 2^2, וכן הלאה. על-כן ערכו של 110 בבסיס שתיים הוא: $1*2^2+1*2^1+0*2^0 = 6$.

עתה ברצוננו לכתוב פונקציה אשר מקבלת מספר בבסיס עשר ומחזירה את המספר בבסיס שתיים. לדוגמה אם פונקציה תקבל את הערך 6 היא תחזיר את 110. הפונקציה תהיה: `unsigned int tobin(unsigned int n10)`.

כדי לכתוב את הפונקציה עלינו לתת את דעתנו לעיקרון הבא: יהיה n_{10} מספר כלשהו בבסיס עשר, ויהיה n_2 תרגומו לבסיס שתיים (לדוגמה: אם $n_{10} = 6$ אזי $n_2 =$ 110). אזי:

א. תרגומו לבסיס שתיים של $2*n_{10}$ הוא $10*n_2$ (לדוגמה: אם תרגומו של 6 לבסיס 2 הוא 110 אזי תרגומו של 12 לבסיס 2 הוא 1100).

ב. תרגומו לבסיס 2 של $2*n_{10}+1$ הוא $10*n_2+1$ (לדוגמה: תרגומו של 13 לבסיס 2 הוא 1101).

נציג עתה את אלגוריתם התרגום:

א. המקרה הפשוט הוא עת `n10 <= 1`, שכן אז תוצאת התרגום היא `n10` עצמו.

ב. במקרה המורכב:

1. הכנס למשתנה עזר `temp` את תוצאת התרגום של `n10/2` לבסיס 2.
2. אם `n10` זוגי אזי החזר את `temp*10`, ואחרת החזר את `temp*10+1`.

נציג עתה את הפונקציה:

```
unsigned int tobin(unsigned int n10) {

   unsigned int temp ;

   if (n10 <= 1) return( n10 ) ;

   temp = tobin(n10 / 2) ;

   if (n10 % 2 == 0)

      return( temp*10) ;

   return(temp*10 +1) ;

}
```

8.1.5 קריאת סדרת מספרים באורך לא ידוע, והדפסתם בסדר הפוך

עתה ברצוננו לכתוב פונקציה אשר קוראת סדרה באורך לא ידוע של מספרים שלמים, עד קריאת אפס, ומדפיסה את המספרים בסדר הפוך לסדר הזנתם. הפונקציה היא בעצם מעין 'תעלול' המשתמש ברקורסיה. נציגה ואחר נסבירה:

```
void reverse() {
   int num ;

   cin >> num ;
   if (num == 0) return ;

   reverse() ;
   cout << num ;
}
```

הקריאה הראשונה לפונקציה (למשל מהתכנית הראשית) תהיה: `reverse() ;`

נסביר את הפונקציה על-ידי שנעקוב אחר דוגמת הרצה. נניח כי הקלט המוזן לתכנית הוא: 17, 3879, 0 (17 מוזן ראשון, 0 מוזן אחרון). העותק הראשון של

הפונקציה מתחיל בפעולתו. הוא קורא את הערך 17 לתוך המשתנה num שלו. ערכו של num שונה מאפס ועל כן העותק הראשון קורא רקורסיבית לעותק השני. עת העותק השני מתחיל להתבצע הוא ראשית קורא את הערך 3879 לתוך המשתנה הלוקלי num (של העותק השני; וכמובן שאין קשר בין המשתנה הלוקלי num של העותק הראשון, לבין המשתנה הלוקלי num של העותק השני). ערכו של num שונה מאפס, ועל כן העותק השני קורא רקורסיבית לעותק השלישי. העותק השלישי עת מתחיל להתבצע קורא את הערך אפס לתוך המשתנה num שלו. ערכו של num בעותק השלישי הוא אפס, ועל כן העותק השלישי מבצע את פקודת ה- return. אנו חוזרים לעותק השני. כתובת החזרה של העותק השלישי היא הפקודה שמיד אחרי הקריאה הרקורסיבית, כלומר פקודת הפלט. העותק השני מציג את ערכו של num שלו, כלומר את הערך 3879. בזאת מסתיים ביצועו של העותק השני, ואנו שבים לעותק הראשון. כתובת החזרה היא, כפי שראינו, פקודת הפלט, ולכן העותק הראשון מציג את ערכו של num שלו, כלומר את הערך 17 (אשר נקרא לפני 3879, ומוצג אחריו). בזאת גם העותק הראשון מסיים.

8.1.6 <u>הצגת האיבר ה- n –י בסדרת פיבונאצ'י</u>

סדרת פיבונאצ'י מוגדרת באופן הבא: שני האיברים הראשונים בסדרה (המצויים במקומות מספר אפס ומספר אחד) הם אפס, ואחד. ערכו של כל איבר מספר n בסדרה, עבור n>=2 שווה לסכום שני האיברים הקודמים לו בסדרה. על-כן אברי הסדרה הם: 0, 1, 1, 2, 3, 5, 8, 13, ...

ברצוננו לכתוב פונקציה המקבלת מספר טבעי place ומחזירה את האיבר הנמצא במקום מספר place בסדרת פיבונאצ'י. ראשית נתאר את האלגוריתם תוך שימוש בסכמה הרקורסיבית:

א. המקרה הפשוט הוא עת place שווה אפס או place שווה אחד, ואז ערכו של האיבר המצוי במקום מספר place הוא place.

ב. במקרה המורכב (עת place >= 2): כדי לחשב את ערכו של האיבר מצוי במקום מספר place בסדרה יש לבצע שלושה צעדים:

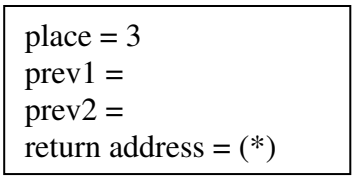
1. לפתור את הבעיה עבור נתונים מעט יותר פשוטים: לחשב את ערכו של האיבר המצוי המקום מספר place-1.
2. לפתור את הבעיה עבור נתונים מעט יותר פשוטים: לחשב את ערכו של האיבר המצוי במקום מספר place-2.
3. לבצע את ה- 'צעד הקטן הנוסף': להחזיר את סכומם של שני הנ"ל.

עתה נציג את הפונקציה

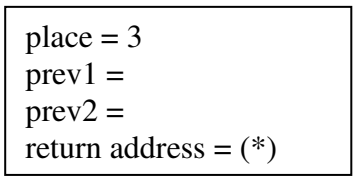
```
usigned int fib(unsigned int place) {
   unsigned int prev1, prev2 ;

   if (place <= 1) return(place) ;

   prev1 = fib(place -1) ;          // (-)
   prev2 = fib(place -2) ;          // (+)
   return( prev1 + prev2 ) ;
}
```

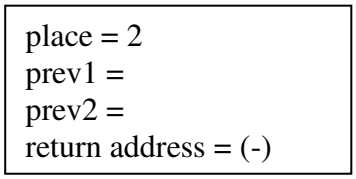
ייחודה של דוגמה זאת בכך שכל קריאה רקורסיבית (אם אינה עבור נתונים פשוטים) מולידה שתי קריאות רקורסיביות (ולא אחת, כפי שראינו עד כה).

נעקוב אחר השתלשלות הקריאות עבור המקרה בו הקריאה הראשונית היא: `a = fib(3)`; כלומר יש לחשב את האיבר מספר שלוש בסדרה (שהוא כזכור האיבר הרביעי, שכן האיבר הראשון מספרו אפס). נסמן פקודה זאת ב- (*). מצב המחסנית יהיה:



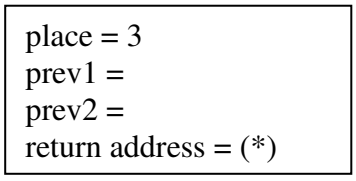
מכיוון שערכו של `place` גדול מאחד אנו מתקדמים להשמה (-) אשר מולידה את הקריאה:



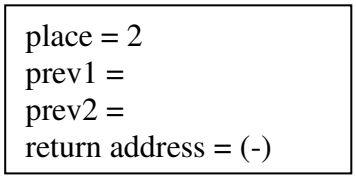


גם בקריאה זאת ערכו של `prev` גדול מאחד. שוב אנו מתקדמים להשמה (-) אשר מולידה את הקריאה השלישית:

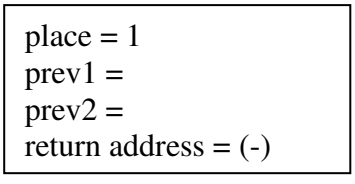
activation record of 1st call



activation record of 2nd call



activation record of 3rd call



בקריאה השלישית ערכו של `place` הוא אחד, ולכן הקריאה מסיימת מייד ומחזירה את הערך 1, לַקריאה השניה. רשומת ההפעלה של הקריאה השלישית

מוסרת מעל המחסנית. הערך 1 המוחזר על-ידי הקריאה השניה מוכנס למשתנה `prev1` של הקריאה השניה (שכן כתובת החזרה של הקריאה השלישית הינה ההשמה (-)). הקריאה השניה ממשיכה בפעולתה, כלומר היא מתקדמת להשמה (+) אשר מובילה אותה לקרוא בשנית לפונקציה `fib`, הפעם עם `place-2` כלומר עם אפס. נציג את מצב המחסנית בעקבות קריאה זו שתקרא הקריאה הרביעית:

activation record of 1st call

```
place = 3
prev1 =
prev2 =
return address = (*)
```

activation record of 2nd call

```
place = 2
prev1 = 1
prev2 =
return address = (-)
```

activation record of 4th call

```
place = 0
prev1 =
prev2 =
return address = (+)
```

אנו שמים לב כי למרות שזו הקריאה הרביעית, על-גבי המחסנית מצויות רק שלוש רשומות הפעלה, של שלושה עותקים של הפונקציה, וזאת משום שרשומת ההפעלה של הקריאה השלישית הוסרה עם סיומה של אותה קריאה, וטרם שבוצעה הקריאה הרביעית.

הקריאה הרביעית מסיימת מיידית, ומחזירה את הערך אפס. למי מוחזר ערך זה? להשמה (+) בקריאה השניה. נציג את מצב המחסנית אחרי סיום הקריאה הרביעית והכנסת הערך המוחזר על-ידה למשתנה `prev2` של הקריאה השניה:

activation record of 1st call

```
place = 3
prev1 =
prev2 =
return address = (*)
```

activation record of 2nd call

```
place = 2
prev1 = 1
prev2 = 0
return address = (-)
```

עתה הקריאה השניה יכולה להתקדם לפקודת ה- `return` שלה, ולהחזיר את סכומם של `prev1, prev2`, כלומר את הערך אחד. לאן מוחזר ערך זה? כתובת החזרה ברשומת ההפעלה של הקריאה השניה מורה לנו כי בערך החוזר יש להשתמש בהשמה (-) בקריאה הראשונה לפונקציה. עתה הקריאה הראשונה לפונקציה יכולה להתעורר (לרגע) מקיפאונה. עם השלמת ההשמה (-) מתקדמת

הקריאה הראשונה להשמה (+) אשר גורמת לקריאה רקורסיבית נוספת, קריאה מספר 5. ערכו של הארגומנט המועבר בקריאה זאת הוא `place -2` כלומר אחד. נציג את מצב המחסנית בעקבות ביצוע אותה קריאה:

activation record of 1st call

```
place = 3
prev1 = 1
prev2 =
return address = (*)
```

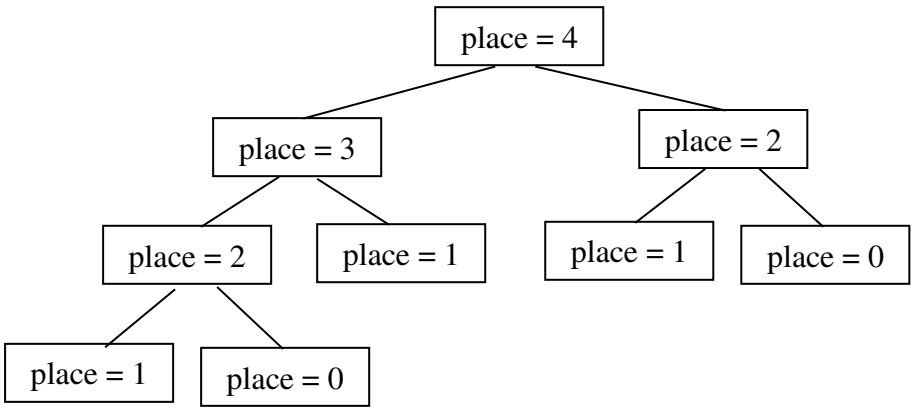
activation record of 5th call

```
place = 1
prev1 =
prev2 =
return address = (+)
```

קריאה מספר חמש היא הקריאה השניה שמולידה הקריאה הראשונה לפונקציה. מכיוון שערכו של הפרמטר `place` שלה הוא אחד, היא מסיימת באופן מיידי ומחזירה את הערך אחד. כתובת החזרה שלה מורה כי ערך זה חוזר להשמה (+) בפונקציה שקראה לה, כלומר בקריאה הראשונה לפונקציה.

עתה הקריאה הראשונה מופשרת שוב. למשתנה `prev2` שלה מוכנס הערך אחד. הפונקציה יכולה להתקדם לפקודת ה- `return` שלה אשר, בשעה טובה, מחזירה את `prev1+prev2` כלומר את שתיים, שהוא אכן ערכו של האיבר מספר שלוש בסדרת פיבונאצ'י.

המעקב אחר התנהלות הפונקציה באמצעות המחסנית הוא כלי חשוב. עתה נציג דרך אחרת להציג מעקב זה. הדרך האחרת תסתמך על 'עץ' שמציג את השתלשלות הקריאות. העץ מורכב מצמתים (במילים אחרות מקודקודים, שמצוירים כמלבנים בשרטוט). כל צומת בעץ מייצג קריאה רקורסיבית לפונקציה. הצומת נקשר בצלע (בקו) לצמתים (למלבנים) של הקריאות הרקורסיביות שהוא מזמן. צמתים אלה יקראו בניו של הצומת. נניח כי הקריאה הרקורסיבית הראשונה היא עבור `place = 4`.



נסביר: שרטוט כנ"ל נקרא עץ שכן הוא נראה כמו עץ הפוך ששורשו למעלה (הריבוע של 4 =`place`) והוא נפרש כלפי מטה. העץ בו אנו דנים נקרא עץ בינארי שכן לכל צומת בו יש שני בנים (שני צמתים אליהם הוא קשור בקו, המייצגים שתי קריאות

רקורסיביות שקראה הקריאה שבצומת האב). הקריאה עם place=4 מולידה שתי קריאות, אחת עם place=3, ואחת עם place=2, לכן הצומת (המלבן) של place=4 קשור בצלעות (בקווים) לצמתים של place= 3 ושל place= 2. באופן דומה נפרש יתר העץ. מכיוון שגם הקריאה עם place= 3 וגם הקריאה עם place= 4 מולידות קריאה עם place= 2 אזי הצומת של place= 2 (וכל מה שנולד ממנו, במילים פורמליות יותר: כל תת-העץ ש- place= 2 שורשו) מופיעים בעץ פעמיים. במקרה הכללי יתכן שתת-עץ כלשהו יחזור כמה וכמה פעמים בעץ. אני ממליץ לכם לצייר כתרגיל את העץ ששורשו place= 5.

8.2 מיון מהיר (Quick Sort)

בעבר ראינו את אלגוריתם המיון 'מיון בועות'. טענו כי מיון בועות הוא אלגוריתם מיון לא יעיל, אולם לא היה ביכולתנו להציג אלגוריתם מיון יעיל ממנו. עתה שהחכמנו, נוכל להציג אלגוריתמי מיון רקורסיביים, יעילים. בפרט נכיר את אלגוריתם המיון מיון מהיר.

אלגוריתם המיון מיון מהיר מקבל מערך של נתונים (שיקרא בשם list, ובדוגמה שלנו נניח כי הנתונים הם מספרים שלמים), וכן שני אינדקסים (שיקראו lo, hi) המתארים את גבולות קטע המערך אותו על האלגוריתם למיין בכל קריאה רקורסיבית לפונקציה. על האלגוריתם למיין את קטע המערך list[lo],..,list[hi] (כולל הקצוות). בתום תהליך המיון יכיל קטע המערך את הנתונים ממוינים (ונניח כי המיון הוא מקטן לגדול).

הרקורסיה מתבצעת על האינדקסים המועברים לאלגוריתם: בכל קריאה רקורסיבית על האלגוריתם למיין קטע מערך שונה.

תיאור האלגוריתם:

א. המקרה הפשוט: אם קטע המערך אותו יש למיין הוא בן תא יחיד, (כלומר אם `lo ==hi`) אזי קטע זה כבר ממוין, ואין צורך לבצע דבר.

ב. המקרה המורכב: בו יש למיין קטע מערך הכולל יותר מתא יחיד (כלומר `lo < hi`). בצע:

1. בחר איבר כלשהו בקטע המערך אותו עליך למיין. איבר זה יקרא הפיבוט. (אנו נבחר את הפיבוט להיות האיבר הנמצא בתא הראשון בקטע המערך שיש למיין, כלומר אנו נבחר את `list[lo]` כפיבוט).
2. העבר (באמצעות אלגוריתם לא רקורסיבי, אותו פגשנו בתרגיל 5.7.4) את כל הערכים בקטע המערך הקטנים או שווים מ- `list[lo]` לתאים המצויים משמאל ל- `list[lo]` (בקטע המערך), ואת כל הערכים הגדולים מ- `list[lo]` לתאים המצויים מימין ל- `list[lo]`, תוך שאתה מזיז את `list[lo]` למקום הדרוש. נסמן ב- `pivot_place` את המקום אליו הוזז `list[lo]` בתהליך הנ"ל.
3. אם קטע המערך שמשמאל לפיבוט כולל יותר מאיבר יחיד, אזי קרא רקורסיבית לאלגוריתם עבור קטע המערך שמשמאל לפיבוט, כלומר עבור קטע המערך `lo..pivot_place-1`.
4. אם קטע המערך שמימין לפיבוט כולל יותר מאיבר יחיד, אזי קרא רקורסיבית לאלגוריתם עבור קטע המערך שמימין לפיבוט, כלומר עבור קטע המערך `pivot_place+1..hi`.

נסביר את האלגוריתם באמצעות דוגמה: נניח כי בתכנית הראשית קיים מערך בשם `int list[LIST_SIZE] ;` (בן שמונה תאים) המכיל נתונים כדלהלן (השורה העליונה מציינת את הנתונים המצויים במערך, השורה התחתונה מציינת את מספרי התאים):

6	5	9	4	10	11	6	0
0#	1#	2#	3#	4#	5#	6#	7#

הקריאה לאלגוריתם מהתכנית הראשית תהיה: `qs(list, 0, LIST_SIZE-1);` שכן יש למיין את המערך מתא מספר אפס, ועד תא מספר `LIST_SIZE-1`.

א. בקריאה זאת ערכי הפרמטרים הם: `lo = 0, hi = 7`. על-כן אנו במקרה המורכב. נבצע את שלושת שלביו:

1. נבחר את `list[lo]` (כלומר את 6, האיבר השמאלי ביותר בקטע) להיות פיבוט. ונעביר את הערכים במערך כמתואר באלגוריתם. בתום תהליך ההעברה יראה המערך באופן הבא:

5	4	6	0	**6**	9	10	11
0#	1#	2#	3#	4#	5#	6#	7#

(כל סידור אחר של המערך בו כל הערכים הקטנים או שווים מ- 6 נמצאים משמאלו, וכל הגדולים מ- 6 נמצאים מימינו ייחשב כסידור תקין). ערכו של `pivot_place` הוא ארבע, שכן הפיבוט (הנתון 6 שהיה בתא מספר אפס במערך) עבר לתא מספר ארבע במערך.

2. מכיוון שקטע המערך שמשמאל לפיבוט כולל יותר מתא יחיד, נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמשמאל לפיבוט, כלומר עבור קטע המערך `lo..pivot_place-1 = 0..3`.
3. מכיוון שקטע המערך שמימין לפיבוט כולל יותר מתא יחיד, נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמימין לפיבוט, כלומר עבור קטע המערך `pivot_place+1..hi = 5..7`

לפני שנמשיך במעקב אחר ריצת האלגוריתם עלינו להבין נקודה מכרעת: בהנחה שהקריאה הרקורסיבית עם קטע המערך שמשמאל לפיבוט תַעשה את עבודתה כהלכה, והקריאה הרקורסיבית עם קטע המערך שמימין לפיבוט תַעשה את עבודתה כהלכה, הקריאה הרקורסיבית הנוכחית השלימה את פעולתה. נסביר מדוע: משמשאל לפיבוט נמצאים כל הערכים הקטנים או שווים מהפיבוט, ורק אברים הקטנים שווים מהפיבוט, באופן סימטרי ממימין לפיבוט נמצאים כל האיברים הגדולים מהפיבוט, ורק אברים הגדולים מהפיבוט, על-כן די שכל קריאה רקורסיבית תמיין את קטע המערך שהיא התבקשה למיין ובכך תושלם המשימה. במילים אחרות: אחרי שהושלם תהליך ההזזה המתואר בסעיף מספר אחד, המערך מקיים כבר כמה תכונות רצויות: (א) לשם השלמת מיונו כבר לא נצרך יותר להזיז את הפיבוט, שכן הוא גדול מכל אלה שמשמאלו, וקטן מכל אלה שמימינו. (ב) אין שום אפשרות שמי שמצוי עתה משמאל לפיבוט יאלץ בהמשך (לשם השלמת המיון) לעבור אל מימין לו (או להפך).

ב. הקריאה הרקורסיבית השניה (זו שנקראה מסעיף א2) מקבלת את הפרמטרים הבאים: `lo = 0, hi = 3`. על-כן אנו במקרה המורכב. נבצע את שלושת שלביו:

1. נבחר את `list[lo]` (כלומר את 5, האיבר השמאלי ביותר בקטע) להיות פיבוט. ונעביר את הערכים במערך כמתואר באלגוריתם. בתום תהליך ההעברה יראה קטע המערך הרלוונטי לקריאה זאת באופן הבא (ביתר תאי המערך הקריאה הנוכחית אינה נוגעת, ולכן לא הצגנו את תוכנם, אשר נשאר כפי שהוא היה קודם לכן):

4	0	5	6				
0#	1#	2#	3#	4#	5#	6#	7#

(כל סידור אחר של המערך בו כל הערכים הקטנים או שווים מ- 5 נמצאים משמאלו, וכל הגדולים מ- 5 נמצאים מימינו ייחשב כסידור תקין). ערכו של

pivot_place הוא שתיים, שכן הפיבוט (הנתון 5 שהיה בתא מספר אפס במערך) עבר לתא מספר שתיים במערך.

2 מכיוון שקטע המערך שמשמאל לפיבוט כולל יותר מתא יחיד, נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמשמאל לפיבוט, כלומר עבור קטע המערך lo..pivot_place-1 = 0..1.

3 מכיוון שקטע המערך שמימין לפיבוט כולל תא יחיד (התא מספר שלוש) לא נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמימין לפיבוט.

ג. הקריאה הרקורסיבית השניה הנפיקה בסעיף ב2 קריאה רקורסיבית עם קטע המערך 0..1. נעקוב אחר קריאה זאת. אנו במקרה המורכב (lo=0, hi=1). נבצע את שלושת שלביו:

1 נבחר את list[lo] (כלומר את 4, האיבר השמאלי ביותר בקטע) להיות פיבוט, ונעביר את הערכים במערך כמתואר באלגוריתם. בתום תהליך ההעברה יראה קטע המערך הרלוונטי לקריאה זאת באופן הבא:

0	4						
0#	1#	2#	3#	4#	5#	6#	7#

ערכו של pivot_place הוא אחד, שכן הפיבוט (הנתון 4 שהיה בתא מספר אפס במערך) עבר לתא מספר אחד במערך.

2 מכיוון שקטע המערך שמשמאל לפיבוט כולל תא יחיד, לא נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמשמאל לפיבוט.

3 מכיוון שקטע המערך שמימין לפיבוט ריק (הפיבוט הוזז לקצהו הימני של קטע המערך אותו היה על הקריאה הרקורסיבית הנוכחית למיין) לא נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמימין לפיבוט.

בזאת הקריאה הרקורסיבית הנוכחית מסתיימת (בלי לייצר קריאות רקורסיביות נוספות). לאן פונה ביצוע התכנית מכאן? מי קרא לעותק הנוכחי של הפונקציה ומאיפה? הקריאה בה דנו בסעיף זה (סעיף ג') זומנה מסעיף ב2. על כן עלינו לחזור לסעיף ב3. אם נבדוק את סעיף ב3 נגלה שהוא אינו מזמן קריאה רקורסיבית. סעיף ב3 הוא תת-הסעיף האחרון של סעיף ב', על-כן בזאת מסתיימת גם הקריאה הרקורסיבית בה דנו בסעיף ב'. ושוב נשאל את עצמנו מניין באנו ואנא אנו פונים? התשובה היא שהקריאה הרקורסיבית בה דנו בסעיף ב' נקראה מסעיף א2, לכן עתה עלינו לפנות לסעיף א3. לפני שנעשה כן נשים לב כי על הקריאה הרקורסיבית מסעיף ב' היה למיין את קטע המערך 0..3, ואכן בתום פעולתה קטע מערך זה ממוין.

ד. עתה אנו יכולים לחזור לסעיף א3 אשר מנפיק את הקריאה הרקורסיבית עם קטע המערך pivot_place+1..hi = 5..7. נדון בקריאה רקורסיבית זאת: אנו במקרה המורכב (lo=5, hi=7). נבצע את שלושת שלביו:

1 נבחר את list[lo] (כלומר את 9, האיבר השמאלי ביותר בקטע) להיות פיבוט. נעביר את הערכים במערך כמתואר באלגוריתם. בתום תהליך ההעברה יראה קטע המערך הרלוונטי לקריאה זאת באופן הבא:

					9	10	11
0#	1#	2#	3#	4#	5#	6#	7#

ערכו של pivot_place הוא חמש, שכן הפיבוט (הנתון 9 שהיה בתא מספר חמש במערך) נשאר בתא מספר חמש במערך.

2 מכיוון שקטע המערך שמשמאל לפיבוט ריק (הפיבוט נותר בקצהו של קטע המערך), לא נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמשמאל לפיבוט.

3 מכיוון שקטע המערך שמימין לפיבוט כולל שני תאים, נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמימין לפיבוט, כלומר עבור קטע המערך `6..7`.

ה. נדון בקריאה הרקורסיבית מסעיף ד3. אנו במקרה המורכב (`lo=6, hi=7`). נבצע את שלושת שלביו:

1 נבחר את `list[lo]` (כלומר את `10`, האיבר השמאלי ביותר בקטע) להיות פיבוט. נעביר את הערכים במערך כמתואר באלגוריתם. בתום תהליך ההעברה יראה קטע המערך הרלוונטי לקריאה זאת באופן הבא:

						10	11
0#	1#	2#	3#	4#	5#	6#	7#

ערכו של `pivot_place` הוא שש, שכן הפיבוט (הנתון 10 שהיה בתא מספר שש במערך) נשאר בתא מספר שש במערך.

2 מכיוון שקטע המערך שמשמאל לפיבוט ריק, לא נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמשמאל לפיבוט.

3 מכיוון שקטע המערך שמימין לפיבוט כולל תא יחיד, לא נקרא רקורסיבית לאלגוריתם עבור קטע המערך שמימין לפיבוט.

בזאת מסתיימת הקריאה ה'. קריאה זאת זומנה על-ידי סעיף ד3 שהינו תת-הסעיף המסיים את סעיף ד', לכן עת סעיף ה' מסתיים אנו חוזרים לסיומו של סעיף ד'. עת סעיף ד' מסתיים עלינו לחזור לסיומו של סעיף א', שכן סעיף ד' זומן על-ידי תת-סעיף א3. ואם גם סעיף א' מסתיים משמע הרקורסיה שלנו תמה. אם נבדוק את מצב המערך נגלה שהוא אכן ממויין.

נציג עתה את מימוש האלגוריתם בשפת C:

```
void qs(int list[],unsigned int lo, unsigned int hi)
{
   unsigned int pivot_place ;

   pivot_place = place_pivot(list, lo, hi) ;

   if (pivot_place -lo > 1)
      qs(list, lo, pivot_place -1) ;

   if (hi - pivot_place > 1)
      qs(list, pivot_place+1, hi) ;
}
```

תלמידים המסתכלים בקוד זה בראשונה חשים מרומים בלי לדעת למה: זה לא יתכן שזה כל-כך פשוט! האמת היא שראשית זה כוחה של הרקורסיה---היא הופכת משימות מורכבות ללכל הפחות קצרות (גם אם לא פשוטות להבנה), שנית עלינו עדיין להשלים את החלק הלא רקורסיבי של הפונקציה, כלומר את פונקצית העזר `place_pivot`. פונקציה זאת מקבלת את המערך, ואת מציינ קטע המערך עליו על העותק הנוכחי של `qs` לפעול (כלומר את `lo, hi`). הפונקציה מזיזה את הערכים במערך כפי שכבר תיארנו, היא מחזירה (באמצעות פקודת `return`) את מספר התא בו הוצב הפיבוט אחרי הזזתו. כתיבת פונקציה זאת היא תרגיל לא רע

בנושא מערכים חד-ממדיים, ואני ממליץ לכל אחד מכם לכתוב אותה בכוחות עצמו. למעשה גם אין זה מתאים להציגה כאן (שכן לא זה נושא דיוננו בַּפרק הנוכחי). אני מצרפה בכל-אופן שכן מניסיוני אני יודע שתלמידים חשים לעיתים שאולי הסיבה ש- qs היא כל-כך קצרה היא שמסדרים אותם באמצעות פונקציה זאת, ובעצם היא נורא מורכבת. לכן אציג את קוד הפונקציה בלי להסבירו בפרוטרוט: אנו סורקים את קטע המערך משמאל לימין. עת אנו נתקלים בערך קטן מהפיבוט אנו מבצעים את הצעדים הבאים:

א. אנו מחליפים בין ערך זה לבין הערך המצוי מימין לפיבוט.

ב. אנו מחליפים בין הערך המצוי מימין לפיבוט לבין הפיבוט.

ג. אנו מעדכנים שהפיבוט זז ימינה תא אחד.

בדקו שאכן זה כל מה שנדרש.

הפונקציה place_pivot תשתמש בפונקצית העזר: void swap(int &var1, int &ar2). הפונקציה swap מחליפה בין הערכים המצויים בפרמטרים שלה. (הצגנו בעבר את הגדרתה).

ועתה לפונקציה place_pivot:

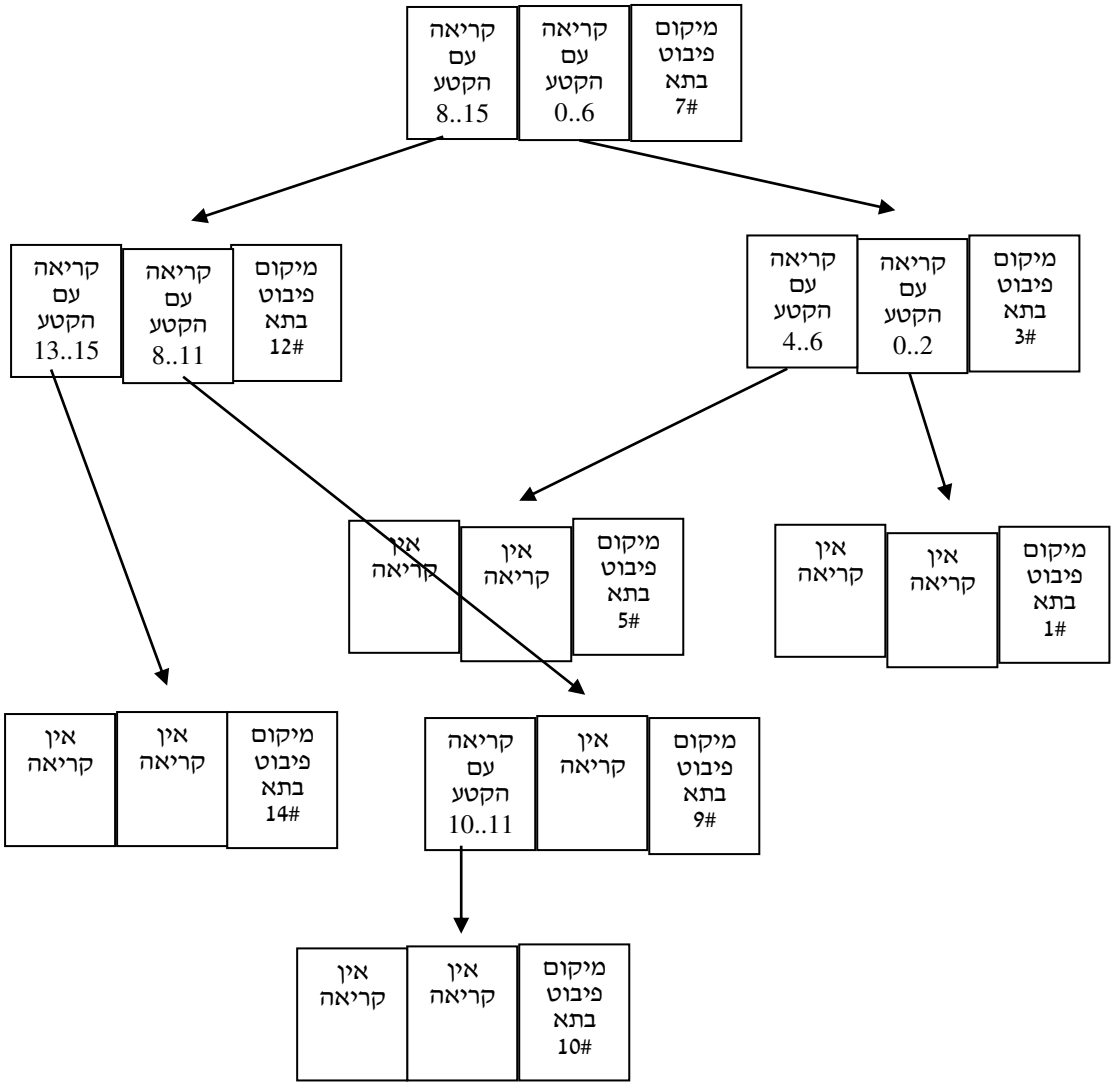
```
unsigned int place_pivot(int list[],
                unsigned int lo, unsigned int hi)
{
   unsigned int pivot_place = lo ;
   for (unsigned int i = lo+1; i<= hi; i++)
      if(list[i] < list[pivot_place])
      {
         swap(list[i], list[pivot_place +1]) ;
         swap(list[pivot_place], list[pivot_place +1]) ;
         pivot_place++ ;
      }

   return( pivot_place ) ;
}
```

שימו לב כי אנו מסתמכים כאן על כך ששינויים המוכנסים לְמערך המועבר כארגומנט לְפונקציה נותרים במערך גם אחרי תום ביצוע הפונקציה.

8.2.1 <u>זמן הריצה של מיון מהיר</u>

התחלנו את הדיון במיון מהיר בטענה שמיון זה יעיל יותר ממיון בועות. עתה ברצוננו להצדיק טענה זאת.

נדון ראשית במקרה האופטימלי: בו בכל שלב הפיבוט ממוקם בדיוק באמצע קטע המערך אותו על הפונקציה למיין (כלומר בתא list[(lo+hi)/2]). נצייר את עץ הקריאות עבור מערך בן 16 תאים (כלומר מערך שמספרי תאיו 0..15). כל צומת בעץ מורכב משלושה מרכיבים: (א) היכן ממוקם הפיבוט, (ב) קטע המערך עבורו נערכת הקריאה הרקורסיבית האחת, (ג) קטע המערך עבורו נערכת הקריאה הרקורסיבית השניה:



נביט בעץ הקריאות: נתייחס לשורש כאל רמה אפס בעץ. בניו של השורש יקראו רמה מספר אחד, נכדיו רמה מספר 2, וכן הלאה. בַּשורש יש לטפל בקטע מערך בגודל 16, בכל-אחד משני בניו של השורש (כלומר, בצמתים מרמה מספר אחת) יש לטפל בקטע מערך בגודל =< 8. בכל אחד מארבעת נכדיו של השורש (כלומר בצמתים מרמה שתיים) יש לטפל בקטע מערך בגודל <= 4. באופן כללי אנו רואים כי בצאצאים מרמה d יש לטפל בקטע מערך בגודל $16/2^d$. עבור מערך בגודל n יהיה גודל קטע המערך בו יש לטפל בַּצאצאים המצויים ברמה d קטן או שווה מ- $n/2^d$. נזכור כי גודלו של קטע המערך בו יש לטפל יהיה תמיד >= 1. על-כן מה מספר הרמות שהעץ יכיל? מספר הרמות הוא d המקיים כי $n/2^d = 1$. עתה נוציא $\log_2$ לשני האגפים, נשתמש בכללי הלוג, ונגלה שמספר הרמות בעץ הוא: $\log_2(n)$. כמה עבודה מתבצעת בכל רמה ורמה בעץ? בשורש מתבצעת עבודה בשיעור n על-ידי הפונקציה place_pivot אשר לצורך מיקום הפיבוט סורקת פעם יחידה את המערך בשלמותו. (זימון כל אחת משתי הקריאות הרקורסיביות דורשת עבודה בשיעור קבוע, ועל כן

אינה נספרת בכמות העבודה הנעשית). בכל אחד משני בניו של השורש נעשית עבודה בשיעור n/2 על-ידי הפונקציה place_pivot, (ושוב איננו סופרים את כמות העבודה הנדרשת לשם ביצוע הקריאות הרקורסיביות, שכן זו עבודה בשיעור קבוע, שאינה מושפעת מגודלו של n). בכל אחד מארבעת נכדיו של השורש מתבצעת עבודה בשיעור n/4 על-ידי place_pivot. וכן הלאה. אנו רואים אם כך שכמות העבודה הנעשית בכל רמה היא בשיעור n (על-ידי סך כל ההרצות של place_pivot המבוצעות באותה רמה). אם נסכם אזי סך כל העבודה המתבצעת על-ידי מיון מהיר במקרה האופטימלי, בו הפיבוט ממוקם בכל קריאה רקורסיבית באמצע קטע המערך, היא בשיעור $n*\log_2(n)$, שכן בעץ יש לכל היותר $\lg_2(n)$ רמות, ובכל רמה נעשית עבודה בשיעור n. אני מזכיר לכם כי כמות העבודה הנעשית במיון בועות כמות היא בסדר גודל של n^2. לדוגמה: עת $n=10^3$ יבצע מיון בועות עבודה בסדר גודל של n^6, בעוד מיון מהיר יבצע עבודה בשיעור n^4, וככל ש- n ילך ויגדל כן יגדל גם הפער בין כמות העבודה שתידרש לשם ביצוע כל אחד משני המיונים.

עד כאן דנו בזמן הריצה של מיון מהיר במקרה האופטימלי. מה יקרה במקרה הגרוע ביותר? במקרה הגרוע ביותר יוצב הפיבוט כל פעם בקצה קטע המערך שעל העותק הנוכחי של הפונקציה למיין (תוך שכל יתר האיברים מועברים לשמאלו, או כל יתר האיברים מועברים לימינו). לדוגמה, אם המערך אותו על הפונקציה למיין כבר ממוין מקטן לגדול, אזי הפיבוט יוותר כל פעם בקצהו השמאלי של קטע המערך. ומה יהיה אז זמן הריצה של הפונקציה? למעשה אז מיון מהיר מתנהג כמו מיון בועות. נסביר: place_pivot תבצע, במקרה זה, עבודה זהה לעבודה שמבצעת הלולאה הפנימית במיון בועות, ואחר תזומן קריאה רקורסיבית עם קטע מערך שגודלו קטן באחד מגודלו של קטע המערך בו טיפל העותק הנוכחי של הפונקציה. לפיכך במערך בן n נתונים תהינה n קריאות רקורסיביות (האנלוגיות ל- n איטרציות בַלולאה החיצונית במיון בועות). בקריאה הראשונה יהיה על place_pivot לטפל במערך בגודל n, בקריאה השניה יהיה עליה לטפל במערך בגודל n-1, וכן הלאה. סכום העבודה המתבצעת על-ידי סך הקריאות ל- place_pivot הוא לפיכך: n+…+1, וכבר ראינו שטור חשבוני זה סכומו הוא בסדר גודל של n^2. זו כל כמות העבודה הנעשית, שכן n הקריאות הרקורסיביות החלות מעבר לעבודה הנעשית על-ידי place_pivot הן עבודה זניחה יחסית ל- n^2. אנו זוכרים שעל-ידי שיפורו של אלגוריתם מיון בועות הבסיסי, נוכל במקרה בו המערך כבר ממוין לעצור את תהליך המיון. לכן במקרה כזה יהיה מיון בועות המשופר עדיף על-פני מיון מהיר.

לבסוף יש מקום לשאול: מה קורה במקרה הממוצע (על קלט מיקרי)? תשובה מלאה ומנומקת לשאלה זאת הינה מעבר למסגרת דיוננו הנוכחי. רק נציין כי המקרה הממוצע דומה למקרה המיטבי, כלומר זמן הריצה של מיון מהיר במקרה הממוצע הוא $n*\lg_2(n)$.

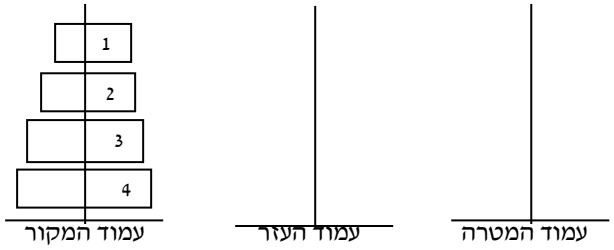
8.3 מגדלי האנוי (Towers of Hanoi)

מגדלי האנוי היא בעיה נוספת אשר מדגימה לנו כי במקרים מסוימים פתרון רקורסיבי עשוי להיות קצר, אלגנטי ופשוט (כלומר פשוט למי שמבין אותו...). בעיה זאת גם תפתח לנו צוהר לבעיות אשר פתרונן מחייב עבודה כה רבה, עד אשר עבור קלט מספיק גדול הן למעשה בלתי פתירות (במילים אחרות כדי לפתור אותן יידרש זמן רב ממידות אנוש).

8.3.1 תיאור הבעיה

הבעיה מגדלי האנוי מוצגת באופן הבא: בחצר המקדש בהנוי הציב נזיר זקן ובר-לבב עמוד ועליו מגדל בן 128 טבעות המסודרות מגדולה (המצויה בבסיס המגדל) לקטנה (המצויה בראש מגדל). לצד העמוד עליו סודר מגדל הטבעות (שיקרא להלן עמוד המקור source), העמיד הנזיר שני עמודים נוספים (שיקראו עמוד המטרה, destination, ועמוד העזר, auxilary). הנזיר הזקן הטיל על צאן מרעיתו את המשימה הבאה: יש להעביר את מגדל הטבעות מעמוד המקור אל עמוד המטרה, באופן שבכל פעם הם רשאים להעביר רק טבעת בודדת, ואסור להם להניח טבעת גדולה על טבעת קטנה ממנה. לצורך משימת ההעברה רשאים הנזירים להשתמש בעמוד העזר (אך כמובן שאין להניח טבעות קדושות על הקרקע הטמאה, או להחזיק ביד טבעות כלשהן כדי שניתן יהיה להעביר טבעות שהיו מצויות מתחתן). הנזיר הקדוש הבטיח לצאן מרעיתו שבתום תהליך ההעברה הם יזכו לנרוונה, והעולם כולו ייהפך לגן-עדן עלי אדמות.

מבחינה גרפית ניתן לתאר את המצב התחילי באופן הבא (עבור מגדל בן ארבע טבעות):



כדי להבהיר את הבעיה נדגיש כי לא ניתן לקחת יחד את כל ארבע הטבעות ולהעבירן ממגדל המקור למגדל המטרה (יען כי זה בניגוד לדרישה שבכל פעם תועבר רק טבעת בודדת). כמו כן לא ניתן להעביר את טבעת #1 למגדל העזר, אחר להעביר את טבעת #2 למגדל העזר (שכן זה בניגוד לדרישה שאסור להניח טבעת גדולה יותר על טבעת קטנה יותר).

8.3.2 הסבר אינטואיטיבי של הפתרון

כיצד יתמודדו הנזירים עם האתגר שהציב בפניהם הנזיר הזקן (העברת מגדל בן n טבעות מעמוד המקור לעמוד המטרה)? כמו בצה"ל בשלושה שלבים:

א. הם יפתרו בעיה מעט יותר קלה: הם יעבירו מגדל בן n-1 טבעות מעמוד המקור לעמוד העזר.

ב. הם יעבירו את טבעת בגודל n, המצויה בבסיסו של עמוד המקור לעמוד המטרה.

ג. הם יפתרו בעיה מעט יותר קלה: הם יעבירו מגדל בן n-1 טבעות המצוי על עמוד העזר לעמוד המטרה.

ומהו המקרה הפשוט בבעיה זאת? עת גודלו של המגדל אותו על הנזירים להעביר הוא טבעת בודדת. במקרה זה ניתן להעביר את הטבעת הבודדת ישירות מעמוד המקור לעמוד המטרה.

בדוגמה שלנו (בה $n=4$). יהיה על הנזירים:

א. להעביר מגדל בן שלוש טבעות מעמוד המקור לעמוד העזר.

ב. להעביר טבעת בגודל ארבע מעמוד המקור לעמוד המטרה.

ג. להעביר את המגדל בן שלוש הטבעות, שהם הציבו בסעיף א' על עמוד העזר, אל עמוד המטרה.

לתלמידים (וחוששני שגם למורים) רבים אלגוריתם זה נראה בין קסם לתרמית. הנקודה המכרעת אותה יש להבין כדי להשתכנע שאין מדובר כאן לא בקסם ולא בתרמית היא הבאה: עת הנזירים מעבירים מגדל בן שלוש טבעות מעמוד המקור אל עמוד העזר *הם אינם נוגעים בטבעת בגודל 4 המצויה בבסיסו של עמוד המקור*. מכיוון שטבעת זאת גדולה מכל שלוש הטבעות אותן הם מעבירים (מפה לשם ומשם לפה) אזי *נוכחותה של הטבעת בגודל 4 בבסיסו של עמוד המקור אינה משנה; במילים אחרות היא אינה מעלה ואינה מורידה את יכולתם של הנזירים להעביר מגדל בן שלוש טבעות ממגדל המקור למגדל העזר!* לכן כדי לשכנע אתכם שהנזירים מסוגלים להשלים את משימתם (כלומר להעביר מגדל בן 4 טבעות מעמוד המקור לעמוד המטרה), די שאשכנע אתכם שהנזירים מסוגלים להעביר מגדל בן שלוש טבעות מעמוד המקור לעמוד העזר. שכן אם הם ידעו לבצע תת-משימה זאת אזי:

א. הם יוכלו להעביר מגדל בן שלוש טבעות מעמוד המקור לעמוד העזר (וכאמור השאלה: האם הם עושים זאת שעה שהטבעת בגודל 4 מצויה בבסיסו של עמוד המקור, או אינה מצויה בבסיסו של עמוד המקור, אינו משנה ולא כהוא זה).

ב. הם יוכלו להעביר את הטבעת בגודל 4 מעמוד המקור לעמוד המטרה.

ג. באופן סימטרי למשימה שהם השלימו בסעיף א', יוכלו הנזירים עתה להעביר מגדל בן שלוש טבעות, המצוי על עמוד העזר, לעמוד המטרה. (ושוב נדגיש כי יכולתם להעביר מגדל בן שלוש טבעות מעמוד העזר לעמוד המטרה אינה מושפעת ולא כהוא זה מנוכחותה של הטבעת מספר ארבע בבסיסו של עמוד המטרה, שכן שעה שהנזירים עמלים על העברת המגדל בן שלוש הטבעות מעמוד העזר אל עמוד המטרה הם אינם נוגעים כלל וכלל בטבעת מגודל 4, ומכיוון שטבעת זאת גדולה מכל הטבעות אותן הם מעבירים הרי הם חופשיים להניח כל אחת ואחת מהטבעות אותן הם מעבירים על הטבעת בגודל 4).

אם כך עתה נותר לי לשכנע אתכם שהנזירים מסוגלים להעביר מגדל בן שלוש טבעות מעמוד המקור אל עמוד העזר. כמובן ששוב אנקוט באותו תעלול:

א. אבקש את הנזירים להעביר מגדל בן שתי טבעות מעמוד המקור אל עמוד המטרה (עליו בינתיים לא שוכנת שום טבעת).

ב. אבקש את הנזירים להעביר את הטבעת בגודל 3 מעמוד המקור את עמוד העזר.

ג. אבקש את הנזירים להעביר את המגדל בן שתי הטבעות ש(בסעיף א') הם הניחו על עמוד המטרה אל עמוד העזר.

ושוב נדגיש כי יכולתם של הנזירים לבצע את המשימות שהוטלו עליהם בסעיפים א' ו-ג' הנוכחיים אינה מושפעת מנוכחותן (או אי נוכחותן) של הטבעות מספר ארבע ומספר שלוש, שכן טבעות אלה גדולות מכל טבעת אותה על הנזירים להעביר בסעיפים א' ו-ג' הנוכחיים, ולכן אין מניעה להניח את הטבעות שעליהם להעביר על טבעות אלה, וכמו כן בזמן העברת המגדל בין שתי הטבעות מעמוד זה לעמוד אחר הנזירים אינם נוגעים, קל וחומר אינם מזיזים, את הטבעות מספר 3, ומספר 4.

נסכם אם כך שכדי להוכיח שהנזירים מסוגלים להעביר מגדל בן ארבע טבעות מעמוד המקור לעמוד המטרה די שאוכיח שהם מסוגלים להעביר מגדל בן שלוש טבעות מעמוד המקור אל עמוד העזר. וכדי להוכיח שהנזירים מסוגלים להעביר מגדל בן שלוש טבעות מעמוד המקור אל עמוד העזר די שאוכיח שהם מסוגלים להעביר מגדל בן שתי טבעות מעמוד המקור אל עמוד המטרה. (ויתר תת-המשימות תיפתרנה באופן סימטרי).

כיצד יעבירו הנזירים מגדל בן שתי טבעות מעמוד המקור אל עמוד המטרה? בשלושה שלבים:

א. הם יעבירו מגדל בן טבעת יחידה מעמוד המקור אל עמוד העזר.

ב. הם יעבירו את הטבעת מגודל 2 מעמוד המקור אל עמוד המטרה.

ג. הם יעבירו מגדל בן טבעת יחידה מעמוד העזר אל עמוד המטרה.

עת הנזירים מתבקשים לבצע את המשימות מסעיפים א' ו-ג' הנוכחיים הם יכולים לנשום לרווחה: אלו הם מקרים בהם הבעיה מוצגת עבור נתונים פשוטים, ולכן הנזירים יכולים להעביר באופן מיידי את הטבעת בגודל 1 מהעמוד הרצוי ואל העמוד הרצוי, ובכך להשלים כל אחת משתי תת-המשימות. כמובן שהשלמת הפעולה מסעיף ב' גם היא מיידית, ולכן השלמת תת-המשימה האחרונה היא פשוטה.

בכך הראיתי לכם כי הנזירים יודעים להעביר מגדל בן שתי טבעות מעמוד המקור אל עמוד המטרה. אחרי שהם השלימו משימה זאת הם יוכלו לקחת את הטבעת בגודל שלוש הנמצאת חשופה בראש מגדל המקור (ללא שום טבעת מעליה) ולהעבירה למגדל העזר. אחר יוכלו הנזירים, בטכניקה דומה לזו בה הם השתמשו עת היה עליהם להעביר מגדל בן שתי טבעות מעמוד המקור את עמוד המטרה, להעביר את המגדל בן שתי הטבעות הניצב על עמוד המטרה אל עמוד העזר (תוך שנוכחותה של הטבעת בגודל 3 בתחתיתו של עמוד העזר, ונוכחותה של הטבעת בגודל 4 בבסיסו של עמוד המקור אינה משנה להם ולא כהוא זה).

בשלב זה יהיה על עמוד העזר מגדל בן שלוש טבעות. עתה יוכלו הנזירים להעביר את הטבעת בגודל 4, הניצבת חשופה בבסיסו של עמוד המקור, אל עמוד המטרה; אז הם יוכלו להעביר את המגדל בן שלוש הטבעות הניצב על עמוד העזר אל עמוד המטרה, (תוך שימוש בטכניקה דומה לזו ששימשה אותם עת הם העבירו את המגדל בן שלוש הטבעות מעמוד המקור אל עמוד העזר. נוכחותה של הטבעת בגודל 4 בבסיסו של עמוד המטרה לא תשנה להם, בדיוק כפי שנוכחותה בעבר בבסיסו של עמוד המקור לא שינתה להם).

8.3.3 <u>תיאור אלגוריתמי של הפתרון</u>

ההסבר שראינו בסעיף הקודם מוביל באופן מיידי לַפונקציה שנציג בסעיף הנוכחי. הפונקציה מקבלת: (א) את גודלו של המגדל אותו עליה להעביר (בפרמטר `size`), (ב) את מספרו של העמוד עַליו ניצב המגדל (כלומר את מספרו של עמוד המקור, בפרמטר `source`), (ג) את מספרו של העמוד אליו עליה להעביר את המגדל (כלומר את מספרו של עמוד המטרה, בפרמטר `dest`), (ד) את מספרו של העמוד בו היא רשאית להשתמש לשם השלמת התהליך (כלומר את מספרו של עמוד העזר, בפרמטר `aux`). הפונקציה מוציאה סדרה של הוראות לנזירים, איזה טבעת עליהם להעביר מאיזה עמוד ולאיזה עמוד לשם השלמת המשימה. הקריאה לפונקציה מהתכנית הראשית תהיה: `hannoi(4, 1, 3, 2);` כלומר על הפונקציה להעביר מגדל בגודל 4, מעמוד מספר 1, אל עמוד מספר 3, תוך שהיא עושה שימוש בעמוד מספר 2. והגדרת הפונקציה:

```
void hanoi(unsigned int size,
           unsigned int source, unsigned int dest,
           unsigned int aux)
{
   if (size == 1)
      cout << "Move ring of size 1 from pole #"
           << source << "to pole #" << dest << endl;
   else
   {
      hanoi(size -1, source, aux, dest) ;
      cout << "Move ring of size " << size
           << "from pole #" << source
           << "to pole #" << dest << endl;
      hanoi(size -1, aux, dest, source) ;
   }
}
```

הסבר הפונקציה: התעלול אותו יש להפְנים בפונקציה זאת הוא 'המשחק' שנעשה בַּפרמטרים (שהופכים להיות ארגומנטים בַּקריאות הרקורסיביות שמזמנת הקריאה הנוכחית). נזכור כי הפרמטר *הראשון* של הפונקציה מציין את גודלו של המגדל שעל העותק הנוכחי של הפונקציה להעביר, הפרמטר *השני* מציין את שם העמוד ממנו יש להעביר את המגדל, הפרמטר *השלישי* מציין את שם העמוד אליו יש להעביר את המגדל, והפרמטר *הרביעי* מציין את מספרו של עמוד העזר. ראשית, נבחן את הַקריאה הרקורסיבית הראשונה מבין השתיים שמזמנת הקריאה שהיא עתה הנוכחית: `hanoi(size-1, source, aux, dest);`. בשל סידור הארגומנטים בקריאה המזומנת יוצא שאם הקריאה הנוכחית התבקשה להעביר, לדוגמה, מגדל בגודל 4, מעמוד מספר 1 (כלומר זה ערכו של הפרמטר `source`), לעמוד מספר 3 (שערכו מצוי בפרמטר `dest`), אזי הקריאה הרקורסיבית שמזמנת הקריאה הנוכחית מתבקשת להעביר מגדל בגודל `size-1` כלומר 3, מעמוד המקור, כלומר מעמוד מספר 1 (הארגומנט השני בקריאה הרקורסיבית המזומנת), לעמוד העזר, כלומר לעמוד מספר 2 (ומכיוון שהארגומנט השלישי בקריאה הרקורסיבית המזומנת הוא `aux` של הקריאה הנוכחית, אזי מה שהיה `aux` של הקריאה הנוכחית יהיה `dest` של הקריאה המזומנת על-ידי הקריאה הנוכחית). *אחרי* שקריאה רקורסיבית זאת תשלים את פעולתה תוכל הקריאה הרקורסיבית שלנו להוציא את פקודת הפלט שנסמנה כאן באופן ציורי $1 \rightarrow^{4} 3$ (כלומר העבר טבעת בגודל 4 מעמוד 1 לעמוד 3). ואחרי שליחת פלט זה תוכל הקריאה הרקורסיבית הנוכחית להשלים את משימתה על-ידי שהיא תקרא רקורסיבית: `hanoi(size-1, aux, dest, source);`. קריאה רקורסיבית זאת מבקשת שוב להעביר מגדל בגודל 3, אך הפעם מהמגדל שהיה מגדל העזר של הקריאה הנוכחית (מכיוון ש- `aux` של הקריאה הנוכחית הוא הארגומנט השני בקריאה הרקורסיבית המזומנת על-ידי הקריאה הנוכחית, אזי הוא יהיה ה- `source` של הקריאה הרקורסיבית הבאה), אל מגדל המטרה של הקריאה הנוכחית.

למרות היגיעה הרבה הכרוכה בכך נעקוב אחר דוגמת הרצה. נניח כי הקריאה הראשונה היא: `hanoi(4, 1, 2, 3);`. כלומר יש להעביר מגדל בגודל 4, מעמוד מספר 1, לעמוד מספר 2, (תוך שימוש בעמוד מספר 3 כעמוד העזר).

א. בקריאה הרקורסיבית הראשונה: `size=4, source=1, dest=2, aux=3`.
אנו במקרה המורכב לכן יש לבצע שלושה צעדים:

1. זימון: `hanoi(size-1, source, aux, dest)` כלומר: `hanoi(3, 1, 3, 2);`.
   אחרי השלמת קריאה זאת:

2. הצגת הפלט: $2 \rightarrow^{4} 1$.

3. זימון: `hanoi(size-1, aux, dest, source);` כלומר: `hanoi(3, 3, 2, 1);`

הפעולה הראשונה המבוצעת על-ידי עותק זה של הפונקציה היא א1'. לכן עתה נדון בה:

ב. סעיף א1' זימן את הקריאה הרקורסיבית: `hanoi(3, 1, 3, 2);` עתה נדון בה: בקריאה זאת: `size=3, source=1, dest=3, aux=2`. אנו במקרה המורכב, לכן יש לבצע שלושה צעדים:

1. זימון: `hanoi(size-1 =2, source =1, aux =2, dest =3)`.
   נסביר את צורת הכתיבה: לצד כל ארגומנט כתבנו את ערכו, לדוגמה לצד `size-1` כתבנו את ערכו (`=2`).
2. הצגת הפלט: $3 \rightarrow^{3} 1$.
3. זימון: `hanoi(size-1 = 2, aux =2, dest =3, source =1);`

נדון ראשית בקריאה המזומנת מסעיף ב1':

ג. סעיף ב1' זימן את הקריאה הרקורסיבית: `hanoi(2, 1, 2, 3);` בקריאה זאת: `size =2, source =1, dest =2, aux =3`. אנו במקרה המורכב לכן יש לבצע שלושה צעדים:

1. זימון: `hanoi(size-1 =1, source =1, aux =3, dest =2);`
2. הצגת הפלט: $2 \rightarrow^{2} 1$.
3. זימון: `hanoi(size-1 = 1, aux =3, sest =2, source =1);`

הפעולה הראשונה המבוצעת על-ידי עותק זה של הפונקציה היא ג1'. לכן עתה נדון בה:

ד. סעיף ג1' זימן את הקריאה הרקורסיבית: `hanoi(1, 1, 3, 2);` בקריאה זאת: `size =1, source =1, dest =3, aux =2`. סוף סוף אנו במקרה הפשוט. לכן ניתן להציג את פקודת הפלט: $3 \rightarrow^{1} 1$, ולסיים. עת העותק הנוכחי של `hanoi` מסיים לאן פונה התכנית? העותק הנוכחי זומן מסעיף ג1', לכן עת עותק זה מסתיים יש לפנות לביצוע ג2', כלומר להציג את הפלט: $2 \rightarrow^{2} 1$. עתה ניתן להמשיך לסעיף ג3' כלומר לקרוא רקורסיבית עם: `hanoi(1, 3, 2, 1);`. נדון בקריאה זאת:

ה. סעיף ג3' זימן את הקריאה הרקורסיבית: `hanoi(1, 3, 2, 1);` בקריאה זאת: `size =1, source =3, dest =2, aux =1`. אנו שוב במקרה הפשוט. לכן ניתן להציג את פקודת הפלט: $2 \rightarrow^{1} 3$, ולסיים. עת העותק הנוכחי של `hanoi` מסיים לאן פונה התכנית? העותק הנוכחי זומן מסעיף ג3', שהוא תת-הסעיף המסיים את סעיף ג'. לכן בזאת מסתיימת הקריאה הרקורסיבית מסעיף ג'. לאן פונה התכנית מכאן? מי קרא לעותק שבוצע בסעיף ג'? רשומת ההפעלה של סעיף ג' מורה לנו כי סעיף זה נקרא מסעיף ב1'. לכן עתה ניתן להמשיך בביצוע תת-סעיף ב2' כלומר להציג את הפלט: $3 \rightarrow^{3} 1$. מסעיף ב2', אנו מתקדמים לסעיף ב3', אשר מזמן את הקריאה הרקורסיבית: `hanoi(2, 2, 3, 1);`. נדון בקריאה זאת.

ו. סעיף ב3' זימן את הקריאה הרקורסיבית: `hanoi(2, 2, 3, 1);` בקריאה זאת: `size =2, source =2, dest =3, aux =1`. אנו במקרה המורכב לכן יש לבצע שלושה צעדים:

1 זימון: `hanoi(size-1 =1, source =2, aux =1, dest =3);`
2 הצגת הפלט: $3 \rightarrow^{2} 2$.
3 זימון: `hanoi(size-1 = 1, aux =1, sest =3, source =2);`

הפעולה הראשונה המבוצעת על-ידי עותק זה של הפונקציה היא ו1'. לכן עתה נדון בה:

ז. סעיף ו1' זימן את הקריאה הרקורסיבית: hanoi(1, 2, 1, 3);. בקריאה זאת: size =1, source =2, dest =1, aux =3. אנו שוב במקרה הפשוט. לכן ניתן להציג את פקודת הפלט: 1→[1]→2, ולסיים. עת העותק הנוכחי של hanoi מסיים לאן פונה התכנית? העותק הנוכחי זומן מסעיף ו1', לכן עתה ניתן לחזור לסעיף ו', ולבצע את ו2', כלומר להציג את הפלט: 3→[2]→2. מ- ו2' אנו מתקדמים ל- ו3' ומזמנים: hanoi(1, 1, 3, 2);. נדון בקריאה זאת:

ח. סעיף ו3' זימן את הקריאה הרקורסיבית: hanoi(1, 1, 3, 2);. זהו המקרה הפשוט. נציג את הפלט 3→[1]→1, ונסיים. עת אנו מסיימים את סעיף ח' אנו חוזרים לסיומו של סעיף ו'. סעיף ו' זומן על-ידי סעיף ב3', לכן אנו חוזרים לסיומו של סעיף ב'. מי זימן את סעיף ב'? סעיף א1, לכן עם סיום סעיף ב' ניתן לפנות ולבצע את א2' כלומר להדפיס: 2→[4]→1.

נעצור לרגע את סימולציית ההרצה כדי לבחון את פקודות הפלט שהוצגו עד כה:

1. סעיף ד' הוציא את פקודת הפלט: 3→[1]→1.
2. סעיף ג2' הוציא את פקודת הפלט: 2→[2]→1.
3. סעיף ה' הוציא את פקודת הפלט: 2→[1]→3.
4. סעיף ב2' הוציא את פקודת הפלט: 3→[3]→1.
5. סעיף ז' הוציא את פקודת הפלט: 1→[1]→2.
6. סעיף ו2' הוציא את פקודת הפלט: 3→[2]→2.
7. סעיף ח' הוציא את פקודת הפלט: 3→[1]→1.
8. סעיף א2' הוציא את פקודת הפלט: 2→[4]→1.

אני מזמין אתכם לעקוב אחר סדרת הפעולות ולגלות שעד כה העברנו את תת-המגדל המורכב מהטבעות מספר 1..3 לעמוד מספר 3, ואת הטבעת מספר ארבע לעמוד מספר 2. לפיכך עתה, כדי להשלים את תהליך, עלינו להעביר את תת-המגדל המורכב מהטבעות 1..3 מהעמוד מספר 3, לעמוד מספר 2. ואכן עתה אנו פונים לביצוע א3, כלומר ל: hanoi(3, 3, 2, 1);.

ט. סעיף א3 זימן את: hanoi(3, 3, 2, 1);. נבחן קריאה זאת. היא מתפצלת לשלוש פעולות:
   1. זימון: hanoi(2, 3, 1, 2);.
   2. הצגת הפלט: 2→[3]→3.
   3. זימון: hanoi(2, 1, 2, 3);.

י. נדון בקריאה שיצאה מ- ט1': hanoi(2, 3, 1, 2);. היא מתפצלת לשלוש פעולות:
   1. זימון: hanoi(1, 3, 2, 1);.
   2. הצגת הפלט: 1→[2]→3.
   3. זימון: hanoi(1, 2, 1, 3);.

יא. נדון בקריאה שיצאה מ- י1': hanoi(1, 3, 2, 1);. זהו המקרה הפשוט. ניתן להציג את הפלט: 2→[1]→3, ולסיים. עם סיום עותק זה של האנוי אנו שבים לתת-סעיף י2', ולכן מציגים את הפלט: 1→[2]→3. מסעיף י2', אנו מתקדמים ל- י3', ולכן מזמנים: hanoi(1, 2, 1, 3);.

יב. נדון בקריאה שיצאה מ- י3': hanoi(1, 2, 1, 3);. זהו המקרה הפשוט. ניתן להציג את הפלט: 1→[1]→2, ולסיים. עם סיום עותק זה של האנוי אנו חוזרים לסיומו של סעיף י'. סעיף י' זומן מסעיף ט1'. לכן עתה נתקדם לסעיף ט2', ונציג את הפלט: 2→[3]→3. מסעיף ט2' אנו ממשיכים ל-ט3', ומזמנים: hanoi(2, 1, 2, 3);. נדון בקריאה זאת:

יג. נדון בקריאה שיצאה מ- ט3': hanoi(2, 1, 2, 3);. היא מתפצלת לשלוש פעולות:

1. זימון: `hanoi(1, 1, 3, 2);`.
2. הצגת הפלט: $2 \rightarrow^{2} 1$.
3. זימון: `hanoi(1, 3, 2, 1);`.

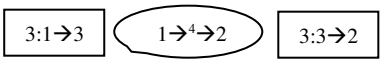
נדון בראשונה מבין שלוש הפעולות:

יד. נדון בקריאה שזומנה מ- יג1': `hanoi(1, 1, 3, 2);`. זה המקרה הפשוט. נציג את הפלט: $3 \rightarrow^{1} 1$, ונסיים. נחזור ל- יג2', נציג את הפלט: $2 \rightarrow^{2} 1$, ונפנה ל- יג3', זימון: `hanoi(1, 3, 2, 1);`.

טו. נדון בקריאה שזומנה מ- יג3': `hanoi(1, 3, 2, 1);`. זהו המקרה הפשוט. נציג את הפלט: $2 \rightarrow^{1} 3$, ונסיים. נחזור לסיומה של הקריאה מסעיף יג'. קריאה זאת זומנה מ- ט3', לכן נפנה לסיומו של סעיף ט'. סעיף ט' זומן מסעיף א3', לכן סיומו של סעיף ט' מבשר גם את סיומו של סעיף א', ובא לציון הגואל.

אתם מוזמנים לעקוב אחר פעולות הפלט שבוצעו בחלק זה של הריצה, ולוודא שהן אכן משלימות המשימה.

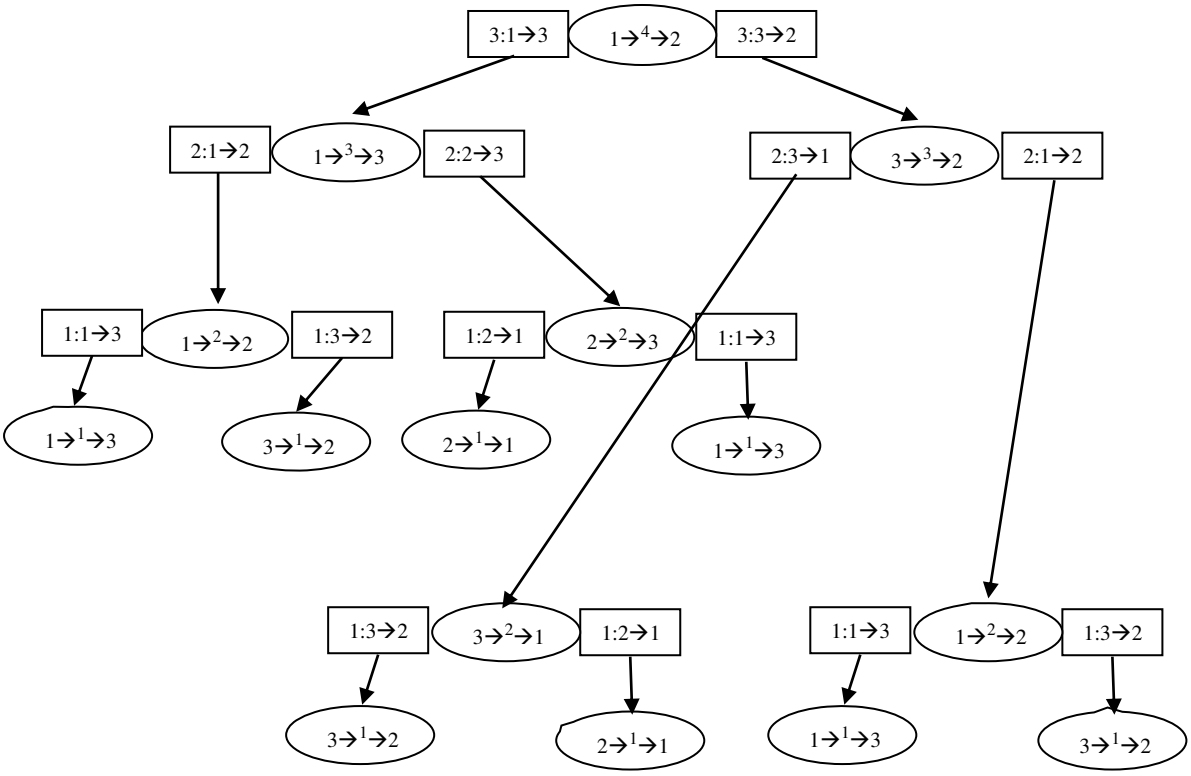
דרך נוספת לעקוב אחר התנהלות `hanoi` היא באמצעות ציור עץ הקריאות.
כל צומת בעץ יכיל לכל היותר שלושה מרכיבים: (א) הקריאה הרקורסיבית הראשונה שהעותק הנוכחי מזמן, (ב) פקודת הפלט שהעותק הנוכחי מבצע, (ג) הקריאה הרקורסיבית השניה שהעותק הנוכחי מזמן. הצמתים שייצגו קריאות רקורסיביות עם 'נתונים פשוטים', יכילו רק את המרכיב השני מבין השלושה. לצמתים אלה לא יהיו צמתים בנים, ולכן הם נקראים עלי העץ. מבחינה ציורית נצייר למשל את הצומת המתאים לקריאה `hanoi(4, 1, 2, 3)` באופן הבא:



נסביר: מרכיבי הצומת מופיעים משמאל לימין: המרכיב הראשון, המסומן במלבן, מייצג את הקריאה הרקורסיבית שאמורה להעביר מגדל בן שלוש טבעות, מעמוד מספר1, לעמוד מספר שלוש. המרכיב השני, המופיע באליפסה, מייצג את פקודת הפלט שהקריאה הנוכחית מבצעת. המרכיב השלישי, המסומן בַּמלבן הימני מייצג את הקריאה רקורסיבית שאמורה להעביר מגדל בן שלוש טבעות מעמוד מספר 3, לעמוד מספר 2. שימו לב שתמיד אותו ערך מופיע מימין לחץ במלבן השמאלי, ומשמאל לחץ במלבן הימני, שכן במלבן השמאלי אנו מעבירים תת-מגדל *אל* עמוד העזר (מעמוד המקור), ובמלבן הימני אנו מעבירים אותו תת-מגדל *מ*עמוד העזר (אל עמוד המטרה).

כרגיל, כל צומת בעץ קשור בחץ לבניו. נזכור כי סדר ביצוע הצמתים הוא שעת לצומת `X` יש בן שמאלי `L` ובן ימני `R`, אזי ראשית מבוצע `L` (על כל צאצאיו), שנית מבוצעת פקודת הפלט של `X`, ושלישית מבוצע `R` (על כל צאצאיו). לביקור כזה בעץ אנו קוראים ביקור בסידור in-order (בניגוד לביקורים אחרים בהם ראשית מציגים את הנתונים שבצומת ורק אחר פונים לבניו, או ראשית פונים לשני הבנים ורק בסוף מציגים את הנתונים שבשורש).

נציג עתה את העץ המתאים לקריאה: `hanoi(4, 1, 2, 3);`



שוב, אתם מוזמנים לעקוב אחר מערכת הקריאות הרקורסיביות, בפרט אחר פקודות הפלט המבוצעות, כדי לבדוק שסדרת הפלטים אכן מתארת תהליך ההעברה חוקי ורצוי.

8.3.4 זמן הריצה של האלגוריתם

כפי שראינו מגדלי האנוי היא פונקציה פשוט ביותר: אין בה אפילו פקודת לולאה. לכן כדי לחשב את כמות העבודה הנעשית על-ידה, עת עליה להעביר מגדל בן n טבעות, עלינו לספור את כמות הקריאות הרקורסיביות המבוצעות, ואת כמות פקודות הפלט הנשלחות.

א. עת האנוי נקרא עם מגדל בגודל 1: הוא מבצע פקודת פלט אחת, ותו לא.

ב. עת האנוי נקרא עם מגדל בגודל 2: הוא קורא פעמיים להאנוי עבור מגדל בגודל 1, וכל ביצוע של האנוי עם מגדל בגודל 1 עולה, כפי שראינו בסעיף א', פעולה יחידה. כמו כן האנוי עם מגדל בגודל 2 מבצע פעולת פלט יחידה. לכן סך כל העבודה נעשית על-ידו היא: 2*hanoi(1) +1 = 2*1 +1 =3. נסביר את צורת הכתיבה: פעמיים כמות העבודה שנעשית על-ידי hanoi עם מגדל בגודל 1, ועוד פעולה יחידה.

ג. עת האנוי נקרא עם מגדל בגודל 3: הוא קורא פעמיים להאנוי עבור מגדל בגודל 2, וכל ריצה של האנוי עם מגדל בגודל 2 עולה, כפי שראינו בסעיף ב', שלוש פעולות. כמו כן האנוי עם מגדל בגודל 3 מבצע פעולת פלט יחידה. לכן סך כל העבודה נעשית על-ידו היא: 2*hanoi(2) +1 = 2*3 +1 =7.

ד. עת האנוי נקרא עם מגדל בגודל 4: הוא קורא פעמיים להאנוי עבור מגדל בגודל 3, וכל ריצה של האנוי עם מגדל בגודל 3 עולה, כפי שראינו בסעיף ג', שבע

פעולות. כמו כן האנוי עם מגדל בגודל 4 מבצע פעולת פלט יחידה. לכן סך כל העבודה נעשית על-ידו היא: 2*hanoi(3) +1 = 2*7 +1 =15.

ה. עת האנוי נקרא עם מגדל בגודל 5: הוא קורא פעמיים להאנוי עבור מגדל בגודל 4, וכל ריצה של האנוי עם מגדל בגודל 4 עולה, כפי שראינו בסעיף ד', 15 פעולות. כמו כן האנוי עם מגדל בגודל 5 מבצע פעולת פלט יחידה. לכן סך כל העבודה נעשית על-ידו היא: 2*hanoi(4) +1 = 2*15 +1 =31.

ובאופן כללי: אנו רואים שתוספת של טבעת למגדל מכפילה לערך את כמות עבודה המבוצעת. מהמקרים שתיארנו ניתן להכליל כי כמות העבודה הנעשית לשם העברת מגדל בן n טבעות היא: 2^n-1. אנו אומרים כי כמות העבודה היא **מעריכית** (בלע"ז אקספוננציאלית) ב- n, שכן n מופיע בביטוי במעריך, (וזאת בניגוד לביטוי פולינומיאלי ב- n, כגון n^{17}, בו n מצוי בבסיס, או ביטוי ליארי ב- n בו n מצוי בבסיס והמעריך הוא 1, או ביטוי לוגריתמי ב- n כגון $lg_{17}(n)$, או דרגות מורכבות אחרות).

אלגוריתמים המחייבים עבודה בסדר גודל מעריכי נחשבים לאלגוריתמים לא מעשיים, שכן עבור n גדול דיו זמן הריצה שלהם רב ממידות אנוש, אולי אפילו ממידות עולמנו. לדוגמה עבור n=100 יהיה זמן הריצה: $2^{100} = (2^{10})^{10}$, מכיוון ש- 2^{10} שווה בערך לאלף אזי, על-פי חוקי חזקה, $(2^{10})^{10}$ שווה בערך ל- $(10^3)^{10}$, כלומר ל- 10^{30}, מספר נאה למדי לכל הדעות. עתה אתם יכולים להניח, למשל, כי כל פעולה במחשב דורשת זמן של מיליונית השניה, ואז לחשב כמה זמן ייקח להריץ את הפונקציה עבור מגדל בגודל 100 טבעות. החישוב יביא אתכם למסקנה כי יש חשש שסוף העולם יגיע טרם שהנזירים החרוצים מהאנוי יביאו אותנו לַנירוונה. (והם קצת פחות מהירים ממיליון פעולות בשניה. בהנחה של מיליון פעולות בשניה, מדובר בסדר גודל של 10^{16} שנים בלבד).

מגדלי האנוי היא דוגמה יפה לבעיה אשר נותנת את עצמה בקלות לפתרון רקורסיבי. יחד עם זאת קיים לבעיה גם פתרון לא רקורסיבי. קווי היסוד של פתרון הם כדלהלן:

א. התייחס לעמודים כאילו הם מסודרים במעגל בו עמוד #2 ניצב אחרי עמוד #1, עמוד #3 ניצב אחרי עמוד #2, ועמוד #1 ניצב אחרי עמוד #3.

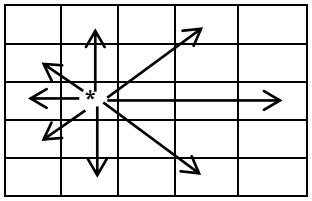
ב. בכל צעד באלגוריתם בחר טבעת שלא הוזזה בצעד הקודם, והזז אותה על-פי כיוון ההתקדמות שתואר בסעיף א' לַעמוד הקרוב ביותר עליו ניתן להניחה.

אלגוריתם זה, גם אם מביא לפתרון הבעיה, הוא פחות אינטואיטיבי מזה שהצגנו לפניו. בפרט הרבה פחות ברור ממנו כיצד משפיע מגדל המטרה שנבחר על התהליך. גם הערכת זמן הריצה של האלגוריתם הנוכחי הרבה פחות מיידית.

8.4 בעיית שמונה המלכות

בעיית שמונה המלכות היא הדוגמה עמה נהוג לפתוח את הדיון בתת-קבוצה של קבוצת האלגוריתמים הרקורסיביים הקרויה אלגוריתמים של **עקיבה לאחור** (back tracking). בכל האלגוריתמים המבצעים עקיבה לאחור המחשב בודק באופן שיטתי את כל האפשרויות לפתרון. המחשב דומה למי שמנסה לצאת ממבוך על-ידי בדיקה שיטתית של כל המסלולים האפשריים; בכל פעם שהוא מגיע ל-'אין מוצא' הוא נסוג אחורנית ומנסה מסלול שטרם נוסה ושדומה ביותר למסלול עמו הוא נכשל עתה. המסלול החדש ייבחר על-ידי שסורק המבוך יפנה (למשל) ימינה במקום שמאלה בהסתעפות האחרונה שלא נוסתה ושבה הוא פנה בעבר שמאלה.

כדי להציג את בעיית שמונה המלכות נשלים, ראשית, רקע שאולי חסר לחלקכם. לוח שחמט מורכב משמונה שורות, בכל שורה יש שמונה משבצות. על-פי חוקי השחמט מלכה רשאית להתקדם מהמקום בו היא מצויה ימינה, שמאלה, מעלה, מטה, או בכל אחד מארבעה כיוונים אלכסוניים מספר צעדים כלשהו. נדגים זאת על לוח בגודל 5X5:



המלכה הניצבת בַמשבצת המסומנת בכוכבית, רשאית להתקדם מספר צעדים כלשהו בכל אחד משמונת הכיוונים המתוארים באמצעות חצים.

בעיית שמונה המלכות מוצגת באופן הבא: הצג את *כל* הסידורים האפשריים של שמונה מלכות על הלוח (בגודל 8X8) כך שאף מלכה לא תוצב במשבצת אליה רשאית להתקדם מלכה אחרת.
לדוגמה, עבור לוח בגודל 4X4 הפתרון הבא הוא פתרון חוקי:

	*		
			*
*			
		*	

אני מזמינכם לנסות ולבדוק שאם מציבים את אחת המלכות בפינה השמאלית העליונה של הלוח, אזי לא ניתן להציב על-גבי הלוח ארבע מלכות כנדרש.

עובדה שקל לתת עליה את הדעת היא שתנאי הכרחי לכך שימצא פתרון לבעיה הוא שכל מלכה תוצב בשורה נפרדת, כלומר שבכל שורה תוצב בדיוק מלכה אחת (לא יותר, ולא פחות).

האלגוריתם שנציג, כדרכם של אלגוריתמים הפועלים בעקיבה לאחור, פותר את הבעיה בהדרגתיות, כל קריאה רקורסיבית לאלגוריתם מנסה להציב מלכה נוספת על הלוח (והיא עשויה להצליח בכך או להיכשל). האלגוריתם מקבל שני פרמטרים:

א. לוח השחמט (עליו ניצבות כבר המלכות שהוצבו על-ידי קריאות רקורסיביות קודמות לאלגוריתם). לפרמטר זה נקרא `board`.

ב. מספר המלכות שכבר ניצבות על הלוח. פרמטר זה יקרא queen_num.

בַּקריאה הראשונה לאלגוריתם הלוח המועבר לשגרה ריק, וערכו של הפרמטר queen_num הוא אפס, שכן הקריאה הראשונה לאלגוריתם מקבלת לוח שאין עליו כל מלכות.

האלגוריתם פועל באופן הבא: כל קריאה רקורסיבית סורקת בלולאה את תאי השורה מספר queen_num, ומנסה, *לחילופין*, להציב מלכה בכל אחד מהתאים. עבור כל תא שנסרק בודק האלגוריתם האם ניתן להציב על-גבי הלוח מלכה נוספת בתא זה, באופן שהמלכה החדשה לא תתנגש עם קודמותיה (נניח כי i הוא משתנה הלולאה שעובר על תאי השורה מספר queen_num) . במידה ומתברר כי ניתן להציב מלכה בתא board[queen_num][i] פועל האלגוריתם באופן הבא:

א. אם זו הייתה המלכה האחרונה שהיה צורך להציב, אזי הקריאה מציגה את מצב הלוח.
ב. אחרת (זו לא הייתה המלכה האחרונה שהיה צורך להציב) הקריאה הנוכחית מזמנת קריאה רקורסיבית נוספת לאלגוריתם. על הקריאה המזומנת יהיה לנסות ולהוסיף מלכות נוספות מעבר למלכה שהציבה הקריאה הנוכחית.
ג. אחרי ביצוע סעיפים א' או ב' מנקה הקריאה הנוכחית את התא board[queen_num][i] (כדי לנסות ולהציב את המלכה הנוכחית בתא אחר בשורה מספר queen_num).

נציג, ראשית, את הקוד המתאים, ואחר נבצע סימולציית הרצה שתסייע לנו להבינו. את הפרמטר board נגדיר באופן הבא: bool board[N][N]. הפונקציה שנכתוב תשתמש בשירותיהן של שתי של פונקציות עזר:

א. bool consistent(bool board[N][N], unsigned int row, unsigned int col) פונקציה זאת תקבל את לוח השח-מט (הפרמטר board) ומשבצת כלשהי בלוח (שתתואר באמצעות row, col). הפונקציה תחזיר את הערך true אם ניתן להציב מלכה במשבצת board[row][col] בלי שאותה מלכה תתנגש עם המלכות המצויות כבר על-גבי הלוח, (במילים אחרות באופן שמלכה זאת תהיה עקבית [קונסיסטנטית] עם המלכות המצויות כבר על הלוח).
ב. void print(bool board[N][N]) פונקציה זאת תציג את מצב הלוח בסיום תהליך ההצבה.

הקריאה לפונקציה מהתכנית הראשית תהיה: queens(main_bord, 0); עבור משתנה main_board של התכנית הראשית שאותחל טרם הקריאה להיות נקי ממלכות.

נגדיר עתה את הפונקציה הדרושה:

```
void queens(bool board[N][N], unsignrd int num_of_queens)
{
   for (unsigned int i=0; i < N; i++)
      if (consistent(board, num_of_queens, i) )
      {
         board[num_of_queens][i] = A_QUEEN ;
         if (num_of_queens = N-1)
             print(board) ;
         else
```

```
            queens(board, num_of_queens +1) ;
         board[num_of_queens][i] = EMPTY ;       //(+)
      }
}
```

לפני שנפנה לדוגמת הרצה נעמוד על כמה מרכיבים בקוד: הפונקציה כוללת כאמור לולאה אשר עוברת על תאי השורה `num_of_queens` (שכן בשורה זאת עלינו למקם את המלכה הנוכחית). עבור כל תא ותא היא קוראת לפונקציה `consistent` אשר בודקת האם ניתן להציב מלכה באותו תא. אם כן:

א. הפונקציה מציבה את המלכה הנוכחית בתא זה.

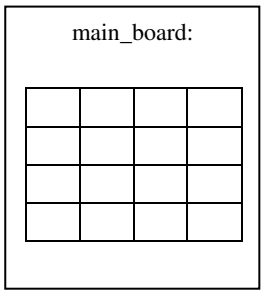
ב. במידה ועל הלוח היו כבר `N-1` מלכות (ולכן הקריאה הנוכחית הציבה את המלכה ה- `N`-ית) הפונקציה מציגה את הלוח.

ג. אחרת הפונקציה קוראת רקורסיבית לשם המשך תהליך ההצבה.

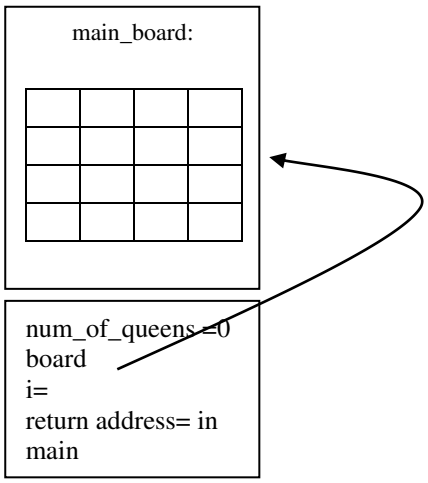
ד. כאשר הקריאה הרקורסיבית חוזרת (וייתכן שהיא כן או לא הצליחה להשלים את התהליך מעבר למלכה שהוצבה כרגע), הקריאה הנוכחית מוחקת את המלכה שהוצבה עתה, כדי להמשיך ולנסות להציב את המלכה הנוכחית בתאים אחרים בשורה `num_of_queens`, ולבחון האם אז ניתן יהיה להשלים את הפתרון (אני מזכיר לכם כי הנחנו שיש להציג את כל הפתרונות).

עתה, נבצע סימולציית הרצה חלקית עבור לוח בגודל `4X4`.
נצייר את מצב המחסנית טרם הקריאה הראשונה:

activation record of main:



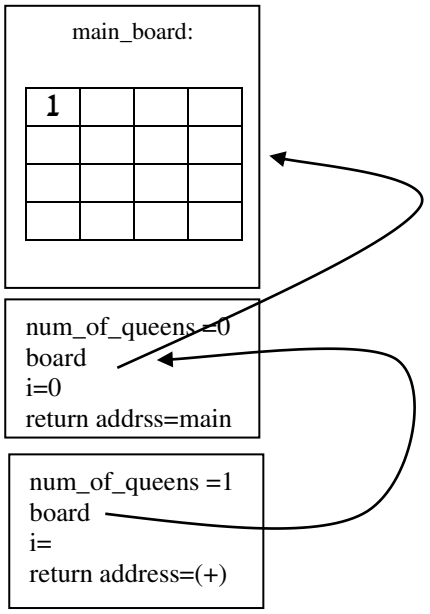
ומייד עם הכניסה לקריאה הראשונה ל- `queens`:



א. עם השלמת תהליך הבניה של רשומת ההפעלה מתחילה להתבצע לולאת ה- `for` שבפונקציה. (1) ל- `i` מוכנס הערך אפס, (2) מתבצעת קריאה ל- `consistent` אשר מחזירה `true`, (3) בתא `borad[0][0]` מוצבת מלכה, (4) מכיוון שמספר

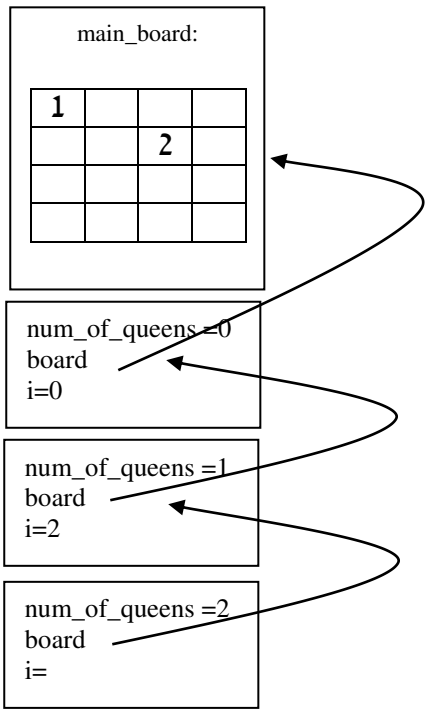
המלכות שהלוח מכיל עדיין קטן מ- `N-1` מתבצעת קריאה רקורסיבית. נדון בקריאה זאת, ונזכור שכתובת החזרה שלה היא הפקודה (+) כלומר ניקוי `borard[0][0]`.

ב. נדון בקריאה שזומנה מסעיף א'. נציג את מצב המחסנית בעקבות ביצוע הקריאה ובניית רשומת ההפעלה שלה על-גבי המחסנית. שימו לב כי ערכו של `i` בקריאה הראשונה הוא אפס (כמו כן, בניגוד לטיפוס המערך אך לשם הבהירות, סימנו את המלכה שהוצבה על הלוח בספרה 1 כדי לציין שזאת המלכה הראשונה):



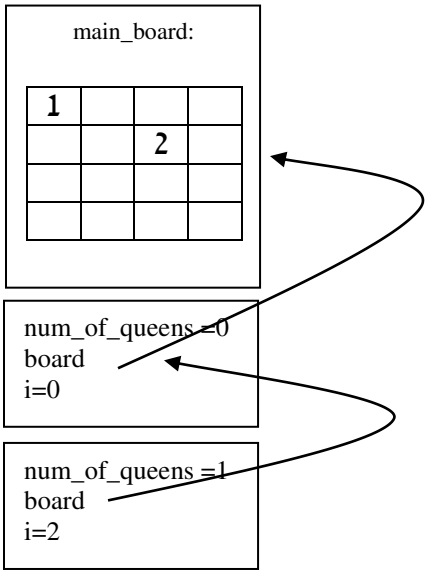
בעקבות בניית רשומת ההפעלה מתחיל ביצוע העותק השני של הפונקציה: עת `i=0` מחזירה `consistent` את הערך `false` ולכן דבר לא נעשה; כך גם עת `i=1`. אך עת `i=2` מחזירה `consistent` את הערך `true` ולכן: (1) בתא `board[1][2]` מוצבת מלכה, (2) מכיוון שמספר המלכות שהלוח מכיל עדיין קטן מ- `N-1` מתבצעת קריאה רקורסיבית. נדון בקריאה זאת, ונזכור שכתובת החזרה שלה היא נקוי `borard[1][2]`.

ג. נדון בקריאה שזומנה מסעיף ב'. על קריאה זאת להציב את המלכה השלישית. נציג את מצב המחסנית בעקבות ביצוע הקריאה ובניית רשומת ההפעלה שלה על-גבי המחסנית. שימו לב כי ערכו של `i` בקריאה השניה הוא שתיים (כמו כן מטעמי קיצור השמטנו את כתובת החזרה המופיעה ברשומת ההפעלה, נזכור שהיא תמיד הפקודה (+) בעותק המצוי במחסנית מקום אחד עמוק יותר מהעותק הנקרא):



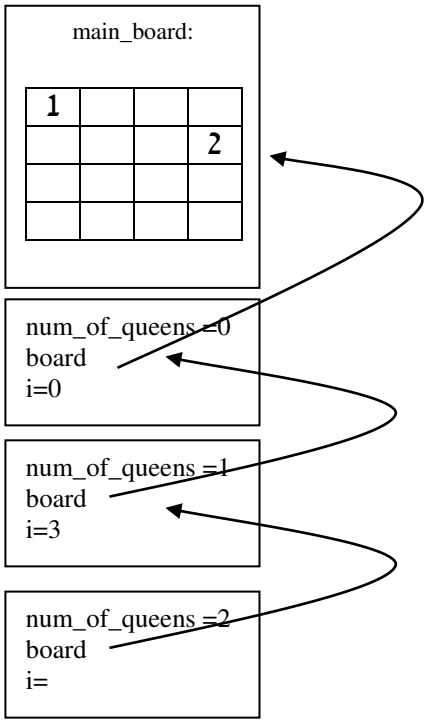
בעקבות בניית רשומת ההפעלה מתחיל ביצוע העותק השלישי של הפונקציה: עבור כל ערכי `i` (כלומר `0..3`) מחזירה `consistent` את הערך `false`, ולכן דבר לא נעשה. לכן הפונקציה מסיימת בלי שהיא הצליחה להציב מלכה כלשהי על הלוח. ראשית, נזכור שכתובת החזרה שלה היא נקוי `borard[1][2]`. ועתה נרצה לדון ברמת המשמעות במצב אליו נקלענו: הגענו לנקודה עדינה אותה חשוב להבין: הלולאה שהריץ העותק הנוכחי של `queens` הסתיימה. מכיוון שבפונקציה `queens` אין עוד פקודות מעבר ללולאה זאת, משמעות הדבר היא כי העותק הנוכחי של פונקציה הסתיים. מה פשר הדבר ברמת המשמעות? במילים אחרות מנין באנו ואנה אנו הולכים ברמת הפתרון של הבעיה? העותק הנוכחי של `queens` ניסה להציב את המלכה השלישית בכל אחד ואחד מתאי השורה השלישית (שורה מספר 2) בַּמערך ונכשל. הדבר מורה לנו כי לא ניתן להוסיף מלכה שלישית מעבר לשתי המלכות שהוצבו כבר. מה אם כך עלינו לעשות? עלינו להזיז את המלכה השניה מעט, ולנסות שוב להציב את המלכה השלישית, (ובעזרת האלגוריתם גם את הרביעית כדי להשלים את הפתרון). נראה כיצד האלגוריתם אכן ידאג לכך.

ד. רשומת ההפעלה השלישית והאחרונה של `queens` מוסרת מהמחסנית, ואנו מפשירים את העותק השני של הפונקציה (זה המצוי עתה בראש המחסנית). כזכור, בעותק זה ערכו של `i` הוא שתיים. נציג שוב את המחסנית:



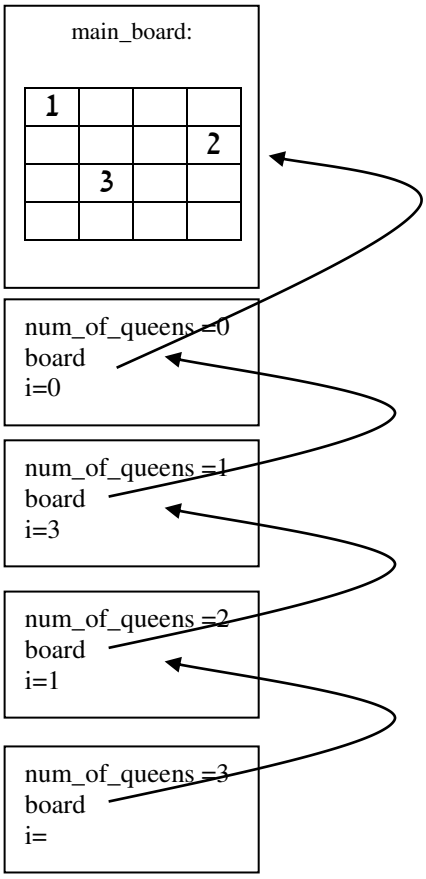
הפקודה שעלינו לבצע עתה היא הפקודה (+) אשר מסירה את המלכה #2 מעל הלוח. העותק השני של הפונקציה ממשיך להתבצע, בפרט הוא ממשיך בביצוע הלולאה אשר מגדילה את ערכו של `i` להיות 3. `consistent` מחזירה את הערך `true` ולכן: (1) בתא `board[1][3]` מוצבת מלכה, (2) נערכת קריאה רקורסיבית. נדון בקריאה זאת, ונזכור שכתובת החזרה שלה, כרגיל, היא הפקודה (+).

ה. נצייר את מצב המחסנית בעקבות בניית רשומת ההפעלה של העותק השלישי של הפונקציה, העותק שנקרא מסעיף ד'. אני מסב את תשומת לבכם כי, טרם הקריאה, בַּעותק השני `i` גדל להיות שלוש, וכי המלכה הוזזה לתא מספר שלוש בשורה השניה.



אחרי בניית רשומת ההפעלה מתחיל ביצוע הפונקציה. עת i שווה אפס לא ניתן להציב מלכה, אך בתא board[2][1] ניתן להציב מלכה. לפיכך: (1) מלכה מוצבת בתא הנ"ל, (2) נערכת קריאה רקורסיבית אשר אמורה להציב את המלכה האחרונה. נדון בקריאה זאת.

ו. נציג את מצב המחסנית (נשים לב למלכה השלישית שהוצבה על הלוח, ולערכו של i בעותק השלישי, שני עדכונים שבוצעו על-ידי העותק השלישי של הפונקציה):

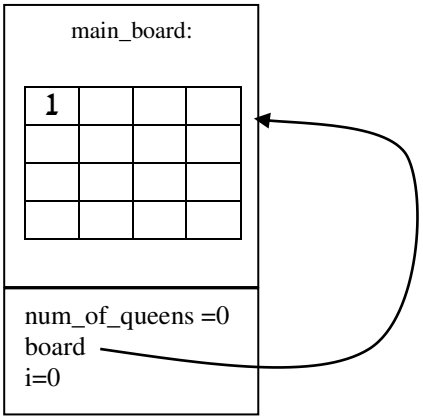


העותק הרביעי מתחיל להתבצע. לדאבוננו הוא אינו מצליח להציב מלכה באף אחד מארבעת התאים שבשורה הרביעית. העותק הרביעי מסיים. לאן אנו שבים? לעותק הקודם. ומה עלינו לעשות בו? לנסות להזיז את המלכה השלישית למקום אחר, ואז לחזור ולנסות להציב את המלכה הרביעית.

ז. מטעמי קיצור, לא נציג את מצב המחסנית אחרי הסרת רשומת ההפעלה של העותק הרביעי. המחסנית נראית בדיוק כפי שהיא מוצגת בשרטוט האחרון, פרט לרשומת ההפעלה הרביעית שכבר אינה עליה. אם נבחן את ערכו של i בעותק השלישי נגלה שערכו 1. עם החזרה אנו פונים לפקודה (+) אשר מנקה את המלכה שבתא board[2][1]. אחר אנו ממשיכים להתגלגל בלולאת ה- for אשר מריצה את i. i גדל להיות 2, ולא ניתן להציב את המלכה השלישית בתא board[2][2], i גדל להיות 3, ושוב לא ניתן להציב מלכה.

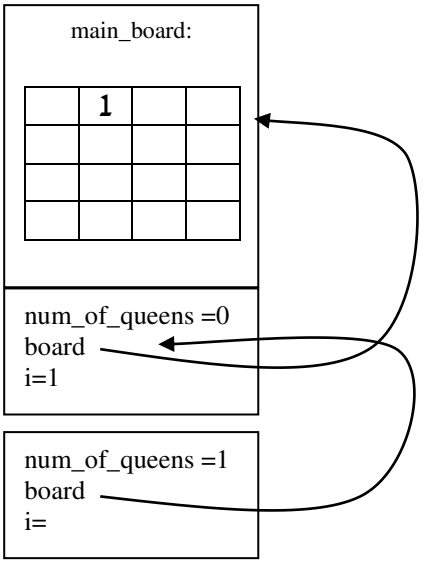
נבחן בדקדקנות את מצבנו: העותק השלישי של הפונקציה הסתיים. הוא מוסר מעל המחסנית, ואנו שבים לעותק השני של הפונקציה. אם נבחן את ערכו של i בעותק השני נגלה שהוא 3, ולכן עלינו לנקות את המלכה שבתא

borad[1][3]. בזאת גמרנו לרוץ בלולאת ה- for של העותק השני של הפונקציה, כלומר גמרנו גם את ביצוע העותק השני של הפונקציה. גם רשומת ההפעלה של העותק השני של הפונקציה מוסרת מעל המחסנית, ואנו שבים לעותק הראשון. נבחן את מצב המחסנית:



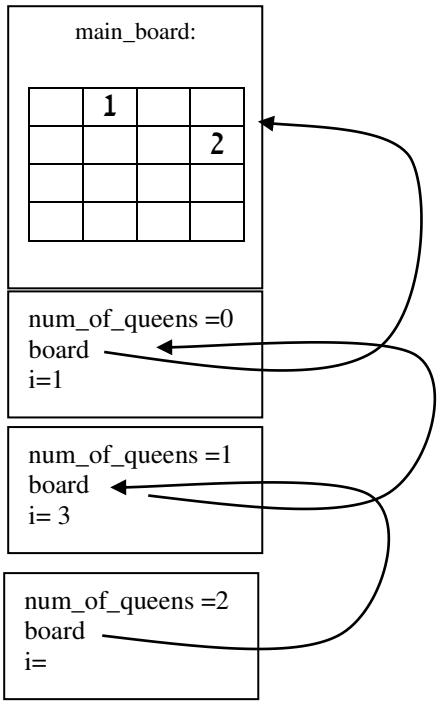
אנו רואים כי המלכות שהציבו הקריאות השניה והשלישית נוקו על-ידי קריאות אלה טרם שהן הסתיימו. ערכו של i בקריאה הראשונה הוא אפס. למעשה מה שהתכנית בדקה, וגילתה עד כה, הוא מה שאתם אולי כבר ידעתם, והוא שלא ניתן להגיע לפתרון עת המלכה הראשונה מוצבת בפינה השמאלית העליונה של הלוח. נמשיך להתגלגל בלולאה שמריצה הקריאה הראשונה. בפרט נגדיל את i להיות 1. כמובן ש- consistent תחזיר את הערך true, ולכן: (1) המלכה מספר 1 תוצב בתא board[0][1], (2) תיערך קריאה רקורסיבית. נדון בקריאה זאת.

ח. אנו דנים בקריאה הרקורסיבית שזומנה מסעיף ז'. נציג את מצב המחסנית אחרי הוספת רשומת ההפעלה של הפונקציה:



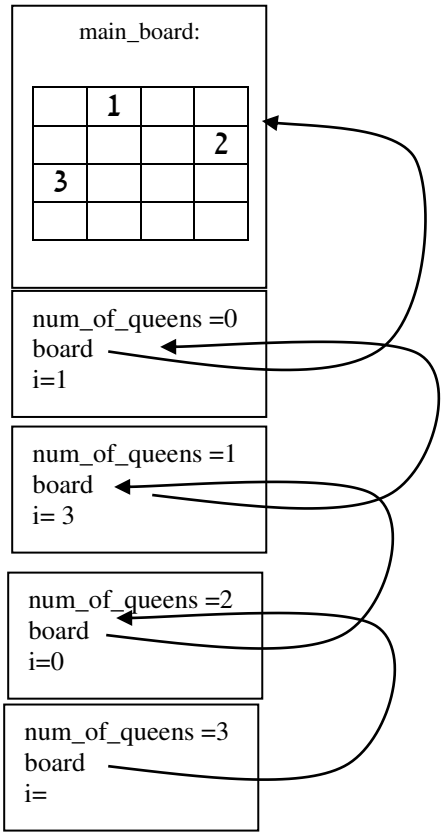
עתה הפונקציה מתחילה להתבצע. עת i=0 לא ניתן להציב מלכה בתא המתאים (board[1][0]), כך גם עת i=1 ועת i=2. אך בסיבוב הרביעי והאחרון של הפונקציה (בו i=3) מחזירה לנו consistent את הערך true, ולכן: (1) בתא board[1][3] מוצבת מלכה. (2) נערכת קריאה רקורסיבית לשם המשך תהליך הצבת המלכות.

ט. נדון בקריאה הרקורסיבית שזומנה מסעיף ח'. נציג את מצב המחסנית אחרי הוספת רשומת ההפעלה של הפונקציה:



עתה הפונקציה מתחילה להתבצע. כבר עת `i=0` ניתן להציב מלכה בתא המתאים, ולכן (1) בתא `board[2][3]` מוצבת מלכה. (2) נערכת קריאה רקורסיבית לשם המשך תהליך הצבת המלכות.

י. נדון בקריאה הרקורסיבית שזומנה מסעיף ט׳. נציג את מצב המחסנית אחרי הוספת רשומת ההפעלה של הפונקציה:



עתה הפונקציה מתחילה להתבצע. בַּתאים `0,1,2` בשורה השישית (והאחרונה) לא ניתן להציב מלכה; אך עת `i=3` מחזירה לנו `consistent` את הערך `true`. בששון ובשמחה אנו מציבים את המלכה הרביעית על הלוח, ומכיוון שזו המלכה האחרונה אנו ממהרים לקרוא ל- `print` אשר מציגה לנו את מצב הלוח. עת `print` חוזרת אנו, ראשית, מוחקים את המלכה שהצבנו. מכיוון שבמקרה יצא ש- `i=3` אזי איננו חוזרים לסיבוב נוסף בַּלולאה. העותק הנוכחי של הפונקציה מסתיים. ולאן פנינו מועדות? לעותק מסעיף ט׳. בַּעותק מסעיף ט׳ ערכו של `i` הוא אפס, על-כן עותק זה ימשיך להתגלגל בַּלולאה שלו. אנו נפרד ממנו בשלב זה לשלום, שכן יש גבול לכל תעלול.

עד כאן הצגנו סימולציית הרצה של בעיית שמונה המלכות אשר הציגה את כל הפתרונות האפשריים. באופן פרדוקסלי לכאורה, פתירת בעיית שמונה המלכות עת נדרש פתרון יחיד היא משימה יותר מורכבת. לתלמידים רבים נראה בַּתחילה שכל שיש לעשות הוא להוסיף לפונקציה שכתבנו, מייד אחרי הקריאה ל- `print`, פקודת `return`. זה פתרון שגוי; ומייד נסביר למה. נבחן למשל את דוגמת ההרצה שהצגנו; ונניח כי הכנסנו את השיפור המוצע. לכן, העותק מסעיף י׳, מייד אחרי הצגת מצב הלוח, מסיים. הביצוע חוזר לעותק מסעיף ט׳. כפי שציינו, בעותק מסעיף ט׳ ערכו של `i` הוא אפס. עתה העותק מסעיף ט׳ צריך לשאול את עצמו: מדוע ובאילו נסיבות הסתיים העותק מסעיף י׳--- האם העותק מסעיף י׳ הסתיים אחרי שהוא הציג פתרון, ולכן אני (העותק מסעיף ט׳) צריך לסיים כבר עכשיו? או שמא העותק מסעיף י׳ הסתיים אחרי שהוא נכשל בהצבת מלכה נוספת, ולכן אני (העותק מסעיף ט׳) צריך לנסות להזיז את המלכה שלי מעט, ולשוב לקרוא רקורסיבית?

לעותק מסעיף ט' אין כל דרך לענות על השאלה. הליקוי שגילינו גם רומז על הדרך הנכונה לבצע את המשימה:

א. עת עותק כלשהו של הפונקציה מסיים עליו להחזיר ערך המעיד האם הוא סיים בהצלחה, כלומר האם הוא או הקריאות הרקורסיביות שנולדו ממנו ישירות ובעקיפין הצליחו למלא את הלוח כנדרש, או לא.

ב. עת עותק כלשהו של הפונקציה מחזיר ערך (לעותק שקרא לו), על העותק הקורא לבדוק האם הוחזר איתות על הצלחה בהשלמת המשימה, או הוחזר איתות על חוסר הצלחה. אם הוחזר איתות על הצלחה אזי העותק הנוכחי (אליו חזר הערך) מסיים מיידית, תוך שגם הוא מחזיר איתות על הצלחה למי שקרא לו. מנגד, אם הוחזר איתות על כשלון ימשיך העותק הנוכחי בהרצת הלולאה שלו.

ג. אם העותק הנוכחי השלים את הרצת הלולאה שלו בלי שהוא חזר קודם לכן עם איתות על הצלחה, אזי על העותק הנוכחי להחזיר איתות על כשלון.

אני משאיר לכם כתרגיל לממש גרסה זאת של הבעיה.

8.4.1 <u>זמן הריצה האלגוריתם</u>

עתה נשאל את עצמנו את השאלה הבלתי נמנעת מהו זמן הריצה של האלגוריתם שכתבנו (עבור לוח בגודל `N x N`)?

א. העותק של הפונקציה אשר אמור להציב את המלכה האחרונה כולל לולאה יחידה אשר עוברת על `N` תאי השורה האחרונה במערך.

ב. העותק האחרון עשוי להיקרא `N` פעמים על-ידי העותק שקדם לו, (זה שאמור להציב את המלכה בשורה האחת לפני אחרונה). לכן ריצת שני העותקים האחרונים תחייב עובדה בשיעור N^2.

ג. העותק האחד לפני אחרון עשוי להיקרא `N` פעמים על-ידי העותק שקדם לו, (זה שאמור להציב את המלכה בשורה ה-'שתיים לפני אחרונה'). לכן ריצת שלושת העותקים האחרונים תחייב עובדה בשיעור N^3.

ד. באופן דומה ריצת `N` העותקים של הפונקציה, אשר אמורים להציב `N` מלכות, תחייב עבודה בשיעור N^N.

ניתן להסביר את זמן הריצה באופן דומה אך שונה:

א. העותק הראשון של הפונקציה קורא `N` פעמים לפונקציה אשר צריכה להציב `N-1` מלכות. לכן כמות העבודה שתתבצע סה"כ תהיה `N*queens(N-1)`, כאשר הביטוי `queens(N-1)` מציין את כמות העבודה הנעשית על ידי הפונקציה עת עליה להציב את `N-1` המלכות האחרונות.

ב. מה ערכו של `queens(N-1)`? כדי להשלים את המשימה קוראת פונקציה זאת לכל היותר `N` פעמים ל- `queens(N-2)`. לכן כמות העבודה המתבצעת לשם השלמת המשימה של הצבת `N-1` מלכות היא `N*queens(N-2)`.

ג. מה ערכו של `queens(N-2)`? כדי להשלים את המשימה קוראת פונקציה זאת לכל היותר `N` פעמים ל- `queens(N-3)`. לכן כמות העבודה המתבצעת לשם השלמת המשימה של הצבת `N-2` מלכות היא `N*queens(N-3)`.

ד. וכך הלאה. עד אשר הצבת המלכה האחרונה דורשת ביצוע לולאה העוברת על `N` תאי המערך.

ה. אם נציב את הביטויים שקיבלנו זה בזה נקבל: `queens(N) = N*queens(N-1) = N*N*queens(N-2)=N*N*N*queens(N-3)`, כלומר בכל ביטוי שאנו מוסיפים נוסף כופל נוסף שערכו `N`, ומצד שני הערך שבַּסוגריים קטן באחד. וזאת עד מתי? עד שבסוגריים ימצא `queens(1)`, כלומר כמות העבודה הנדרשת לשם הצבת מלכה אחת; וביטוי זה קל להעריך: ערכו הוא `N`.

אם נסכם אזי בסופו של התהליך יהיו לנו N-1 כופלים שערכם N, ועוד כופל נוסף שגם ערכו N, ולכן ערכו של הביטוי יהיה N^N.

אנו זוכרים כי עת כמות העבודה הנעשית הינה בשיעור מעריכי (כלומר עת N מופיע במעריך) אנו אומרים כי במקרה הכללי (עבור N גדול דיו) הבעיה אינה פתירה.

לסיום נאמר כי גם לבעיית שמונה המלכות קיים פתרון לא רקורסיבי. פתרון זה מותאם לערכו של N, במובן זה שאם השתנות ערכו של N יש לשנות (במעט) גם את האלגוריתם, (תכונה מאוד לא רצויה, כמובן). נציג אותו בקצרה ובקווים כללים עבור לוח בגודל NxN. מבנה הנתונים בו נשתמש יהיה מערך חד-ממדי בן N תאים בשם place. תא מספר i במערך יציין באיזה עמודה מצויה המלכה שהוצבה בשורה מספר i במערך. לכן כל תא במערך יוכל להכיל ערך שבין אפס ל- N-1 (אנו מציינים את שורות ועמודות הלוח במספרים 0..N-1). ניתן להתייחס למערך גם כמכיל מספר בן N ספרות בבסיס N. לדוגמה אם N=4 ובמערך מצויים הערכים:

2	0	3	1
3	2	1	0

אזי ניתן לחשוב על המערך כמכיל את המספר 2031 (שהוא מספר בן ארבע ספרות בבסיס ארבע). שימו לב כי ציירנו הפעם את המערך הפוך מדרכנו בדרך-כלל: התא מספר אפס מופיע עתה מימין, והתא האחרון (מספר שלוש) מופיע משמאל.

המערך הנ"ל מורה לנו כי המלכה בשורה אפס נמצאת בעמודה מספר 1, המלכה בשורה 1 מצויה בעמודה 3, וכן הלאה. (זהו המערך שהכיל את הפתרון שהצגנו בסימולציית ההרצה שערכנו).

על האלגוריתם הלא רקורסיבי לפתרון בעיית שמונה המלכות למנות את כל המספרים בני N ספרות בבסיס N. (האלגוריתם יעשה זאת תוך שימוש ב- N לולאות; ודבר זה יחייב אותנו לשנות את התכנית עת N משתנה). עבור כל מספר שהאלגוריתם מייצר עליו לבדוק (תוך שימוש בפונקציה כדוגמת consistent שראינו) האם המספר מציין פתרון של הבעיה.

8.5 איתור מסלול יציאה ממבוך

איתור מסלול יציאה ממבוך היא בעיה נוספת המצריכה פתרון הנוקט בעקיבה לאחור. מבוך הוא מערך דו-ממדי שכל תא בו עשוי להיות: (א) קיר (שייוצג על-ידי קבוע, למשל על-ידי הערך אפס), (ב) מקום שהוא חלק מנתיב היציאה הנסרק על-ידי פותר המבוך (ייוצג על-ידי הערך 2), (ג) מקום שהתגלה כמביא לאין מוצא (ייוצג על-ידי הערך 3), (ד) מקום שאינו קיר, אך שפותר המבוך טרם בדק האם הוא מוביל החוצה מהמבוך (ייוצג על-ידי 1). לדוגמה המערך הבא עשוי לייצג מבוך (בשוליים הוספנו את מספרי השורות והעמודות; כמו כן, לשם בהירות הציור השארנו ריקים את התאים בהם מצוי הערך 1):

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0		2	3	3	0
2	0	0	2	0	0	0
3	0		2			0
4	0		0	0		0
5	0			0	0	0
6	0					0
7	0	0	0	0		0

<u>ב</u>מבוך המתואר בציור יצא הפותר מהתא `lab[1][2]`. הוא ניסה לפנות ימינה ונכשל, ועתה הוא נמצא בתא `lab[3][2]`. פתח המבוך מצוי ב- `lab[7][4]`.

בבעיה שלנו נניח כי הפותר רשאי להתקדם בכל פעם מהתא בו הוא נמצא ימינה, שמאלה, מעלה, מטה, אך לא באלכסון. עוד נניח כי אם קיימים מספר מסלולי יציאה מהמבוך אזי אנו מעונינים באחד מהם (ולא משנה איזה).

האלגוריתם ליציאה מהמבוך מקבל את מצב המבוך, ואת התא בו נמצא הפותר. האלגוריתם פועל באופן הבא:

א. סמן כי התא בו אתה נמצא הוא חלק ממסלול ההתקדמות, (על-ידי הצבת ערך מתאים בתא).

ב. אם הגעת לפתחו של המבוך (כלומר לשורה מספר אפס, או לשורה מספר `ROWS-1` או לעמודה מספר אפס, או לעמודה מספר `COLS-1`) חזור תוך שאתה מאותת על פתרון. הפתרון מסומן בַּמבוך, ולכן מי שקרא לפונקציה יוכל להציגו.

ג. אם התא שמעליך אינו קיר (יש בו ערך שונה מאפס), וכן טרם ביקרת בו (הערך בתא הוא ערכו של תא שטרם נוסה) אזי קרא לעצמך רקורסיבית עם המבוך ועם מספר התא שמעליך.

ד. אם הקריאה הרקורסיבית מסעיף ב' החזירה הצלחה אזי סיים מיידית תוך שאתה מאותת על הצלחה.

ה. אם התא שמימינך אינו קיר (יש בו ערך שונה מאפס), וכן טרם ביקרת בו, אזי קרא לעצמך רקורסיבית עם המבוך ועם מספר התא שמימינך.

ו. אם הקריאה הרקורסיבית מסעיף ד' החזירה הצלחה אזי סיים מיידית תוך שאתה מאותת על הצלחה.

ז. באופן דומה עבור כל אחד משני הכיוונים הנותרים.

ח. אם לא החזרת איתות על הצלחה עד כה אזי סמן את התא שלך כתא שמוביל ל-'אין מוצא' והחזר איתות על כישלון.

נציג את הפונקציה המתאימה בשפת C, ואחר נסבירה. קטע התכנית שנכתוב יסתמך על ההגדרות הבאות:

```
const int ROWS = …, COLS = …    // size of labirint
const bool SUCCESS = true, FAILURE = false ;
                                // vals lab[][] may hold
enum lab_vals = {WALL, NOT_TRIED, PART_OF_PATH, TRIED} ;
```

הקריאה לפונקציה תיעשה עם מערך שכולל את תיאור קירות המבוך (בתאים אלה ימצא הערך WALL); יתר תאי המערך יכילו את הערך NOT_TRIED. כמו כן יועברו לפונקציה האינדקסים של התא בו נמצא הפותר בתחילת התהליך. לדוגמה: כדי לפתור את המבוך שהצגנו בציור יש לקרוא לפונקציה באופן הבא: success = run_away(lab, 1, 2);. נסביר את הקריאה: המערך lab הוא שמכיל את המבוך. בתחילת התהליך נמצא הפותר בתא lab[1][2]. הפונקציה run_away תחזיר ערך שיורה האם המבוך נפתר; ערך זה יוכנס למשתנה success.

הגדרת הפונקציה:

```
bool run_away(int lab[ROWS][COLS], int row, int col)
{
   lab[row][col] = PART_OF_PATH ;
   if (row == 0 || row == ROWS-1 ||
       col == 0 || col == COLS -1)
      return SUCCESS ;

   if (lab[row -1][col] == NOT_TRIED)
   {
      success = run_away(lab, row-1, col) ;
      if (success) return SUCCESS;            //(1)
   }

   if (lab[row][col +1] == NOT_TRIED)
   {
      success = run_away(lab, row, col+1) ;
      if (success) return SUCCESS; ;          //(2)
   }

   if (lab[row +1][col] == NOT_TRIED)
   {
      success = run_away(lab, row+1, col) ;
      if (success) return SUCCESS; ;          //(3)
   }

   if (lab[row][col-1] == NOT_TRIED)
   {
      success = run_away(lab, row, col-1) ;
      if (success) return SUCCESS; ;          //(4)
   }
   lab[row][col] = TRIED ;
```

```
   reteun FAILURE ;
}
```

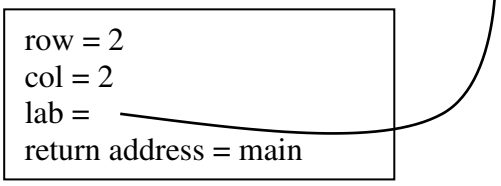
נציג דוגמת הרצה של הפונקציה על המבוך הבא:

	0	1	2	3	4
0	0	0	0	0	0
1	0			0	0
2	0	0		0	0
3	0				0
4	0		0	0	0

וננית כי בתחילה נמצא הפותר בתא `lab[2][2]`.

א. בקריאה הראשונה לפונקציה `row =2, col = 2` נציג את רשומת ההפעלה:

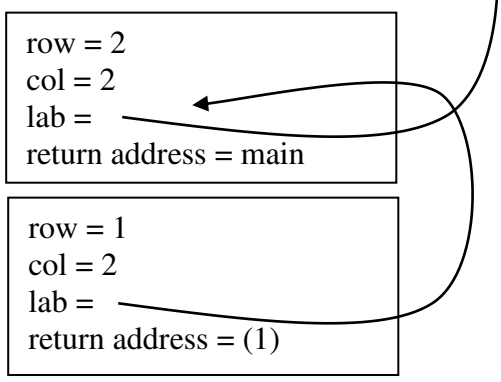
	0	1	2	3	4
0	0	0	0	0	0
1	0			0	0
2	0	0		0	0
3	0				0
4	0		0	0	0



הפונקציה מתחילה להתבצע. בתא `lab[2][2]` היא מציבה את הערך `PART_OF_PATH` שאנו נצייר בהמשך באות `P`. עתה היא בודקת האם התא שמעליה טרם נוסה. הוא אכן טרם נוסה ולכן הפונקציה קוראת רקורסיבית עם התא `lab[1][2]`. נדון בקריאה זאת.

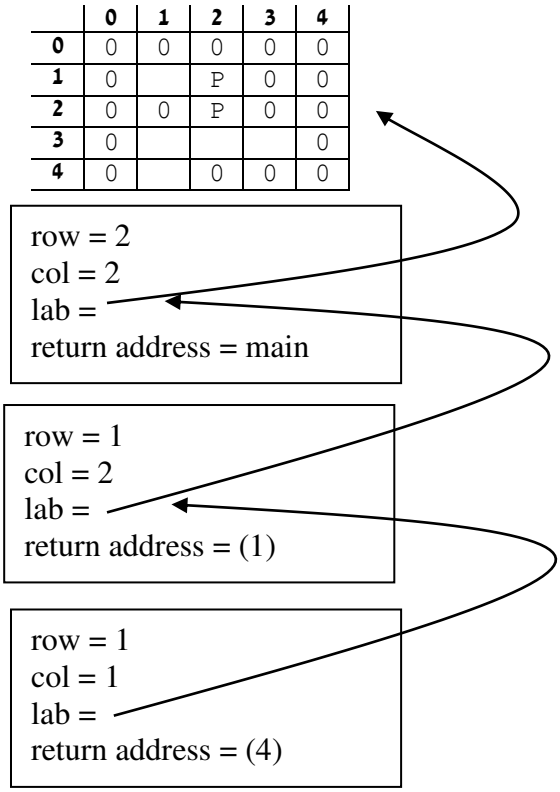
ב. נדון בקריאה שזומנה מסעיף א'. כתובת החזרה שלה היא פקודת התנאי (1) בעותק מסעיף א'. ערכי הפרמטרים שלה הם: `row =1, col =2`. נציג את מצב המחסנית:

	0	1	2	3	4
0	0	0	0	0	0
1	0			0	0
2	0	0	P	0	0
3	0				0
4	0		0	0	0



הפונקציה מתחילה להתבצע: בתא lab[1][2] מושם הערך P. הפונקציה בודקת האם ניתן להתקדם אל התא שמעל התא lab[2][1] ומגלה שלא ניתן (אין בו ערך NOT_TRIED, יש בו ערך אפס). היא בודקת האם ניתן להתקדם אל התא שמימין לתא lab[2][1] ומגלה שלא ניתן. היא בודקת האם ניתן להתקדם אל התא שמתחת לתא lab[2][1] ומגלה שלא ניתן (אין בו ערך NOT_TRIED אלא ערך P). היא בודקת האם ניתן להתקדם אל התא שמשמאל לתא lab[2][1] ומגלה שניתן להתקדם לתא זה. על כן העותק ב' מזמן קריאה רקורסיבית עם התא lab[1][1]. נדון בקריאה זאת.

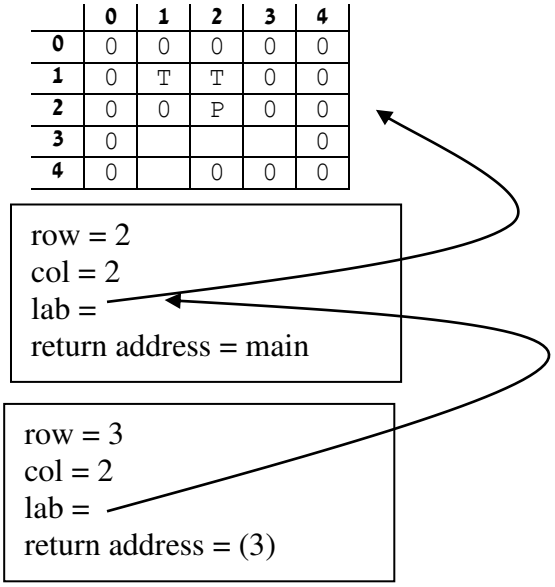
ג. נדון בקריאה שזומנה מסעיף ב'. כתובת החזרה שלה היא פקודת התנאי (4) בעותק מסעיף ב'. ערכי הפרמטרים שלה הם: row =1, col =1. נציג את מצב המחסנית:



הפונקציה מתחילה להתבצע: בתא lab[1][1] מושם הערך P. עבור כל אחד מארבעת כיווני ההתקדמות הפוטנציאליים הפונקציה מגלה כי היא לא יכולה להתקדם באותו כיוון. על כן הפונקציה אינה קוראת רקורסיבית לשום פונקציה אחרת, והיא מתדרדרת ישירות לפקודה ; lab[row][col] = TRIED ומשם לפקודת החזרת הערך אשר מחזירה כישלון: מהתא lab[1][1] לא ניתן להמשיך ולהתקדם החוצה מהמבוך. לאן חוזר העותק הנוכחי של הפונקציה? לפקודת התנאי (4) בסעיף ב'. התנאי בַּפקודה if (success) אינו מתקיים, ולכן העותק ב' אינו יכול להחזיר את הערך SUCCESS. העותק ב' ממשיך לפקודה lab[1][2]= TRIED, ואחר גם הוא מחזיר איתות על כשלון. לאן ממשיך ביצוע התכנית? לפקודת התנאי (1) בעותק א'. התנאי שבפקודה אינו מתקיים, ולכן העותק א' אינו יכול לחזור עם איתות על הצלחה. אולם עוד לא אבדה תקוותו של העותק א', עד כה הוא בסך הכל בדק האם ניתן לפתור את המבוך על-ידי התקדמות כלפי מעלה, והוא גילה שלא ניתן. עומדים לפניו עוד שלושה כיוונים נוספים לבדיקה, ואולי מאחד מהם יבוא עזרו. לכן עתה העותק א' ממשיך לבדיקת התנאי: if (lab[row][col+1]==NOT_TRIED),

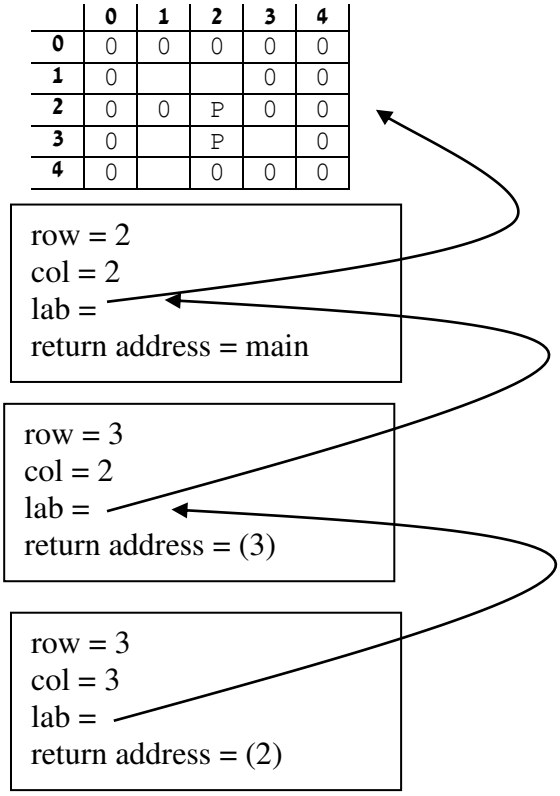
לדאבונו תנאי זה אינו מתקיים. עותק א, ממשיך לתנאי: `if (lab[row+1][col]==NOT_TRIED)`, ותנאי זה מתקיים. לכן עותק א' מזמן קריאה רקורסיבית עם האינדקסים: `row = 3, col =2`. כתובת החזרה של הקריאה תהיה פקודת התנאי (3).

ד. נדון עתה בקריאה הנ"ל: בקריאה זאת `row =3, col = 2`, וכתובת החזרה שלה היא פקודת התנאי (3) בעותק מסעיף א'. נציג את מצב המחסנית (אני מסב את תשומת לבכם שה- `P` שהיו בתאים `lab[1][1], lab[1][2]` הוחלפו בערך `TRIED` טרם שהקריאות הרקורסיביות שהציבו אותם סיימו).



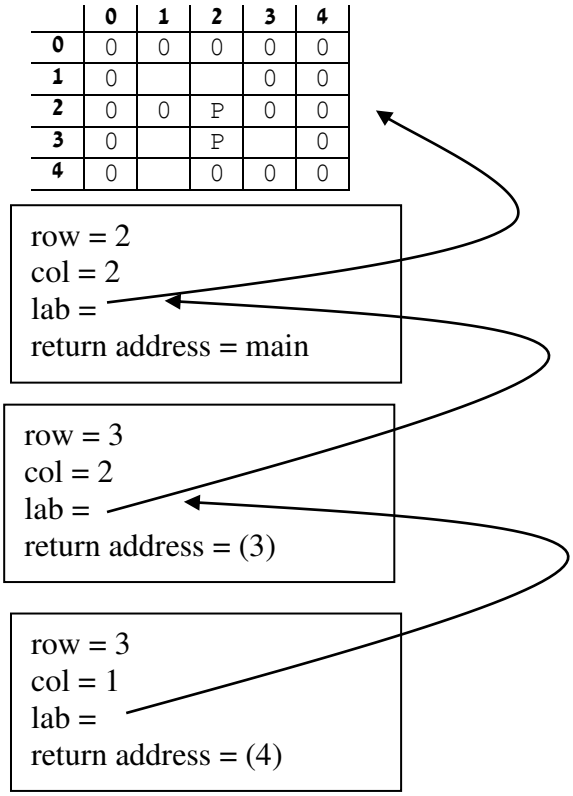
הפונקציה מתחילה להתבצע: ראשית היא מציבה ערך בתא `lab[3][2]`. אחר היא מגלה שאין ביכולתה להתקדם כלפי מעלה, אך היא יכולה להתקדם ימינה לתא `lab[3][3]`, ולכן העותק הנוכחי של הפונקציה מזמן קריאה רקורסיבית עבור התא `3,3`. נדון בקריאה זאת.

ה. בקריאה הנוכחית: `row = 3, col =3`. כתובת החזרה היא הפקודה (2) בעותק מסעיף ד'. נציג את מצב המחסנית:



הפונקציה מתחילה להתבצע: ראשית היא מציבה ערך בתא `lab[3][3]`. אחר היא מגלה שאין ביכולתה להתקדם לאף לא אחד מארבעת הכיוונים הפוטנציאליים. הפונקציה מציבה את הערך `TRIED` בתא `lab[3][3]` ומסיימת, תוך שהיא מחזירה איתות על כשלון. אנו שבים לפקודה (2) בעותק מסעיף ד׳. לַעותק מסעיף ד׳ חזר איתות על כשלון. על כן הוא אינו מחזיר בעצמו איתות על הצלחה. העותק מסעיף ד׳ ממשיך בפעולתו. הוא מגלה שאין ביכולתו להתקדם כלפי מטה (מהתא `lab[3][2]`), אך יש ביכולתו להתקדם שמאלה. על כן העותק מסעיף ד׳ מזמן קריאה רקורסיבית עם התא בשורה #3, עמודה #1. נדון בקריאה זאת.

ו. בקריאה בה אנו דנים עתה: `row =3, col =1`, כתובת החזרה היא הפקודה (4) בעותק מסעיף ד׳. נציג את מצב המחסנית:



הפונקציה מתחילה להתבצע: ראשית היא מציבה ערך בתא `lab[3][1]`. אחר היא מגלה שאין ביכולתה להתקדם מעלה או ימינה, אך היא יכולה להתקדם כלפי מטה. לכן הקריאה הנוכחית מזמנת קריאה עבור התא שבשורה #4, עמודה #1. נדון בקריאה זאת.

ז. אנו דנים בקריאה בה: `row =4, col =1`, כתובת החזרה היא הפקודה (3) בעותק מסעיף ו'. הקריאה מציבה את הערך `P` בתא `lab[4][1]`. אחר היא מגלה שהיא הגיעה לשולי המבוך ולכן היא מסיימת את ריצתה תוך שהיא מחזירה איתות על הצלחה. האיתות חוזר לעותק מסעיף ו', לפקודת התנאי (3). התנאי בפקודה מתקיים ולכן גם העותק מסעיף ו' מסיים, ומחזיר איתות על הצלחה. העותק מסעיף ו' נקרא על-ידי העותק מסעיף ד', ולכן אנו שבים לעותק מסעיף ד' (לפקודת התנאי (4)). עתה העותק מסעיף ד' מסיים ומחזיר איתות על הצלחה. העותק מסעיף ד' נקרא על-ידי העותק מסעיף א' אשר מסיים ומחזיר לתכנית הראשית איתות על הצלחה. עתה תוכל התכנית הראשית להדפיס את המבוך הפתור. (אגב, סביר שהתכנית הראשית לא תרצה להציג מסלולים שנוסו לשווא, ולכן סביר שתאים בהם מצוי הערך `TRIED` יוצגו באופן דומה לתאים בהם מצוי הערך `NOT_TRIED`).

אחרי שהצגנו את האלגוריתם, והבנו אותו, עלינו להתמודד עם השאלה מהו זמן הריצה של האלגוריתם? אשאיר לכם לענות בכוחות עצמכם על השאלה...

8.6 חלוקת קבוצה לשתי תת-קבוצות שקולות

נסיים את הדיון באלגוריתמים של עקיבה לאחור בדוגמה הבאה: נניח כי בתכנית הוגדר המערך: `unsigned int arr[N];` , ולמערך הוזנו `N` ערכים טבעיים. המשימה הניצבת בפנינו עתה היא לחלק את הנתונים המצויים במערך לשתי קבוצות, כך שסכום הנתונים בשתי הקבוצות יהיה זהה. לדוגמה, אם המערך `aar` כולל את הנתונים הבאים: `{5, 2, 2, 9, 6, 17, 1, 8}` אזי החלוקה הבאה תתקבל כפתרון: `{5, 2, 9, 1, 8}, {2, 6, 17}`, שכן סכום האיברים בכל קבוצה הוא 25. מטבע הדברים ייתכנו מצבים בהם הבעיה אינה ניתנת לפתרון (לא קיימת חלוקה כמבוקש), וקיימים מצבים בהם קיימים מספר פתרונות. הפונקציה שנכתוב תחזיר איתות האם היא מצאה פתרון, ובמידה וכן היא תחזיר פתרון כלשהו.

הפונקציה שנכתוב תקבל את הפרמטרים הבאים:

א. `unsigned int list[]` הוא המערך שמכיל את כלל הנתונים.

ב. `unsigned int alloc` מציין כמה נתונים מהמערך המקורי כבר חולקו לשתי הקבוצות (לא משנה לאיזו מבניהן). בתחילת התהליך הוקצו אפס נתונים, בסופו הוקצו כל `N` הנתונים.

ג. `unsigned int list1[]` הוא המערך שמכיל את הנתונים שהוקצו לקבוצה הראשונה.

ד. `unsigned int alloc1` מציין כמה נתונים מהמערך המקורי הוקצו לקבוצה הראשונה.

ה. `unsigned int list2[]` הוא המערך שמכיל את הנתונים שהוקצו לקבוצה השניה.

ו. `unsigned int alloc2` מציין כמה נתונים מהמערך המקורי הוקצו לקבוצה השניה.

ז. `half` יציין את סכום הערכים שכל קבוצה אמורה להכיל. (מין הסתם, הוא יחושב על-ידי הפונקציה הקוראת טרם הקריאה לפונקציה שלנו).

האלגוריתם על-פיו פועלת הפונקציה:

א. הקצה את הערך מספר `alloc` לקבוצה הראשונה (תוך עדכון הפרמטרים המתאימים).

ב. אם הקצת את הנתון האחרון, אזי אם סכום הערכים בקבוצה הראשונה שווה ל- `half` אזי החזר הצלחה.

ג. אם סכום הערכים בקבוצה הראשונה קטן או שווה מ- `half` וכן יש נתונים נוספים להקצות אזי קרא רקורסיבית לאלגוריתם. אם הקריאה הרקורסיבית מחזירה איתות על הצלחה אזי חזור תוך שאתה מחזיר איתות על הצלחה.

ד. העבר את הערך מספר `alloc` לקבוצה השניה (תוך עדכון הפרמטרים המתאימים).

ה. אם בסעיף ד' העברת את הנתון האחרון, אזי אם סכום הערכים בקבוצה השניה שווה ל- `half` אזי החזר הצלחה, אחרת אין מנוס מלהחזיר כשלון.

ו. (לסעיף זה נגיע אם בסעיף ד' לא הקצנו את הערך האחרון). אם סכום הערכים בקבוצה השניה קטן או שווה מ- `half` וכן יש נתונים נוספים להקצות אזי קרא רקורסיבית לאלגוריתם. אם הקריאה הרקורסיבית מחזירה איתות על הצלחה אזי חזור תוך שאתה מחזיר איתות על הצלחה.

ז. החזר איתות על כשלון.

נציג את הגדרת הפונקציה:

```
bool split(unsigned int list[],unsigned int alloc,
           unsigned int list1[],unsigned int alloc1,
           unsigned int list2[],unsigned int alloc2,
           unsigned int half)
{  unsigned int curr_sum ;

   list1[ alloc1++ ] = list[ alloc++ ] ;
   curr_sum = sum(list1, alloc1) ;

   if (alloc == N && curr_sum == half)
      return( SUCCESS ) ;

   if (curr_sum <= half && alloc < N)
   {
      bool success = split(list, alloc,
                list1, alloc1, list2, alloc2, hlaf ) ;
      if (success=SUCCESS) return( SUCCESS ) ;     // (-)
   }

   list2[ alloc2++ ] = list1[  alloc1-- ] ;
   curr_sum = sum(list2, alloc2) ;

   if (alloc == N)
      return( (curr_sum == half)? SUCCESS : FAILURE ) ;

   if (curr_sum <= half && alloc < N)
         return(  split(list, alloc,
                        list1, alloc1,
                        list2, alloc2, hlaf ) );   // (+)
}
```

הפונקציה שכתבנו משתמשת בשרותי הפונקציה `unsigned int sum(unsigned int list[], unsigned int num)`. הפונקציה מחזירה את סכומם של `num` הנתונים הראשונים במערך `list` (כלומר את סכומם של הנתונים הנמצאים בתאים `0..num-1`).

כדרכנו, נבצע סימולציית הרצה:

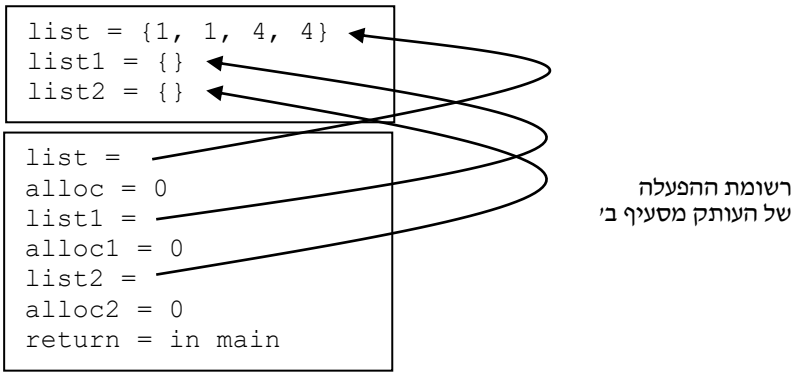
נניח כי בתכנית הוגדר הקבוע `N=4`, הוגדרו המערכים: `unsigned int list[N]= {1, 1, 4, 4}, list1[N], list2[N] ;` והוגדר המשתנה: `bool success`.

הקריאה לפונקציה מהתכנית הראשית תהיה, לפיכך: `success = split(list, 0, list1, 0, list2, 0, 5)`.

א. נציג ראשית את מצב המחסנית טרם הקריאה לפונקציה:

```
list = {1, 1, 4, 4}
list1 = {}
list2 = {}
```

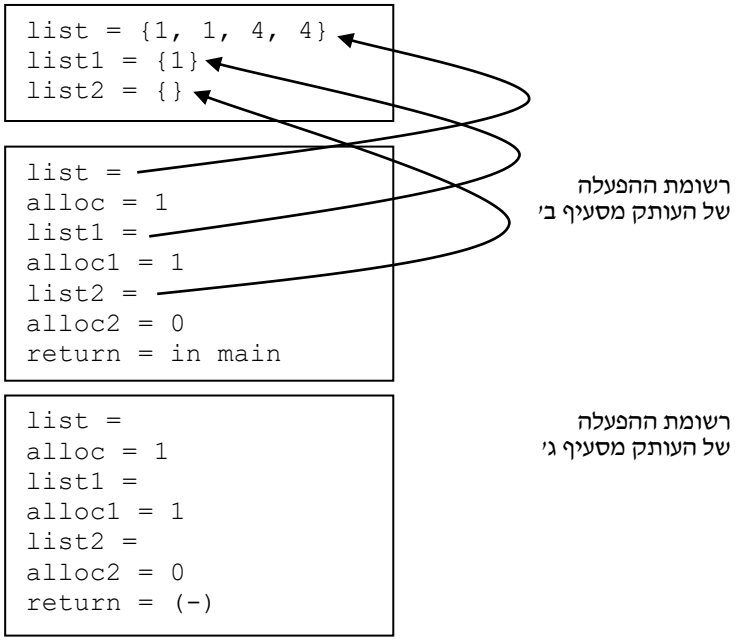
ב. מצב המחסנית עם הכניסה לעותק הראשון של הפונקציה:



לצורך הפשטות השמטנו מציור המחסנית מספר פרמטרים ומשתנים: (א) הפרמטר half, זהו פרמטר ערך שערכו בכל הקריאות הוא 5, ואין בו עניין מיוחד. (ב) המשתנה curr_sum שמחזיק את סכום הערכים בקבוצה לה הוקצה ערך לאחרונה. (ג) המשתנה הבולאני success לתוכו מוכנס הערך שחוזר מהקריאה הרקורסיבית המזומנת על-ידי הקריאה הנוכחית.

העותק ראשון מתחיל להתבצע: הוא מקצה את הערך שבתא מספר alloc במערך list, כלומר את הערך אחד שבתא מספר אפס, לקבוצה הראשונה תוך עדכון הפרמטרים המתאימים. סכום הערכים בקבוצה הראשונה הוא עתה 1, ולכן ערכו של curr_sum קטן מערכו של half, וכן לא הוקצה עדיין הערך האחרון מהקבוצה list, לכן אנו פונים לקריאה הרקורסיבית שכתובת החזרה של היא הפקודה (-) בעותק שבסעיף הנוכחי.

ג. נדון בקריאה שזומנה מסעיף ב'. נציג את מצב המחסנית בעקבות זימון הפונקציה:



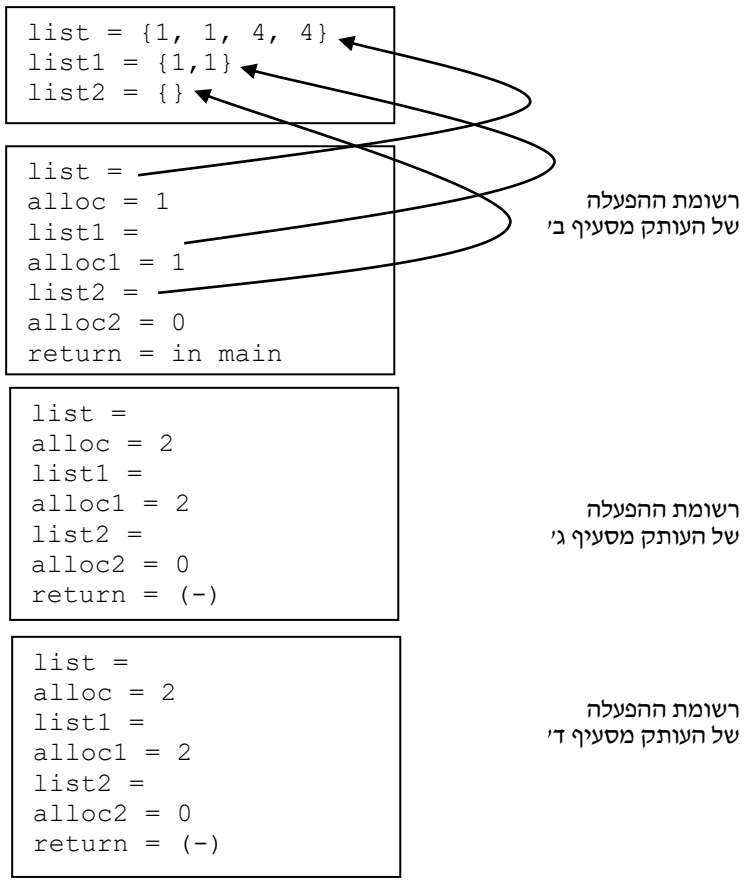
גם משרטוט זה השמטנו מספר פרטים לשם הבהירות: לא ציירנו את החיצים מהפרמטרים list, list1, list2 שבעותק השני של הפונקציה לארגומנטים list, list1, list2 שבעותק הראשון. שימו לב לערכם של הפרמטרים השונים: list1 = {1} שכן הערך אחד הוקצה (על-ידי הקריאה

הראשונה) לקבוצה הראשונה; בהתאמה `alloc = 1` (הוקצה ערך יחיד מהקבוצה `list`), וכן `alloc1 = 1` (הוקצה ערך יחיד לַקבוצה `list1`).

העותק השני מתחיל להתבצע. הוא מקצה את הערך `list[alloc]` כלומר את הערך אחד לקבוצה הראשונה (ותוך כדי כך הוא מעדכן: `alloc =2`, `alloc1 = 2`). סכום הערכים בקבוצה הראשונה הוא עתה שתיים.

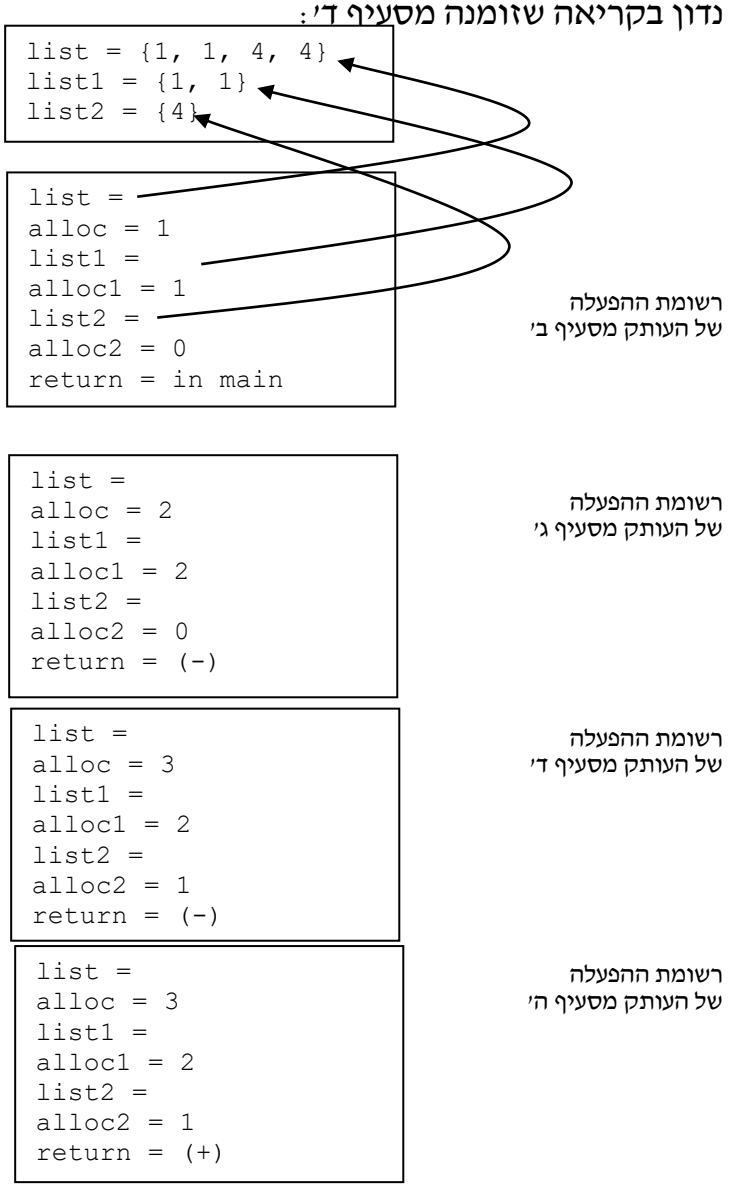
עדיין לא הוקצה הערך האחרון, וכן `curr_sum <= half`, ולכן מבוצעת קריאה רקורסיבית שכתובת החזרה שלה היא הפקודה (-).

ד. נדון בקריאה שזומנה מסעיף ג'. נציג את מצב המחסנית:



הפונקציה מתחילה להתבצע: היא מקצה את הערך מספר `alloc`, כלומר את הערך מספר שתיים לקבוצה הראשונה (תוך עדכון: `alloc = 3`, `alloc1 = 3`). סכום ערכים בקבוצה הראשונה (כפי ששמור ב- `curr_sum`) הוא עתה 6, כלומר הוא גדול מערכו של `half`. לכן הקריאה הנוכחית מנסה להעביר את הערך שהיא הקצתה (כלומר את הערך 4) לקבוצה השניה (תוך עדכון: `alloc1 = 2, alloc2 = 1`). עתה הפונקציה מזמנת קריאה רקורסיבית שכתובת החזרה שלה תהיה הפקודה (+) בעותק הנוכחי.

ה. נדון בקריאה שזומנה מסעיף ד׳:

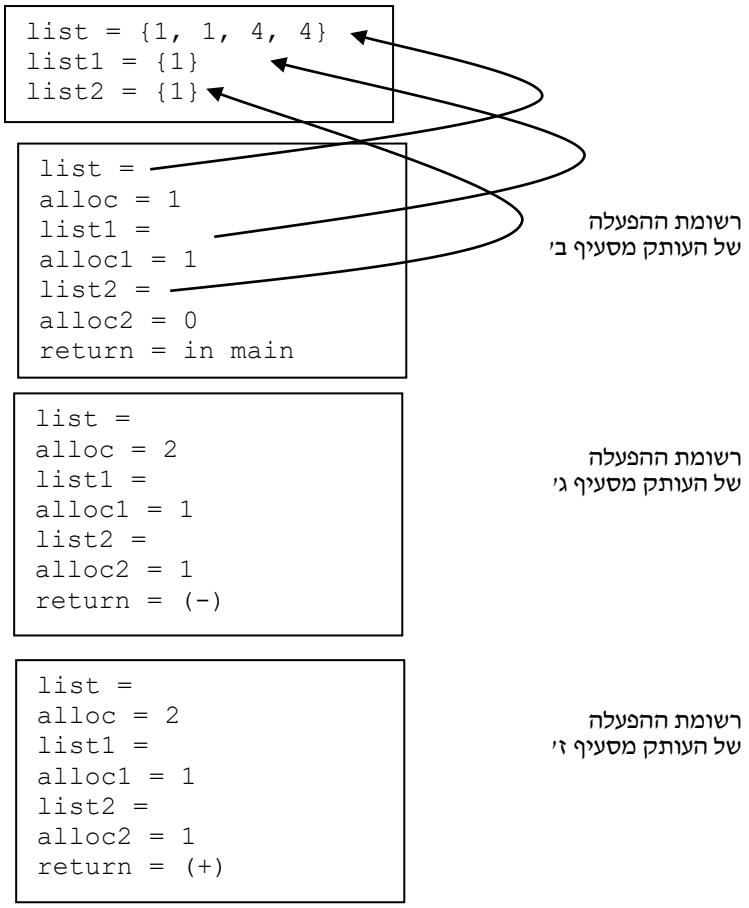


הקריאה הרקורסיבית הרביעית מתחילה להתבצע: היא מקצה את הערך `list[alloc]` כלומר את הערך 4 לקבוצה הראשונה. `curr_sum` מקבל את הערך שש, כלומר ערכו גדול מערכו של `half`. לכן עלינו להעביר את הערך אחד לקבוצה השניה (תוך עדכון: `alloc1 = 2, alloc2 = 2`). עתה ערכו של `curr_sum` הוא שמונה, וגם זה לא טוב. לכן הקריאה הנוכחית מסתיימת ומחזירה כשלון. על מי זורקים את האשמה לכישלון? על פקודה (+) בסעיף ד׳ אשר מחזירה גם היא כישלון. כישלון זה חוזר לפקודה (-) בקריאה שמסעיף ג׳. ברמת המשמעות גילינו כי אם נקצה את שני הערכים הראשונים לאותה קבוצה לא נוכל להגיע לפתרון; כלומר עלינו לנסות להקצות את הערך השני לקבוצה השניה, ולחזור ולנסות לבנות את הפתרון. נראה כיצד הדברים קורים.

ו. שַבנו לַקריאה מסעיף ג׳. קריאה זאת ניסתה להקצות את הערך אחד (מ-`list[1]`) ל-`list1`, עתה היא תנסה להקצותו ל-`list2`. חזרנו להשמה (-) ובה הוכנס הערך כשלון למשתנה `success` של העותק מסעיף ג׳. לכן עתה עותק זה מתקדם לפקודה אשר מעבירה את הערך (אחד מהתא השני של המערך) שהוקצה בעבר ל-`list1` ל-`list2`. עתה העותק מסעיף ג׳ מזמן קריאה רקורסיבית אשר תבדוק האם צורת הקצאה זאת (בה לכל קבוצה

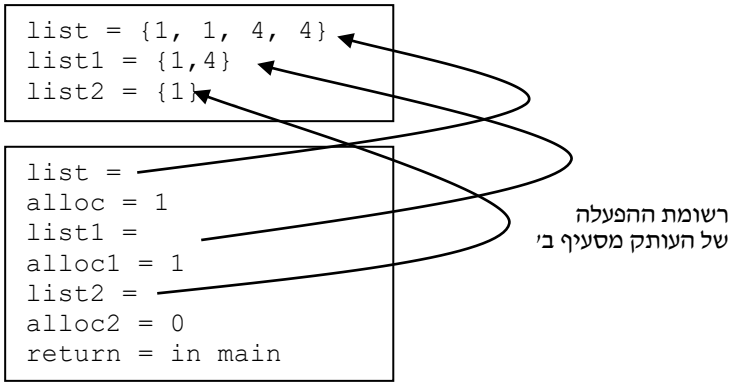
הוקצה ערך יחיד, הערך אחד) מאפשרת השלמת הפתרון. (להזכירכם העותק ג׳ עושה זאת אחרי שהוא גילה שהקצאת שני האחד-ים הראשונים לַקבוצה `list1` אינה מאפשרת השלמת הפתרון). נדון בקריאה המזומנת. כתובת החזרה שלה היא הפקודה (+) בעותק ג׳.

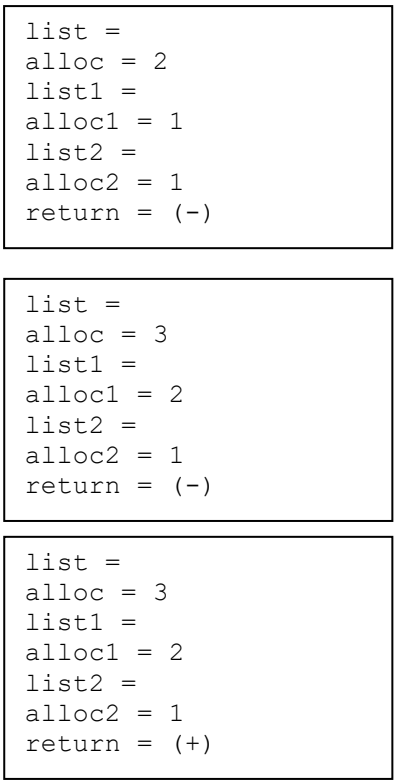
ז. נציג את מצב המחסנית:



העותק מתחיל להתבצע. הוא מקצה את הערך `list[alloc]`, כלומר את הערך 4, לקבוצה הראשונה. סכום האברים בקבוצה הראשונה הוא עתה 5, כלומר הוא שווה ל- `half`. הערך האחרון טרם הוקצה ולכן הקריאה הנוכחית מזמנת קריאה רקורסיבית נוספת. כתובת החזרה של הקריאה המזומנת היא ההשמה (-).

ח. נדון בקריאה שזומנה מסעיף ז׳. נציג את מצב המחסנית:





כרגיל, אנו מסב את תשומת לבכם למצבם של המערכים ושל המשתנים המתארים את מספר הערכים שהוקצו. העדכונים האחרונים (הוספת 4 ל-`list[1]`, ועדכון: `alloc =3, alloc1 = 2`) בוצעו על-ידי העותק מסעיף ז׳, טרם שהוא קרא לעותק הנוכחי.

העותק הנוכחי מתחיל להתבצע. הוא מקצה את `list[alloc]` כלומר את `list[3]` שערכו ארבע, לקבוצה הראשונה. תוך כדי כך הוא מגדיל את `alloc` להיות ארבע, ואת `alloc1` להיות שלוש. אחַר, העותק הנוכחי של הפונקציה מגלה שסכום האיברים בקבוצה הראשונה (בעקבות ההקצאה) הוא תשע, כלומר הסכום גדול מדי. לכן הערך 4 שהוקצה לקבוצה הראשונה מועבר לשניה תוך עדכון: `alloc1 = 2, alloc2 = 2`. בעקבות ההעברה מתעדכן ערכו של `curr_sum` להיות סכום הערכים בקבוצה השניה, כלומר חמש. מכיוון שהקצנו את הערך האחרון שהיה להקצות אזי הפונקציה מחזירה את תוצאת ההשוואה בין `curr_sum` ל- `half`, ולכן את הערך `SUCCESS`.

ט. הערך שהחזיר העותק ח׳ מוחזר לעותק ז׳ לַפקודה (-) המשימה ערך לַמשתנה `success` של העותק ז׳. אחרי השלמת ההשמה (בַעותק ז׳) נערכת בדיקה מה ערכו של המשתנה, ומכיוון שערכו הוא `SUCCESS` אזי עותק ז׳ מסיים, תוך שגם הוא מחזיר את הערך `SUCCESS` . העותק מסעיף ז׳ נקרא על-ידי העותק מסעיף ג׳. כתובת החזרה של העותק מסעיף ז׳ היא הפקודה (+) בעותק מסעיף ג׳. לכן העותק מסעיף ג׳ מחזיר את מה שהחזיר לו העותק מסעיף ז׳, כלומר את הערך `SUCCESS`. הערך המוחזר על-ידי העותק מסעיף ג׳, חוזר להשמה (-) בעותק מסעיף ב׳. בעקבות השלמת ההשמה (-) יכול גם העותק מסעיף ב׳ להחזיר את הערך `SUCCESS`, ובכך מסתיים התהליך.

לפני שאנו נפרדים מבעיה זאת נציג לעצמנו כמה שאלות בתור חומר למחשבה (ולתרגול):

א. מה זמן הריצה של התכנית שכתבנו?
ב. כיצד היה עלינו לכתוב את הפונקציה לו הערכים אותם היה צורך לחלק בין שתי הקבוצות היו מספרים שלמים כלשהם (לא בהכרח טבעיים, כפי שאנו הנחנו)?
ג. כיצד היה עלינו לכתוב את הפונקציה לו היה צורך לחלק את הערכים לא בין שתי קבוצות, אלא בין `S` קבוצות, עבור `S` שאינו בהכרח שתיים (כפי שהיה במקרה שלנו)? ניתן מספר רמזים לתיקון הנדרש במקרה זה:
1. במקום להחזיק זוג מערכים חד-ממדיים `list1, list2` היה עלינו להחזיק מערך דו-ממדי בן `S` שורות, כל שורה תכיל את הערכים שהוקצו לתת-הקבוצה המתאימה.
2. במקום להחזיק זוג מונים `alloc1, alloc2` היה עלינו להחזיק מערך חד-ממדי של מונים. התא מספר `i` במערך היה מורה כמה ערכים הוקצו לתת-הקבוצה מספר `i`.
3. במקום להקצות את הערך ראשית לקבוצה הראשונה (ואז לבדוק האם מכך ניתן להמשיך ולבנות פתרון), ואחר לקבוצה השניה (ואז לבדוק האם מכך ניתן להמשיך ולבנות פתרון), היה עלינו לבצע לולאה שכל פעם תוסיף את הערך המוקצה על-ידי העותק הנוכחי של הפונקציה לתת-קבוצה כלשהי, ואחר תקרא רקורסיבית כדי לבדוק האם ניתן להמשיך ולבנות פתרון מלא בעקבות ההקצאה שבוצעה.

8.7 תרגילים

8.7.1 תרגיל מספר אחד: תכניות רקורסיביות פשוטות

בכל הפונקציות שעליכם לכתוב בסעיף זה אין להשתמש בפקודות לולאה, בכולן ניתן (ולעיתים אף יש) להוסיף פרמטרים נוספים מעבר לאלו המתוארים בשאלה. תלמידים שאינם מכירים עדיין את מושג הסטרינג לא יענו, בשלב זה, על סעיפים ג', ד'.

א. כתבו פונקציה המקבלת מערך של מספרים שלמים בגודל N ומחזירה את איבר המרבי במערך. (הסיקו מכך כיצד ניתן להחליף כל לולאה ברקורסיה).
ב. כתבו פונקציה המקבלת מערך של מספרים שלמים בגודל N ומחזירה את סכום האברים במערך.
ג. כתבו פונקציה המקבלת string (מערך של תווים), ומחזירה true אם ורק אם הסטרינג הוא פלינדרום.
ד. כתבו פונקציה המקבלת שלושה סטרינגים ומכניסה לתוך הראשון בין השלושה את תוצאת השירשור של שני האחרים. הפונקציה תחזיר את ערך true אם ורק אם השרשור הצליח, במובן זה שמערך השלישי היה גדול דיו להכיל את תוצאת השירשור (כולל ה- '\0'). אין להשתמש בפונקציות מ- string.h.
ה. כתבו פונקציה המקבלת שני מספרים טבעיים a, b ומחזירה את a % b בלי שהיא עושה שימוש בפעולות כפל ושארית (אלא רק בחיבור וחיסור).
ו. כתבו פונקציה המקבלת מספר טבעי num ומדפיסה את פירוקו לגורמים ראשוניים (לדוגמא עבור 12 יודפסו: 2,2,3. רמז: לפונקציה יהיה פרמטר נוסף divider, בקריאה הראשונה לפונקציה יועבר לה בפרמטר הנוסף הערך 2, הפונקציה תבדוק האם divider מחלק את num אם כן תדפיסו ותקרא רקורסיבית עם אותו מחלק ועם num/divider, אם לא תקרא רקורסיבית עם אותו num ועם divider+1. הרקורסיה תיעצר עת...)
ז. כנ"ל, אך במקום להדפיס את הפירוק הוא יוחזר במערך.

כתבו תכנית ראשית בה המשתמש יוכל לבחור איזה פונקציה ברצונו להפעיל או לציין שברצונו לסיים. עבור כל פונקציה הוא יזין את הקלט הדרוש (לדוגמה עבור סעיף א' הוא יזין N מספרים שלמים), יוצג לו הקלט והערך המוחזר על-ידי הפונקציה.

8.7.2 תרגיל מספר שתיים: מיון מהיר (Quick Sort)

מיון מהיר מוגדר באופן הבא:

א. עת יש למיין קטע מערך הכולל תא יחיד אין כל עבודה לבצע, קטע המערך כבר ממוין.

ב. עת יש למיין קטע מערך הכולל $n > 1$ תאים בצע:

1. מיין את חציו השמאלי של קטע המערך (תוך שימוש ב- merge sort).
2. מיין את חציו הימני של קטע המערך.
3. מזג את שני החצאים (הממוינים) לכדי קטע מערך ממוין בן n תאים.

הערה: מיזוג שני קטעי מערך **ממוינים** מתבצע באופן הבא: (1) התחל בכך שאתה מצביע על התא הראשון בכל-אחד משני הקטעים. (2) חזור על התהליך הבא עד שאתה מסיים לעבור על **אחד** משני קטעי המערכים: **אם** האיבר עליו אתה מצביע בקטע המערך האחד קטן או שווה מהאיבר עליו אתה מצביע בקטע המערך השני **אזי** העתק את האיבר עליו אתה מצביע בקטע המערך האחד למערך שיכיל את תוצאת המיזוג, והתקדם על קטע המערך האחד, **אחרת** העתק את האיבר עליו אתה מצביע בקטע המערך השני למערך שיכיל את תוצאת המיזוג, והתקדם על קטע המערך השני. (3) **אם** סיימת לעבור על קטע המערך האחד **אזי** העתק את יתרת האיברים מקטע המערך השני למערך שיכיל את תוצאת המיזוג, **אחרת** להפך. (4) העתק את תוצאת המיזוג חזרה למקומה במערך המקורי.

עליכם לכתוב תכנית אשר קוראת מהמשתמש את הקלט הבא:

א. ראשית, יזין המשתמש את המילה asc אם ברצונו למיין את הטקסט שהוא יזין בהמשך בסדר מיון עולה, הוא יזין את המילה dsc אם ברצונו למיין את הטקסט בסדר מיון יורד.

ב. עתה יזין המערך טקסט (סדרה של מחרוזות) עד איתות על סוף קובץ הנתונים (eof).

התכנית תקרא את המחרוזות למערך (דו-ממדי של תווים), ואחר תמיין את המערך תוך שימוש באלגוריתם המיון Merge Sort בסידור המבוקש. לבסוף תציג תכנית את הטקסט הממוין. למותר לציין שהשוואת מחרוזות תיעשה באמצעות strcmp. את מערך המחרוזות קבעו בגודל 17 מחרוזות, כל-אחת בגודל 13 (לכל היותר, כולל ה-'\0').

8.7.3 תרגיל מספר שלוש: הצגת צעד רצוי במגדלי האנוי

כתבו תכנית אשר קוראת מהמשתמש את הקלט הבא:

א. גודלו של המגדל שברצונו להזיז.

ב. על איזה עמוד ניצב המגדל בראשית התהליך.

ג. לאיזה עמוד יש להעביר את המגדל בסיום התהליך.

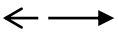
ד. מה מספר הצעד בו הוא מתעניין.

התכנית תדפיס למשתמש מה תהיה הטבעת שתועבר בצעד המבוקש על-ידו, ומאיזה מגדל לאיזה מגדל תועבר הטבעת הרצויה. במידה והעברת המגדל מתבצעת תוך פחות צעדים מהערך שהוזן בסעיף ד' תודיע התכנית למשתמש גם על כך.

<u>רעיון הפתרון</u>: לפונקציה hanoi הוסיפו פרמטר משתנה באמצעותו יועבר לפונקציה כמה טבעות כבר הועברו טרם שהקריאה הנוכחית לפונקציה זומנה (מה יהיה ערכו של הפרמטר בקריאה הראשונה מהתכנית הראשית? למה זה צריך להיות פרמטר משתנה?), ופרמטר ערך שיציין את מספר הצעד הרצוי. עת קריאה רקורסיבית כלשהי 'מזיזה' טבעת (גם אם לא בהכרח מדווחת על כך) היא תוסיף 1 לפרמטר העבודה שנעשתה ותשווה אותו לפרמטר הצעד הרצוי, אם ... אז ... **וכן לא תבוצענה קריאות רקורסיביות נוספות** (לשם ההעברה ממגדל העזר למגדל המטרה). שנאמר תחשבו על זה...

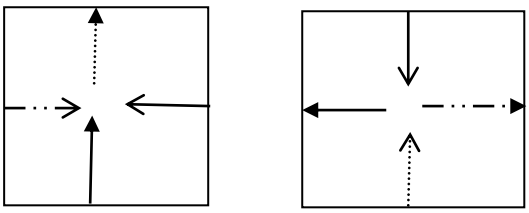
8.7.4 תרגיל מספר ארבע: סידור כרטיסים בתבנית רצויה

בתרגיל זה נרצה להשתמש בטכניקה של עקיבה לאחור על-מנת לפתור משחק מוכר. ראשית נציג את המשחק: בפני המשחֵק במשְחק מוצגים תשעה כרטיסים ריבועיים. בכל אחד מארבעת צדי כל כרטיס מצוייר חצי דג; חצי הדג עשוי לכלול ראש או זנב, של דג בצבע כזה או אחר. לצורך הפשטות נצייר אנו כל דג כחץ, ראש הדג יומר בחץ עם ראש משולש, וזנב הדג יומר בחץ עם ראש בצורת v. דג שלם יראה אצלנו בצורה הבאה:



על כן בשולי הכרטיסים שאנו נצייר יופיע ראש חץ עם ראש משולש או בצורת v. את צבע הדג נמיר בציורנו בתבנית קו כזו או אחרת. נציג שני דגים בעלי צבעים שונים:

נציג שני כרטיסים לדוגמה:



מטרת המשחק היא לסדר את הכרטיסים במבנה ריבועי של 3x3 באופן שכל דג יושלם כהלכה, כלומר זנב בצבע X יחובר לראש בצבע X. (לחצי דג שפונה לשולי המבנה הריבוי אין השפעה על הסידור הדרוש).

כדי לייצג את המשחק בתכנית מחשב נזדקק להגדרות הבאות:

```
enum fish {head, tail, yellow, green, blue, red } ;
enum coordinate {UP_FISH_PART=0, UP_COLOR=1,
                 LEFT_FISH_PART=2, LEFT_COLOR=3,
                 DOWN_FISH_PART=4, DOWN_COLOR=5,
                 RIGHT_FISH_PART =6, RIGHT_COLOR=7 };
fish cards[9][8] ;
```

נסביר: המערך `cards` כולל תשע שורות. כל שורה מתארת כרטיס במשחק. כדי לתאר כרטיס אנו זקוקים לשמונה תכונות: התכונה `cards[i][UP_FISH_PRT]` תכלול את הערך `head` או `tail` ותעיד האם בכרטיס מספר `i`, בכיוון כלפי מעלה מצויר ראש או זנב של דג כלשהו. התכונה `cards[i][UP_COLOR]` תציין מה צבעו של הדג המצוייר בכרטיס מספר `i`, בכיוון כלפי מעלה. באופן דומה, התכונה `cards[i][LEFT_FISH_PART]` תציין מה מצוייר בצדו השמאלי של הכרטיס

מספר i, בעוד cards[i][LEFT_COLOR] תורה מה צבע הציור בשוליו השמאליים של הכרטיס מספר i.

על המשתמש יהיה להזין ערכים למערך בהתאם לכרטיסים שבידו.

בנוסף למבנה הנתונים הנ"ל נחזיק מבנה נתונים נוסף:

```
unsigned int sort[3][3][2] ;
```

במבנה נתונים זה נחזיק את סידור הכרטיסים כפי שהתכנית יצרה בכל שלב בתהליך החיפוש (בתהליך העקיבה לאחור). sort[i][j][0] יחזיק את מספרו של הכרטיס שהוצב בשורה מספר i בעמודה מספר j. sort[i][j][1] יחזיק את הכיוון בו הוצב הכרטיס: כשראשו (כפי שהכרטיס מתואר במערך cards) כלפי מעלה, שראשו מסובב כלפי שמאל, כשראשו כלפי מטה, או שכראשו מסובב לכיוון ימין.

התכנית שנכתוב תקרא, כאמור, את תיאור הכרטיסים מהמשתמש, ואחר תנסה, תוך שימוש בעקיבה לאחור, לסדר את הכרטיסים.

רעיון אלגוריתם הוא כדלהלן: כל קריאה רקורסיבית תנסה להוסיף כרטיס בתא עליו היא מופקדת. הקריאה הראשונה תהיה מופקדת על התא הראשון במערך, הקריאה השניה על התא השני, וכן הלאה (תהינה על-כן תשע קריאות רקורסיביות). כל קריאה רקורסיבית תבצע:

א. עבור על כל הכרטיסים שטרם הוצבו בידי קריאות קודמות לפונקציה. עבור כל כרטיס כזה נסה להציבו בכל אחד מארבעת הכיוונים האפשריים (כיוון אחר כיוון):
ב. נסה למקם את הכרטיס הנוכחי בתא הנוכחי בכיוון הנוכחי.
ג. במידה ולא התגלתה סתירה עם הדרך בה הוצבו כרטיסים קודמים (על-ידי קריאות רקורסיביות קודמות) בצע: אם זו הקריאה הרקורסיבית התשיעית (והאחרונה) אזי הצג את הפתרון שנמצא, אחרת: קרא רקורסיבית כדי למקם כרטיס נוסף בַּתא הבא. במידה והקריאה הרקורסיבית מחזירה הצלחה סיים, אחרת:
ד. נסה לסובב את הכרטיס הנוכחי לכיוון שטרם נוסה. עבור כל כיוון אפשרי שלא יוצר סתירה קרא לעצמך רקורסיבית.
ה. במידה וכל ארבעת האופנים להציב את הכרטיס הנוכחי בתא הנוכחי לא הובילו לפתרון, נסה להציב כרטיס אחר (שטרם הוצב על-ידי קריאות רקורסיביות קודמות) במקום הכרטיס הנוכחי.
ו. במידה ולא הצלחת להציב אף כרטיס בתא הנוכחי אותת למי שזימן עותק זה של הפונקציה כי לא ניתן להשלים את הפתרון (ועל-כן מי שקרא לעותק זה של הפונקציה צריך לנסות לסובב את הכרטיס שהוא הציב, או להציב כרטיס אחר בתא שעליו העותק שקרא לעותק הנוכחי מופקד).

8.7.5 תרגיל מספר חמש: מילוי תשבץ במילים

גם בתרגיל זה נרצה להשתמש בעקיבה לאחור כדי להשלים משימה רצויה כלשהי. המשימה היא הבאה: נתון מערך דו-ממדי המייצג תשבץ, וכולל על-כן תאים פנויים בתוכם ניתן לשבץ מילים, ותאים מלאים המפרידים בין מילים. המשימה שעל התכנית שלנו לבצע היא להציב בתשבץ מילים מתוך מאגר נתון של מילים, באופן שאחַר כותב התשבץ יצטרך רק עוד להוסיף הגדרות למילים, ובידיו יהיה תשבץ מוכן לפרסום.

מכיוון שאנו טרם למדנו לטפל במילים נמיר כל מילה במספר טבעי. נניח על כן כי בתכנית הוגדר: `int matrix[N][N];` . נניח גם כי המערך אותחל כך שכך תאיו מכילים את הערך `EMPTY` (שהינו קבוע שערכו שלילי). בַּתחִלה, עלינו לקרוא מהמשתמש את מציינ‍י התאים המלאים (כאלה שבתשבץ רגיל מצויירים בשחור). בכל תא מלא נכניס את הערך `BLOCKED` (קבוע אחר שערכו שלילי).

בשלב השני עלינו לקרוא מהמשתמש סדרה של מספרים טבעיים (כל מספר אנלוגי למילה בתשבץ רגיל) מתוכה עלינו לשבץ מילים בתשבץ שנכין. כל ספרה מתוך כל מספר שיוכנס בסופו של דבר לתשבץ תמוקם בתא נפרד. המשתמש רשאי להזין יותר מספרים מכפי שיש צורך לשבץ בתשבצו, במקרה כזה בחלק מהמספרים לא נעשה שימוש.

הפונקציה הרקורסיבית שנרצה לכתוב, ושתהווה את לֵבה של התכנית תשתול מספרים טבעיים בתאים הריקים שבתשבץ. אחרי מילוי התשבץ נרצה להציגו למשתמש.

בתרגיל זה לא נתאר את האלגוריתם בצורה מפורטת. נסתפק ברמז כי כל קריאה רקורסיבית תהיה אחראית על הוספת מילה אפשרית החל בתא במערך בו מתחילה מילה בכיוון מאוזן או מאונך (זכרו כי לא בכל תא ריק במערך מתחילה מילה). הקריאה תנסה להוסיף את כל המילים שלא הוספו על-ידי קריאות שקדמו לה, ותקרא רקורסיבית לשם השלמת תהליך המילוי.

8.7.6 תרגיל מספר שש

נתונה פונקציה:

```
int what(const int a[], int x, int y, int& t)
{
    int tmp1, tmp2, ttt;

    if( x == y ) {
        t = 1;
        return a[x];
    }

    tmp1 = what(a , x  , (x+y)/2 , t  );
      tmp2 = what(a , (x+y)/2 + 1, y  , ttt);

    if( tmp1 == tmp2 ) {
        t += ttt;
        return tmp1;
    }

    if( tmp1 > tmp2 )
        return tmp1;

    t = ttt;
    return tmp2;
}
```

א. בתוכנית הראשית הוגדר:

```
int arr[] = {2,1,0,1,2};
int d, r;
```

```
r = what(arr, 0, 4, d);
```
אחרי הביצוע של הנ"ל, מה יהיו ערכי r ו- d ?

ב. כמה קריאות לפונקציה תתבצענה עבור הנתונים מסעיף א' (כולל הקריאה מהתוכנית הראשית)? תארו את עץ הקריאות של הרקורסיה.

ג. מה עושה הפונקציה what() ?

ד. האם what() היא פונקציה יעילה (מבחינת זמן ריצה) ?

ה. האם הפונקציה הייתה עובדת כהלכה לו זימנו אותה באופן הבא (הסבירו):
```
r = what(a,4,0,d);
```

8.7.7 תרגיל מספר שבע: התאמת תבניות (Pattern Matching)

התאמת תבניות היא פעולה המקבלת תבנית וסטרינג. על הפעולה לקבוע האם הסטרינג מתאים לתבנית או לא.

כתבו פונקציה המקבלת סטרינג שעשוי לכלול כל אחד מהתווים a..z, ותבנית שעשויה לכלול את התווים: a..z,*,? .
הסטרינג והתבנית יתאימו זה לזה אם:

א. בשניהם במקום המתאים ניצב אותו תו.

ב. בתבנית במקום המתאים ניצב התו '?', ובסטרינג ניצב צו כלשהו.

ג. בתבנית במקום המתאים נמצא התו '*', ובסטרינג נמצאים אפס או יותר תווים כלשהם המתאימים לכוכבית, באופן שיתר הסטרינג יתאים ליתר התבנית.

לדוגמה:

א. הסטרינג 'abc' מתאים לתבניות: '*b*', '*abc', '*bc', 'a*', '??c', 'a?c', 'abc', אך לא לתבניות: 'ab*c?', 'ab??', 'ab', 'abcd' .

ב. הסטרינג: 'ababab' מתאים לתבניות: '*ab', 'ab??***', 'ab*', 'ab??ab', אך לא לתבניות: 'ab*a', 'abb*'.

8.7.8 תרגיל מספר שמונה: איתור מסלולים במערך

נתון מערך דו-ממדי: a[N][N] .
נגדיר: **מסלול** במערך המתחיל בתא [x1][y1] ומסתיים בתא [x2][y2] להיות: סדרה של תאים במערך, באופן שמתקיים כי:

א. התא הראשון בסדרה הוא: [x1][y1].

ב. התא האחרון בסדרה הוא: [x2][y2].

ג. כל תא בסדרה פרט לתא הראשון מצוי במערך מעל, מתחת, מימין או משמאל לתא שקדם לו בסדרה.

ד. כל תא במערך מופיע לכל היותר פעם יחידה במסלול.

לדוגמה במערך בן חמש שורות וחמש עמודות קיים המסלול הבא בין התא [0][1] לתא [2][4]: [0][1], [0][2], [0][3], [1][3], [1][4], [2][4] , וקיימים, כמובן עוד מסלולים רבים.

א. כתבו פונקציה המקבלת זוג תאים במערך (כלומר האלכסון a[i][i] עבור i=0..N-1) ומחזירה את מספר המסלולים הקיימים בין שני התאים.

ב. כתבו פונקציה המקבלת זוג תאים במערך המצויים מתחת לאלכסון הראשי (כלומר האלכסון a[i][i] עבור i=0..N-1) ומחזירה את מספר המסלולים שאינם חוצים את האלכסון הראשי (אך עשויים להכיל תאים המצויים באלכסון זה) הקיימים בין שני התאים.

8.7.9 תרגיל מספר תשע: איתור תולעים במערך

עבור מערך דו-ממדי של מספרים שלמים, נגדיר **תולעת** באורך k במערך להיות סדרה של k תאים, שכל אחד מהם סמוך לקודמו (כלומר הינו אחד משמונת שכניו במערך, שֶׁכן הוא מצוי מעליו, מתחתיו, מימינו, משמאלו או באלכסון לו) כך שהערכים בתאים מהווים סדרה עולה של מספרים שלמים עוקבים n, n+1, …, n+k-1 (n, האיבר הראשון בסדרה, עשוי להיות כלשהו). למשל במערך:

3	13	15	17	30
50	51	28	29	55
51	10	52	54	56
52	12	7	53	11

בתא [0][1] מתחילה התולעת באורך שלוש הבאה: 50,…,52 (הנגמרת בתא [0][3]).
בתא [0][1] מתחילה התולעת באורך שבע הבאה: 50,…,56 (הנגמרת בתא [4][2]).
בתא [2][1] מתחילה התולעת באורך שלוש הבאה: 28,…,30 (הנגמרת בתא [4][0]).

כתבו פונקציה המקבלת מערך דו-ממדי ומחזירה את אורכה ואת נקודת התחלה של התולעת הארוכה ביותר במערך. במידה וקיימות מספר תולעים מקסימליות באורכן ניתן להחזיר אחת מהן כפי רצונכם.

8.7.10 תרגיל מספר עשר: מסעי הפרש

נתון לוח שחמט המיוצג על-ידי מערך דו-ממדי: `board[N][N]`. כתבו תכנית אשר קוראת מהמשתמש קואורדינטה של תא רצוי במערך ואחר מבצעת את המשימה הבאה: על התכנית לאתר ולהציג את כל האופנים בהם ניתן למלא את הלוח בצעדי פרש לפי הכללים הבאים:

א. הפרש יוצא מהתא שהוזן על-ידי המשתמש.
ב. הפרש מבקר בכל משבצת בלוח פעם אחת בדיוק.

לאלה מכם אשר אינם בקיאים במשחק השחמט נסביר כי פרש במחשק השחמט רשאי להתקדם מהתא בו הוא ניצב לאחד משמונה תאים סמוכים על-ידי שהוא פוסע שתי משבצות בכיוון מסוים, ואז משבצת אחת הצידה. כמובן שאם הפרש ניצב בשולי הלוח אזי פתוחות בפניו פחות משמונה משבצות אליהן הוא יכול להתקדם.
לדוגמה, בלוח הבא, אם הפרש ניצב בתא המסומן ב- * אזי הוא יכול לפסוע לכל אחד מהתאים המסומנים ב- + :

			+	
	*			
			+	
+		+		

פלט התכנית יכלול סדרה של תיאורי לוח. בכל תיאור לוח, עבור כל תא יופיע מספר הצעד בו ביקר הפרש בתא זה.

חיזרו וכתבו את התכנית בשנית, אולם עתה הציגו פתרון יחיד.

מה הוא זמן הריצה של התכנית שכתבתם?

8.7.11 תרגיל מספר אחת-עשרה: תמורות

בהינתן סדרה של מספרים, **תמורה** (permutation) של המספרים היא דרך נוספת לסדר את המספרים. לדוגמה הסדרה {5, 9, 17} היא תמורה של הסדרה {5, 17, 9}; גם הסדרה {17, 9, 5} היא תמורה של אותה סדרה. למעשה בהינתן סדרה בת n מרכיבים קיימות ... תמורות שונות של n המרכיבים (אתם מוזמנים להשלים את השלוש נקודות).

כתבו פונקציה המקבלת מערך הכולל n מספרים שלמים שונים זה מזה (ופרמטרים נוספים על פי הבנתכם). על הפונקציה להציג את כל התמורות של n המספרים.

8.7.12 תרגיל מספר שתיים-עשרה: מגדלי האנוי עם טבעות ורודות וירוקות

נדון בבעיית מגדלי האנוי כפי שהכרנו בַּפרק, עם שנוי קל: הפעם נניח כי עבור כל גודל טבעת קיימת הן טבעת וְרוּדה, והן טבעת ירוקה מגודל זה. עוד נניח כי מותר להניח טבעות שוות גודל זו על זו, וכי בְּתְחילה המגדל בנוי כך שמכל גודל טבעת הטבעת הורודה מונחת על-גבי הטבעת הירוקה.

כתבו תכנית הקוראת מהמשתמש את גודלו של המגדל שיש להעביר, ואת זהותם של עמוד המקור ועמוד היעד, ומציגה את סדרת הצעדים שיש לבצע על-מנת להעביר את המגדל הנתון מעמוד המקור לעמוד היעד, תוך שבסיום התהליך יראה המגדל בדיוק כפי שהוא נראה בתחילת התהליך (כלומר עבור כל גודל טבעת, הטבעת הורודה מונחת על הטבעת הירוקה).

8.7.13 תרגיל מספר שלוש-עשרה: כמות העבודה הנדרשת בעת העברת מגדלי האנוי

נדון בבעיית מגדלי האנוי כפי שהכרנו בַּפרק, עם שנוי קל: הפעם נניח כי עת אנו מעבירים טבעת בגודל n מעמוד source לעמוד dest אנו משקיעים עבודה שהינה בשיעור מכפלת גודל הטבעת במרחק בין העמודים. המרחק בין שני העמודים הקיצוניים מוגדר להיות שתיים, והמרחק בין העמוד האמצעי לשני העמודים הקיצוניים מוגדר להיות אחד.

כתבו תכנית אשר קוראת מהמשתמש את גודלו של המגדל שיש להעביר, ואת זהותם של עמוד המקור ועמוד היעד. התכנית תציג למשתמש כמה עבודה נדרשת לשם העברת המגדל כנדרש.

9 משתנים תוויים (chars) ומחרוזות (strings)

עד כה הכרנו משתנים פרימיטיביים מספריים (כדוגמת `int, float`), ומשתנים פרימיטיביים בולאניים. בפרק זה נכיר סוג נוסף של משתנים פרימיטיביים: **משתנים תוויים** (`character variables`).

בתכניות שאנו כותבים, לעיתים עלינו להחזיק תווים (כדוגמת אותיות או סימנים אחרים, למשל: !) ומחרוזות (כגון שם של תלמיד) במשתנים של התכנית. משתנים מטיפוס `char`, ומערכים של משתנים מטיפוס `char` הם שיאפשרו לנו להשיג מטרה זאת.

9.1 משתנים תוויים

עת מתכנני מערכת ההפעלה עמלים על עיצובה עליהם להחליט, בין היתר, אילו תווים מערכת ההפעלה שהם מתכננים תכיר, במילים אחרות באילו תווים ניתן יהיה להשתמש במחשב, (לדוגמה: האם במחשב יהיה התו a כך שניתן יהיה לעשות בו שימוש? האם הוא יכיר את התו A? האם הוא יכיר את התו מפתח סול? האם הוא יכיר את האות אלף? את האות היוונית פי? ועוד). תוצאת ההחלטה של מתכנני מערכת ההפעלה נקראת **אוסף התווים** (character set) של המחשב. חלקכם ודאי שמע על ה- ASCII שהוא דוגמה אחת לאוסף תווים אפשרי. ה- ASCII משמש במחשבים רבים שכן הן מערכת ההפעלה חלונות, והן מערכת ההפעלה יוניקס אמצו אוסף תווים זה. (דוגמה אחרת לאוסף תווים הוא ה- EBCEDIC ששימש בעבר חשבים של חברת IBM).

אחת התכונות של אוסף תווים היא 'מספר התווים הנכללים באוסף התווים'. כלומר כמה סימנים שונים נכללים באוסף התווים (למשל, באלף בית העברי, אם סופרים גם אותיות סופיות יש 26 תווים). הערכים המקובלים לתכונה זאת הם 128 או 256. תכונה שנייה של אוסף תווים היא המיקום של כל תו באוסף התווים. לדוגמה באוסף התווים 'האלף בית העברי' מצוי התו 'י' במקום העשירי, והתו 'מ' במקום השלושה עשר. ב- ASCII הקיים ברוב המחשבים ישנם 256 תווים. נדון בחלק מהם ובמיקומם באוסף זה של תווים:

א. הספרות '0'..'9' ממוקמות במקומות 48 ואילך ברצף, זו אחרי זו (התו '0' נמצא בַּמקום מספר 48, התו '1' מצוי במקום מספר 49, ... התו '9' נמצא בַּמקום מספר 57).

ב. האותיות 'a'..'z' ממוקמות במקומות 97 עד 122 (ברצף, זו אחרי זו).

ג. האותיות 'A'..'Z' ממוקמות במקומות 65 עד 90 (ברצף, זו אחרי זו).

ד. התו רווח מצוי במקום מספר 32, והתו מעבר-שורה מצוי במקום מספר 10.

ה. כל תו אחר שאתם יכולים להקיש על המקלדת (כגון /, ;, ~) מצוי לו אי שם באוסף התווים.

ו. במחשבים בהם ניתן לעבוד בעברית מותקנות גם האותיות העבריות במקומות מספר 128 ואילך.

ז. תו בעל תפקיד מיוחד עבור תכניות בשפת C, הוא התו הראשון ב- ASCII, התו שבמַקום מספר אפס. זהו תו שאינו ניתן להצגה על המסך. בתכניות בשפת C אנו מציינים תו זה בַּאופן הבא: '\0'. נסביר: כפי שכבר ראינו, (עם ה- \n), עת אנו כותבים את הלוכסן השמאלי (ה- backslash) לפני תו, הלוכסן השמאלי משמש כ- **'תו מבריח'** (escape character) אשר משנה את משמעותו של התו העומד

אחריו. בַּמקרה של ה- \n השינוי הוא שבמקום שהתו n יציין את האות האנגלית En הוא מציין את התו מעבר-שורה (התו מספר 10 ב-ASCII). במקרה של ה- \0 השינוי הוא שבמקום שהתו 0 יציין את הספרה אפס, הוא מציין את המקום מספר אפס ב- ASCII. בהמשך נעמוד על התפקיד המיוחד שיש, בשפת C, לתו '\0'.

אתם מוזמנים לכתוב את קטע הקוד הבא ולגלות בעצמכם מהו אוסף התווים במחשב:

```
for (int i=0; i<256; i++)
   cout << i << "=" << char(i) << endl ;
```

נסביר: עבור כל אחד ואחד מ- 256 התווים במחשב (התווים במקומות אפס עד 255), אנו מציגים את מקומו של התו (ערכו של המשתנה השלם i), ואת התו המצוי במקום המתאים: char(i) מחזיר את התו במקום מספר i, במילים אחרות הוא מחזיר את התו השקול לערכו של i. (במקום לכתוב char(i) ניתן לכתוב: I (char), למעשה, בשפת סי [בניגוד ל: C++] חייבים לכתוב את הדבר השני בין השניים).

אם תריצו את התכנית הנ"ל תגלו שחלק מהתווים אינם ניתנים להצגה על-גבי מסך המחשב (למשל '\0' ועמיתיו שבתחילת ה- ASCII), חלקם מוכרים לכם היטב, וחלקם עשויים להפתיע אתכם, באשר עד עצם הרגע הזה לא דמיינתם לעצמכם שבחובו של מחשבכם שוכנים תווים שכאלה.

מכיוון שהתווים באוסף התווים מסודרים זה אחרי זה אזי ניתן גם לשאול איזה מהם קודם או אחרי משנהו. לדוגמה: בעברית האות 'י' קודמת לאות 'כ', בַּ- ASCII התו 'A' מקדים את התו 'a', כמו גם את התו 'B'.

בשפת C ניתן להגדיר משתנים מטיפוס char. משתנים אלה יוכלו להחזיק *תו בודד* בכל נקודת זמן. לדוגמה, אם נגדיר: char c1, c2; אזי בהמשך נוכל לכתוב: c1 = 'a'; c2 = '?'; אך לא נוכל לכתוב: c1 = 'yosi'; (שכן c1 יכול להכיל רק תו יחיד, ולא מחרוזת שלמה).

ניתן לקרוא ערך מהקלט לתוך משתנה תווי באמצעות הפקודה: cin >> c1;. פקודה זאת תגרום למחשב לפסוח על כל הרווחים, מעברי שורה ו- tab-ים שהמשתמש (אולי) הזין, ולקרוא לתוך c1 את הערך הראשון שאינו white space (לַתווים: רווח, מעבר-שורה, ו- tab קוראים בשם הקיבוצי white spaces). שימו לב כי רווח, tab, ומעבר שורה גם הם תווים לגיטימיים, וכעיקרון ניתן לקרוא גם את ערכם לתוך c1, אולם פקודת הקלט cin מדלגת על תווים אלה ומכניסה לתוך c1 את התו הראשון שאינו רווח, מעבר-שורה או tab.

מכיוון שבין התווים קיים יחס סדר, אזי אנו יכולים לבדוק האם הערך שמכיל משתנה תווי מצוי בטווח ערכים מסוים. לדוגמה:

```
cin >> c1 ;
if (c1 >= 'a' && c1 <= 'z')
   cout << "it is a lowercase letter\n" ;
else if (c1 >= 'A' && c1 <= 'Z')
   cout << "it is an uppercase letter\n" ;
```

בקטע תכנית זה קראנו תו יחיד לתוך המשתנה c1, ואחר בדקנו האם ערכו של המשתנה c1 בַּתחום 'a'..'z', ובמידה וכן הצגנו הודעה מתאימה. באופן דומה בדקנו האם ערכו בתחום 'A'..'Z', ושוב הצגנו הודעה מתאימה. יכולתנו לבדוק את התנאים, ולהציג במידה והם מתקיימים הודעות מתאימות, מסתמכת על כך

שהתווים מסודרים, וכן אנו מובטחים שבין 'a' ל- 'z', למשל, יש רק אותיות קטנות (ולא שום תו אחר).

אתם רשאים לכלול בתכניתכם את פקודת ה- include: #include <cctype> אשר תאפשר לכם להשתמש במאקרו הבאים (מכיוון שאנו לא הצגנו את נושא המאקרו, אתם יכולים להתייחס למאקרו כאל פונקציה, זה יהיה לא מדויק, אך נסבל):

א. isspace(c1) יחזיר true אם ערכו של המשתנה c1 הוא רווח, מעבר שורה, או tab.
ב. isdigit(c1) יחזיר true אם c1 מחזיק ספרה בתחום '0'..'9'.
ג. isalpha(c1) יחזיר true אם c1 מחזיק אות כלשהי (קטנה או גדולה).
ד. isalnum(c1) יחזיר true אם c1 מחזיק תו **אלפה-נומרי**, כלומר אות או ספרה.
ה. ispunct(c1) יחזיר true אם c1 מחזיק סימן פיסוק כלשהו (כדוגמת: '.', '!' ועוד).

לדוגמה: בהנחה שכללנו את פקודת ה- include המתאימה, אנו רשאים לכתוב:

```
cin >> c1 ;
if (isdigit(c1))
   cout << "read a digit\n" ;
else if (isalpha(c1))
   cout << "read a letter\n" ;
```

עת המשתמש מזין קלט לתכניתכם, מערכת ההפעלה, אשר אחראית על העברת הנתונים מהמקלדת לתכניתכם, מעתיקה תו, תו מהמקלדת אל שטח בזיכרון הראשי הקרוי **חוצץ הקלט** (input buffer). בחוצץ הקלט ממתין הקלט עד שתכניתכם תואיל בטובה להתעניין בו. נניח שהמשתמש הקיש על המקלדת את סדרת ההקשות הבאה: (א) רווח, (ב) רווח, (ג) התו '1', (ד) התו '7', (ה) התו רווח, (ו) התו '3', (ז) התו '8', (ח) התו '7', (ט) התו Enter. (ולכן קלט זה הושם על-ידי מערכת ההפעלה בחוצץ הקלט). עתה נבחן מה יקרה עת תכניתכם תבצע את פקודת הקלט: cin >> a >> c1; (עבור a שהינו משתנה שלם). המחשב, בתבונתו כי רבה, ידע לדלג על הרווחים המובילים, אלה המופיעים לפני הספרה '1'; ידע 'לאכול' את ה- '1' ואת ה- '7' ולהכניס ל- a את הערך 17. עתה הוא ידע לפסוח שוב על הרווח, ולהכניס ל- c1 את התו הבודד '3'. התווים '8', '7', ו- Enter יאלצו להמשיך להמתין בחוצץ הקלט. אם בהמשך תכניתנו תופיע הפקודה cin >> x; אזי אם x הוא משתנה מספרי, אזי יכנס לתוכו המספר 87; ואם x הוא משתנה תווי, אזי יכנס לתוכו התו '8'. שימו לב כי אם הקלט היה כולל במקום את התווים '8' ו- '7', את התו '^', ואם טיפוסו של x אינו תווי, אזי התכנית הייתה נכשלת במשימת הקריאה, וכל תהליכי הקלט שלה היו נחסמים, שכן היא ניסתה לקרוא לתוך משתנה מספרי ערך שלא ניתן לפרש כמספר.

9.1.1 קריאת מספר טבעי מהקלט תו אחרי תו

שפת C++ (אך לא שפת C) מעמידה לרשותכם את הפונקציה cin.get() אשר מאפשרת לכם לקרוא תו בודד לתוך משתנה תווי, בלי לדלג על white spaces. (הסיבה לכך ששם הפונקציה כולל בתוכו נקודה תובהר לכם כאשר תלמדו תכנות מונחה עצמים). הפונקציה cin.get() אינה מקבלת כל פרמטר, והיא מחזירה את התו הבא בקלט, כאמור בלי לדלג על רווחים. נראה כיצד ניתן להשתמש בה. נניח כי ברצוננו לכתוב את הפונקציה: bool read_num(unsigned int &num). הפונקציה מנסה לקרוא מספר טבעי מהקלט. היא מחזירה איתות האם הקריאה

הצליחה או לא. במידה והקריאה הצליחה יכיל הפרמטר המשתנה num את ערכו של המספר שנקרא. קריאת מספר שלם תצליח אם:

א. לפני המספר עשויים להופיע white spaces.

ב. אחר-כך מופיע רצף של ספרות המייצגות מספר טבעי הניתן לשמירה במשתנה מטיפוס unsigned int.

ג. רצף הספרות מסתיים ב- white space.

נציג את הקוד:

```
bool read_num(unsigend int &num) {
   char c ;

   num = 0 ;

   do                    // skip leading spaces
   {  c = cin.get();   }
   while (isspace(c)) ;

   if (!isdigit(c))      // after spaces u should meet
      return(false) ;    // a digit

   while (isdigit(c))    // read the number
   {
      num = num*10 + (c - '0') ;
      c = cin.get() ;
   }

   if (!isspace(c))    // the number should end with
      return(false) ; // a white space
   return(true) ;
}
```

הפונקציה שכתבנו פשוטה למדי, וגם תיעדנו אותה תיעוד יתר כדי לסייע לכם להבינה. יש בה רק נקודה אחת הראויה לדיון, והיא הפקודה: num = num*10 + (c - '0') ; . פקודה זאת כוללת אריתמטיקה מתמיהה המחייבת הסבר עקרוני:

בשפת C משתנים תוויים (כדוגמת המשתנה c שהגדרנו בפונקציה האחרונה), וקבועים תוויים (כדוגמת: '2', '%') הם יצורים דו-פרצופיים: את הפן האחד שלהם, הפן התווי, הכרנו; אך יש להם גם פן נוסף: פן מספרי; נסביר: כל משתנה תווי הוא גם משתנה שלם שעשוי להכיל מספרים בתחום 0..255, כל קבוע תווי הוא גם מספר שלם, שהוא מקומו של התו באוסף התווים, (לדוגמה: הפן המספרי של הקבוע התווי 'a' הוא 97, שכן 'a' ניצב במקום ה- 97 ב- ASCII). לכן במקום לכתוב: c = 'a'; אנו רשאים לכתוב: c = 97; , זה אולי יהיה פחות קריא, אך זה לגיטימי, וייתן אותה תוצאה בדיוק. כמו כן הפקודה: cout << c << int(c); תציג את שני פניו של c: את הפן התווי, ואת זה המספרי.

מהסיבה הנ"ל ניתן גם לבצע אריתמטיקה על משתנים תוויים. אם המשתנה c מכיל ספרה אזי בַּביטוי (c - '0') יש הגיון: הוא מפחית את מיקומו בַּ-ASCII של התו '0' ממיקומו בַּ-ASCII של התו השמור במשתנה c; במילים אחרות הביטוי הנ"ל מחזיר את המרחק בין התו אפס לַתו השמור ב- c. מרחק זה מייצג את הערך המספרי של התו שמכיל המשתנה c (אשר כזכור לנו מכיל בהכרח ספרה). למשל,

אם ערכו של c הוא '0' אזי c-'0' הוא אפס, אם ערכו של c הוא '1' אזי '0'-'c' ערכו אחד: המקום ב- ASCII של התו שמכיל c (התו '1') הוא המקום מספר 49, פחות המקום ב- ASCII של התו אפס, שהוא המקום מספר 48.

הזכרנו כי בשפת C (בניגוד ל- C++) לא עומדת לרשותנו הפקודה: cin.get(). כבר בעבר אמרנו כי כל נושא הקלט והפלט שונה בין שפת C ל- C++, בפרט קריאת תווים. לשם הדיוק ראוי לציין כי cin.get() מחזירה ערך שלם (ולא ערך תווי). הסיבה לכך תובהר בהמשך; לעת עתה נסתפק בכך שנבין שהדבר אינו יוצר קושי שכן למשתנה תווי ניתן להתייחס גם כאל מספר, ולכן ניתן להכניס לתוכו ערך שלם (ובלבד שהערך יהיה בַּתחום 0..255).

9.1.2 קריאת שורת מספרים מהקלט: הפקודה cin.putback()

שפת C++ מעמידה לרשותנו עוד שתי פונקציות שעשויות להיות שימושיות עת אנו עוסקים בתווים: הפקודה

```
c = cin.peek() ;
```

תכניס למשתנה c את התו המצוי בראש חוצץ הקלט (כלומר את התו הבא שהתכנית תקרא עת היא תבצע cin.get()), אך תו זה לא 'יאכַּל' מחוצץ הקלט אלא יישאר בחוצץ. שקולות לפקודה הנ"ל הן שתי הפקודות הבאות:

```
c = cin.get() ;
cin.putback(c) ;
```

הפקודה: cin.putback(c) ניתנת לביצוע רק עת המשתנה c מכיל את התו האחרון שנקרא מהקלט; הפקודה מחזירה תו זה לחוצץ הקלט (בלי לפגוע בערכו של המשתנה c). על כן, כאמור, זוג הפקודות התחתונות, שקולות בדיוק לפקודה העליונה יותר.

נראה דוגמה העושה שימוש בכלים הנ"ל: התכנית שלנו צריכה לקרוא מהקלט n שורות של מספרים שלמים (n נקרא מהקלט). כל שורה מכילה מספר אחד לפחות; אולם אורכה של כל שורה אינו ידוע, ולשורות שונות יש אורך שונה. על התכנית להדפיס את ממוצע הערכים בכל שורה ושורה.
דוגמה לקלט אפשרי:

```
4
17 3
5 -15 0
77
66 77 88 99
```

<u>הסבר</u>: הנתון הראשון (ארבע) מציין כי יש לקרוא ארבע שורות. כל שורה, כפי שנקל לראות מכילה מספר כלשהו של נתונים (בהכרח תקינים).

התכנית הראשית תכלול לולאה:

```
cin >> n ;
for (int i  = 0; i < n; i++)
   cout << average_of_a_line() ;
```

בלולאה הנ"ל אין כל רבותא. הרבותא היא בפונ' average_of_a_line אשר אמורה לקרוא שורה בודדת, ולהחזיר את ממוצע הערכים בשורה.

שימו לב כי הפונ' average_of_a_line צריכה לזהות ששורה נגמרה (ואז עליה להחזיר את ממוצע הערכים בַּשורה). אם הפונ' תקרא רק מספרים, ולא תתעניין בתווים בודדים, היא לא תוכל לזהות זאת, שכן עת אנו קוראים לתוך משתנה

מספרי: `cin >> num;` המחשב מדלג על מעברי שורה ואין לנו כל אינדיקציה לכך ששורה הסתיימה. על כן ברור שהפונ' תצטרך, בין היתר, לקרוא תווים בודדים, ובאמצעותם לזהות ששורה נגמרה (ועליה להחזיר את ממוצע הערכים בשורה).

נציג את קוד הפונ', ואחר-כך נסבירו:

```
double average_of_a_line()
{
   int num,
       sum = 0 ,
       counter = 0;
   char c ;

   while (true)
   {
      do {                        // skip over spaces
         c = cin.get() ;
      } while (c == ' ') ;

      if (c == '\n')             // if you reach end of line
           return( ((double) num) / counter );

      cin.putback(c) ;          // else: you got another num
      cin >> num ;
      sum += num ;
      counter++ ;
   }
   return 0 ;                   // only 2 clm dwn th complr
}
```

נסביר: כל סיבוב בלולאה הלכאורה אינסופית מתחיל בכך שאנו חולפים על פני הרווחים שאולי מופיעים לפני המספר (הבא). אם בתום המעבר על הרווחים הגענו לסוף שורה אזי אנו מסיימים את ביצוע הפונ', ומחזירים את הערך הדרוש. אחרת (בהינתן שהקלט תקין) הגענו לנתון חדש, שאת ספרתו הראשונה 'אכלנו' בלולאת ה-`do-while`; על כן נפלוט ספרה זאת חזרה לקלט, ונקרא (לתוך משתנה שלם) את המספר בשלמותו. נוסיף את המספר לַסכום, ונגדיל את מספר הערכים שקראנו באחד.

9.2 מחרוזות

9.2.1 מערך של תווים ככל מערך

משתנה תווי הוא יצור נחמד אך קצת חלש. במקרים רבים בהם יש לנו צורך לטפל בתווים אנו זקוקים למשתנה שיוכל להחזיק מחרוזת שלמה (כגון "yosi"), ולא רק תו בודד. מערך חד-ממדי של תווים הוא הכלי שיאפשר לנו להחזיק מחרוזת שלמה, ומייד נראה כיצד. (לעיתים נשתמש במילה האנגלית 'סטרינג' במקום עמיתתה העברית 'מחרוזת'.)

ראשית נבהיר כי אם הגדרנו מערך: `char s[LEN];` אזי `s` הוא מערך ככל מערך אחר. לדוגמה אנו רשאים לכתוב לולאה אשר מכניסה לכל תאי המערך את התו 'x':

```
for (i=0; i<LEN; i++)
   s[i] = 'x' ;
```

באופן דומה אנו רשאים לכתוב לולאה אשר קוראת תווים לתוך המערך:

```
i = 0 ;
do {
   cin >> s[i++] ;
}
while (!isspace(s[i-1]) && i < N) ;
```

נשים לב כי הלולאה שלנו מכניסה למערך תווים עד קבלת הערך רווח (או עד מילוי המערך), על כן נוכל להציג את תוכנו של המערך ע"י לולאה מקבילה אשר תרוץ על תאי המערך עד התקלות בתא המכיל את הערך רווח (או עד סיום).
הקוד:

```
for (i = 0; i < N && s[i] != ' '; i++)
   cout << s[i] ;
```

בדוגמות שהצגתי עד כה רציתי להדגיש שמערך של תווים הינו מערך ככל מערך אחר, וניתן לעשות בו שימוש כבכל מערך אחר. הדגשתי זאת שכן מערך של תווים מתייחד מכל מערך אחר בכך שקיימות פעולות אותן ניתן לבצע על המערך *בשלמותו*, ואשר מאפשרות לנו לטפל בנוחות במחרוזות (כלומר בסדרה של תווים). רוב השימושים שלנו במערכים של תווים, לכל הפחות בדיון הנוכחי, יהיו כאלה שיטפלו במערך בשלמותו; למרות זאת, אל לנו לשכוח שביסודו מדובר במערך ככל מערך.

9.2.2 קלט ופלט של מחרוזת, null terminated string

דוגמה ראשונה לפעולה המתבצעת על מערך של תווים בשלמותו: על מערך של תווים, ורק עליו, אנו רשאים לבצע את הפקודה:

```
cin >> s;
```

פקודה זאת תקרא מחרוזת לתוך המשתנה s על-פי הכללים הבאים:

א. היא תדלג על פני כל ה- white spaces המובילים שאולי הוזנו, ולא תכניסם ל- s.
ב. היא תכניס את רצף התווים שהוזן עד ה- white space הראשון לתוך המשתנה. באחריות המתכנת לדאוג שהמערך s יהיה גדול דיו כדי להכיל את כל התווים המוזנים על-ידי המשתמש.
ג. היא תסיים את תהליך הקריאה עת היא נתקלת ב- white space. white space זה לא יוכנס ל- s, הוא יישאר בחוצץ הקלט.
ד. בתא שמעבר לתווים שנקראו למערך תשתול הפקודה את התו '\0' (התו הקרוי null)

על-כן אם תכניתנו מבצעת את הפקודה: cin >> s; , והמשתמש מזין בקלט את התווים ben-lulu yosi, אזי לתוך s יכנס הסטרינג ben-lulu. (המשך הקלט, הכולל את הרווח, את הסטרינג yosi, ואת התו סוף שורה, יישאר להמתין בחוצץ הקלט).

נניח שהגדרנו משתנה: char s[5];
נניח שבהמשך התכנית אנו קוראים לתוכו מחרוזת:

```
cin >> s;.
```

לבסוף, נניח שהקלט של המשתמש הוא: abcdefg
השאלה: מה יקרה?
התשובה: בשפה המכבדת את בעליה יקראו לתוך המשתנה רק ארבעה תווים (abcd), כדי לשמור מקום גם לַ- '\0' שאמור להישתל בתא האחרון; או

לחילופין: התכנית תועף. אולם שפת סי היא 'המערב הפרוע', בו מניחים שאם זה מה שהמתכנת רוצה לעשות, אזי זה גם מה שיש לעשות. על כן מה שיקָרה בפועל הוא שלתוך חמשת תאי המערך ייקראו הערכים a עד e; והערכים f, g כמו גם ה- '\0' שמוסף אחריהם יוכנסו לתאי זיכרון שמחוץ למערך! ובכך שלושת הערכים הללו עלולים 'לדרוך' על משתנים אחרים בתכנית. זה כמובן רע מאוד!! כדי להימנע מתקלה חמורה זאת נכתוב כל פקודת קלט באופן הבא:

```
cin >> setw(5) >> s ;
```

הפקודה setw(5) מורה למחשב שיש להגביל את מספר התווים שיקָראו ללכל היותר ארבעה (כדי לשמור מקום גם ל- '\0'). יש להקדים את הפקודה setw(...) לכל פקודת קלט של מחרוזת.

על מנת שהמהדר יכיר את הפקודה יש להוסיף לתכנית את הנחיית המהדר:

```
#include <iomanip>
```

באופן סימטרי לפקודת הקלט, עם מערך של תווים אנו רשאים לבצע את הפקודה:

```
cout << s;
```

אשר תציג את המחרוזת השמורה במשתנה s. נניח עתה כי גודלו של המערך s הוא 50 תווים. עוד נניח כי s כולל עתה את המחרוזת "ben-lulu". נשאלת השאלה: כיצד יודע המחשב, עת הוא מציג את המחרוזת, להציג רק את תכנם של שמונת התאים הראשונים (ולא את יתר 42 התאים, בהם קיים ערך 'זבל')? התשובה היא שכזכור לנו, עת המחשב קרא את המחרוזת "ben-lulu" לתוך s, הוא הציב בתא התשיעי (התא הפנוי הראשון, שאינו מכיל תו כלשהו מהמחרוזת שנקראה) את התו '\0' (התו הראשון ב- ASCII, שאינו תו גרפי, ולא ניתן להזינו מהמקלדת). בכך סימן לעצמו המחשב כי בזאת מסתיימים התאים 'שמעניינים אותנו' במערך, ובכל יתר תאי המערך יש לפי שעה 'זבל'. אנו אומרים כי המחרוזות היא: **null terminated** כלומר מסתיימת בתו null (התו מספר אפס ב- ASCII). פקודת ה- cout על-כן רצה על המערך עד שהיא נתקלת ב- '\0' (ואם היא לא תיתקל בתו זה אזי בתכנית יקרו דברים רעים...)

9.2.3 פונ' הספרייה strcpy, strlen, strcmp, strcat

שפת C מעמידה לרשותכם פונ' ספרייה רבות לטיפול במחרוזות. על-מנת שהקומפיילר יכיר פונ' אלה עליכם לכלול בתכניתכם את ההוראה: #include <cstring> (אשר תופיע, מטבע הדברים, לצד הוראות ה- include האחרות). בסעיף זה נמנה *חלק* מהפונקציות בהם אתם יכולים לעשות שימוש כדי לטפל במחרוזות.

לא ניתן להשתמש בפקודת ההשמה על מחרוזות (כלומר לא ניתן לכתוב: s = "yosi"); כדי להכניס מחרוזת למשתנה עליכם להשתמש בפונקציה strcpy. לדוגמה תוכלו לכתוב: strcpy(s, "yosi"); הפונקציה strcpy תעתיק את תווי המחרוזת "yosi" על תאי המערך s (החל בתא מספר אפס במערך). באחריותכם לדאוג ש- s יכיל די תאים כדי ש- strcpy תוכל להעתיק את המחרוזת בשלמותה. כמו כל הפונקציות שמעמידה לרשותכם השפה, גם strcpy תשים בסוף המחרוזת את התו '\0', לכן מספר תאי המערך צריך להיות גדול לפחות באחד מאורכה של המחרוזת שרוצים להכניס לתוכו. שימו לב לסדר הפרמטרים המועברים ל- strcpy: הפונקציה מכניסה לתוך הפרמטר הראשון את הפרמטר השני, סדר כתיבת הפרמטרים דומה לַסדר בו אנו כותבים אותם בפקודת ההשמה: ראשית המשתנה לתוכו משימים, ושנית הערך אותו יש לשים.

שפת C מעמידה לרשותכם גם את הפונקציה strlen. פונקציה זאת מקבלת מחרוזת (שהינה null terminated), ומחזירה את מספר התווים שהמחרוזת

כוללת, לא כולל ה- '\0' (שאינו חלק מהמחרוזת, הוא רק מציין כי כאן המחרוזת מסתיימת). הפונ' strlen, כמו כל הפונ' האחרות שהשפה מעמידה לרשותנו, מצפה לקבל מחרוזת שהינה null terminated. בהתאמה: כל פונקציות הספרייה עת מייצרות מחרוזת מייצרות מחרוזת שהינה null terminated.

נציג דוגמה קטנה אשר עושה שימוש בַפונקציות שהכרנו. נניח כי בתכנית הוגדרו:

```
char s[100], name[10] ;
```

עתה נוכל לכתוב את קטע התכנית הבא:

```
cin >> setw(100) >> s ;
if (strlen(s) < 10)
   strcpy(name, s) ;
```

הסבר: אנו קוראים מחרוזת לתוך המשתנה s. עתה אם המחרוזת שנקראה אינה ארוכה מדי (יש בה לכל היותר תשעה תווים), אנו מעתיקים אותה על המשתנה name, (מכיוון שהמערך name כולל עשרה תאים, ניתן לאחסן בתוכו מחרוזת בת לכל היותר תשעה תווים).

אין מניעה לטפל במערך של תווים הן באמצעות הכלים שמעמידה לרשותכם השפה, והן כבכל מערך אחר. הכלל היחיד עליו יש לשמור הוא שאם רוצים שהכלים שמעמידה לרשותכם השפה יפעלו כהלכה, אזי יש לדאוג כי המחרוזת השמורה במערך תהיה null terminated כלומר בסופה יופיע תמיד '\0'.
נראה דוגמה: נניח כי ברצוננו לקרוא מהמשתמש מחרוזת, ואחר להוריד מהמחרוזת את כל מופעיו של התו 'x'. (לדוגמה אם המחרוזת שקראנו הייתה: "abxcdxxd" אזי בתום תהליך הניקוי המחרוזת תהיה: "abcdd".) בתום התהליך יש להציג את המחרוזת.

```
cin >> setw(N) >> a_string ;

place = 0 ;
while (a_string[place] != '\0')
{
   if (a_string[place] == 'x')
      for (int i=place; a_string[i] != '\0'; i++)
         a_string[i] = a_string[i+1] ;
   if (a_string[place] != 'x')
       place++ ;
}

cout << a_string ;
```

הסבר: אנו קוראים את המחרוזת בַתחילה, ומציגים אותה בסיום, תוך שימוש בכלים לטיפול במחרוזות שמעמידה לרשותנו השפה, (איננו קוראים את המחרוזת בעצמנו תו, תו, ואיננו מציגים את תוכנה תו, תו). בין שתי פעולות אלה אנו מטפלים במחרוזת בעצמנו, באמצעות פקודות לולאה. הלולאות שלנו מסתמכות על כך שהמחשב, עת קרא את המחרוזת, שם בסופה את התו '\0'. הלולאות שלנו מצדן דואגות לכך שתו זה יישאר בסוף המחרוזת גם אחרי פעולתן (ולכן cout תוכל להציג את המחרוזת כהלכה). איני נכנס להסבר הלולאות עצמן שכן בהן אין לנו עניין בשלב זה; אני רק מסב את תשומת לבכם כי הלולאות פונות לתאי המערך a_string באופן זהה לטיפול במערכים כפי שמוכר לנו מימים ימימה, וכי הלולאות משאירות את המחרוזת null terminated.

על מחרוזות מוגדר יחס סדר הקרוי '**סדר לקסיקוגרפי**' (lexicographical order) או **סדר מילוני.** יחס סדר זה הוא יחס הסדר המוכר לכם עת אתם מחפשים מילה במילון. לדוגמה המילה: "abcd", קטנה מהמילה "abce" (הראשונה תופיע לפני השנייה במילון), והמילה "abcd" גדולה מהמילה "abc" ("abcd" תופיע אחרי "abc" במילון). שפת C מעמידה לרשותכם פונקציה אשר בודקת מה יחס הסדר בין שתי מחרוזות שונות. הפונקציה נקראת `strcmp`. היא מקבלת שתי מחרוזות ומחזירה אחד משלושה ערכים אפשריים:

א. הפונקציה תחזיר ערך שלילי כלשהו (על-פי חישקה) אם המחרוזת הראשונה קטנה (מבחינת סדר לקסיקוגרפי) מהמחרוזת השנייה.

ב. הפונקציה תחזיר את הערך אפס אם שתי המחרוזות שהועברו לה זהות.

ג. הפונקציה תחזיר ערך חיובי כלשהו (על-פי חישקה) אם המחרוזת הראשונה גדולה (מבחינת סדר לקסיקוגרפי) מהמחרוזת השנייה.

שימו לב כי יש משהו די מבלבל בערך ההחזרה של הפונ'. התנאי: `if (!strcmp(s1, s2))` שקול לתנאי: `if (strcmp(s1, s2) == 0)` ושניהם אומרים ששתי המחרוזות *שוות*, די בניגוד לאינטואיציה, לכל הפחות שלי, אשר מצפה שהערך `false` או אפס יוחזר עת אין שוויון, שעה שבפועל הוא מוחזק עת *יש* שוויון, שכן הפונ' אינה בולאנית: עליה להחזיר אחד משלושה מצבים אפשריים: הראשונה קטנה, השתיים שוות, הראשונה גדולה.

נראה דוגמה פשוטה לשימוש ב- `strcmp`. נכתוב קטע תכנית אשר קורא מחרוזת (לתוך מערך של תווים בשם `wanted`), ואחר עוד סדרת מחרוזות (לתוך המערך `curr`) עד קריאת הסטרינג ".". קטע התכנית יספור כמה פעמים הופיעה המחרוזת שהוזנה ראשונה בקרב המחרוזות שהוזנו אחריה:

```
cin >> setw(N) >> wanted;
counter = 0 ;

cin >> setw(N) >> curr ;
while (strcpy(curr, ".") != 0)
{
   if (strcmp(wanted, curr) == 0)
      counter++ ;
   cin >> setw(N) >> curr ;
}
```

הסבר: לפני הלולאה אנו: קוראים את המחרוזת המבוקשת, ומאפסים את המונה. כמו כן אנו קוראים את המחרוזת הראשונה מבין אלה שיש לבדוק את השוויון בינן לבין ה- `wanted` (כך נוכל שלא להיכנס ללולאה אף לא פעם אחת, אם המחרוזת שמזין המשתמש ל- `curr` כבר בפעם הראשונה היא "."). הלולאה מתנהלת כל עוד ערכו של `curr` שונה מהמחרוזת ".", כלומר כל עוד `strcmp` עת מקבלת את `curr` ואת הסטרינג "." מחזירה ערך שונה מאפס (להזכירכם ערך מוחזר אפס מעיד על שוויון של המחרוזות שהועברו ל- `strcmp`). בגוף הלולאה אנו בודקים האם המחרוזת השמורה במשתנה `curr` שווה לסטרינג המצוי במשתנה `wanted`, ואם כן מגדילים את ערכו של המונה באחד, ולקראת סיבוב נוסף בלולאה קוראים מחרוזת חדשה.

פונקצית ספרייה נוספת שמעמידה לרשותנו השפה היא הפונקציה `strcat`. פונקציה זאת מקבלת שתי מחרוזות, **ומשרשרת** (concatenation, במילים אחרות

מוסיפה) את המחרוזת השנייה בהמשכה של המחרוזת הראשונה. לדוגמה: אם ערכו של s1 הוא: "abc" וערכו של s2 הוא: "xy" אזי אחרי הקריאה ל: strcat(s1, s2) יכיל s1 חמישה תווים: abcxy (ואחריהם כמובן '\0'). כמו בכל המקרים האחרים באחריותכם לדאוג לכך ש- s1 יהיה גדול דיו כדי להכיל את תוצאת השרשור. שימו לב כי שתי המחרוזות משורשרות ברצף (בלי כל רווח ביניהן). אם אתם חפצים בכך אתם יכולים לכתוב גם את זוג הפקודות הבא:

```
strcat(s1, " ") ;
strcat(s1, s2) ;
```

פקודות אלה ראשית שרשרו בסופו של s1 את המחרוזת הכוללת את התו רווח בלבד, ואחַר למחרוזת שהתקבלה שורשר s2. באופן כזה בין שתי המחרוזות הוכנס רווח.

אעיר כי השפה מעמידה לרשותכם פונ' ספרייה רבות עבור מחרוזות, כאן תיארתי רק קמצוץ מתוכן. אתם מוזמנים לחפש וללמוד בעצמכם פונ' נוספות (לדוגמה: atoi, strstr ואחרות).

9.2.4 מערך של מחרוזות

אמרנו כי למערך של תווים ניתן, ואף נהוג, להתייחס כאל מחרוזת. באופן דומה למערך דו-ממדי של תווים אנו יכולים להתייחס כאל מערך חד-ממדי של המחרוזות: כל שורה בַמערך הדו-ממדי מכילה מחרוזת יחידה. נציג עתה קטע תכנית שיעשה שימוש במערך כנ"ל: קטע התכנית קורא סדרה של המחרוזות (עד מילוי המערך או קבלת המחרוזת ".") המחרוזות הנקראות מוכנסות לתוך מערך של מחרוזות שמוחזק ממוין (בסדר לקסיקוגרפי). כמו כן נרצה שמחרוזת אשר מוזנת מספר פעמים על-ידי המשתמש לא תוכנס מספר פעמים למערך (אלא תופיע במערך פעם יחידה בלבד).

ראשית נגדיר בתכנית הראשית את המשתנים הדרושים (נניח כי הקבועים המתאימים הוגדרו):

א. המערך שיכיל את המחרוזות: `char dict[MAX_STRS][MAX_STR_LEN];`
(המערך יוכל להכיל לכל היותר MAX_STRS מחרוזות, כל אחת באורך קטן מ-MAX_STR_LEN).

ב. מונה שיציין כמה מחרוזות הוספו למילון: `int dict_size = 0;`

ננצל את הדוגמה גם כדי לחזור שוב על עקרונות התכנות המודולארי, ועל עיצוב מעלה-מטה. התכנית הראשית תזמן פונ' לקריאת הקלט:

```
read_dict(dic, dict_size) ;
```

מין הסתם, בַהמשך, תשתמש התכנית הראשית במילון שהוזן, אולם אנו לא נתעניין בהמשך התכנית הראשית, אלא נסתפק בפונ' read_dict.

הפונ' read_dict בלולאה תקרא מהמשתמש מחרוזות עד מלוי המערך, או עד קליטת המחרוזת "end" באמצעות קלט זה יאותת המשתמש כי בזאת הוא סיים להזין את המילון.

עבור כל מחרוזת שהתכנית תקרא, ראשית עליה לבדוק האם המחרוזת כבר מצויה במערך (ואז אין להוסיפה שוב למילון). לשם ביצוע תת-משימה זאת נזמן פונ' בשם search. במידה והמחרוזת טרם הוספה למילון אזי יש להוסיפה, ולשם כך נזמן את הפונ' insert.

הפונ' search, כאמור, צריכה לקבל את מערך המחרוזות, את מספר המחרוזות במערך (כדי לדעת כמה תאים במערך כבר מצויים בשימוש; רק בתאים אלה יש

טעם לחפש את המחרוזת שהוזנה עתה), ואת המחרוזת שהוזנה (ושיש לחפש במילון). הפונ' תחזיר האם המחרוזת כבר מצויה במילון או לא; מעבר לכך, כדי לייעל את תהליך הוספת המחרוזת למילון (באמצעות insert), תחזיר הפונ' search גם את מספר התא במערך המחרוזות (כלומר את מספר השורה במערך הדו-ממדי) אליו יש להוסיף את המחרוזת עת היא לא מצויה במילון; או את מקומה של המחרוזת במילון, אם היא כבר מצויה בו.

קוד הפונ' read_dict:

```
void read_dict(char dict[][MAX_STR_LEN],
               int &dict_size)
{
   const char END[] = "end" ;
   char current[MAX_STR_LEN] ;
   int place ;

   dict_size = 0 ;
   cin >> setw(MAX_STR_LEN) >> current ;

   while (dict_size < MAX_STRS &&
          strcmp(current, END) != 0)
   {
         if (!search(dict, dict_size, current, place))
            insert(dict, dict_size, current, place) ;
         cin >> setw(MAX_STR_LEN) >> current ;
   }
}
```

<u>הסבר</u>: הפונ' קוראת מחרוזות בלולאה עד מילוי המילון, או עד קריאת המחרוזת end. עבור כל מחרוזת היא ראשית מחפשת האם המחרוזת מצויה במילון, וזאת עושה הפונ' search. search מחזירה ערך בולאני: האם המחרוזת כן\לא נמצאת כבר במילון. כמו כן באמצעות הארגומנט place מוחזר המקום בו המחרוזת כבר מצויה, או המקום אליו יש להכניס את המחרוזת.

נפנה עתה לכתיבת הפונקציה search. פונקציה זאת מקבלת את מערך המחרוזות, ואת המחרוזת שעליה לחפש. היא מחזירה באמצעות פקודת return איתות האם המחרוזת המבוקשת נמצאת במילון או לא. במידה והמחרוזת נמצאת מוחזר בפרמטר where התא בו היא נמצאת. במידה והמחרוזת לא נמצאת מציין הפרמטר where לאיזה תא במערך יש להוסיף את המחרוזת.

מכיוון שמערך המחרוזות מוחזק ממוין, אזי עת מחפשים בו מחרוזת מבוקשת ראוי מאוד, מאוד לעשות זאת באמצעות חיפוש בינארי. מכיוון שאני חושש שלא כולם זוכרים כיצד מתנהל החיפוש הבינארי, ומכיוון שלא זה המוקד של הדוגמה הנוכחית, וכדי שתראו שגם אני לעתים עושה דברים מאוד לא ראויים, אזי קוד הפונ' שנציג לא ישתמש בחיפוש בינארי, אלא יסרוק את מערך המחרוזות סדרתית עד מציאת המחרוזת הרצויה, או עד הגעה למחרוזת גדולה ממנה (בסדר מילוני) ואז הדבר מורה שהמחרוזת לא מצויה במערך:

```
bool search(const char dict[][MAX_STR_LEN],
            const char a_str[],
            int str_counter,
            int &where)
```

```
{
   for (where = 0;
        where < str_counter &&
                strcmp(a_str, dict[where]) > 0;
        where++)
        ;

   if (where < str_counter &&
       strcmp(a_str, dict[where]) == 0)
         return(true) ;
   return(false) ;
}
```

נסביר: ראשית, ברשימת הפרמטרים השמטנו מחלק מהמערכים את ציון גודלם של חלק מהמימדים. אני מזכיר לכם כי שפת C מאפשרת לכם להשמיט את ציון גודלו של המימד הראשון. הסיבה לכך היא שמערך N מימדי הוא למעשה סדרה של מערכים N-1 מימדיים; כלומר זהו מערך חד-ממדי שכל תא בו הוא מערך N-1 מימדי. שפת C מאפשרת לכם שלא לציין כמה תאים יש בַּמערך החד-ממדי המועבר לפונקציה; היא מחייבת אתכם לציין מה גודלו של כל תא במערך, ולכן אם אתם מעבירים מערך שכל תא בו הוא מערך N-1 מימדי, יש לציין את גודלם של N-1 הממדים האחרונים.

אני גם מסב את תשומת לבכם כי תיאור שני הפרמטרים הראשונים מתחיל במילה const. בכך אנו קובעים כי עבור הפונקציה search (אשר רק אמורה לחפש את הפרמטר השני בקרב הפרמטר הראשון), שני פרמטרים אלה הם קבועים, כלומר הפונקציה לא רשאית לשנותם. ניסיון לכתוב בגוף הפונקציה השמה כגון: a_str[0]='!'; יגרום לשגיאת קומפילציה.

מעבר לכך קוד הפונקציה פשוט למדי: בלולאת ה- for, אנו סורקים את המערך עד מיצוי התאים שמכילים מחרוזות קיימות או עד הגעה למחרוזת גדולה או שווה מזו שאנו מחפשים. בגוף הלולאה איננו עושים דבר (גוף הלולאה כולל את הפקודה הריקה.) אחרי הלולאה אנו בודקים: אם טרם סיימנו לעבור על כל המחרוזות שבמילון (where < str_counter) וכן המחרוזת עימה יצאנו מהלולאה שווה לזו שאנו מחפשים (strcmp(a_str, dict[where]) == 0) אזי אנו מחזירים את הערך true (והמשתנה whereכבר מכיל את הערך הדרוש). אם התנאי הנ"ל לא התקיים, ועל כן לא עפנו החוצה מהפונ' תוך כדי החזרת הערך true, אזי אנו מתדרדרים להחזרת הערך false.

אעיר כי התנאי:

```
if (where < str_counter &&
       strcmp(a_str, dict[where]) == 0)
```

מכיל גם את המרכיב: where < str_counter עבור המקרה בו יש להוסיף למילון מחרוזת הגדולה מכל המחרוזות הקיימות כבר במילון. לדוגמה: נניח שהמילון ריק, כלומר ערכו של str_counter הוא אפס. במקרה זה לא ניכנס ללולאת ה- for כלל. נגיע לתנאי בו אנו דנים ושם נשאל: האם where < str_counter התשובה תהיה לא, על כן גם לא נרצה להשוות בין המחרוזת הנוכחית לבין dict[where], שכן תא זה במערך המחרוזות מכיל ערך זבל, ונרצה להחזיר ערך false, וזה בדיוק מה שאנו עושים.

עתה נציג את הפונקציה insert:

```
void insert(char dict[][MAX_STR_LEN],
```

```
            int counters[],
            const char a_str[],
            int &str_counter,
            int where)
{
   for (int place = str_counter; place > where; place--)
      strcpy(dict[place], dict[place -1]) ;

   strcpy(dict[where], a_str) ;
   str_counter++ ;
}
```

קוד הפונ' פשוט למדי: אנו רצים על מערך המחרוזות מסופו כלפי התא אליו יש להכניס את המחרוזת הנוכחית (התא מספר `where`). בכל סיבוב בלולאה אנו מעתיקים (מסיטים) מחרוזת שכבר מצויה במערך תא אחד ימינה (או למטה, איך שנח לכם לחשוב על המערך), ובכך אנו 'פותחים' מקום למחרוזת החדשה. לבסוף, מחוץ ללולאה אנו מוסיפים את המחרוזת החדשה למילון, ולא שוכחים להגדיל את מונה המחרוזות באחד.

לסיכום, שאלה למחשבה: נניח כי פעולה אטומית (בפרט פעולת השוואה) מתבצעת על תו בודד, מהו זמן הריצה של הפונקציה `read_n_sort`?

cin.getline 9.2.5

פונקציה נוספת שמעמידה לרשותכם השפה (`c++` בלבד, אך לא `C` טהור), ושעשויה להיות לכם לעזר היא `cin.getline`. ראשית נכיר גרסה של הפונקציה המקבלת שני פרמטרים: הראשון הוא מערך של תווים, השני הוא מספר שלם המציין את מספר התאים במערך שהועבר בארגומנט הראשון. הפונקציה קוראת מחרוזת לתוך המערך עד אשר:

א. המערך מתמלא (הפונקציה דואגת להכניס בסוף המערך את התו `'\0'`, ולכן מספר התווים שיקראו יהיה לכל היותר קטן באחד מערכו של הפרמטר השני).

<u>או</u>

ב. הפונקציה נתקלת בַּתו סוף-שורה. תו זה 'נאכל' על-ידי הפונקציה (כלומר הוא מוסר מחוצץ הקלט) אך אינו מוכנס לַמערך.

לדוגמה: אם הגדרנו: `char s[15];` אזי אנו רשאים לקרוא לַפונקציה: `cin.getline(s, 15);`. אם הקלט יהיה: `yosi cohen` ואחר יקיש המשתמש על מקש ה- Enter, אזי לתוך עשרת תאי המערך הראשונים יוכנס הסטרינג "yosi cohen", לתוך התא האחד-עשר יוכנס התו '\0', וארבעה תאים יישארו ללא שימוש. אם לעומת זאת הקלט יהיה: 12345678901234567890 ואחר יקיש המשתמש על מקש ה- Enter, אזי לתוך ארבעה-עשר התאים הראשונים במערך יוכנס הסטרינג "12345678901234", ולתוך התא האחרון יוכנס התו '\0'.

ניתן לזמן את הפונ' cin.getline גם עם שלושה ארגומנטים (ולא עם שניים) כפי שתיארתי מעל. שני הארגומטים הראשונים יהיו כמו קודם, והארגומנט השלישי יהיה תו. הפונ' תפעל כמו קודם, אולם במקום לעצור על סוף שורה, הפעם היא תעצור על התו שהועבר לה בפרמטר השלישי.

<u>לדוגמה</u>: נניח שהמערך s הוגדר: `char s[15];` ואנו מזמנים:

```
cin.getline(s, 15, '?') ;
```

עוד נניח כי הקלט של המשתמש הוא:

```
12345?67
```

89?
אזי למערך s יכנסו התווים 12345 (ואחריהם, כמובן, יוכנס ה- '\0' לתא מספר חמש). כמו כן ה- ? 'יאכל' מחוצץ הקלט ויישלח לאבדון. פקודת קלט נוספת:

```
cin.getline(s, 15, '?') ;
```

תכניס למערך את התווים: 6, 7, סוף-שורה, 8, 9 (כלומר ביוניקס חמישה תווים). שימו לב שעתה התו סוף שורה אינו שורה במעמדו מכל תו אחר, הנקלט ומוכנס למערך; עתה יש מעמד מיוחד לתו '?' ורק לו.

9.2.6 אתחול של מחרוזות

ראינו כבר כי אנו יכולים לאתחל סטרינג בעת הגדרתו באופן הבא:

```
char name[50] = "yosi cohen" ;
```

ניתן גם להגדיר סטרינג באופן הבא: `. char her_name[]= "dina levi";` הפעם לא ציינו את גודלו של המערך, לכן הקומפיילר יקצה עבורנו את המערך המזערי אשר יהיה גדול דיו להכיל את הסטרינג (כולל '\0'). במקרה שלנו יוקצה מערך בן עשרה תאים. בהמשך נוכל להשתמש במערך זה כרצוננו.

9.2.6 מימוש הפונ' strcmp

לפני סיום נשאל את עצמנו כיצד נראה הקוד של הפונקציה strcmp? נשאל וגם נציג תשובה אפשרית:

```
int strcmp( char s1[], char s2[]) {
   int i;

   for (i = 0 ;
        s1[i] == s2[i]  &&
        s1[i] != '\0' && s2[i] != '\0';
        i++)
     ;

   return(s1[i] - s2[i]) ;
}
```

נסביר: אנו סורקים את שני הסטרינגים כל עוד התווים המצויים בשניהם זהים, וכן כל עוד הם לא הסתיימו. לולאת ה- for הכוללת את הפקודה הריקה עושה עבורנו את העבודה. אחרי הלולאה אנו מחזירים את ההפרש בין התאים אליהם הגענו: אם בשתי המחרוזות הגענו ל: '\0' אזי נחזיר אפס, שכן הן שוות, ואחרת תוצאת ההפרש תהייה חיובית או שלילית על פי התווים אליהם הגענו בכל אחת משתי המחרוזות.

אתם מוזמנים לכתוב בכוחות עצמכם גם את strlen, strcat. אני מזכיר לכם שפונ' אלה כבר עומדות לרשותכם ואל לכם לממשן בתכניות שאתם כותבים. מימשנו אחת מהן כאן רק לצורך התרגול.

9.2.7 argc, argv

בכל התכניות שכתבנו עד כה הוגדרה התכנית הראשית בצורה: `; int main()` כלומר לתכנית הראשית לא היו כל פרמטרים. בסעיף זה נציג את הפרמטרים *היחידים* אשר ניתן להעביר ל- main. (אעיר כי בניוד לכל פונ' אחרת, לה אתם יכולים לקבוע את הפרמטרים על-פי שיקול דעתכם, התכנית הראשית יכולה לא לקבל פרמטרים כלל, כפי שעשינו עד היום, או לקבל פרמטרים בדיוק כפי שנתאר בסעיף זה.)

נציג את הנושא באמצעות דוגמה: נניח שברצוננו לכתוב תכנית פשוטה הקוראת שתי מחרוזות ומודיעה האם המחרוזת הראשונה קטנה בסדר מילוני מהמחרוזת השנייה, שווה לה, או גדולה ממנה. התכנית הראשית תהיה פשוטה למדי:

```
int main()
{
      char s1[N], s2[N] ;

      cin >> setw(N) >> s1
            >> setw(N) >> s2 ;
      if (strcmp(s1, s2) < 0)
            cout << "first smaller\n" ;
      else if (strcmp(s1, s2) == 0)
            cout << "both are equal\n" ;
      else
           cout << "first greater\n" ;

      return(EXIT_SUCCESS) ;
}
```

אעיר כי ניתן לטעון שמטעמי יעילות עדיף לזמן את `strcmp` פעם יחידה, להכניס את הערך המוחזר על-ידה למשתנה, ואחר לבדוק את ערכו של המשתנה, ולא לזמן את הפונ' פעמיים. זו תהיה טענה נכונה, אולם לא היא המוקד של הדוגמה הנוכחית.

נניח שקימפלנו את התכנית, וקובץ ההרצה (הקובץ המכיל את תרגום התכנית לשפת מכונה) נקרא `my_prog`. הרצת התכנית תהיה באופן הבא:

```
my_prog
yosi YOSI
```

כלומר יהיה עלינו (ב- `Shell` הטקסטואלי של מערכת ההפעלה) להקליד את שם התכנית, אחר-כך `Enter` ואז את שני הקלטים (שבדוגמה הנוכחית הם `yosi YOSI`).

זוהי הרצה אפשרית של התכנית אולם אנו עשויים למצוא בה טעם לפגם: בד"כ עת אנו מריצים פקודות בַּ- `Shell` איננו מקישים ראשית את הפקודה (כלומר את התכנית שיש להריץ), ורק אחר-כך את הקלט לה, אלא אנו מקלידים את הפקודה עם הקלט לה, ובסוף `Enter` יחיד. לדוגמה, אנו כותבים: `cp file1 file2` על מנת להעתיק את `file2` על גבי `file1`; איננו כותבים:

```
cp
file1 file2
```

נרצה שגם התכנית שאנו נכתוב, אשר משווה שתי מחרוזות, תתנהג באותו אופן, כלומר שהרצתה מה- `Shell` תהיה: `my_prog yosi YOSI`
כיצד נשיג את המבוקש?

התשובה היא שלשם כך על התכנית לקבל פרמטרים. בדוגמה שלנו את שתי המחרוזות שיש להשוות. תכנית בשפת `C/C++` עשויה לקבל מספר כלשהו של ארגומנטים (כפי שמקליד מי שמריץ את התכנית) *הארגומנטים יהיו בהכרח מחרוזות*. כלומר אם מי שמריץ את התכנית מקליד ב- `Shell` את הפקודה:

```
my_prog yosi YOSI
```

אזי התכנית מקבלת שני ערכים. אם לעומת זאת מי שיריץ את התכנית יקליד:

```
my_prog
```

אזי התכנית אינה מקבלת כל ערך עת היא מתחילה לרוץ (היא יכולה, כמובן, תוך כדי ריצתה, לקרוא קלטים מהמשתמש).
לבסוף, אם ההרצה היא:

```
my_prog yosi 17 YOSI x
```

אזי התכנית מקבלת ארבעה ארגומנטים; כולם, כאמור, בהכרח מחרוזות (כמובן שקבלת ארגומנטים אינה מונעת מהתכנית, אם היא זקוקה לכך, גם לקרוא קלט בהמשך).

כיצד תראה התכנית שלנו, אשר מסוגלת לקבל ארגומנטים משורת הפקודה?
פרוטוטיפ התכנית הראשית יהיה:

```
int main(int argc, char **argv)
```

או לחילופין:

```
int main(int argc, char *argv[])
```

שתי צורות הכתיבה שקולות זו לזו. השם `argc` הוא קיצור של: `argument counter` כלומר מונה הארגומנטים; השם `argv` הוא קיצור של `argument vector` כלומר וקטור הארגומנטים.

<u>נסביר:</u> הכוכבית המופיעה בתוך הסוגריים, לצד הפרמטר `argv` אינה מוכרת לנו לעת עתה. נכיר אותה עת נדון במצביעים. בשלב זה נסתפק בכך שנאמר שהפרמטר `argv` הוא מערך של מחרוזות. כמה תאים יש במערך? כלומר, כמה מחרוזות הועברו לתכנית עם הרצתה? על כך מורה לנו הפרמטר הראשון: `argc`.

פרט נוסף שעלינו לדעת: תמיד, המחרוזת הראשונה ב- `argv`, כלומר התא `argv[0]` מחזיק את שם קובץ ההרצה (בדוגמות שלנו הוא מחזיק את `my_prog`).

על כן, אן התכנית שלנו הורצה באופן:

```
my_prog
```

אזי ערכו של `argc` הוא 1, כלומר מערך המחרוזות `argv` מכיל רק מחרוזת יחידה שהינה כפי שתיארתי מעל.
אם, לעומת זאת, התכנית שלנו הורצה באופן:

```
my_prog yosi 17 YOSI x
```

אזי ערכו של `argc` הוא חמש, והמערך `argv` נראה באופן הבא:

```
argv[0] = "my_prog"
argv[1] = "yosi"
argv[0] = "17"
argv[0] = "YOSI"
argv[0] = "x"
```

נחזור עתה לתכנית עימה פתחנו את הסעיף (תכנית המשווה שתי מחרוזות): התכנית שלנו, על מנת שניתן יהיה להריצה בצורה: `my_prog yosi YOSI` אמורה לקבל שתי מחרוזות, על כן `argc` בה אמור להיות שלוש (נזכור ששם התכנית מועבר בכל מקרה). המחרוזות ישבו בתאים `argv[1], argv[2]`.

בתכנית המצפה לקבל ארגומנטים באמצעות וקטור הארגומנטים נהוג בתחילת התכנית לבדוק האם הועברו ארגומנטים כמצופה, ובמידה ולא לתת הודעת שגיאה קצרה המתארת כיצד יש להפעיל את התכנית:

```
if (argc != 3)
{
   cerr << "Usage: " << argv[0]
        << "<string1> <string2>\n" ;
```

```
   return(EXIT_FAILURE) ;
}
```

הנוהג לגבי פורמט הודעת השגיאה הוא לפתוח אותה במילה: `Usage` ואז להציג את שם התכנית (כפי שמוחזק ב- `argv[0]`), ואח"כ את הארגומנטים שיש להעביר לתכנית, עטופים בסוגרי זווית (<>), בדוגמה שלנו יש להעביר שתי מחרוזות, ולכן זה מה שאנו מסבירים.

נציג עתה את התכנית הראשית בשלמותה:

```
int main(int argc, char *argv[])
{
    char s1[N], s2[N] ;

    if (argc != 3)
    {
       cerr << "Usage: " << argv[0]
            << "<string1> <string2>\n" ;
       return(EXIT_FAILURE) ;
    }
    strcpy(s1, argv[1]) ;
    strcpy(s2, argv[2]) ;
    int cmp_res = (int) strcmp(s1, s2) ;

    if (cmp_res < 0)
           cout << "first smaller\n" ;
    else if (cmp_res == 0)
           cout << "both are equal\n" ;
    else
          cout << "first greater\n" ;

    return(EXIT_SUCCESS) ;
}
```

9.3 תרגילים

9.3.1 תרגיל מספר אחד: הורדת תיעוד מתכנית, וספירת מילים המופיעות בה

כתבו תכנית הקוראת תכנית בשפת C++ ומדפיסה את התכנית ללא הערות תיעוד. תוספת אפשרית: התכנית גם תציג כמה פעמים הופיעה כל מילה בתכנית. הרשימה תוצג בסדר יורד של מספר הפעמים בהם הופיעה כל מילה.

הגדרה: מילה הינה רצף של תווים המתחיל בקו תחתון או באות, וכולל אותיות ספרות וקווים תחתונים.

9.3.2 תרגיל מספר שתיים: ספירת מילים בטקסט

כתבו תכנית הקוראת טקסט ומציגה כמה פעמים הופיעה כל מילה בטקסט. יש להציג את הפלט ממוין לפי סדר לקסיקוגרפי.

מילה תחשב לרצף של אותיות. הופעת תו שאינו אות יגרום לתכנית להסיק כי בזאת תמה מילה. התכנית תתעלם מסטרינגים שאינם מילים. לדוגמה בטקסט הבא:
yosi 123 cohen-levi lev 51lev?
מופיעות המילים yosi, cohen, levi פעם אחת, המילה lev מופיעה פעמיים.

9.3.3 תרגיל מספר שלוש: הצפנת ופיענוח טקסט

בתרגיל זה עליכם לכתוב זוג תכניות: אחת מהן מצפינה טקסט, השניה מפענחת אותו.

התכנית הראשונה תקרא מהמשתמש קלט המורכב משני חלקים:

א. מפתח הצפנה שהינו סדרה של 26 תווים. התו הראשון שייקרא יהיה זה שיצפין את האות a, התו השני יצפין את b, וכך הלאה. יש לוודא שכל תו מופיע במרכיב זה של הקלט פעם יחידה, כלומר שכל תו מצפין רק אות אחת.

ב. טקסט אותו על התכנית להצפין. כל אות בתחום a..z תוחלף בתו שמצפין אותה, תווים אחרים יוותרו בלא שינוי. התכנית תדפיס את הטקסט המוצפן.

התכנית השניה שתכתבו תפענח טקסט מוצפן. התכנית תפעל באופן הבא:

א. התכנית תקרא מהמשתמש טקסט שאינו מוצפן ותספור כמה פעמים מופיעה כל אות בטקסט הגלוי. במהלך תהליך הקריאה תייצר התכנית טבלת שכיחות אותיות שתמוין בסדר יורד של שכיחות הופעת האותיות השונות. לדוגמה: אם הקלט הנקרא הינו: yosi is nice אזי תיוצר הטבלה הבאה: i מופיע 3 פעמים, s מופיעה פעמיים, y,o,n,c,e מופיעים פעם יחידה.

ב. התכנית תקרא פעם ראשונה את הטקסט המוצפן ותבנה גם עבורו טבלת שכיחות אותיות. לדוגמה: אם הטקסט המוצפן הינו: ab bbc a אזי תיוצר הטבלה: b מופיע שלוש פעמים, a מופיע פעמיים, c מופיע פעם יחידה.

ג. התכנית תניח כי: האות שמופיעה מספר מרבי של פעמים בטקסט המוצפן מצפינה את האות שמופיעה מספר מרבי של פעמים בטקסט הגלוי (בדוגמה שלנו b מצפינה את i), האות שמופיעה בשכיחות השניה בטקסט המוצפן מקודדת את האות שמופיע בשכיחות השניה בטקסט הגלוי (בדוגמה שלנו: a מצפינה את s), וכך הלאה. באופן זה תנחש תכנית הפענוח את קוד ההצפנה.

ד. תכנית הפענוח תקרא שנית את הטקסט המוצפן ותפענח אותו על-פי קוד ההצפנה שהיא נחשה בשלב הקודם. התכנית תציג את תוצאת הפענוח.

9.3.4 תרגיל מספר ארבע: מספרים גדולים

בתכנית זאת עלינו לטפל במספרים טבעיים גדולים מאוד, בני עשרות ספרות. כדי לעשות זאת נייצג כל מספר באמצעות מערך של תווים; כל תא במערך יכיל ספרה של המספר. לדוגמה: המספר 123 ייוצג באמצעות המערך הבא:

3	2	1	0

0#	1#	2#	3#

כלומר, ספרת אחדות של המספר תשמר בתא מספר אפס, ספרת העשרות בתא מספר אחד, וכן הלאה. בתאים בהם אין שימוש יישמר הערך אפס.

מבנה הנתונים המרכזי של תכניתכם יהיה, לפיכך, מערך דו-ממדי של תווים. כל שורה במערך תאחסן מספר טבעי יחיד.

התכנית תאפשר למשתמש לבצע את הפעולות הבאות:

1. הזנת מספר נוסף לרשימת המספרים שהתכנית מחזיקה.
2. חיבור שני מספרים שהוזנו והצגת סכומם. על המשתמש יהיה להזין את קוד הפעולה הרצוי ואת מספרי השורות במערך בהם שמורים שני האופרנדים.
3. חיסור שני מספרים שהוזנו והצגת הפרשם.
4. כפל שני מספרים שהוזנו והצגת המכפלה.
5. חילוק שני מספרים שהוזנו והצגת המנה והשארית.
6. בדיקה האם המספר ראשוני. (באמצעות אלגי הסתברותי?).
7. הצגת רשימת המספרים.
8. סיום.

לכל פעולה מבוקשת נקבע קוד באמצעותו יבקש המשתמש לבצע את הפעולה המתאימה:
הזנה = a, חיבור= +, חיסור= -, כפל = *, חילוק= /, בדיקת ראשוניות= p, הצגת המספרים= d, סיום= e.

ניתן להניח כי הזנת כל סטרינג תסתיים ב- white space. במידה והסטרינג כולל תווים שאינם ספרות יש להודיע כי הוזן קלט שגוי, ולחזור לתפריט, ממנו יוכל המשתמש לבחור פעולה כרצונו.

9.3.5 <u>תרגיל מספר חמש: התאמה חלקית של מחרוזות</u>

ממשו את הפונקציה הבאה:

```
int inexact_pos(const char pattern[], const char text [],
                int from_pos, const int mismatches);
```

הניחו כי שני הקלטים הראשונים מכילים מחרוזות חוקיות (null terminated). שני הקלטים האחרים יכולים להכיל שלמים כלשהם.

עליכם לחפש בתוך `text` החל מהמקום `from_pos` (כאשר אינדקס התו הראשון הוא אפס), תת-מחרוזת שאורכה בדיוק כאורך תווי pattern (למעט '\0') <u>ובדיוק</u> mismatches תווים בה שונים מהתווים המתאימים בַּ- pattern. אם הקלט אינו חוקי או אם לא מצאתם תת מחרוזת מתאימה החזירו (-1), אחרת החזירו את האינדקס האבסולוטי של תחילת תת-המחרוזת הראשונה ב text. לדוגמא:

```
   inexact_pos("goo", "foobarfoo", 1, 1) should return 6.
inexact_pos("foo", "foobarfoo", 0, 3) should return 2.
inexact_pos("foo", "foobarfoo", 0, 1) should return (-1).
inexact_pos("foobarfoo", "foo", 3, 0) should return (-1).
```

9.3.6 <u>תרגיל מספר שש: פלינדרום מקסימלי</u>

ממשו את הפונקציה הבאה:

```
bool maximal_palindrome (const char s[],
                         int &from_pos, int &length);
```
עליכם לחפש בתוך המחרוזת s פלינדרום (כלומר תת מחרוזת הנקראת באופן זהה משמאל לימין ומימין לשמאל, דוגמת: flippilf או oh-ho) מאורך מקסימלי. הניחו כי s כולל מחרוזת חוקית.
הערות והנחיות:

א. בין מספר פתרונות מאורך שווה העדיפו את הקרוב יותר לתחילת s (אין צורך למיין!).

ב. פלינדרום טריביאלי, מאורך אחד, אינו תשובה מותרת.

ג. במידה ולא מצאתם תת מחרוזת מתאימה החזירו false לקורא, אחרת החזירו true ועדכנו את תוכנו של from_pos להיות אינדקס תחילת תת המחרוזת בתוך s , ואת תוכנו של length לאורכה.

ד. חשבו את משך זמן ריצת האלגוריתם שמימשתם כלומר את מספר פעולות ההשוואה המבוצעות) כפונקציה של n, עבור n שהינו הוא אורך מחרוזת הקלט s. הניחו לשם פשטות כי מספר ההשוואות עבור תת מחרוזת מאורך m הוא m/2.

9.3.7 תרגיל מספר שבע: חילוץ מספרים מטקסט

בתכנית זאת עליכם לקרוא סדרת שורות של תווים מהקלט, ועבור כל שורה להציג את המספרים שהופיעו באותה שורה.
המספרים שעשויים להופיע בקלט יכולים להיות מאחת מחמש תבניות:

א. תבנית [A] שצורתה רצף של ספרות. לדוגמה: 0, 123, 99, 00.

ב. תבנית [B] שצורתה: [A]- או [A]+, כלומר רצף של ספרות ולפניהן התו '+' או התו '-'. לדוגמה: +0, -0, +123, -00.

ג. תבנית [C] שצורת: [B] או [A].[B], כלומר מספר שלם או מספר ממשי המוצג עם נקודה עשרונית. לדוגמה: +00.00, 123.123, 45, -45.123.

ד. תבנית [D] שצורתה: [C] או [C]E[B] או [C]e[B] לדוגמה: -45.123E+123 , +17e-3879. ערכו של מספר המוצג בתבנית זאת הוא: $[C]*10^{[B]}$. לדוגמה: ערכו של –45.23E-3 הוא $-45.23*10^{-3}$ כלומר: 0.04523 =0.001*45.23.

שימו לב כי התו רווח אינו חלק לגיטימי מכל תבנית שהיא, ועל כן 23 .45 יובן כזוג מספרים שלמים 23, ואחריו 45, באופן דומה גם 1E 2 יובן כ: 1 ואחריו 2.

כל המספרים יופיעו בפלט בצורה עשרונית רגילה עם דיוק של לכל היותר שלוש ספרות מאחורי הנקודה (כלומר 45.23E-3– יוצג כ: 0.045). אפס יופיע תמיד בהצגה: 0.

כל שורה בפלט תכיל את מספר שורת הקלט המתאימה, ואחר את המספרים שהופיעו באותה שורה. תווים אחרים שאולי הופיעו בשורת הקלט לא יופיעו בפלט.

לדוגמה, הקלט הבא:

```
abc+-23E+01.12$
yosi cohen ..
23.4e 17
```

יגרום להצגת הפלט:

```
line 1: -230 12
line 2:
```

line 3: 23.4 17

10 קלט פלט וטיפול בקבצים בשפת C++

10.1. הקדמה

לצורך הפרק הנוכחי יהיה לנו מועיל (גם אם לא הכרחי, כפי שאסביר בהמשך), להניח את ההנחה הבאה: נניח שאנו עובדים על תכנית שצריכה ראשית לקרוא עשרה מספרים מהמשתמש, ושנית לבצע עליהם עיבוד כלשהו (למשל לאתר את שונות המספרים). במהלך העבודה על התכנית, בשלבי ה- debugging שלה, עלינו להריצה שוב ושוב, ובכל הרצה, כמובן, לספק לה עשרה מספרים. בכך יש משהו מוגיע. על כן, כדי להקל עלינו, נעדיף לעבוד באופן הבא: נניח שקובץ המקור בו שוכנת תכניתנו נקרא prog.cc (ביוניקס): או (בחלונות): prog.cpp, ואנו מקמפלים אותו ומקבלים קובץ ניתן להרצה בשם prog (ביוניקס; או prog.exe בחלונות). עתה, בעזרת העורך, נצור קובץ שיכיל את עשרת המספרים עליהם ברצוננו להריץ את תכניתנו (בשורה אחת, או בכמה שורות, מופרדים ברווח זה מזה). הקובץ פשוט יכיל את עשרת הנתונים ותו לא. נניח שלקובץ זה נקרא inp.txt . שימו לב שאין כל מניעה לצור קובץ כזה בעזרת העורך (למרות שאנו רגילים בעזרת העורך לייצר רק קובצי מקור בשפת C++). אם עתה, ב- shell של יוניקס, או בחלון DOS אותו נפתח בחלונות, נפנה למדריך בו מצוי הקובץ הניתן להרצה, ושם נקליד את הפקודה: prog < inp.txt (אותה פקודה יש להקליד בשתי מערכות ההפעלה השונות) אזי יקרה הדבר הבא: מערכת ההפעלה תריץ את התכנית prog (או prog.exe), אולם התכנית כבר לא תצפה לקלט מהמקלדת, אלא הקלט יועבר לה *ע"י מערכת ההפעלה* מהקובץ inp.txt. מבחינת התכנית שלנו, היא ממשיכה לקרוא נתונים *מהקלט הסטנדרטי* (cin); מערכת ההפעלה היא שדואגת שהפעם הנתונים לא יגיעו מהמקלדת אלא מהקובץ inp.txt. מבחינתנו כמשתמשים בתכנית, נחסך הצורך להקליד לתכנית את כל עשרת המספרים שוב ושוב בכל הרצה, שכן המספרים כבר מצויים בקובץ, ובכל הרצה הם נקראים ממנו. הנקודה הקריטית אותה יש להפנים היא כי *מבחינתה של התכנית דבר לא השתנה* יחסית למצב בו הקלט הוזן לה מהמקלדת. התכנית כלל לא מודעת לכך שעתה היא מקבלת את הקלט מהקובץ inp.txt. מבחינתה של התכנית, היא קראה בעבר, וקוראת גם עתה, את הקלט שלה מקובץ אותו היא מכנה בשם cin. מערכת ההפעלה היא שבאופן מחדלי דואגת לכך שעת התכנית קוראת נתון מ- cin יועבר לה נתון מהמקלדת, ואילו עתה יועבר לתכנית נתון מהקובץ. מעבר לכך הכול נותר בעינו: כמו בעבר כל נתון ייקרא פעם יחידה; כמו בעבר יש לדאוג להכין את הנתונים לתכנית בדיוק כפי שהיא מצפה לקלוט אותם (אם התכנית מצפה לקרוא מספר שלם, אחריו מספר ממשי, ואחריו מספר שלם, אזי בקובץ יש להכין לה בדיוק את שלושת הנתונים הללו בסדר המתאים). לפעולה שעשינו אנו קוראים *הכוונה מחודשת של הקלט הסטנדרטי* (*redirection of the standard input*), שכן אנו משנים את המקור ממנו מגיע הקלט הסטנדרטי (cin) לתכנית: במקום שהוא יגיע מהמקלדת, הוא מגיע מקובץ.

על-כן בהמשך הדיון, נניח לשם הפשטות, כי את הקלט אנו מכינים בקובץ, כפי שתיארתי בפסקה הקודמת, ואת התכנית אנו מריצים על-ידי בצוע הכוונה מחודשת של הקלט הסטנדרטי.

נניח שברצוננו לכתוב תכנית אשר קוראת סדרת מספרים, ומחשבת את ממוצע הסדרה. נשאלת השאלה: כיצד יאותת המשתמש לתכנית על תום הסדרה? תשובות אפשריות פשוטות:

א. הערך אפס (או כל ערך אחר, יסמן את סוף הקלט). מגבלתה של אפשרות זאת: לא ניתן לכלול בסדרה, שאת הממוצע שלה יש לחשב, את הערך אפס (ודאי שלא ניתן להכלילו בסדרה מספר פעמים).

ב. אחרי הזנת כל מספר המשתמש יישאל האם ברצונו להזין נתון נוסף. מגבלתה של אפשרות זאת: הוגעת המשתמש.

ג. ניתן, כמובן, לחשוב על אפשרויות אחרות.

בדיוננו הנוכחי ברצוני להציע פתרון אחר: שימוש בעיקרון הקרוי 'תום קובץ הקלט' או end of file (eof). מייד אסבירו, כמו גם מושגים אחרים הקשורים אליו.

לפני כן אעיר כי בפרק זה אנו משתמשים בפונקציות שמעמידה לרשותנו שפת C++, ושאינן קיימות בשפת C. בשפת C קיימות פונקציות מקבילות, יחסית דומות, אולם התחביר בשפת C מעט שונה.

10.2 הפונקציה cin.eof()

כאמור, בשלב זה ברצוני להציע דרך, יחסית טבעית, אחרת להשיג את המטרה: קריאת הקלט עד תומו וחישוב ממוצע הערכים שנקראו. הדרך מסתמכת על פונקציה שהשפה מעמידה לרשותנו ושנקראת cin.eof() . היא מתגברת על המגבלות של שתי הדרכים שהצגנו מעל (וכדרכו של עולם, יש לה מגבלות אחרות, משלה.)

ראשית אעיר כי יש משהו מוזר בצורת הכתיבה: cin.eof() ; הסיבה למוזרות היא שאיננו מכירים את גישת התכנות מונחה העצמים בו צורת כתיבה זו מקובלת וטבעית. על כן מבחינתנו ננהג בבחינת 'נעשה ונשמע': כך כותבים.

כיצד תסייע לנו הפונקציה cin.eof() להשלים את משימתנו? עת הפונקציה נקראת היא מחזירה את הערך true אם ורק אם ניסינו לקרוא נתון *פעם יחידה* מהקלט ונכשלנו *שכן הקלט תם* (בקובץ הקלט אין יותר נתונים).

הקוד של תכניתנו יראה:

```
sum = counter = 0 ;

cin >> num ;
while (!cin.eof())
{
   sum += num ;
   counter++ ;
   cin >> num ;
}
cout << ((counter == 0) ? 0 : sum/counter) ;
```

נסביר: המשתנה sum יכיל את סכום הערכים שקראנו, ו- counter את מספר הערכים שקראנו. על כן אנו מאפסים אותם בתחילה.
נניח כי קובץ הקלט שלנו כולל שלושה נתונים: 4 0 2 .
פקודת ה- cin שמעל הלולאה מנסה לקרוא מספר שלם לתוך num וגם מצליחה בכך, וערכו של num הופך להיות 2.
בשלב הבא אנו מתקדמים לכותרת הלולאה, ובה מזמנים את הפונ' cin.eof() .
כפי שאמרנו פונ' זו שואלת: האם קרה שניסינו לקרוא נתון (פעם יחידה) ונכשלנו, שכן הקלט נגמר? התשובה לשאלה היא 'לא' (ניסינו לקרוא נתון, והצלחנו יפה

מאוד), ועל-כן הפונ' מחזירה לנו את הערך false. בכותרת הלולאה אנו שואלים האם הפונ' החזירה לנו את הערך false, ולכן התשובה לשאלה היא כן. על כן אנו נכנסים לגוף הלולאה, sum גדל להיות 2, counter גדל להיות 1, ואנו מנסים לקרוא מספר חדש לתוך num. מנסים וגם מצליחים, שכן בקובץ מחכה לנו נתון נוסף, ולמשתנה num מוכנס הנתון השני בקובץ: 0. בזאת סיימנו סיבוב ראשון בלולאה.

אנו חוזרים לכותרת הלולאה, ומזמנים שוב את הפונ' cin.eof() אשר, כמו תמיד שואלת: האם קרה שניסינו לקרוא נתון (פעם יחידה) ונכשלנו, שכן הקלט נגמר? התשובה לשאלה היא 'לא' (ניסינו לקרוא נתון, וגם הצלחנו), ועל-כן הפונ' מחזירה לנו את הערך false. בכותרת הלולאה אנו שואלים האם הפונ' החזירה לנו את הערך false, ולכן התשובה לשאלה בכותרת הלולאה היא כן. אנו נכנסים לגוף הלולאה, sum 'גדל' להיות 2, counter גדל להיות 2, ואנו מנסים לקרוא מספר חדש לתוך num. מנסים וגם מצליחים, שכן בקובץ מחכה לנו נתון נוסף, ולמשתנה num מוכנס הנתון השלישי והאחרון בקובץ: 4. בזאת סיימנו סיבוב שני בלולאה.

אנו חוזרים לכותרת הלולאה, ומזמנים שוב את הפונ' cin.eof() אשר, כמו תמיד שואלת: האם קרה שניסינו לקרוא נתון (פעם יחידה) ונכשלנו, שכן הקלט נגמר? התשובה לשאלה היא 'לא' (ניסינו לקרוא נתון, וגם הצלחנו), ועל-כן הפונ' מחזירה לנו את הערך false. בכותרת הלולאה אנו שואלים האם הפונ' החזירה לנו את הערך false, ולכן התשובה לשאלה היא כן. אנו נכנסים לגוף הלולאה, sum 'גדל' להיות 6, counter גדל להיות 3.

נשים לב מה קורה עתה: אנו מנסים לקרוא מספר חדש לתוך num. אולם בקובץ הקלט לא נותרו יותר נתונים. על כן ל- num לא נכנס נתון חדש, וערכו נותר כשהיה (4 במקרה שלנו. אעיר כי ב- Visual Studio 6, בניגוד להגדרת השפה, מוכנס ל- num הערך אפס). בזאת גמרנו סיבוב שלישי בלולאה.

אנו שבים לכותרת הלולאה, ומזמנים שוב את הפונ' cin.eof() אשר, כמו תמיד שואלת: האם קרה שניסינו לקרוא נתון (פעם יחידה) ונכשלנו, שכן הקלט נגמר? הפעם התשובה לשאלה היא: 'כן': אכן קרה שניסינו פעם יחידה לקרוא נתון ונכשלנו, שכן קובץ הקלט נגמר. על-כן הפעם cin.eof() מחזירה לנו את הערך true, ולכן התנאי: !cin.eof() שבכותרת הלולאה אינו מתקיים, אנו איננו נכנסים לסיבוב נוסף בלולאה, אלא פונים לפקודת הפלט שמעבר לולאה, ואשר מציגה את הערך שתיים כנדרש.

שימו לב שתכניתנו תפעל כהלכה גם אם קובץ הקלט יהיה ריק. במצב זה לא ניכנס כלל ללולאה, ונציג את הפלט אפס. זו הסיבה שאנו כותבים פקודת cin אחת לפני הלולאה (כדי שהשאלה: האם קרה שניסינו לקרוא נתון ונכשלנו...? תוכל להיענות בחיוב כבר בפעם הראשונה בה אנו מתעתדים להיכנס ללולאה, אם הקלט ריק), ואחת בסוף כל סיבוב בלולאה, לקראת בדיקת התנאי, ובמידת הצורך כניסה לסיבוב נוסף.

עד כאן תיארתי את מהלך העניינים בהנחה שאנו קוראים את הקלט מקובץ. מה יקרה אם נרצה להריץ את התכנית הנ"ל ולקרוא את הקלט מהמקלדת? התשובה היא שכדי לאותת לתכנית על סיום הקלט ביוניקס עלינו להקליד בתחילת שורה את התווים: ^d (Ctrl + d), ובחלונות נקיש *פעמיים* על מקש F6 ואחריו Enter או לחילופין על ^z ואחריו Enter *שוב, פעמיים*.

ברצוני להדגיש כי אם ניהלנו לולאה עד eof אזי מעבר ללולאה זאת לא תוכל התכנית לקרוא מהמשתמש כל קלט שהוא! התכנית תוכל, כמובן, לעשות שימוש כאוות נפשה בנתונים שנקראו, אולם אם קראנו עד eof אזי לא ניתן אחר-כך לחדש שוב את הקלט בשם דרך שהיא. זהו המחיר אותו עלינו לשלם אם ברצוננו לקרוא נתונים באופנות כזאת.

כדי להשתמש בפונ' cin.eof() איננו זקוקים ל- include נוסף מעבר ל- iostream, או לפקודת using נוספת מעבר ל: using std::cin;

10.3 הפונקציות cin.fail() וחברותיה

נניח שכתבנו ואנו מרצים את התכנית הפשוטה הבאה:

```
int num1 = 17, num2 = 17;

cin >> num1 ;
cin >> num2 ;
cout << num1 + num2 << endl ;
```

עד כה הנחנו שאם המשתמשת מצופה להזין מספר שלם אזי זה אכן הקלט שהיא מזינה. עתה נשאל את השאלה: ומה יקרה אם המשתמשת תזין לתכניתנו במקום מספר שלם אות כלשהי (האות a למשל), כלומר קלט שגוי? תשובה טבעית תהיה: המחשב יעיף את התכנית, כלומר יקטע את ביצועה. זו אולי תשובה טבעית, אך לא התשובה הנכונה. בשפת C++ מה שיקרה הוא הדבר הבא: פקודת הקלט תכשל, על כן ערכו של המשתנה יוותר כפי שהוא היה, וכן המחשב יַשהֶה את כל תהליכי הקלט של התכנית, במלים פשוטות: הוא יתעלם מכל פקודות הקלט הבאות, ידלג עליהן, תוך שהוא ממשיך לבצע רק פקודות שאינן פקודות קלט.

על-כן, בדוגמה שראינו מעל, אם המשתמשת כקלט ראשון תזין את האות a, אזי פקודת ה- cin תכשל, ערכו של num1 יוותר 17 (כפי שהוא היה טרם ביצוע הפקודה), המחשב יתעלם מפקודת הקלט השנייה (ולא יקרא נתון לתוך num2); את פקודת הפלט, כמו את כל הפקודות האחרות שאינן cin, המחשב יבצע כרגיל, ועל כן הפלט שיודפס יהיה 34 (ערכם של num1, num2 נותר 17 כפי שהוא נקבע באיתחול).

כיצד נוכל אנו, המתכנתים, לבדוק האם פקודת הקלט הצליחה או נכשלה? שפת C++ מעמידה לרשותנו את הפונ' cin.fail() פונ' זו בודקת האם פעולת הקלט האחרונה נכשלה מכל סיבה שהיא (בין אם משום שקובץ הקלט הסתיים, ועל כן חל גם cin.eof(), ובין מסיבה אחרת). הפונ' cin.fail() מחזירה את הערך true אם פקודת הקלט האחרונה נכשלה, ו- false אחרת. במידה וחל כישלון, אזי כפי שאמרנו, נחסמו כל תהליכי הקלט שלנו. כדי לחדשם נשתמש בפקודה: cin.clear() . פקודה זו אומרת למחשב: 'אני, המתכנת, מודע לתקלה שחלה בקלט, ולוקח עליה פיקוד, ואתה המחשב הסר את החסימה שבצעת על תהליכי הקלט שלי'. במידה וחל כישלון, ואיתרנו זאת באמצעות cin.fail(), וחידשנו את תהליכי הקלט באמצעות cin.clear() עלינו לבצע צעד נוסף והוא הסרת התו הסורר שגרם לכישלון הקלט, נוכל לעשות זאת בפשטות על-ידי שנבצע: cin.get() . cin.et() 'אוכלת' תו יחיד בקלט, במקרה זה אין לנו עניין בתו, ועל-כן איננו מכניסים אותו לשום משתנה שהוא (דבר שיכולנו לעשות באמצעות הפקודה: c = cin.get()).

הקוד השלם שלנו יראה, על-כן:

```
cin >> num ;
while (cin.fail())
{
  cin.clear() ;
  cin.get() ;
  cout << "enter an int: " ;
  cin >> num ;
}
```

עתה נצעד צעד קטן קדימה, ונכתוב פונ' אשר תפקידה לקרוא מהקלט מספר שלם. הפונ' תחזיר באמצעות ערך ההחזרה איתות בולאני: האם הקריאה הצליחה או נכשלה, הערך הנקרא יוחזר באמצעות פרמטר הפניה. באיזה נסיבות עשויה הקריאה להיכשל, והפונ' תחזיר אז ערך `false`? התשובה היא שהקריאה תיכשל, ראשית אם הגענו לסוף הקלט (חל `eof`), ושנית אם יש בעיה שאינה ניתנת לפתרון: מישהו ניתק את המקלדת, אנו מנסים לקרוא מדיסקט ומישהו שלף את הדיסקט, אנו מנסים לקרוא מקובץ ממנו איננו רשאים לקרוא, וכולי. כדי לאתר שקיימת בעיה שאינה ניתנת לפתרון נשאל האם חל: `cin.bad()`, במידה ואכן זה המצב אזי 'כלו כל קיצים' אין טעם שנבצע `cin.clear()` וננסה לקרוא ערך חדש, ויש להרים ידיים (מהניסיון לקרוא מספר שלם), ולהחזיר ערך `false`.
קוד הפונ':

```
bool read_int(int &int_var)
{
    while (true)
    {
        cout << "enter an int: " ;
        cin >> int_var ;
        if (cin.good())
             return true ;
        if (cin.eof() || cin.bad())
             return false ;
        //else it is cin.fail() so:
       cin.clear() ;
       cin.get() ;
     }
     return true ;
}
```

זימון הפונ' (למשל מהתכנית הראשית) עשוי להיות כדוגמת הבא:

```
while(read_int(num))
   cout << num << endl;
```

הפונ' משתמשת ב- `cin.good()` פונ' זו מחזירה ערך `true` אם `'so far so good'`, כלומר עד כה לא חלה כל תקלה בקלט.

לעתים מתכנתים כותבים קוד כדוגמת הבא:

```
sum = 0 ;
while (cin >> num)
   sum += num ;
cout << sum ;
```

התנאי: (cin >> num) מתקיים כל עוד הקלט תקין, כלומר כל עוד לא חל כישלון. במידה ואנו מובטחים שהקלט יהיה תקין אזי באופן כזה אנו קוראים נתונים עד eof.

10.4 שימוש בקבצים חיצוניים

כמו מסייה ז'ורדן במחזה 'גם הוא באצילים' של מולייר, שכל חייו דיבר פרוזה, רק לא היה מודע לכך, גם אנו כל חיינו (התכנותיים) השתמשנו בקבצים, רק לא ידענו זאת. למעשה, עת תכניותינו קוראות נתונים מ: cin או פולטות נתונים ל: cout או ל: cerr הן, מבחינתן, משתמשות בקבצים. קבצים אלה 'אנו מקבלים בחינם' במובן זה שמערכת ההפעלה פותחת אותם עבורנו, ומגדירה שמחדלית הם יקושרו לַמקלדת ולמסך. כפי שראינו בסעיפים הקודמים, ביכולתנו גם להורות למערכת ההפעלה לשַנות ממנהגה זה, ולהעביר את הנתונים הנקראים\נכתבים מ: cin/cout מ\לקבצים. מדוע אני כל-כך מדגיש זאת? שכן לא אחת אני נשאל ע"י תלמידים: האם עת משתמשים בקבצים חיצוניים ניתן לעשות ...? ותשובתי במרבית המקרים היא: האם עת השתמשת ב: cin/cout יכולת לעשות ...? אם כן, אזי הדבר אפשרי גם עת אתה משתמש בקובץ חיצוני שאתה פתחת מפורשות, ואם לא אז לא. אומר זאת אחרת: מרגע שפתחת את הקובץ, אתה יכו לעשות עליו *בדיוק* את אותן הפעולות שיכולת בעבר לעשות על cin/cout.

בשפת C/C++ אנו מבחינים בין <u>קובצי טקסט</u>, אותם ניתן לטעון לעורך, להציג על המסך, להדפיס, שכן הם כוללים תווים, לבין <u>קבצים בינאריים</u>, עליהם לא ניתן לבצע את הפעולות שציינתי. קבצים בינאריים משמשים אותנו לשם העברת נתונים בין תכנית לתכנית (בפרט בהרצה אחת התכנית עשויה לייצר קובץ בינארי, ובהרצה השנייה לקרוא את הכתוב בו). אופן הטיפול בשני סוגי הקבצים דומה למדי, אולם בפרק זה אדון רק בקובצי טקסט.

נפנה עתה לדוגמה אשר תציג לנו כיצד נשתמש בקבצים חיצוניים. עבור הדוגמה נגדיר ראשית מהו **<u>מיזוג (merge)</u>** של שתי סדרות: בהינתן שתי סדרות *ממוינות* מיזוגן הוא סדרה ממוינת אחת הכוללת את הנתונים שהיו בשתי הסדרות גם יחד (כולל כפילויות). ניתן למזג *רק* סדרות ממוינות. <u>לדוגמה</u>: אם הסדרה הראשונה היא: 1, 2, 7, 7 והסדרה השנייה היא: 0, 2, 4, 7, 13, 17 אזי מיזוגן נותן את הסדרה: 0, 1, 2, 3, 4, 7, 7, 7, 13, 17. כדי למזג את הסדרות די לעבור עליהן במקביל, החל בתחילתן, בכל פעם נשאל: באיזה סדרה אנו ניצבים על איבר קטן יותר? איבר זה נשלח לפלט, ועל אותה סדרה נתקדם לאיבר הבא. עת אחת הסדרות מסתיימת נמשיך ונפלוט את האיברים בסדרה השנייה עד תומה. לדוגמה: בסדרות שהצגתי נתחיל עם 1 בסדרה האשונה ו- 0 בשנייה. אפס קטן יותר על כן נשלח אותו לפלט ונתקדם ל – 2 בסדרה השנייה. עתה 1 (מהסדרה הראשונה) קטן מ-2 מהסדרה השנייה, על כן הוא שיפלט, ונתקדם ל- 2 בסדרה הראשונה. נקבע שרירותית כי עת הערכים בשתי הסדרות שווים פולטים את האיבר מהסדרה הראשונה, על כן עתה 2 מהסדרה הראשונה יישלח לפלט, ונתקדם ל- 7 בסדרה זאת. וכך הלאה, עד תום הסדרה הראשונה. עתה נפלוט את 7, 13, 17 מהסדרה השנייה, ובכך נסיים. זמן הריצה של תהליך המיזוג הוא על כן ליניארי במספר האיברים שמיזגנו.

עתה נניח כי הקבצים inp1.txt, inp2.txt כוללים סדרות של מספרים שלמים ממוינים (כמו שהוצגו בדוגמה מעל). משימתנו היא למזג את הסדרות, ולשלוח לפלט הסטנדרטי (cout) את תוצאת מיזוגן. כמובן שלשם ביצוע המשימה לא נוכל לבצע רק הפניה מחודשת (redirection) של הקלט הסטנדרטי, שכן אנו זקוקים לשני קבצי קלט, על כן נזדקק לכלי חדש: נזדקק למשתנים אשר יאפשרו לנו לקרוא נתונים משני קובצי הקלט. מישהו יכול היה להציע שנכתוב משהו

כדוגמת: `inp1.txt >> num1 ;` כלומר: קרא מהקובץ `inp1.txt` נתון למשתנה `num1`. פעולה זאת לא נוכל לבצע, אך נבצע פעולה דומה.

ראשית, נגדיר זוג משתנים שיאפשרו לנו לקרוא מקבצים (תפקידם יהיה דומה לזה של `cin, cout` שלא היינו צריכים להגדיר):

```
std::ifstream inp_fd1, inp_fd2 ;
```

טיפוס המשתנים הוא `std::ifstream`. זהו טיפוס משתנים חדש, שלא הכרנו עד כה. אסביר את מקור שם הטיפוס: האות `i` מורה שמדובר על טיפוס המיועד לקלט (`i= input`), האות `f` מציינת קובץ (`f = file`), המילה `stream` (זרם, בעברית), מורה שאת הנתונים מהקובץ נקרא כמו מזרם: בזה אחר זה, תוך שכל נתון נקרא פעם יחידה, ואחריו נקרא הנתון הבא (בדיוק כפי שעשינו עד היום עת קראנו מ-`cin`, שגם הוא קובץ קלט מסוג זרם של תווים). התחילית `std::` מורה שהטיפוס `ifstream` אינו טיפוס מובנה הקיים בשפת `C++`, אלא הוא חלק מהתוספות הסטנדרטיות המוכרות לכל מהדר. עבורנו כמתכנתים אין חשיבות טכנית לכך שהטיפוס אינו בסיסי (פרט לצורך להגדירו כ: `std::ifstream`), אולם מתכנני השפה רצו שנהיה ערים לכך, ולכן הם מחייבים אותנו לציין מפורשות את העובדה, ע"י התחילית `std::`. כמו עם מרכיבים אחרים בשפה המגיעים מהספרייה הסטנדרטית, גם כאן נוכל בתחילת התכנית להצהיר פעם יחידה: `using std::ifstream;` ובהמשך לכתוב רק: `ifstream`).

בחרתי לקרוא למשתנים בשמות `inp_fd1, inp_fd2`. התחילית `inp` מציינת, כמובן קלט (`input`), והסופית `fd` היא ראשי תיבות של המונח 'מתאר קובץ' (`file descriptor`) שהוא המונח המציין משתנה המאפשר לנו לקרוא או לכתוב מ\לקובץ. בזאת קיבלנו שני משתנים, אך משתנים אלה עדיין לא נקשרו לשני קובצי הקלט שלנו.

עתה נניח כי אנו קוראים מהמשתמש בזה אחר זה את שמותיהם של קובצי הקלט הדרושים לו, לתוך משתנה מחרוזת שהגדרנו:

```
char file_name[MAX_FILE_NAME_LEN] ;
```

פעולת הקריאה היא, כרגיל, מהקלט הסטנדרטי, `cin`:

```
cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
```

המשתנה `file_name` מכיל, בשלב זה, את שמו של קובץ הקלט הראשון.
עתה נבצע את הפעולה:

```
inp_fd1.open(file_name) ;
```

פעולה זאת קושרת בין המשתנה `inp_fd1` שהגדרנו, לבין הקובץ ששמו שמור במשתנה המחרוזת `fil_name` ופותחת את הקובץ לקריאה (הפתיחה היא בהכרח לקריאה שכן המשתנה `inp_fd1` הוא מטיפוס `ifstream`, טיפוס המיועד לקובצי קלט.
נחזור על אותן שתי פעולות שוב עבור הקובץ השני:

```
cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
inp_fd2.open(file_name) ;
```

אם המשתמש הקיש שם קובץ שגוי (לדוגמה: אין במערכת הקבצים שלנו קובץ שזה שמו, או שאין לנו הרשאת קריאה על הקובץ שאולי כן קיים), אזי פעולת הפתיחה תכשל. על כן, לפני שנפנה לתהליך המיזוג, בפרט לפני שנקרא נתונים מהקבצים, עלינו לברר האם חלילה פתיחת אחד הקבצים נכשלה. נעשה זאת ע"י:

```
if (!inp_fd1.is_open() || !inp_fd2.is_open()) {
     cerr << "Cannot open input file\n" ;
     exit(EXIT_FAILURE) ;
}
```

הפעולה, ליתר דיוק השאלה: `inp_fd1.is_open()` בודקת האם הקובץ אליו קשרנו את מתאר הקובץ `inp_fd1` נפתח בהצלחה. במידה והתשובה היא לא, עבור לפחות אחד הקבצים, אנו שולחים הודעת שגיאה, ועוצרים את ביצוע התכנית. לחילופין, ניתן לשאול: `if (!inp_fd1 || !inp_fd2)` תנאי זה שקול לזה שהצגתי מעל.

במידה והגענו עד הלום, משמע שני הקבצים נפתחו בהצלחה לקריאה. מרגע זה ואילך `inp_fd1, unp_d2` דומים לגמרי ל- `cin` וכל מה שיכולנו לעשות על `cin` (אותו לא היינו צריכים לפתוח), אנו יכולים לעשות עליהם. למשל:

```
inp_fd1 >> num1 ;
inp_fd2 >> num2 ;
while (!inp_fd1.eof() && !inp_fd2.eof()) {
     if (num1 <= num2) {
        cout << num1 << " " ;
       inp_fd1 >> num1 ;
     }
     else {
          cout << num2 << " " ;
          inp_fd2 >> num2 ;
     }
}
while (!inp_fd1.eof()) {
        cout << num1 << " " ;
       inp_fd1 >> num1 ;
}
while (!inp_fd2.eof()) {
        cout << num2 << " " ;
       inp_fd2 >> num1 ;
}
```

הפעולה: `inp_fd1 >> num1 ;` שקולה לפעולה: `cin >> num1` . רק שבעוד שהפעולה השנייה קוראת נתון מקובץ הקלט הסטנדרטי, הפעולה הראשונה קוראת נתון מהקובץ אליו נקשר המשתנה `inp_fd1`. (פעולת הקריאה נקראת חילוץ (extraction)). באותו אופן: `inp_fd1.eof()` שקולה לפעולה: `cin.eof()` ובודקת האם קרה שניסינו לקרוא נתון מהקובץ פעם יחידה ונכשלנו שכן הקובץ נגמר.

מעבר לכך, לולאת ה- `while` הראשונה מתנהלת כל עוד בשני הקבצים גם יחד יש נתונים. עת אחד הקבצים יסתיים, למשל `inp_fd1`, השאלה: `!inp_fd1.eof()` תחזיר את הערך `false` ונצא מהלולאה הראשונה. נגיע ללולאה השנייה, אשר שואלת האם הקובץ הראשון טרם הסתיים. בדוגמה שלי התשובה תהיה לא, ועל כן לא ניכנס, אף לא פעם אחת ללולאה זאת. נתקדם לולאה השלישית, ובה התנאי: `!inp_fd2.eof()` יתקיים, על כן בה נתגלגל, ונפלוט את הנתונים שנותרו בקובץ הקלט השני. כלומר תמיד נתגלגל בדיוק באחת משני הלולאות האחרונות: או בשנייה, וזאת אם קובץ הקלט השני נגמר קודם, או בשלישית, וזאת אם קובץ הקלט הראשון הסתיים לפני קובץ הקלט השני. באופן כזה אנו ממזגים את שני הקבצים.

לבסוף, כפי שפתחנו את מתארי הקבצים `inp_fd1, unp_fd2` עלינו גם לסגרם, וזאת נעשה באמצעות הפעולה:

```
inp_fd1.close() ;
```

```
inp_fd2.close() ;
```
כל קובץ שפתחתם ראוי גם לסגור. אומנם עת תכניתכם מסתיימת מערכת ההפעלה אמורה לסגור בעצמה את כל הקבצים שפתחתם, גם אם אתם לא הוריתם לה מפורשות לעשות כן, אולם כאנשים מסודרים ראוי שתסגרו בעצמכם כל קובץ שפתחתם, ולא תשאירו למ.ה. לעשות זאת עבורכם. לפעולת הסגירה יש חשיבות במיוחד עבור קבצים עליהם כותבים (מה שלא עשינו בדוגמה זאת, בה רק קראנו מקבצים חיצוניים), שכן עת התכנית פולטת נתונים לקובץ הנתונים אינם מגיעים מיידית לקובץ, הם ראשית נצברים בשטח בזיכרון הראשי הקרוי חוצץ (buffer). רק עת החוצץ מתמלא (או עת נשברת שורה בפלט), מ.ה. מעבירה את הנתונים שבחוצץ לַקובץ השמור על-גבי הדיסק. פעולת הסגירה מורה למ.ה. כי עליה להעביר את תוכנו של החוצץ לקובץ, גם אם החוצץ אינו מלא ולא נשברה שורה.

כמה הערות נוספות:

א. כדי לבדוק האם פעולת הפתיחה נכשלה שאלנו האם: `inp_fd1.is_open()`! במקום זאת ניתן לשאול האם `inp_fd1`!.

ב. אם שם הקובץ אותו יש לפתוח ידוע למתכנת בעת כתיבת התכנית אזי ניתן להגדיר את המשתנה, ולפתוח את הקובץ בפעולה יחידה: `ifstream inp_fd1("inp1.txt") ;` . כמובן שיש לבדוק שהפתיחה הצליחה.

ג. בדוגמה שמעל ראינו כיצד קוראים נתון מקובץ חיצוני, וכיצד בודקים האם הגענו לסוף הקובץ. באופן דומה ניתן לבצע על קובץ חיצוני את כל הפעולות שראינו קודם עבור `cin`: `inp_fd1.fail()`, `inp_fd1.clear()`, `inp_fd1.get()`, `inp_fd1.getline(...)`.

ד. עת מתאר קובץ מועבר לפונ' חובה להעבירו כפרמטר הפניה (גם אם הפונ' אינה אמורה לפתוח את הקובץ או לסגור אותו, כלומר אינה אמורה לשנות את מתאר הקובץ).

ה. כדי להשתמש בקובץ קלט חיצוני יש צורך בהוראת המהדר: `#include <fstream>`

ו. מבחינת פקודות `using` יש צורך ב: `using std::ifstream;`

לסיכום קטע זה אציג את התכנית השלמה המבצעת את המשימה שראינו:

```
#include <iostream>
#include <iomanip>
#include <fstream>


using  std::cin ;
using  std::cout ;
using  std::cerr ;
using  std::endl ;
using  std::setw ;
using  std::ifstream ;

const int MAX_FILE_NAME_LEN =20;
//-----------------------------------------------------------------
int main() {
      int num1, num2 ;
      char file_name[MAX_FILE_NAME_LEN] ;

      ifstream inp_fd1, inp_fd2 ;

      cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
      inp_fd1.open(file_name) ;
      cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
```

```
        inp_fd2.open(file_name) ;

        if (!inp_fd1.is_open() || !inp_fd2.is_open()) {
              cerr << "Cannot open input file\n" ;
              exit(EXIT_FAILURE) ;
        }

        inp_fd1 >> num1 ;
        inp_fd2 >> num2 ;
        while (!inp_fd1.eof() && !inp_fd2.eof()) {
              if (num1 <= num2) {
                 cout << num1 << " " ;
               inp_fd1 >> num1 ;
              }
              else {
                    cout << num2 << " " ;
                    inp_fd2 >> num2 ;
              }
        }
        while (!inp_fd1.eof()) {
                 cout << num1 << " " ;
               inp_fd1 >> num1 ;
        }
        while (!inp_fd2.eof()) {
                 cout << num2 << " " ;
               inp_fd2 >> num1 ;
        }

        inp_fd1.close() ;
        inp_fd2.close() ;

        cout << "\n" ;
        return EXIT_SUCCESS ;
}
```

עד כאן ראינו כיצד נקרא קלט מקבצים חיצוניים. שליחת פלט נעשית באופן סימטרי לגמרי, תוך שימוש במשתנים דומים:
טיפוס המשתנים הדרוש: `ofstream` (האות o עומדת, כמובן, עבור `output`, וכל יתר מרכיבי השם זהים לטיפוס `ifstream` שראינו בהקשר של הקלט), ונגדיר משתנה: `ofstream out_fd`.
הפתיחה נעשית בדומה לקובץ לקריאה: `out_fd.open(file_name);` (בהנחה שמשתנה המחרוזת `file_name` מכיל את שם הקובץ הדרוש). מכיוון שעתה טיפוס הקובץ הוא `ofstream` אזי הפתיחה היא בהכרח לכתיבה. עת הקובץ נפתח לכתיבה: במידה ולא היה קיים עד כה קובץ בשם זה הוא ייווצר, במידה וכבר היה קובץ קיים בשם זה הוא ירוקן (כלומר כל מה שהיה בו יימחק). מספר דגלים אותם ניתן להעביר לפעולת הפתיחה כארגומנט שני משנים התנהגות מחדלית זאת. לדוגמה: `out_fd.open(file_name, std::ios::app);` מורה למחשב שאם הקובץ כבר קיים אזי אין לרוקנו, אלא יש להוסיף בסופו (`app = append`). באופן דומה הפקודה: `out_fd.open(file_name, std::ios::noreplace);` מורה שאם הקובץ קיים על פעולת הפתיחה להיכשל, כלומר ניתן לפתוח לכתיבה רק קובץ חדש. ניתן גם לשלב בין הדגלים באמצעות | (הקו האנכי כמו ב- `OR` בביטויים בולאניים, אולם בודד): למשל, הפקודה: `out_fd.open(file_name, std::ios::nocreate | std::ios::app);` מורה שאין לייצר קובץ חדש (כלומר על הפעולה להיכשל אם מבקשים לפתוח לכתיבה קובץ שאינו קיים), ומנגד אם הקובץ קיים (והפעולה מצליחה) אזי אין לרוקנו, אלא יש להוסיף בסופו.

כתיבה על קובץ שנפתח נעשית בדומה לכתיבה על קובץ הפלט הסטנדרטי: `out_fd << num1 << " " << num2 ;` (כלומר ההבדל היחיד מכתיבה על קובץ הפלט הסטנדרטי הוא שהקובץ עליו כותבים אינו `cout` אלא משתנה הקובץ שפתחנו).

כפי שהזכרתי קודם, עת אנו כותבים נתון על קובץ הנתון אינו נשלח מיידית לקובץ עצמו (המצוי על מצע האחסון, למשל על הדיסק), אלא מערכת ההפעלה צוברת את הנתונים הנכתבים בשטח בַּזיכרון הראשי הקרוי חוצץ (`buffer`); רק עת החוצץ מתמלא (או עת נשברת שורה בפלט), הנתונים מועברים לקובץ עצמו. לכן אם תכניתכם תדפיס נתון על קובץ, ועתה תעצור (אך לא תסיים, למשל תמתין לקלט), ובזמן שהתכנית עוצרת תבקשו לראות את תוכנו של הקובץ, סביר שלא תראו את הנתון שנכתב (בהסתברות גבוהה, הוא עדיין בחוצץ, וטרם הועתק לקובץ עצמו). כמובן שאם המחשב ייפול בין פרק הזמן בו התכנית פלטה את הנתון למועד בו מערכת ההפעלה העתיקה את תוכנו של החוצץ למצע האחסון, אזי הנתון שנכתב, ושנשמר בינתיים רק בזיכרון, יאבד. כדי לאלץ את מ.ה. לרוקן את החוצץ לדיסק נשתמש בפקודה: `out_fd.flush();` או לחילופין נכתוב: `out_fd << num1 << " " << num2 << flush ;`

לסיכום, אציג תכנית הקוראת סדרת מספרים שלמים מהקלט הסטנדרטי. את כל החיוביים היא פולטת לקובץ פלט אחד (ששמו נקרא מהמשתמש), את כל השליליים היא פולטת לקובץ פלט שני (שגם שמו נקרא מהמשתמש), וכל פעם שהיא קוראת אפס היא שוברת שורה בשני קובצי הפלט.

```
#include <iostream>
#include <iomanip>
#include <fstream>


using  std::cin ;
using  std::cout ;
using  std::cerr ;
using  std::endl ;
using  std::setw ;
using  std::ofstream ;

const int MAX_FILE_NAME_LEN =20;
//---------------------------------------------------------------------
int main() {
      int num;
      char file_name[MAX_FILE_NAME_LEN] ;

      ofstream out_fd1, out_fd2 ;

      cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
      out_fd1.open(file_name) ;
      cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
      out_fd2.open(file_name) ;

      if (!out_fd1.is_open() || !out_fd2.is_open()) {
            cerr << "Cannot open output file\n" ;
            exit(EXIT_FAILURE) ;
      }

      cin >> num ;
      while (!cin.eof()) {
            if (num > 0)
               out_fd1 << num << " " ;
```

```
            else if (num < 0)
               out_fd2 << num << " " ;
            else {
               out_fd1 << endl ;
               out_fd2 << endl ;
            }
            cin >> num ;
      }

      out_fd1.close() ;
      out_fd2.close() ;

      return EXIT_SUCCESS ;
}
```

10.5 מצביעי get ו- put

נניח כי עומד לרשותנו קובץ בשם stud.txt המכיל נתונים אודות תלמידים, וכולל שורות, שורות בפורמט הבא:

```
333444555 yosi cohen      M 17/10/1990
222345666 dana levi       F 01/01/2001
45678913  nechama buzaglo F 31/12/1999
```

אסביר: תשעת התווים הראשונים בכל שורה כוללים את שדה את מספר הזהות, אחריהם באים רווח אחד או יותר, התווים במקומות האחד-עשר עד עשרים-וחמש כוללים את השם, אחריהם מופיע(ים) רווח(ים), אחר-כך אות בודדת המציינת את מין התלמיד, ולבסוף עשרה תווים המציינים את תאריך הלידה. לשם הפשטות אניח כי אורכן של כל השורות אחיד, וכל שדה מתחיל תמיד באותה עמודה (שדה מספר הזהות בעמודה הראשונה, שדה השם בעמודה האחת-עשרה, וכן הלאה), לכן עת חסרים תווים במספר הזהות או בשם מוספים רווחים. אורכה של כל שורה הוא על-כן: 9 (עבור מ.ז.) + 1 (רווח) + 15 (עבור השם) + 1 (רווח) + 1 (מין) + 1 (רווח) + 10 תאריך לידה + 1 (סוף שורה, בהנחה שאנו במערכת יוניקסית), כלומר סך הכל: 39 תווים.

עת הקובץ הנ"ל שמור במחשב הוא מהווה למעשה סדרה של תווים כך שהתו הראשון בקובץ מצוי במקום #0 בקובץ התו השני במקום #1, וכך הלאה. נוכל לצייר זאת באופן הבא:

3	3	3	4	4	4	5	5	5		y	o	s	i		c	o	h	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	

בציור, השורה העליונה מתארת את תווי הקובץ (stud.txt), והשורה התחתונה מתארת את מקומם בקובץ. כפי שהזכרנו הרווח הינו תו ככל תו אחר (ב- ASCII הוא התו מספר 32), וסוף שורה מצוין במערכות יוניקסיות על-ידי תו יחיד (התו מספר 10 ב- ASCII), ובמערכות חלונות באמצעות שני תווים (התו מספר 13, ואחריו התו מספר 10 ב- ASCII).

עתה נניח שהמשימה הניצבת בפנינו היא לספור כמה סטודנטיות נשים וכמה סטודנטים גברים לומדים במוסדנו. נוכל לעשות זאת בנקל אם נפנה לתו ה- 27-י בקובץ, הכולל את מינו של התלמיד הראשון, ואחר נדלג לתו במקום ה- 39+27 כדי לקרוא את מינו של התלמיד השני, ומשם לתו במקום ה: 39+39+27 שכולל את מינו של התלמיד השלישי, וכך הלאה עד תום הקובץ.

הכלי ששפת C++ מעמידה לרשותנו לשם ביצוע המשימה נקרא **<u>מצביע\מכוון ה-get</u>** של הקובץ (get_pointer). מצביע זה מורה כל פעם על התו הבא *שיקרא* מהקובץ (ע"י פקודת הקריאה הבאה). עת הקובץ נפתח הוא ניצב במַקום מספר אפס בקובץ, וכל פעולת קריאה מקדמת אותו בשיעור מספר התווים שנקראו, כך שהוא יוצב על התו שיקַרא בפעולת הקלט הבאה.

ביכולתנו לשאול היכן נמצא מצביע ה- get של הקובץ, באמעצות הפונ': inp_fd1.tellg() , בהנחה ש: inp_fd1 הוא משתנה מטיפוס ifstream אשר נקשר לקובץ שנפתח לקריאה. הפונ' מחזירה לנו את מקומו של המצביע בקובץ (החל במקום מספר אפס).

אנו יכולים גם לשנות את המקום עליו נמצא המצביע, ועל-ידי כך לשנות את הנתון שיקרא בפעולת הקלט הבאה, וזאת באמצעות הפקודה:
inp_fd1.seekg(3879); אשר מורה כי יש להזיז את מצביע הקובץ לתו במקום מספר 3879 בקובץ (הפקודה עשויה להיכשל במידה ולא קיים מקום כזה בקובץ, הפונ' inp_fd1.fail() תוכל ללמדנו האם חל כישלון, ו- inp_fd1.clear() תעזור לנו להתאושש מהכישלון). לפקודת ה- seek יש גם צורה נוספת בה אנו מורים להיכן ברצוננו לנוע *יחסית* למקום הנוכחי בו אנו ניצבים, או יחסית לסוף הקובץ. לדוגמה: inp_fd1.seekg(17, std::ios::cur) ; מורה שיש להתקדם בקובץ 17 מקומות יחסית למיקום הנוכחי (כמובן בתקווה שיש עוד 17 תווים בקובץ). הארגומנט השני, בפרט המרכיב cur (קיצור של current) מורה כי יש לזוז יחסית למקום הנוכחי. התזוזה עשויה להיות גם שלילית, וזאת כדי לנוע אחורנית בקובץ. באופן דומה:
inp_fd1.seekg(-17, std::ios::end) ; מבקשת לזוז יחסית לסוף הקובץ (end מורה זאת), 17- מקומות, כלומר 17 תווים לפני סוף הקובץ. לבסוף אפשר לבקש גם לזוז למקום:
inp_fd1.seekg(17, std::ios::beg) ; אך פקודה זאת שקולה ל: inp_fd1.seekg(17) ;. לכן, בפרט, כדי לדעת מה גודלו של הקובץ (בבתים) נוכל לבצע את הפעולות הבאות:

```
inp_fd1.seek(0, std::ios::end) ;
```

כלומר, ראשית נוע לסוף הקובץ.

```
unsigned long file_size = inp_fd1.tellg() ;
```

שנית, לתוך משתנה מטיפוס unsigned long שבחרתי (שרירותית) לקרוא לו בשם file_size הכנס את המקום בו אתה ניצב בקובץ, וזהו בדיוק גודלו של הקובץ.
שלישית, חזור לתחילת הקובץ, או לכל מקום רצוי אחר:

```
inp_fd1.seek(0) ;
```

התכנית אשר משלימה את המשימה של ספירת הנשים והגברים (בחלונות) מוצגת להלן:

```
#include <iostream>
#include <iomanip>
#include <fstream>

using  std::cin ;
using  std::cout ;
using  std::cerr ;
using  std::endl ;
using  std::setw ;
using  std::ifstream ;

const int MAX_FILE_NAME_LEN =20;
```

```
const int LINE_LEN = 40,
          PLACE_IN_ROW = 27;
//---------------------------------------------------------------
int main() {
      int f_counter = 0, m_counter = 0;
      unsigned long file_size ;
      char file_name[MAX_FILE_NAME_LEN] ;
      char gender ;

      ifstream inp_fd ;

      cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
      inp_fd.open(file_name) ;

      if (!inp_fd.is_open()) {
            cerr << "Cannot open input file\n" ;
            exit(EXIT_FAILURE) ;
      }
```

אתר את גודל הקובץ:

```
      inp_fd.seekg(0, std::ios::end) ;
      file_size = inp_fd.tellg() ;
      inp_fd.seekg (PLACE_IN_ROW-1, std::ios::beg);
```

חזור כל עוד לא הגעת לסוף הקובץ:

```
      while (inp_fd.tellg() < file_size) {
```

קרא את מינו של התלמיד\ה הנוכחי\ת:

```
            gender = inp_fd.get() ;
            if (gender == 'F')
                  f_counter++ ;
            else if (gender == 'M')
                  m_counter++ ;
            else {
                cerr << "Wrong char was read: *" << gender << "*\n"
;
                exit(EXIT_FAILURE) ;
            }
```

דלג למינו של התלמיד הבא בתור:

```
            inp_fd.seekg(LINE_LEN -1, std::ios::cur) ;

      }
      cout << f_counter << " " << m_counter << endl ;

      inp_fd.close() ;

      return EXIT_SUCCESS ;
}
```

<u>הסבר התכנית</u>: מרביתה של התכנית אמורה להיות מובנת. אעיר רק כמה הערות. הפקודה `inp_fd.seekg (PLACE_IN_ROW-1, std::ios::beg);` מזיזה אותנו לתו המציין את מינה של התלמידה הראשונה, ולכן: `gender = inp_fd.get() ;` קוראת תו זה. הפקודה: `inp_fd.seekg(LINE_LEN -1, std::ios::cur) ;` מקדמת אותנו למינו של התלמיד הבא (ה-1- נובע מכך שקראנו תו יחיד, את המין של התלמיד הנוכחי, ועל-כן עלינו להתקדם רק `LINE_LEN` פחות אחד תווים), וכך הלאה בלולאה. הלולאה מתנהלת כל עוד המקום בו אנו ניצבים בקובץ קטן מגודלו של הקובץ.

עד כאן דנו במצביע ה- `get` הקיים בקובץ הנפתח לקריאה. באופן סימטרי לגמרי, עת אנו מגדירים משתנה: `ofstream out_fd`, ופותחים את הקובץ לכתיבה, אנו

מקבלים מצביע\מכוון put המורה לנו להיכן בקובץ תבוצע פעולת הכתיבה הבאה. כמו עם מצביע ה- get גם את מקומו של מכוון ה- put אנו יכולים לבדוק, והפונ' שמחזירה לנו את מקומו היא: out_fd.tellp(); . כמו עם מצביע ה- get גם את מקומו של מצביע ה- put אנו יכולים לשנות, וזאת באמצעות הפקודה: out_fd.seekp(17); אשר תזיז את המצביע למקום מספר 17 בקובץ. כמו בקריאה, גם בכתיבה קיים לפקודה גם וריאנט הזזה יחסית למקום רצוי, כדוגמת: out_fd.seekp(-17, std::ios::cur); . אעיר כי ניתן לנוע גם מעבר לסוף הקובץ, ועל-ידי כך לייצר 'חור' בקובץ. עוד כדאי לזכור כי הכתיבה דורסת את מה שהיה קודם לכן בקובץ (ואינה דוחפת את מה שהיה, כפי שיודע לעשות עבורכם עורך). על כן אם זזתם למקום קיים בקובץ, ואתם כותבים עליו, אתם מאבדים את המידע שהיה בו בעבר.
XXX

10.6 קריאה וכתיבה מאותו קובץ

עד כאן פתחנו קובץ לקריאה *או* לכתיבה, אך לא לשם ביצוע שתי הפעולות גם יחד. לעתים עלינו לפתוח קובץ על-מנת לקרוא ממנו נתונים, ובמידת הצורך גם לעדכנו; במצבים כאלה נזדקק לקובץ שנפתח לקריאה וכתיבה כאחת.

ניתן דוגמה: נניח שאנו צנזורים והמשימה שבפנינו היא לקרוא מחרוזת רצויה מהמשתמש (מהקלט הסטנדרטי), ולמחוק את כל מופעיה מקובץ כלשהו (ששמו מסופק לנו ע"י המשתמש). לשם הפשטות נניח כי מחיקת המחרוזת מהקובץ תיעשה ע"י החלפת המחרוזת האסורה במחרוזת של X-ים. לדוגמה, אם עלינו לצנזר את המחרוזת yosi נמיר אותה ב- xxxx. (אעיר כי פעילות מורכבת יותר על קובץ, כגון מחיקה גמורה של תווים בו, כמו גם הוספה של תווים באמצעיתו, לא ניתן לבצע בפשטות: יש להעתיק את כלל התווים העוקבים בקובץ ממקום למקום על מנת 'לכסות' על נתונים שרוצים למחוק, או כדי 'לפתוח מקום' לתווים חדשים; בדומה למה שצריך לעשות עם רוצים להוסיף או למחוק נתון באמצע מערך).

טיפוס המשתנה לו נזדקק הוא fstream, ונגדיר משתנה: fstream in_out_fd; בעזרתו נקרא ונכתוב נתונים מהקובץ הדרוש. פתיחת הקובץ נעשית, כרגיל, באמצעות פקודת ה- open, אולם כדי לציין שהקובץ נפתח הן לקריאה והן לכתיבה נעביר לפקודת הפתיחה את שילוב הדגלים: std::fstream::in |std::fstream::out, ועל כן פקודת הפתיחה בשלמותה תראה: in_out_fd.open(file_name, std::fstream::in |std::fstream::out); . כמו תמיד עלינו לבדוק שהפתיחה הצליחה. במידה והיא הצליחה המצביע לקובץ נמצא בתחילתו.

בקובץ לכתיבה עומדים לרשותנו *שני* המצביעים הן מצביע ה- get, והן מצביע ה- put אולם הם *תמיד יורו לאותו מקום בקובץ*. הקריאה והכתיבה יעשו כרגיל, לדוגמה: in_out_fd >> current ; תקרא ערך מהקובץ לתוך המשתנה current, ו: in_out_fd << 'x' ; כותבת על הקובץ את התו x.

אציג את קוד התכנית, ואשלב בו הסברים:

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstring>

using  std::cin ;
using  std::cout ;
using  std::cerr ;
```

```
using  std::endl ;
using  std::setw ;
using  std::fstream ;

const int MAX_FILE_NAME_LEN =20;
const int MAX_STR_LEN =20;
//---------------------------------------------------------------------
int main() {
      char file_name[MAX_FILE_NAME_LEN] ;
      char wanted[MAX_STR_LEN],
           current[MAX_STR_LEN] ;
      fstream in_out_fd ;

      cin >> setw(MAX_FILE_NAME_LEN) >> file_name ;
      in_out_fd.open(file_name,                        std::fstream::in
|std::fstream::out);

      if (!in_out_fd.is_open()) {
            cerr << "Cannot open input file\n" ;
            exit(EXIT_FAILURE) ;
      }

      cin >> setw(MAX_STR_LEN) >> wanted ;
```

אנו שומרים את אורך המחרוזת שיש לצנזר, כדי לדעת כמה x-ים יש לכתוב על הקובץ:

```
      unsigned len = strlen(wanted) ;
```

שוב ושוב נקרא מחרוזת מהקובץ ונבדוק האם יש לצנזרה,
נצא מהלולאה עת ניכשל; סיבת הכישלון היא שננסה להתקדם עם מכוון הקריאה מעבר לסוף הקובץ:

```
      in_out_fd >> current ;
      while (!in_out_fd.fail()) {
            if (strcmp(current, wanted) == 0) {
```

אם יש לצנזר אזי: נזוז אחורנית בקובץ, למקום בו מתחילה המחרוזת שיש לצנזר, ונכתוב עליה x-ים:

```
                  in_out_fd.seekp(-len, std::ios::cur) ;
                  for (int i=0; i < len; i++)
                        in_out_fd << 'x' ;
```

לקראת פעולת הקריאה הבאה נטפל במצביע הקריאה, אין צורך להזיזו, רק להכניסו שוב לפעולה (אחרי *שכתבנו* על הקובץ):

```
                  in_out_fd.seekg(0, std::ios::cur) ;
            }
```

לקראת סיבוב נוסף בלולאה נקרא מחרוזת חדשה:

```
            in_out_fd >> current ;
      }

      in_out_fd.close() ;

      return EXIT_SUCCESS ;
}
```

10.7 תרגילים

10.7.1 תרגיל מספר אחד: מיזוג קבצים.

נניח כי נתונים זוג קבצים:

א. קובץ המכיל מספרי זהות ושמות, בפורמט כדוגמת הבא:

333444555 cohen yosi
345345345 levi dana
444444444 ben-lulu yael
66699966 salomon yoel moshe

כלומר כל שורה כוללת: (א) מספר זהות בן לכל היותר תשע ספרות, (ב) שם משפחה הכולל רצף של אותיות ומקפים ללא רווח, (ג) שם פרטי אשר עשוי להיות מורכב ממספר שמות (כדוגמת yoel moshe).

ב. קובץ הכולל מספרי זהות וציונים, כדוגמת הבא:

333444555 88 77 66
345345345 77 77 77 77
66699966 55

כלומר כל שורה כוללת מספר זהות וסדרה של ציונים.

שני הקבצים ממוינים על-פי מספר הזהות.

עליכם לכתוב תכנית אשר קוראת במקביל את שני הקבצים (**פעם יחידה**), ויוצרת מהם קובץ בו יופיע מספר הזהות, השם וסדרת הציונים של כל תלמיד ותלמיד.

הניחו כי כל מספר זהות המופיע בקובץ הציונים אמור להופיע בקובץ השמות, במידה ולא כך המצב שלחו הודעת שגיאה כי התלמיד שמספרו ... אינו מופיע בקובץ השמות. את הודעת השגיאה הדפיסו הן על קובץ השגיאה הסטנדרטי (`cerr`), והן על קובץ חריגים מיוחד בו תרוכזנה הודעות השגיאה. עבור תלמיד כזה הוסיפו בקובץ הפלט שאתם יוצרים את השם הפרטי ושם המשפחה כ: "???". מנגד, ייתכן כי תלמיד כלשהו יופיע בקובץ השמות אך לא בקובץ הציונים, ובכך אין כל פסול (במקרה כזה אין להוציא הודעת שגיאה).

כמו כן במידה ואחד הציונים אינו בתחום 0..100 התרָאו על-כך (ל- `cerr`, ולקובץ החריגים), אך העתיקו את הציון כפי שהוא מופיע.

קריאת הנתונים מהקבצים השונים תעשה עד `eof` .

שמות הקבצים מהם יש לקרוא את הנתונים, ועליהם יש לכתוב את הפלט, יועברו כפרמטרים לתכנית (באמצעות `argv`). סדר הארגומנטים המועברים יהיה:

א. קובץ השמות.
ב. קובץ הציונים.
ג. קובץ הפלט.
ד. קובץ החריגים.

10.7.2 תרגיל מספר שתיים: דחיסת נתונים

נתון קובץ המכיל רצפים של תווים זהים (כדוגמת: aaaaaaaaaaaabcccc%%%xXXXXXXXXXXXXX).

יש לבצע דחיסה של הקובץ, כלומר לבנות קובץ חדש, קטן יותר, שיכיל מידע כיצד ניתן לשחזר את הקובץ המקורי.

עליכם לכתוב שתי תכניות: הראשונה מבצעת את תהליך הדחיסה, ושניה את תהליך השחזור.

בצעו את המשימה תוך הקפדה על העקרונות הבאים:

א. בעקבות תהליך הדחיסה נפח הקובץ יקטן ככל שניתן.

ב. בשום מקרה נפח הקובץ לא יגדל בעקבות דחיסתו.

ניתן להניח כי התו backslash (\) אינו מופיע בקובץ אותו עליכם לדחוס, אך כל תו אחר עשוי להופיע בו.

תכנית הדחיסה תקבל באמצעות `argv` את שם הקובץ אותו עליה לדחוס, ואת שם הקובץ עליו תיכתב תוצאת הדחיסה. תכנית השחזור תקבל את שם הקובץ בו מצוי המידע הדחוס, ואת שם הקובץ עליו יש לכתוב את המידע המשוחזר.

11. מצביעים ומערכים

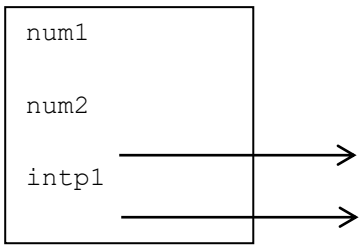
בקצרה ובפשטות ניתן לומר כי מצביע הוא משתנה המכיל כתובת של משתנה אחר, במילים אחרות מצביע הוא משתנה אשר מַפנה אותנו למשתנה אחר.

11.1 עקרונות בסיסיים

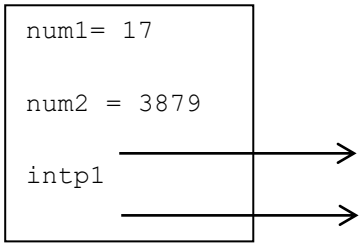
נניח כי בתכנית כלשהי הגדרנו את המשתנים הבאים:

```
int num1, num2, *intp1, *intp2 ;
```

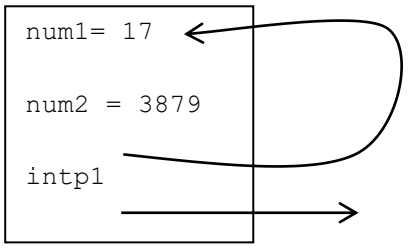
מה קיבלנו? קיבלנו ארבעה משתנים, טיפוסם של שני הראשונים ביניהם מוכר לנו היטב מימים ימימה: אלו הם משתנים אשר מסוגלים לשמור מספר שלם; טיפוסם של השניים האחרונים, אלה שלפני שמם כתבנו כוכבית (*), הוא `int *`, ומהותם חדשה לנו: משתנים אלה הם מצביעים לתאי זיכרון (כלומר למשתנים) המכילים מספרים שלמים. ראשית, נצייר כיצד נראית המחסנית בעקבות הגדרת המשתנים הללו, ואחר נמשיך בהסבר. שימו לב כי מצביעים אנו מציירים עם חץ שראשו הוא בצורת v, וזאת בניגוד לפרמטרים משתנים אותם אנו מציירים עם חץ שראשו הוא משולש שחור:



המשתנים `num1, num2` מסוגלים לשמור מספר שלם, על-כן נוכל לכתוב: `num1 = 17; num2 = 3879;` ובתאי הזיכרון המתאימים ישמרו הערכים הרצויים. מצב המחסנית אחרי ביצוע שתי השמות אלה יהיה:



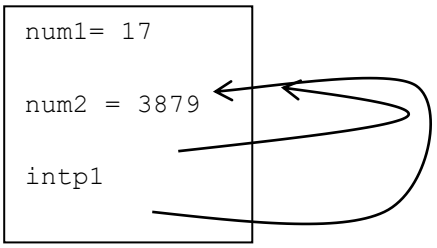
לעומת זאת, intp1, intp2 הם מצביעים. עליהם אנו רשאים לבצע פעולה כגון: intp1 = &num1; . נסביר את משמעות הפעולה: אופרטור ה- & מחזיר את כתובתו של המשתנה (של האופרנד) עליו הוא מופעל, במקרה שלנו הוא מחזיר את כתובתו של num1, כתובת זאת אנו משימים למשתנה intp1, ועל כן עתה intp1 מצביע על num1, מבחינה ציורית נציג זאת:



עתה אנו יכולים לפנות לתא הזיכרון הקרוי num1 גם באופן הבא: *intp1 . למשל אנו רשאים לכתוב: *intp1 = 0; והדבר יהיה שקול לכך שנכתוב: num1 = 0; . אנו גם רשאים לכתוב: (*intp1)++; , והדבר יהיה שקול לפקודה: num1++; . כלומר *intp1 הוא עתה שם נרדף ל- num1. נדגיש כי intp1 הוא מצביע ל- int, לעומת זאת: *intp1 הוא תא הזיכרון עליו intp1 מצביע. הפעולה בה אנו פונים באמצעות מצביע לשטח הזיכרון עליו המצביע מורֶה נקראת dereferncing.

באופן דומה אנו רשאים לבצע גם את ההשמה: intp2 = &num2; והאפקט שלה יהיה דומה. עתה נוכל לבצע את ההשמה: *intp1 = *intp2; , מה עושה השמה זאת? היא מכניסה לתוך תא הזיכרון עליו מצביע intp1, כלומר למשתנה num1 את הערך המצוי בתא הזיכרון עליו מצביע intp2, כלומר את הערך המצוי ב- num2. במילים אחרות ההשמה: *intp1 = *intp2; שקולה להשמה: num1=num2 ; .

לעומת ההשמה האחרונה שראינו, ההשמה: intp1 = intp2; מכניסה *למצביע* intp1 את הערך המצוי *במצביע* intp2. במילים אחרות ב- intp1 תהיה אותה כתובת המצויה ב- intp2; ובמילים פשוטות, עתה שני המצביעים יצביעו על תא הזיכרון num2. מבחינה ציורית נציג זאת כך:



במצב הנוכחי *intp1, *intp2, num2 הם שלושה שמות שונים לאותו תא זיכרון. הדבר עלול לגרום לתוצאות לכאורה מפתיעות. לדוגמה נביט בקטע התכנית הבא (אשר מניח כי מצב המצביעים הוא כפי שמתואר בציור האחרון):

```
num2 = 3 ;
*intp1 = 5 ;
cout << num2 ;
```

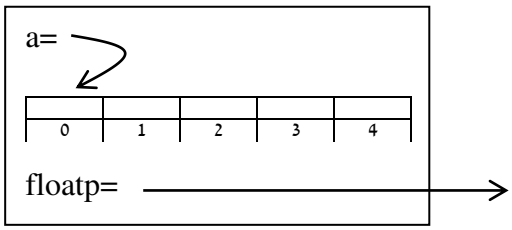
הפלט שיוצג יהיה חמש, וזאת למרות שהפקודה האחרונה אשר הכניסה ערך למשתנה num2 שמה בו את הערך שלוש. הסיבה לכך שהפלט יהיה חמש היא שגם הפקודה *intp1 = 5; השימה ערך לאותו שטח זיכרון, ועשתה זאת אחרי ההשמה: num2 = 3; . לַמצב בו לאותו שטח בזיכרון ניתן לפנות באמצעות מספר שמות אנו קוראים alias. זהו מצב לא רצוי, שכן הוא עלול לגרום לתופעות

לכאורה משונות, כפי שראינו. למרות שהוא מצב לא רצוי, בהמשך נעשה בו שימוש רב.

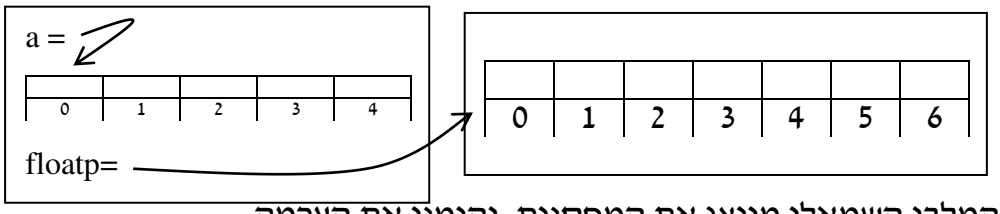
שימו לב כי הפקודה: `intp1 = 3;` היא פקודה שגויה, שכן אנו מנסים להכניס למצביע (שטיפוסו: `int *`) ערך שלם (כלומר ערך מטיפוס `int`). זה לא יהיה בלתי סביר לחשוב שכתובות במחשב מיוצגות באמצעות מספרים טבעיים טבעיים, ולמרות זאת השמה כגון הנ"ל היא שגויה.

11.2 הדמיון והשוני בין מצביע לבין מערך

כל השימוש שעשינו עד כאן במצביעים היה כדי להפנותם לתאי זיכרון קיימים. יבוא הקנטרן וישאל: אם זה כל השימוש שניתן לעשות במצביעים, אז מה תועלתם? התשובה היא שבמצביעים ניתן לעשות גם שימושים אחרים. מצביע הוא למעשה *פוטנציאל למערך*. נסביר: נניח שהגדרנו: `float a[5], *floatp;`. נציג את המחסנית:



מה קיבלנו על-גבי המחסנית? קיבלנו מערך בשם `a` בן חמישה תאים, ומצביע בשם `floatp`. (בהמשך יובהר מדוע גם לצד `a` ציירנו חץ). עתה אנו רשאים לבצע את פקודה: `floatp = new float[7];`. לְמה תגרום פקודה זאת? *פקודה זאת מממשת את הפוטנציאל הגלום במצביע, והופכת אותו למערך ככל מערך אחר*. מייד ניתן הסבר מפורט; אך נקדים לתשובה רקע הכרחי: כל המשתנים שראינו עד כה הוקצו על-גבי המחסנית. המחסנית היא שטח בזיכרון הראשי אשר מערכת ההפעלה מקצה לתכניתכם. בשלב זה של חיינו אנו גם מבינים מדוע שטח הזיכרון הזה מכונה מחסנית: בשל שיטת ה- last in first out על-פיה הוא מתנהל, ואשר קובעת באיזה אופן חלקים נוספים מזיכרון זה מוקצים לתכנית, ואחר משוחררים. פרט למחסנית, מקצה מערכת ההפעלה לתכניתכם גם שטח נוסף בזיכרון הראשי. שטח זה נקרא **ערמה** (heap). הוא נקרא כך שכן הוא מנוהל בבלגן יחסי, באופן דומה לדרך בה אנו זורקים דברים נוספים על ערמה, או מושכים דברים מערמה. עת תכנית מבצעת פקודה כדוגמת: `floatp = new float[7];` קורה הדבר הבא: על-גבי הערמה מוקצה מערך בן 7 תאים ממשיים, והמצביע `floatp` עובר להצביע לתא מספר אפס במערך זה. מבחינה ציורית נציג זאת כך:



המלבן השמאלי מייצג את המחסנית, והימני את הערמה.

המערך שהתקבל יקרא `floatp`. אל תאיו נפנה כפי שאנו פונים לכל מערך אחר. לדוגמה אנו רשאים לכתוב: `floatp[0]=17;` או: `cin >> float[6];`. אנו רואים, אם כן, כי *אחרי שהקצנו למצביע מערך* (באמצעות הפקודה: `floatp = new...`), כלומר אחרי שמימשנו את הפוטנציאל הגלום במצביע, *אנו מתייחסים ל-`floatp` כאל כל מערך אחר*.

אנו אומרים כי המערך floatp הוקצה **דינמית**, בעת ריצת התכנית, וזאת בניגוד למערך a שהוקצה **סטטית**, וגודלו נקבע בעת שהמתכנת כתב את התכנית.

מספר הערות על פקודת ה- new:

א. כמובן ששם הטיפוס שיופיע אחרי המילה השמורה new יהיה תואם לטיפוס המצביע. כלומר לא יתכן שנגדיר מצביע: inp *p; , ובהמשך נכתוב: p = new float[7];.

ב. הקפידו שגודל המערך אותו ברצונכם להקצות יופיע בתוך *סוגריים מרובעים*, (ולא בתוך סוגריים מעוגלים!).

ג. פקודת ה- new עלולה להיכשל, כלומר היא עלולה שלא להצליח להקצות מערך כנדרש. במקרה זה ערכו של המצביע יהיה הקבוע NULL. הערך NULL מורה שהמצביע אינו מצביע לשום כתובת שהיא. נדון במשמעותו ביתר הרחבה בהמשך. כלל הזהב למתכנת המתחיל הוא כי אחרי ביצוע new יש לבדוק האם ערכו של המצביע המתאים שונה מ- NULL. במידה וערכו של המצביע הוא NULL, סביר שתרצו לעצור את ביצוע התכנית, תוך שאתם משגרים הודעה מתאימה. נציג דוגמה:

```
floatp = new int[7] ;
if (floatp == NULL) {
   cout << "Can not allocate memory\n" ;
   exit(EXIT_FAILURE) ;
}
```

הסבר: אם בעקבות ביצוע פקודת ה- new ערכו של floatp הוא NULL אזי אנו משגרים הודעת שגיאה, ועוצרים את ביצוע התכנית באמצעות הפקודה exit(1) . פקודה זאת תחזיר את הקוד המופיע בסוגריים למערכת ההפעלה. על-מנת שהקומפיילר יכיר את פקודת ה- exit , ואת הקבוע NULL יש לכלול בתכניתכם את ההוראה: #include <stdlib.h> . בדרך כלל נוהגים לכלול את זוג הפקודות המבוצעות במקרה וההקצאה נכשלה בפונקציה קטנה נפרדת שמקבלת מחרוזת, וערך שלם; הפונקציה מציגה את המחרוזת, וקוטעת את ביצוע התכנית תוך החזרת ערך השלם.

האפשרות להקצות מערך דינמית גורמת לכך שעתה יש ביכולתנו לכתוב קטע קוד כדוגמת הבא (נניח את ההגדרות: int num_of_stud, *bible;):

```
cin >> num_of_stud ;
bible = new int[num_of_stud] ;
if (bible == NULL)
    terminate("can not allocate memory\n", 1) ;
for (int i = 0; i < num_of_stud; i++)
   cin >> bible[i] ;
```

הסבר: ראשית אנו קוראים מהמשתמש כמה תלמידים יש בכיתתו, אחַר אנו מקצים מערך בגודל הדרוש, ולבסוף (בהנחה שההקצאה הצליחה), אנו קוראים נתונים לתוך המערך. במידה וההקצאה נכשלה, אנו מזמנים את הפונקציה terminate כפי שתואר בפסקה הקודמת.

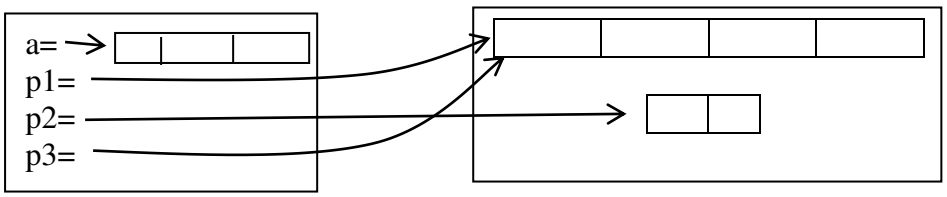
11.2.1 <u>נוטציה מערכית לעומת נוטציה מצביעית</u>

אמרנו כי מצביע הוא פוטנציאל למערך. הוספנו שאנו מממשים את הפוטנציאל באמצעות פקודת ה- new, ואחרי שמימשנו את הפוטנציאל המצביע והמערך היינו הך הם (כמעט). עוד אמרנו כי המצביע, אחרי שהוקצה לו מערך מצביע למעשה על תא מספר אפס במערך. נעקוב אחרי קביעות אלה שוב: נניח כי הגדרנו:

```
int a[5], *ip ;
```
ואחר הקצנו ל- `ip` מערך: `ip = new int[5];` (נניח כי ההקצאה הצליחה).
עתה יש לנו בתכנית שני מערכים בגודל של חמישה תאים. על כן אנו יכולים לכתוב קוד כדוגמת הבא:
```
for (int i=0; i<5; i++)
   a[i] = ip[i] = 0 ;
```
האם אנו יכולים לכתוב גם: `*ip = 3879;` ? ואם כן מה משמעות ההשמה הנ"ל?
התשובה היא שאנו יכולים לכתוב השמה כנ"ל, וההשמה תכניס לתא מספר אפס במערך `ip` את הערך 3879; שכן כפי שאמרנו `ip` מצביע על התא מספר אפס בַּמערך שהוקצה דינמית. מה שאולי יפתיע אתכם יותר הוא שאנו רשאים לכתוב גם: `*a = 3879;` ופקודה זאת תכניס את הערך 3879 לתא מספר אפס במערך `a`.
כלומר גם `a` הוא למעשה מצביע: מצביע אשר מורה על התא מספר אפס בַּמערך שהוקצה סטטית על-גבי המחסנית. עתה אתם גם מבינים מדוע גם לצד `a` ציירנו חץ שהצביע על התא מספר אפס במערך המתאים.

את לולאת איפוס המערכים שכתבנו קודם בצורת כתיבה 'מערכית', אנו רשאים לכתוב גם בצורת כתיבה 'מצביעית':
```
for (i=0; i<5; i++)
   *(a + i) = *(ip + i) = 0 ;
```
נסביר: אנו מבצעים כאן **אריתמטיקה של מצביעים**. ערכו של הביטוי: `(a + i)` הוא המצביע אשר מורה על התא שבהסטה `i` מהתא עליו מורה `a`. לכן אם `a` מורה על התא מספר אפס במערך, וערכו של `i` הוא שלוש, אזי הביטוי `(a + i)` הוא המצביע לתא מספר שלוש במערך. לכן אם אנו כותבים: `*(a + i)`, כלומר אנו הולכים בעקבות המצביע, אזי אנו פונים לתא מספר שלוש במערך, ובלולאה שלנו אנו מאפסים אותו. במילים פשוטות: הכתיבה: `*(a + i)` שקולה לכתיבה: `a[i]`. אנו רואים אם כן שוב את הדמיון בין מערך לבין מצביע שהוקצה לו מערך.

האם קיימת זהות מלאה בין מערך שהוקצה סטטית (על-גבי המחסנית), לבין כזה שהוקצה דינמית (על-גבי הערמה)? לא לגמרי. עתה נציין את ההבדלים. נבחן את קטע הקוד הבא, תוך שאנו מניחים את ההגדרות:
```
int a[3], *p1, *p2, *p3 ;

p1 = new int[4] ;
p2 = new int[2] ;
p3 = p1 ;
p1 = p2 ;
a = p2 ;
```
נסביר: ראשית מקֻטע הקוד השמטנו את הבדיקה האם ההקצאה הדינמית הצליחה. אנו עושים זאת לצורך הקיצור, וההתמקדות בַּעיקר. בתכניות שלכם הקפידו תמיד לערוך את הבדיקה המתאימה. מעבר לכך: אנו מקצים ל- `p1` ול- `p2` שני מערכים. עתה אנו מפנים את `p3` להצביע על המערך `p1`, ולכן אם נכתוב `p3[0]` או נכתוב `p1[0]` נִפנה לאותו שטח זיכרון. נתאר זאת על-גבי המחסנית:



הפקודה הבאה: `p1 = p2;` מפְנה את `p1` להצביע על אותו מערך כמו `p2` (ולכן עתה רק `p3` מצביע על המערך בן ארבעת התאים שהוקצה דינמית). הפקודה

האחרונה: `a = p2` היא שגיאה שתמנע מהתכנית להתקמפל בהצלחה. `a` הוא מצביע אשר קשור לנצח נצחים לַמערך שהוקצה לו על-גבי המחסנית; שלא כמו `p1` את `a` לא ניתן להפנות להצביע על מערך אחר.

מאותה סיבה הפעולה `p1++;` אשר מקדמת את `p1` להצביע על התא הבא במערך, היא תקינה: את `p1` ניתן להזיז באופן חופשי. לעומתה הפקודה `a++;` היא שגיאה. שימו לב כי הפקודה `p1++;` שונה לחלוטין מהפקודה `(*p1)++;` אשר שקולה לפקודה: `p1[0]++;`. שתי הפקודות האחרונות מגדילות באחד את ערכו *של תא הזיכרון עליו מצביע* `p1`, במילים אחרות הן מגדילות באחד את הערך המצוי בַּתא מספר אפס במערך `p1`, כלומר הן מבצעות אריתמטיקה במספרים שלמים. בעוד הפקודה `p1++` היא אריתמטיקה במצביעים: היא מסיטה את `p1` להצביע מקום אחד ימינה יותר במערך.

נִראה שימוש לפקודת ההגדלה העצמית במצביעים. נניח כי הגדרנו: `char *cp;`, אחר הקצנו ל- `cp` מערך (באמצעות הפקודה: `cp = new char[3879];`). עתה אנו יכולים לאפס את המערך גם באופן הבא:

```
for (int i=0; i<3879; i++)
{  *cp = 0; cp++ ; }
```

הסבר: בכל סיבוב בלולאה אנו מאפסים את התא במערך עליו מצביע `cp`, (וכזכור בתחילת התהליך `cp` מצביע על התא מספר אפס במערך), ואחר מקדמים את `cp` כך שהוא יצביע על התא הבא במערך.

להיכן מצביע `cp` אחרי ביצוע הלולאה? אל מחוץ לגבולות המערך, ליתר דיוק לַתו הראשון שאחרי המערך. כדי להשיב את `cp` להצביע על התא מספר אפס במערך אנו יכולים לבצע את הפקודה: `cp -= 3879;` פקודה זאת מחזירה את `cp` להצביע 3879 תאים אחורנית, כלומר לראש המערך. כמובן שלוּ היינו כותבים במקום את הפקודה: `cp -= 3879;` את הפקודה: `cp -= 4000;` אזי היינו שולחים את `cp` להצביע אי שם למקום לא ידוע בַּזיכרון, וזאת פעולה מסוכנת, שאת אחריתה מי יישור.

הדוגמה האחרונה רמזה לנו על כמה מהסכנות הטמונות בשימוש לא די זהיר וקפדני במצביעים. נראה דוגמה נוספת: נניח שהגדרנו: `bool *bp;` ואחר, בלא שהפננו את `bp` להצביע על תא זיכרון מוגדר כלשהו, אנו מבצעים: `*bp = true;`. מה אנו עושים בכך? ב- `bp`, כמו בכל משתנה שלא נקבע לו עדיין ערך, יש ערך 'זבל' מקרי כלשהו, כלומר הוא מצביע לתא מקרי כלשהו בזיכרון. עתה אנו נגשים לאותו תא מקרי ומכניסים לתוכו את הערך `true`. התא המקרי שעליו 'דרכנו' עשוי להכיל משתנים אחרים של התכנית, טבלות של מערכת ההפעלה, או נתונים כאלה או אחרים. התוצאה עלולה להיות שתכניתנו תתנהג בצורה ביזארית, שלא לומר אכזרית. לצערי, כבר ראיתי מתכנתים מתחילים שבאמצעות שימוש לא די זהיר במצביעים יצרו באגים שאת האפקט שלהם ניתן לתאר בלשון המעטה כ-'מפתיע, יצירתי, עולה על כל דמיון'. הצד המרגיע של הנושא הוא שכל התקלות שתחוללו באמצעות שימוש קלוקל במצביעים תוגבלנה לזיכרון הראשי, ולכן תעלמנה עת תכבו את המחשב; במילים אחרות, לא תגרמו לנזק בלתי הפיך למחשב, או לתוכנו של הדיסק.

אחד ממנגנוני הבטיחות שעוזר לנו להישמר מפני תקלות כדוגמת זו שראינו הוא השימוש ב- `NULL`. כאשר ערכו של מצביע `p` הוא `NULL` משמעות הדבר היא כי `p` אינו מצביע לשום תא בזיכרון, ועל-כן לא ניתן לפנות באמצעותו לַזיכרון, כלומר לא ניתן לבצע: `*p = ...;`. כיצד נוכל להיעזר ב- `NULL` בתכניות שאנו כותבים?

א. בעת הגדרת המצביע נאתחל אותו לַערך `NULL`: `char *bp = NULL;`.

ב. בַהמשך אולי כן ואולי לא נפנה את bp להצביע על תא כלשהו בַזיכרון (למשל, באמצעות פקודה כגון bp = new …).
ג. לבסוף, לפני שאנו פונים לתא הזיכרון עליו מצביע bp (כלומר לפני שאנו מבצעים: … = *bp) נישְאל האם ערכו של bp שונה מ- NULL. אם ערכו של bp שונה מ- NULL אנו מובטחים כי bp הופנה לתא כלשהו בַזיכרון, ועל כן הפנִיה … = *bp היא בטוחה. אם, לעומת זאת, bp לא הופנה להצביע על תא כלשהו בַזיכרון, אזי ערכו של bp יישאר כפי שהוא היה בעקבות האיתחול, כלומר ערכו יהיה NULL, ואז לא ננסה לפנות לתא הזיכרון עליו bp מצביע.

נראה דוגמה:

הגדרת המצביע: char *bp = NULL; .

אחר- כך, אי שם בהמשך התכנית נכתוב:

```
if (…) {
   bp = new char[17] ;
   if (bp == NULL)
      terminate("Can not allocate memory\n", 1) ;
}
```

כלומר אם תנאי כלשהו (שלא פרטנו) מתקיים אנו מקצים מערך, ומפנים את bp להצביע עליו.

ולבסוף, הלאה יותר בתכנית אנו כותבים:

```
if (bp != NULL)
   bp[1] = bp[3] = 3879 ;
```

בזכות האיתחול של bp ל- NULL בעת הגדרתו, מתקיים כי הבדיקה: if(bp != NULL) מעידה האם ל- bp הוקצה מערך, ועל כן אנו יכולים לפנות בבטחה לתאים מספר אחד, ומספר שלוש במערך, או שמא ל- bp לא הוקצה מערך, ועל-כן ערכו נותר כפי שהוא היה בעת הגדרתו, כלומר NULL.

כלל הזהב האחרון אותו עלינו ללמוד בַהֶקשר של מצביעים ומערכים הוא: כל מערך שהוקצה באמצעות פקודת new, ישוחרר בשלב זה או אחר, (אולי רק לקראת סיום ריצת התכנית), באמצעות פקודת delete. לדוגמה, כדי לשחרר את המערך עליו מצביע bp נכתוב: delete [] bp; . משמעות הפקודה היא שאנו מורים למערכת ההפעלה כי איננו זקוקים יותר למערך, ולכן ניתן למחזר את שטח הזיכרון שלו. פניה לזיכרון שמוחזר מהווה שגיאה; על כן היזהרו מקוד כדוגמת הבא:

```
intp1 = new int[17] ;
inp2 = intp1 ;

…

delete [] intp2 ;
intp1[0] = 12 ;
```

בקוד זה intp2 מצביע לאותו שטח זיכרון כמו intp1. על-כן עת שחררנו את שטח הזיכרון, באמצעות הפקודה: delete [] intp2; , ישמטנו את השטיח גם מתחת לרגלי 'intp1', כלומר שטח הזיכרון עליו מצביע (עדיין) intp1 כבר אינו עומד לרשותנו, וזו שגיאה לפנות אליו (בַפקודה: intp1[0] = 12;).

פקודת ה- delete אינה מכניסה למצביע ערך NULL. היא מותירה במצביע ערך 'זבל'.

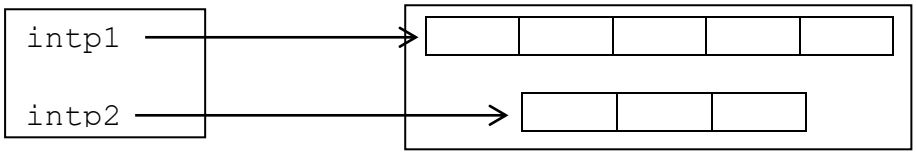
טעות שכיחה אצל מתכנתים מתחילים היא לכתוב קטע קוד כדוגמת הבא:

```
intp2 = new int[ … ] ;
```

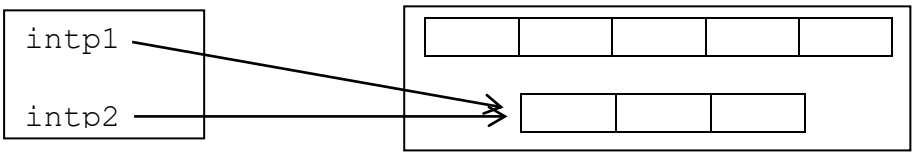
…

```
intp1 = new int[ … ] ;
intp1 = intp2 ;
```

נסביר: הפקודה הראשונה בין השלוש המתוארות מקְצה מערך ל- `intp2`. בהמשך התכנית, (בַּקטע המתואר בשלוש הנקודות), אנו עושים שימושים שונים במערך. עתה אנו רוצים להפנות את `intp1` להצביע לאותו מערך כמו `intp2`, וזה לגיטימי; אולם להקדים לַפקודה: `intp1 = intp2;` את הפקודה: `intp1 = new int[ … ] ;` זו שטות. נסביר מדוע: נתאר את מצב הזיכרון אחרי ביצוע שתי פקודות ההקצאה:



עתה, הפקודה האחרונה: `intp1 = intp2;` יוצרת את המצב הבא:



התוצאה היא שהמערך עליו הצביע `intp1` עד עתה הפך להיות מין zombie, שמחד מצוי בזיכרון, ומאידך אין כל דרך לפנות אליו. לכן אם ברצונכם להפנות מצביע `p` להצביע על מערך עליו מורה מצביע `q`, *לעולם* אל תקדימו לפקודת ההשמה `p = q;` פקודה: `p = new …` .

עד כאן הצגנו את אופן השימוש במצביעים לשם יצירת מערכים. הדוגמות שבהמשך הפרק תסייענה לכם, ראשית, להטמיע את שהוסבר עד כה, ושנית ללמוד כיצד לשלב את השימוש במצביעים ובפונקציות.

11.3 פונקציות המקבלות מצביעים או מערכים

בעבר הכרנו את הפונקציה `strlen`. להזכירכם, פונקציה זאת מקבלת מערך של תווים, ומחזירה את אורכו של הסטרינג השמור במערך (לא כולל ה- `'\0'`). אנו יכולים לכתוב את הפונקציה באופן הבא:

```
int strlen( char s[]) {
   for (int i=0; s[i] != '\0'; i++)
      ;
   return(i) ;
}
```

הסבר: לולאת ה- `for` מתקדמת על המערך כל עוד התו המצוי בתא הנוכחי במערך אינו `'\0'`. בגוף הלולאה מתבצעת הפקודה הריקה. אחרי הלולאה אנו מחזירים את ערכו של `i`.

עתה, שלמדנו את השקילות בין מערכים ומצביעים, אנו יכולים לכתוב את הפונקציה בצורה דומה אך שונה:

```
int strlen( char *s) {
   for (int i=0; s[i] != '\0'; i++)
      ;
   return(i) ;
}
```
הסבר: את הפרמטר תיארנו הפעם בנוטציה מצביעית, במקום בנוטציה מערכית. בשל השקילות בין מערך למצביע הדבר לגיטימי. את גוף הפונקציה לא שינינו. נדגיש כי נוסח זה, כמו הנוסח שראינו לפניו, וכמו אלה שנראה בהמשך, אינו מחייב קריאה עם מערך שהוקצה דינמית או כזה שהוקצה סטטית; ניתן לקרוא לכל אחת מהפונקציות שאנו כותבים בסעיף זה הן עם מערך שהוקצה סטטית על המחסנית, והן עם כזה שהוקצה דינמית על הערמה.

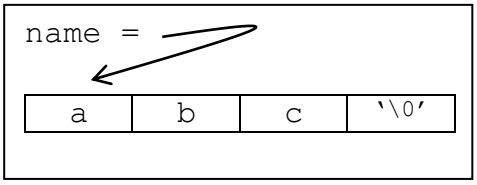
אנו יכולים לכתוב את אותה פונקציה גם בצורה שלישית:

```
int strlen( char *s) {
   for (int i=0; *(s +i) != '\0'; i++)
      ;
   return(i) ;
}
```
הסבר: הפעם גם את תנאי הסיום בלולאה המרנו מנוטציה מערכית לנוטציה מצביעית. כפי שאנו יודעים הביטוי `*(s+i)` פונה לתא מספר `i` במערך עליו מצביע `s`. במילים אחרות `(s+i)` הוא מצביע לתא המצוי בהסטה `i` מהתא עליו מצביע `s`, ולכן `*(s+i)` פונה לתא המצוי בהסטה `i` מהתא עליו מצביע `s`.

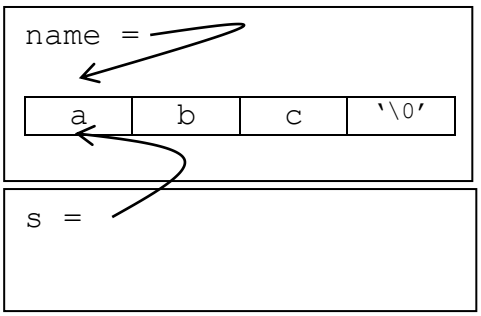
ולבסוף נציג את אותה פונקציה בצורת כתיבה רביעית:

```
int strlen( char *s) {
   for (int i=0; *s != '\0'; i++, s++)
      ;
   return(i) ;
}
```
הסבר: הפעם אנו משתמשים באריתמטיקה על מצביעים. אנו מקדמים את המצביע `s` כל עוד התא עליו הוא מצביע כולל ערך שונה מ- `'\0'`. במקביל אנו סופרים כמה פעמים קידמנו את `s`, וזהו אורכו של הסטרינג.

הגרסה האחרונה שכתבנו מדגימה נקודה אותה חשוב להבין: המצביע המועבר לפונקציה מועבר *כפרמטר ערך*, ועל כן שינויים שהפונקציה עושה לפרמטר שלה s, לא ישפיעו, אחרי תום ביצוע הפונקציה, על הארגומנט עמו הפונקציה נקראה. ארגומנט זה ימשיך להצביע על התא הראשון במערך. עת מצביע מועבר כפרמטר ערך לפונקציה, המחשב פועל באותו אופן בו הוא פועל עת משתנה פרימיטיבי כלשהו מועבר כפרמטר ערך: המחשב *מעתיק* את ערכו של הארגומנט המתאים על הפרמטר המתאים. במילים אחרות הכתובת המוחזקת בארגומנט מועתקת לפרמטר. מבחינה ציורית לפרמטר יוקצה חץ משלו, חץ זה יאותחל להצביע על אותו מקום כמו הארגומנט. נדגים זאת בציור. נניח כי בתכנית הוגדר מערך name[4], ועתה אנו קוראים לפונקציה strlen (בכל אחד מארבעת הנוסחים שכתבנו). נציג את מצב המחסנית טרם הקריאה:



אנו זוכרים כי מערך (כדוגמת name) הוא למעשה מצביע לתא מספר אפס במערך. עתה נציג כיצד תראה המחסנית אחרי בניית רשומת ההפעלה של strlen (נשמיט מרשומת ההפעלה את כתובת החזרה, ואת המשתנה הלוקלי i, שאינם מעניינים אותנו עתה).



הסבר: גם s הוא מצביע (אפילו אם הוא מוגדר בכותרת הפונקציה כ- char s[]), ולכן ציירנו אותו עם חץ בצורת v (כפי שאנו מציירים מצביעים; בניגוד למשולש שחור המשמש לציור פרמטרים משתנים). לאן מצביע s? לאותו מקום כמו הארגומנט המתאים לו, כלומר name. במילים אחרות ערכו של name הועתק על s. אם עתה נסיט את s למקום אחר איננו משנים בכך את המקום אליו מצביע name (כפי שמתאים שיקרה עם פרמטר ערך).

הסבר זה מסייע לנו להבין תופעה שמוכרת לנו כבר מזמן: עת מערך מועבר כפרמטר לְפונקציה, הפונקציה יכולה לשנות את תוכנם של תאי המערך. עתה נניח כי בפונקציה שלנו אנו מבצעים: *s = 'x'; או במילים אחרות: s[0]='x'; מה יקרה? אנו פונים לתא מספר אפס במערך עליו מצביע s ולשם מכניסים את הערך 'x'. ערך זה יוותר כמובן במערך גם אחרי תום ביצוע הפונקציה.

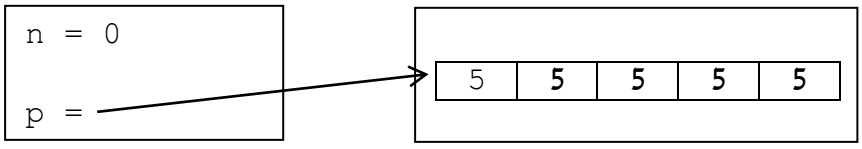
כדי לחדד את התופעה נציג את הפונקציה הבאה:

```
void f( int *pp, int nn) {
 np = 3879; *pp = 17; pp = NULL;
}
```

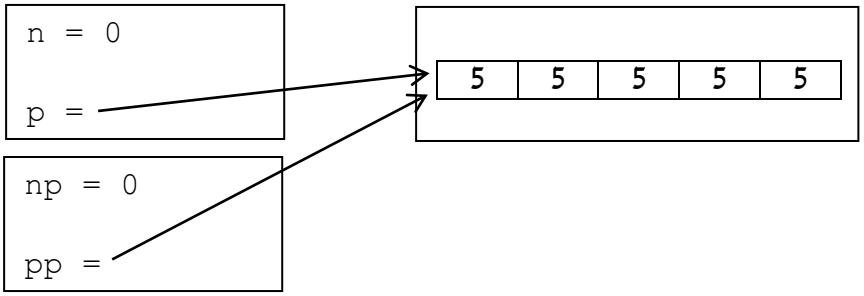
לפונקציה יש שני פרמטרי ערך: הראשון בהם הוא מצביע (במילים אחרות מערך), והשני הוא מספר שלם.

נניח כי בתכנית הראשית הוגדרו: `int n = 0, *p = new int[5];` ,
ולחמשת תאי המערך `p`, הוכנס הערך חמש.

הזיכרון הכולל את משתני התכנית הראשית נראה:

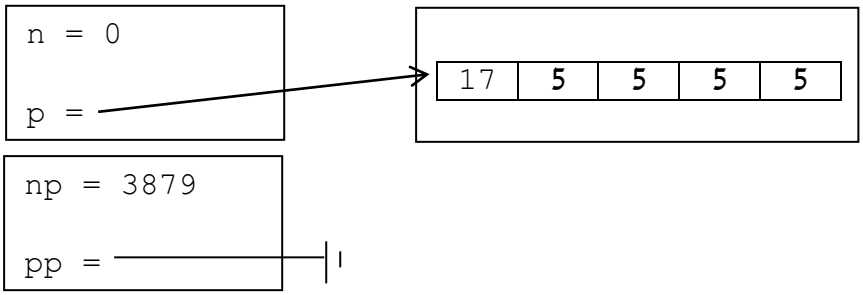


עתה אנו קוראים לפונקציה: `f(p, n);`
בעקבות הקריאה לפונקציה נוספת על-גבי המחסנית רשומת ההפעלה של `f`. נציג את מצב הזיכרון:



נסביר: כפי של- `np` הועתק ערכו של `n`, כך ל- `pp` הועתק ערכו של `p`, ולכן `pp` מצביע לאותו מערך כמן `p`.

עתה הפונקציה `f` מתחילה להתבצע. ההשמה: `np = 3879;` מכניסה ערך ל- `np`, ואינה משנה את ערכו של `n`. ההשמה: `*pp = 17;` (השקולה להשמה: `pp[0] = 17;`) מכניסה את הערך 17 לתא עליו מצביע `pp`. ההשמה: `pp = NULL;` מכניסה ל- `pp` את הערך `NULL`, ואינה משנה את ערכו של `p`. מצב הזיכרון אחרי ביצוע שלוש הפעולות הללו יהיה:



סימון 'ההארקה' לצד `pp` הוא הדרך בה אנו מציירים מצביע שערכו הוא `NULL`.

עתה הפונקציה `f` מסתיימת, ורשומת ההפעלה שלה מוסרת מעל המחסנית. כפי שאנו יודעים השינויים שהפונקציה הכניסה לתוך `np` אינם נותרים ב- `n`. באופן דומה, השינויים שהפונקציה הכניסה ל- `pp` אינם נותרים ב- `p`. ולבסוף, בהתאמה למה שאנו כבר מכירים היטב, שינויים שהפונקציה הכניסה לתוך המערך עליו הורה `pp` נותרים במערך גם אחרי תום ביצוע הפונקציה.

בעבר ראינו כי אם אנו רוצים למנוע מפונקציה המקבלת מערך לשנות את ערכם של תאי המערך אנו יכולים להגדיר את הפרמטר של הפונקציה באופן הבא: `const int a[]`. אם מתכנת כלשהו ינסה בגוף הפונקציה לכתוב פקודה כגון:

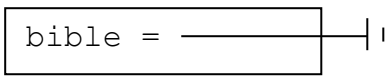
a[0]=17; אזי התכנית לא תעבור קומפילציה (ולכן העברת מערך כפרמטר שהינו קבוע אינה שקולה להעברת משתנה פרימיטיבי כפרמטר ערך). עתה למדנו כי אם פונקציה מקבלת מערך כפרמטר אנו יכולים לתאר את הפרמטר של הפונקציה גם באופן הבא: int *a . גם במקרה זה כדי למנוע מהפונקציה לשנות את ערכם של תאי המערך אנו רשאים לכתוב: const int *a. אולם כתיבה זאת לא תמנע מהפונקציה לשנות את ערכו של המצביע. בגוף הפונקציה נוכל לכתוב פקודה כגון: a = NULL; *a = 3879; ולא יהיה בכך כל פסול. מה שלא נוכל לכתוב הוא: כלומר אין ביכולתנו לשנות את ערכם של תאי המערך עליו מצביע a, אך יש ביכולתנו לשנות את ערכו של a. כדי למנוע מפונקציה לשנות את ערכו של המצביע המועבר לה נכתוב: int * const a . עתה פקודה כגון: a= NULL; תגרום לשגיאת קומפילציה. לעומתה פקודה כגון: *a = 1; היא עתה לגיטימית, שכן הפעם לא אסרנו לשנות את ערכם של תאי המערך עליו מצביע a, רק אסרנו לשנות את המקום עליו a מצביע. כמובן שניתן לשלב את שני האיסורים גם יחד: const int * const a . לבסוף נזכור כי אם מצביע הועבר כפרמטר ערך אזי בדרך כלל לא כל-כך יפריע לנו שהפונקציה תוכל לשנות את ערכו של הפרמטר שלה. שהרי השינוי לא יוותר בארגומנט המתאים לאותו פרמטר.

11.4 מצביע כפרמטר הפניה

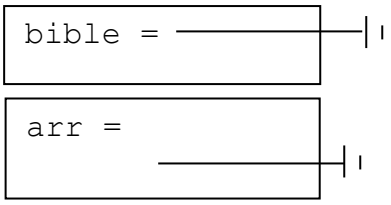
עתה נניח כי ברצוננו לכתוב תכנית אשר מגדירה מצביע (int *p = NULL;), קוראת מהמשתמש גודל מערך רצוי (לתוך המשתנה size), מקצה את המערך (באמצעות פקודת new), ולבסוף, קוראת נתונים לתוך המערך. עוד נניח כי כדי לסמן את סוף המערך (שגודלו אינו ידוע לתכנית הראשית), מציבה הפונקציה בתא האחרון במערך את הערך הקבוע THE_END. נציג את קטע התכנית הבא, אשר מבצע את המשימה באמצעות פונקציה?

```
void read_data(int *arr) {
   int size ;

   cin >> size ;

   arr = new int[size] ;

   for (int i=0; i< size -1; i++)
      cin >> arr[i] ;
   a[size -1] = THE_END ;
}
```

נניח כי אחרי שהגדרנו את: int *bible = NULL; , אנו מזמנים את הפונקציה: read_data(bible); . האם קטע תכנית זה עונה על צרכינו? התשובה היא: לא ולא. נסביר: הפונקציה read_data מקבלת את המצביע כפרמטר *ערך*. על כן שינויים שהפונקציה מכניסה ל- arr, אינם נותרים בתום ביצוע הפונקציה ב- bible, וערכו נותר NULL כפי שהוא היה לפני הקריאה. נציג את התהליך בעזרת המחסנית: נתחיל בהצגת מצב המחסנית עליה מוקצה המשתנה bible של התכנית הראשית:

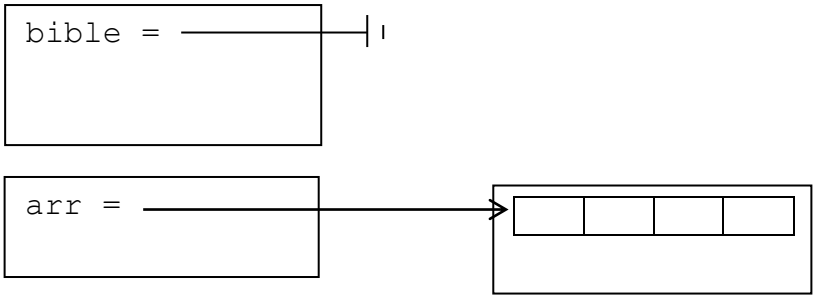


ערכו של bible אותחל בהגדרה להיות NULL, ועל כן ציירנו אותו כמתואר. עתה נערכת קריאה לפונקציה. מצב המחסנית בעקבות הקריאה יהיה הבא (מהציור השמטנו פרטים שאינם מהותיים לצורך דיוננו הנוכחי):

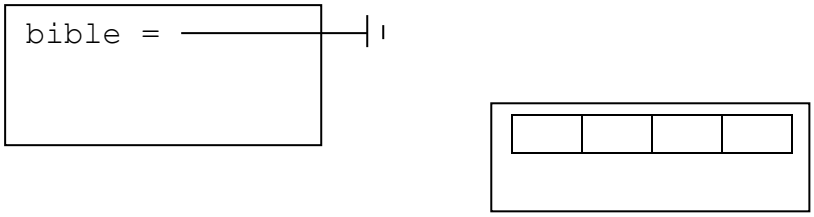


מציור המחסנית השמטנו את המשתנים הלוקליים של read_data, ואת כתובת החזרה. ערכו של arr הוא NULL שכן הארגומנט המתאים ל- arr הוא bible, ערכו של bible הוא NULL, וערך זה *מועתק* ל- arr.

עתה הפונקציה מתחילה להתבצע. הפקודה: arr = new …; יוצרת את המצב הבא:



כלומר על המערך שהוקצה מצביע arr (אך לא bible). אחַר הפונקציה read_data קוראת נתונים לתוך המערך, ועת היא מסתיימת, ורשומת הפעלה שלה מוסרת מעל-גבי המחסנית, מצב הזיכרון הינו:



כלומר המערך הפך ל- zombie שאין דרך לפנות אליו, והמצביע bible נותר בתומתו: משמע ערכו עדיין NULL.

אנו רואים אם כן ש- read_data לא עשתה את המצופה. הסיבה היא שהיא קבלה את המערך כפרמטר ערך. כיצד נשנה את הפונקציה על-מנת שהיא כן תשיג את המבוקש? כלומר שבתום פעולתה המצביע bible יורה על המערך שהפונקציה הקצתה? התשובה היא שעלינו להעביר את המערך כפרמטר *משתנה*; ואז שינויים שהפונקציה תבצע על הפרמטר arr, יוותרו בתום ביצוע הפונקציה בארגומנט המתאים bible.

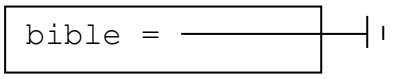
עתה, אם כך, נשאל כיצד מעבירים מצביע כפרמטר משתנה? התשובה היא שבהגדרת הפרמטר, אחרי שם הטיפוס יש להוסיף את התו &. טיפוסו של מצביע ל-

int הוא int *, ועל-כן כדי שהפרמטר מטיפוס int * יהיה פרמטר משתנה עלינו להגדירו כ- int *&.

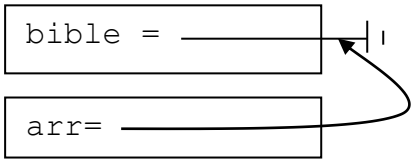
נראה עתה את הפונקציה read_data בגרסתה המתוקנת:

```
void read_data(int *&arr) {
   int size ;

   cin >> size ;

   arr = new int[size] ;

   for (int i=0; i< size -1; i++)
      cin >> arr[i] ;
   arr[size -1] = THE_END ;
}
```

השינוי היחיד שהכנסנו הוא תוספת ה- & אחרי שם הפרמטר. נתאר את התנהלות העניינים מבחינת מצב הזיכרון: ראשית, כמו קודם, מוגדר המשתנה bible של התכנית הראשית. מצב הזיכרון הוא:



עתה נערכת קריאה לפונקציה read_data. אולם בגרסתה הנוכחית ל- read_data יש פרמטר משתנה. כדרכם של פרמטרים משתנים, אנו שולחים חץ (עם ראש משולש שחור), מהפרמטר לארגומנט המתאים. נצייר את מצב המחסנית (תוך שאנו מתרכזים רק בפרמטר arr, ומשמיטים מהציור את יתר הפרטים):



אנו רואים כי החץ נשלח מ- arr, אל bible, וכל שינוי שיערך ב- arr, יתבצע למעשה על bible.

עתה הפונקציה מתחילה לרוץ. פקודת הקצאת הזיכרון תיצור, בגרסה הנוכחית של הפונקציה, את המצב הבא:

כלומר המצביע שערכו השתנה הוא bible, שכן עת הפונקציה בצעה arr = new ... , ומכיוון ש- arr הוא פרמטר משתנה, הלך המחשב בעקבות החץ (עם הראש השחור), וביצע את השינוי על הארגומנט המתאים.

בהמשך קוראת הפונקציה נתונים לתוך המערך. עת הפונקציה מסיימת, ורשומת ההפעלה שלה מוסרת מעל-גבי המחסנית, המערך לא נותר zombie, ו- bible אינו NULL: כי אם bible מצביע על המערך כמבוקש.

11.5 פונקציה המחזירה מצביע

הפונקציה read_data שכתבנו שינתה את הפרמטר היחיד שלה. כפי שמיצינו בעבר, פונקציה אשר מחזירה רק נתון יחיד ראוי לכתוב לא עם פרמטר משתנה, אלא כפונקציה המחזירה את הנתון המתאים באמצעות פקודת return. נראה עתה גרסה של read_data הפועלת על-פי עקרון זה. נקרא לה read_data2. הקריאה לפונקציה מהתכנית הראשית תהיה:

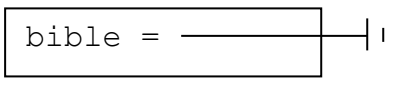
```
bible = read_data2() ;
```

מהקריאה אנו למדים, באופן לא מפתיע, שהפונקציה מחזירה מצביע ל- int (שכן הערך המוחזר על-ידה מוכנס לתוך משתנה שזה טיפוסו). כיצד מוגדרת הפונקציה:

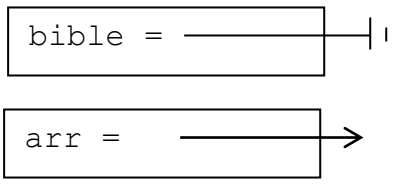
```
int *read_data2(){
   int *arr, size ;

   cin >> size ;

   arr = new int[size] ;

   for (int i=0; i< size; i++)
      cin >> arr[i] ;
   arr[size -1] = THE_END ;

   return( arr ) ;
}
```

נסביר: הפונקציה אינה מקבלת כל פרמטרים. יש לה משתנה לוקלי מטיפוס int * ששמו הוא arr. הפונקציה מקצה מערך, מפנה את המצביע להצביע על המערך, וקוראת נתונים לתוך המערך. בסוף פעולתה הפונקציה מחזירה את ערכו של המצביע (כלומר את הכתובת עליה המצביע מורה). מכיוון שכתובת זאת מוכנסת למשתנה bible של התכנית הראשית, אזי מעתה bible יצביע על המערך שהוקצה.

כדרכנו נציג את התנהלות העניינים בזיכרון. נתחיל עם המחסנית עליה הוגדר המשתנה bible של התכנית הראשית:

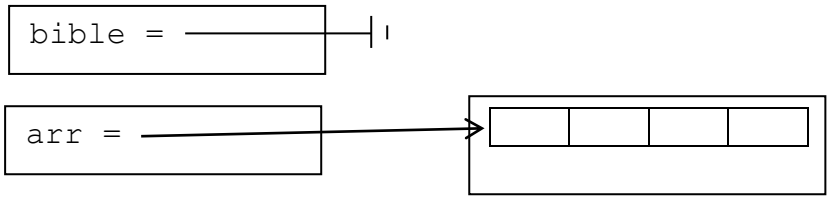


עתה נערכת קריאה לפונקציה read_data2. נציג את מצב המחסנית (תוך התרכזות בעיקר בלבד):

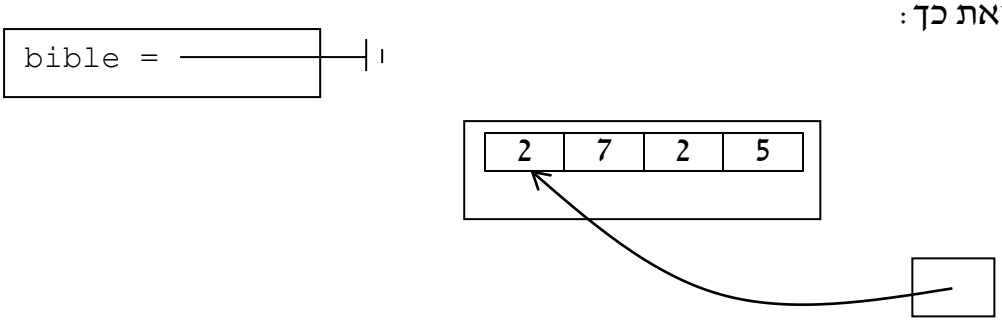


הסבר: בפונקציה מוגדר משתנה לוקלי בשם arr. המשתנה אינו מאותחל בהגדרה, ועל כן הוא מצביע למקום מקרי כלשהו.

עתה הפונקציה מתחילה לרוץ, בפרט היא מקצה מערך, ומפנה את arr להצביע עליו. מצב הזיכרון הוא עתה:

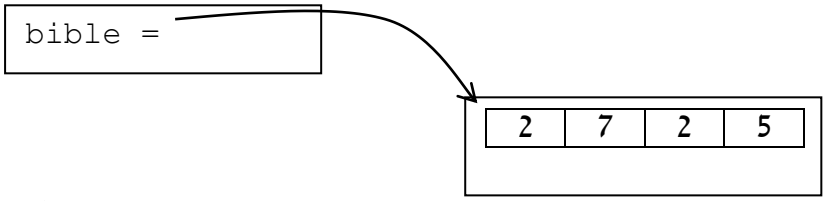


אַחר הפונקציה קוראת ערכים לתוך המערך, ולבסוף היא מסיימת, תוך שהיא מחזירה את arr. אנו זוכרים כי עת פונקציה מחזירה ערך, הערך שהיא מחזירה 'מושאר בצד' (ואחר מוכנס לַמשתנה המתאים). בַּמקרה של מצביע הערך ש-'מושם בצד' היא הכתובת אליה המצביע מורה, ומבחינה ציורית המקום עליו החץ מורה. נציג זאת כך:



המלבן הימני התחתון מציג את הערך ש- 'הושאר בצד', והחץ שבו מורה לאותו מקום עליו הורה arr.

עתה, מכיוון שהקריאה לפונקציה הייתה: bible = read_data2(); , אזי הערך ש-'הושאר בצד' מוכנס למשתנה bible, כלומר bible עובר להצביע על המערך שהוקצה על-ידי הפונקציה. והציור המתאים:



ושוב, השגנו את האפקט הרצוי: בתום ביצוע הפונקציה מצביע bible על מערך שהוקצה על-ידי הפונקציה, ומכיל נתונים כנדרש.

11.6 שינוי גודלו של מערך

עתה ברצוננו לכתוב פונקציה אשר תוכל לקרוא סטרינג באורך לא ידוע. הפונקציה תקרא את תווי הסטרינג בזה אחר זה עד קליטת התו מעבר-שורה, אשר יציין את סופו של הסטרינג. הקריאה לפונקציה תהיה: a_s = read_s(); , עבור משתנה char *a_s; . כלומר הפונקציה תחזיר מצביע לַסטרינג שהיא קראה לתוך מערך שהוקצה דינמית על-גבי הערמה.

הפונקציה שונה מזו שראינו בסעיף הקודם בכך שהיא מקצה מערך בגודל N תאים, וקוראת לתוכו תווים. עת המערך מתמלא הפונקציה מגדילה אותו בשיעור N וממשיכה בתהליך הקריאה; וכך שוב ושוב כל עוד יש בכך צורך.

נציג את הפונקציה, ואחר נסבירה ביתר פירוט:

```
char *read_s() {
   char *the_s,    // pointer to the array the func alloc
        *temp ;
   int arr_size,   // the size of the_s[]
       s_size ;    // how many chars were read, so far

   the_s = new char[N] ;        // alloc initial array
   arr_size = N ;

   do {              // read the string until u read '\n'
      if (s_size == arr_size) {     // if the_s[] is full
         temp = new char[arr_size + N] ;// alloc bigger 1
         for (int i = 0; i < s_size; i++)
            temp[i] = the_s[i] ;    // copy old onto new
         delete [] the_s ;          // free old
         the_s = temp ;             // the_s point to new
         arr_size += N ;            // update size
      }
      the_s[ s_size++ ] = cin.get() ;
   } while (the_s[s_size -1] != '\n') ;

   the_s[s_size -1] = '\0' ;
   return the_s ;
}
```

נסביר את הפונקציה (אף שתיעדנו אותה תיעוד יתֵר כדי להקל על הבנתה): אנו מתחילים בכך שאנו מקצים למצביע `the_s` מערך (וכמובן בודקים שההקצאה הצליחה!). המשתנה `arr_size` שומר את גודלו של המערך שהוקצה. עתה אנו נכנסים ללולאת `do-while` אשר קוראת תו תו עד קליטת `'\n'`. בכל סיבוב בלולאה אנו בודקים האם כבר אין מקום במערך להוספת תו נוסף (כלומר האם `s_size == arr_size`). אם אכן זה המצב אנו: (א) מקצים מערך *חדש* בגודל `arr_size +N`, כלומר מערך הגדול ב- `N` תאים מהמערך הנוכחי, ומפְנים את המשתנה `temp` להצביע על המערך החדש. (ב) אנו מעתיקים לַמערך החדש את תוכנו של המערך הישן. (ג) אנו משחררים את המערך הישן (באמצעות פקודת ה-`delete`). (ד) אנו מפְנים את המצביע `the_s` (שכבר אינו מצביע על מערך כלשהו) להצביע על המערך עליו מורה `temp`. (ה) אנו מעדכנים את `arr_size` על-ידי הגדלתו בשיעור `N`. אחרי שהגדלנו (או לא הגדלנו) את המערך, אנו פונים לקריאת התו הבא מהקלט.

11.7 מערך של מצביעים כפרמטר ערך

בסעיפים הקודמים ראינו כי במקום להגדיר מערך סטטי: `char s[N];` אנו יכולים להגדיר מצביע: `char *sp;` אשר בהמשך יהפוך למערך (שמוקצה דינמית). אנו גם זוכרים כי בשפת `C` מערך דו-ממדי הוא למעשה סדרה של מערכים חד-ממדיים, או במילים אחרות מערך שכל תא בו הוא מערך חד-ממדי. אם נצרף את שתי המסקנות הללו יחד נוכל להסיק שבמְקום להגדיר מערך דו-ממדי: `char ss[N][M] ;`, אנו יכולים להגדיר מערך של מצביעים: `char *ssp[N];`. כל תא במערך זה הוא מצביע (כלומר יֵצור מטיפוס `char *`) שיכול בהמשך להפוך למערך חד-ממדי, אשר יוכל לשמור סטרינג.

עתה נניח כי הגדרנו בתכנית הראשית את המשתנה: `char *strings[N];` (כלומר הגדרנו מערך של מצביעים, שהינו מערך של פוטנציאלים למערכים חד-ממדיים, כלומר פוטנציאל למערך דו-ממדי); אחר אתחלנו את תאיו (שכל אחד מהם הוא מצביע) להכיל את הערך NULL:

```
for (int i=0; i < N; i++)
   strings[i] = NULL ;
```

עתה אנו קוראים לפונקציה: `.read_strings(strings);`

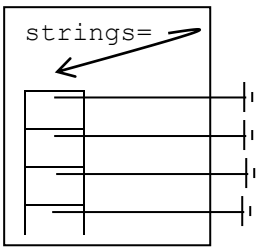
הגדרת הפונקציה:

```
void read_strings( char *strings[N] ) {
   char temp[M] ;
   int i = 0 ;

   while (i < N) {
      cin >> temp ;
      if (strcmp(temp, ".") == 0)
         break ;
      strings[i] = new char[ strlen(temp) +1 ] ;
      strcpy(strings[i++], temp) ;
   }
}
```

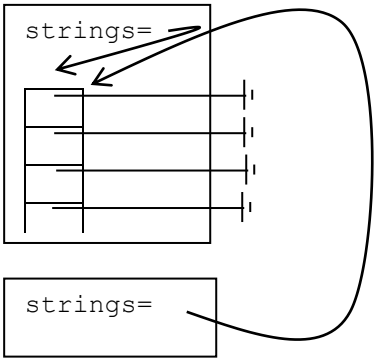
נסביר: הפונקציה מתנהלת כלולאה. בכל סיבוב בלולאה אנו קוראים סטרינג לתוך משתנה העזר temp (שהינו מערך סטטי). במידה והסטרינג הוא "." אנו עוצרים את תהליך הקריאה (ולכן גם את ביצוע הפונקציה באופן כללי). במידה והסטרינג אינו "." אנו מקצים מערך בַגודל הדרוש (כלומר כאורכו של הסטרינג השמור ב-temp ועוד תא אחד, בו יאוחסן ה- '\0'), מפְנים את strings[i] להצביע על אותו מערך, ומעתיקים את הסטרינג המצוי ב- temp על strings[i]. שימו לב שהתוצאה המתקבלת היא לא בדיוק מערך דו-ממדי, אלא מערך בו כל שורה היא באורך שונה.

עתה ברצוננו לשאול שאלה כללית יותר: האם הפונקציה תבצע את משימתה כהלכה? האם טיפוס הפרמטר של הפונקציה הוא כנדרש? האם שינויים שהפונקציה תכניס לתוך הפרמטר strings יישארו בארגומנט המתאים בתום ביצוע הפונקציה? התשובה היא: כן! וכן!. נסביר מדוע: אנו יודעים היטב כי שינויים שפונקציה מכניסה לתוך תאי מערך שהינו פרמטר של הפונקציה נשארים בתום ביצוע הפונקציה בארגומנט המתאים; וְכלל זה נכון גם עבור מערך שכל תא בו הוא מצביע (אשר הופנה בַפונקציה להצביע על סטרינג שהוקצה דינמית על הערמה). נתאר את השתלשלות העניינים בזיכרון.

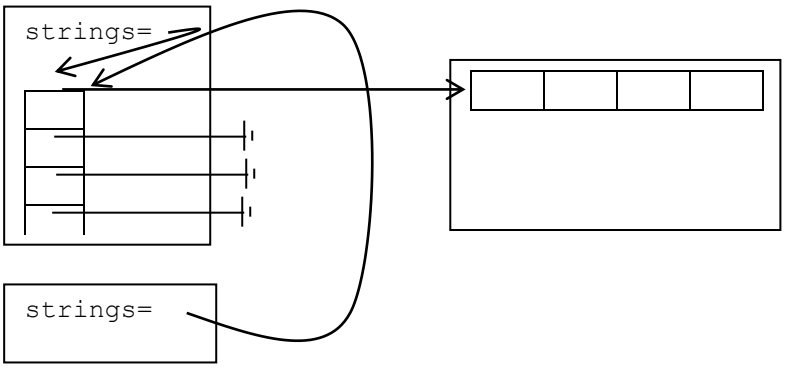
ראשית, בתכנית הראשית, על-גבי המחסנית, מוגדר מערך של מצביעים, ותאי מאותחלים לערך NULL. נציג את המחסנית:



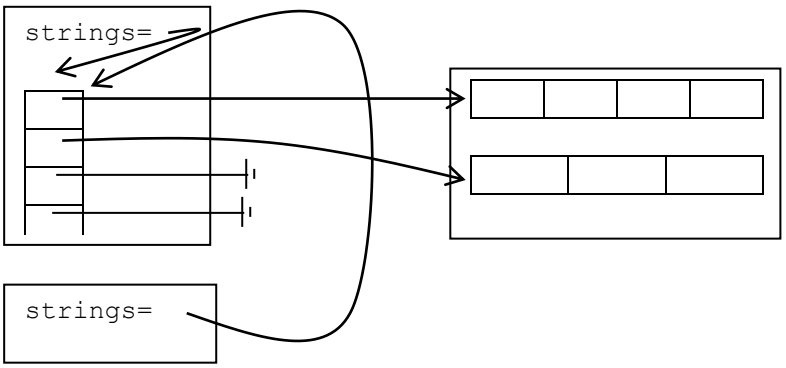
עתה מתבצעת קריאה לפונקציה. המערך מועבר כפרמטר ערך, כלומר ערכו של הארגומנט strings, מועתק על הפרמטר strings. מבחינה ציורית משמעות הדבר היא שהפרמטר יצביע לאותו מקום כמו הארגומנט. נציג את מצב המחסנית:



עת הפונקציה מבצעת פקודה כגון: `strings[0] = new char[4];` מתקבל המצב הבא בזיכרון:



כלומר המצביע שבתא מספר אפס במערך המצביעים המקורי (והיחיד הקיים) הוא שעובר להצביע על הסטרינג המוקצה. אם אחרי קריאת הסטרינג השני יוקצה מערך בגודל שלושה תאים אזי מצב הזיכרון יהיה:



בַּפונקציה read_strings שכתבנו הגדרנו את הפרמטר באופן הבא: char *strings[N] . כפי שאנו יודעים במקום לתאר פרמטר כמערך (ואפילו הוא מערך של מצביעים) אנו יכולים לתארו כמצביע. לכן במקום להגדיר את הפרמטר כ: char *strings[N] אנו יכולים להגדירו כ: char **strings, במילים: כמצביע למצביע לתו (מבחינה תחבירית אנו ממרים [N] ב- *). גוף הפונקציה לא יצטרך לעבור שינוי גם אם נבחר לתאר את הפרמטר באופן זה.

11.8 מערך של מצביעים כפרמטר הפניה

מערך המצביעים char **strings, שראינו בסעיף שעבר, הועבר כפרמטר ערך, ולכן ביכולתה של הפונקציה היה לשנות רק את ערכם של תאי המערך, אך לא את גודלו של המערך. נניח כי ברצוננו לכתוב פונקציה אשר קוראת סדרת מחרוזות. מספר המחרוזות אינו ידוע למתכנת, ולכן בתכנית הראשית לא ניתן להגדיר משתנה: char *strings[N], שכן משתנה שכזה מגדיר את מספר המחרוזות להיות לכל היותר N. מה יהיה הפתרון? אנלוגי לזה שראינו עם מערך חד-ממדי שגודלו היה לא ידוע למתכנת: במקום להגדיר מערך char s[N], הגדרנו מצביע char *s (כלומר פוטנציאל למערך. מבחינה תחבירית המרנו את [N] ב- *). את המצביע העברנו כפרמטר משתנה לְפונקציה אשר יכלה לממש אותו לכדי מערך בַּגודל הרצוי לה (ואף לשנות את גודלו של המערך עת זה מה שנדרש). גם עם מערך של מצביעים ננקוט באותו פתרון: בְּמקום להגדיר מערך סטטי של מצביעים char *strings[N] נגדיר מצביע למצביע char **strings. (שוב המרנו את ה- [N] ב- *). את המצביע למצביע נעבר כפרמטר משתנה לפונקציה אשר תקצה מערך של מצביעים בגודל הרצוי (ואף תוכל לשנות את גודלו של המערך עת זה מה שיידרש). כיצד מתארים פרמטר משתנה שהינו מצביע למצביע? כותבים אחרי שם הטיפוס, לדוגמה אחרי char **, את התו &.

נציג עתה את הפונקציה alloc_arr_n_read_strings אשר תקבל מצביע למצביע כפרמטר משתנה, תקרא מהמשתמש כמה מחרוזות ברצונו להזין, תקצה מערך של מצביעים בגודל הדרוש, ואחַר תקרא סדרת סטרינגים תוך שימוש בפונקציה read_strings שכתבנו בסעיף שעבר, ואשר מקבלת מערך מוכן של מצביעים. בפונקציה read_strings עלינו לבצע שינוי קל: הפונקציה תקבל פרמטר ערך נוסף, int arr_size, אשר יורה לה מה גודלו של מערך המצביעים המועבר לה. (להזכירכם, בגרסה שכתבנו הנחנו כי גודלו של המערך הוא הקבוע N, אשר מוכר לפונקציה מהֵיותו גלובלי). הפונקציה read_strings תקְרא לכל יותר arr_size מחרוזות (בְּמקום לכל היותר N מחרוזות, כפי שהיא עשתה במקור).

הפונקציה alloc_arr_n_read_strings שנכתוב תחזיר באמצעות פרמטר משתנה את גודלו של מערך המצביעים שהוקצה.

בתכנית הראשית נגדיר, אם כן:

```
char **arr_of_strings ;
int arr_size ;
```

ונזמן את הפונקציה באופן הבא:

```
alloc_arr_n_read_strings(arr_of_strings, arr_size) ;
```

הגדרת הפונקציה:

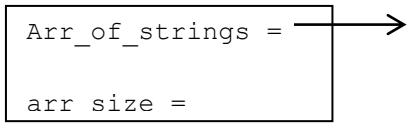
```
void alloc_arr_n_read_strings(char **&arr_of_strings,
```

```
                                    int &arr_size)
{
   cin >> arr_size ;
   arr_of_strings = new char *[arr_size] ;

   for (int i = 0; i < arr_size; i++)
      arr_of_strings[i] = NULL ;

   read_strings( arr_of_strings, arr_size ) ;
}
```

הסבר: הפונקציה מקבלת את המצביע למצביע (כלומר את הפוטנציאל למערך של מצביעים) כפרמטר משתנה. לכן שינויים שהפונקציה תערוך לַפרמטר שלה יוותרו בתום ביצוע הפונקציה בארגומנט המתאים. פקודת ה- `new` המבוצעת על-ידי הפונקציה מקצָה מערך של מצביעים. שימו לב כי עת אנו כותבים `new char[…]` אנו מקצים מערך של תווים, ועת אנו כותבים `new char *[…]` אנו מקצים מערך של מצביעים לתווים. כמובן שאחרי ביצוע ההקצאה יש לבדוק שערכו של המצביע שונה מ- `NULL`, בדיקה שאנו משמיטים לשם הקיצור.

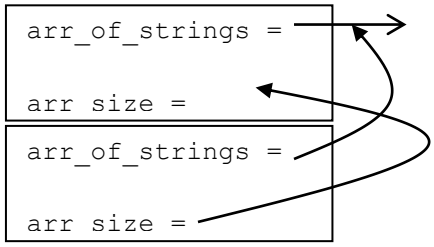
אחרי שהקאצנו מערך של מצביעים, ואיתחלנו את תאיו לערך `NULL`, אנו יכולים לזמן את הפונקציה `read_strings` אשר מצפה לקבל מערך קיים של מצביעים.

נציג את התנהלות התכנית מבחינת ציור הזיכרון. כמו תמיד נתרכז רק בַהיבטים המשמעותיים לצורך דיוננו. בַתחילה, אנו מגדירים בַתכנית הראשית מצביע למצביע `arr_of_strings`, ומשתנה שלם `arr_size`. נציג את מצב המחסנית:

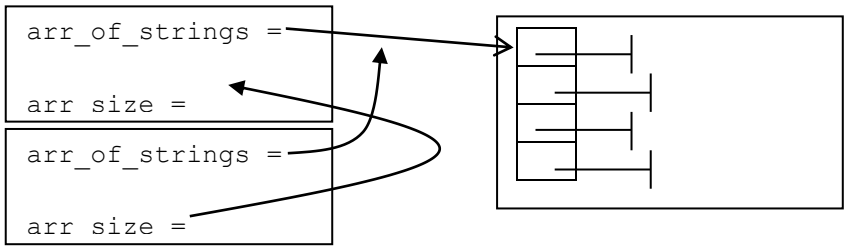


מכיוון שהמצביע לא אותחל ציירנו אותו כמורה למקום מקרי.

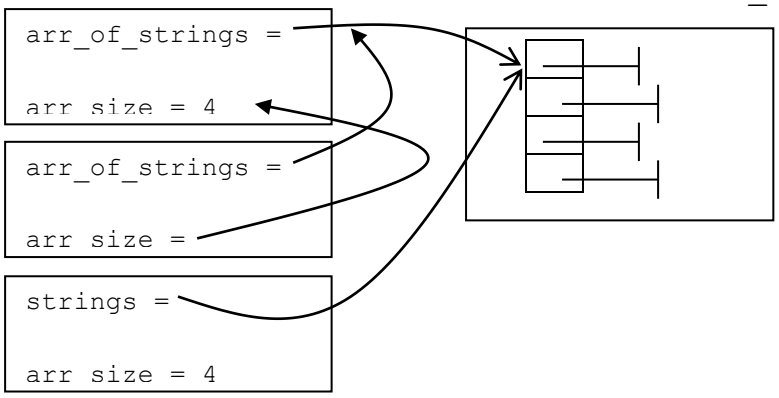
עתה אנו מזַמנים את הפונקציה `alloc_arr_n_read_strings`. לַפונקציה יש שני פרמטרים משתנים, ועל כן אנו שולחים חץ (עם ראש בצורת משולש שחור) מכל אחד משני הפרמטרים אל הארגומנט המתאים לאותו פרמטר. נציג את מצב המחסנית:



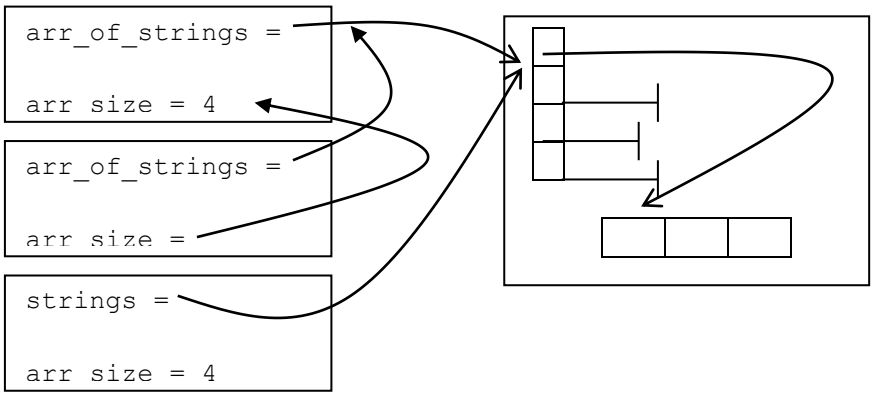
עתה הפונקציה מתחילה להתבצע. עת הפונקציה מבצעת את פקודה ה- new מוקצה על-גבי הערמה מערך של מצביעים. מי יצביע על מערך זה? הפרמטר arr_of_strings של הפונקציה הוא פרמטר משתנה (חץ עם ראש משולש שחור), כדרכו של פרמטר משתנה אנו הולכים בעקבות החץ ומטפלים בארגומנט המתאים, כלומר במשתנה arr_of_strings של התכנית הראשית; ומצביע זה (שכדרכו של מצביע צויר עם חץ שראשו בצורת v) הוא שמצביע על המערך שהוקצה. נתאר זאת ציורית:



אחַר, אנו מאתחלים את תאי המערך (שהִנם, כזכור לנו, מצביעים לתווים). ולכן בציור תארנו כל תא כמורה על NULL. בשְלב הבא אנו קוראים לפונקציה read_strings ומעבירים לה את מערך המצביעים כפרמטר ערך, כלומר הפרמטר של הפונקציה (שהנו חץ עם ראש בצורת v) יצביע גם הוא על מערך המצביעים. גם גודלו של המערך מועבר כפרמטר ערך, ולכן הערך המתאים מועתק לפרמטר arr_size. נציג זאת בציור:



עת הפונקציה read_strings מתחילה להתבצע אנו חוזרים למצב שתיארנו בסעיף הקודם. לדוגמה, עת הפונקציה מבצעת: strings[0] = new char[3]; מוקצה מערך, ותא מספר אפס בַמערך strings עובר להצביע על מערך זה. הציור המתאים יהיה:



11.9 מצביעים במקום פרמטרי הפניה

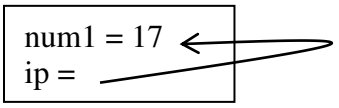
בעבר, ראינו את הפונקציה swap אשר מחליפה בין ערכם של שני הפרמטרים המועברים לה. כתבנו אותה באופן הבא:

```
void swap( int &var1, int &var2) {
   int temp = var1 ;
   var1 = var2 ;
   var2 = temp ;
}
```

כלומר לפונקציה swap יש שני פרמטרים משתנים. אם בתכנית מוגדרים: int a, b ; ולתוך זוג המשתנים הכנסנו ערכים, אזי אנו יכולים לקרוא לפונקציה swap(a, b); והפונקציה תחליף בין ערכיהם של a ושל b.

יש המסתייגים מפונקציה כדוגמת swap הנ"ל. המסתייגים טוענים כי מהקריאה לפונקציה (swap(a, b); לא ניתן לדעת שהפרמטרים של הפונקציה משתנים, ולכן: (א) לַפונקציה יש את הכוח לשַנות את ערכם של הפרמטרים, וכן: (ב) לא ניתן לזמן את הפונקציה באופן הבא: swap(a, 17); . המסתייגים טוענים כי עת תכניות נעשות גדולות מאוד, ונכתבות לכן על-ידי צוות של מתכנתים, יש חשיבות עליונה לכך שמאופן הקריאה לַפונקציה נוכל לדעת את סוג הפרמטרים של הפונקציה. מצביעים עשויים לבוא לעזרנו גם בסוגיה זאת.

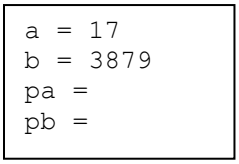
התחלנו את דיוננו בנושא מצביעים בכך שראינו שניתן להפנות מצביע להצביע על משתנה קיים. אם הגדרנו בתכנית: int num1 =17, *ip ; , אזי הפקודה: ip = &num1; תפנה את המצביע ip להורות על המשתנה num1. מבחינה ציורית אנו מתארים זאת כך:



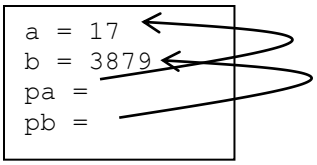
בַּעקרון שהצגנו בפִסקה האחרונה נעשה שימוש כדי לכתוב את swap בצורה שונה, שתענה על דרישות המסתייגים. שוב נניח כי בתכנית הוגדרו המשתנים: int a, b; ולצורך דיוננו הנוכחי נניח כי הוגדרו גם: int *pa, *pb; . עוד נניח כי ל- a ול- b הוכנסו ערכים, ועתה ברצוננו להחליף בין ערכי המשתנים הללו. נוכל לבצע זאת בדרך הבאה: ראשית בתכנית הראשית נכתוב: pa = &a; pb = &b; . ועתה נזמן את הפונקציה: new_swap(pa, pb); . נציג את הגדרת הפונקציה new_swap:

```
void new_swap( int *pvar1, int *pvar2) {
   int temp = *pvar1 ;
   *pvar1 = *pvar2 ;
   *pvar2 = temp ;
}
```

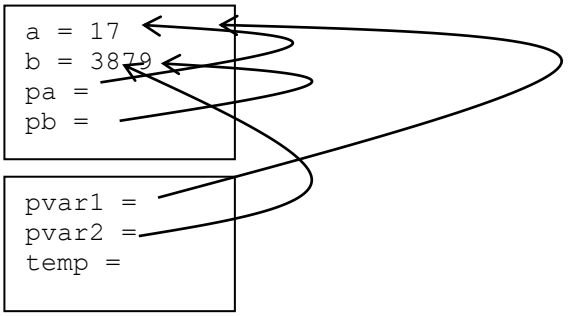
עתה נבחן מדוע ואיך הפונקציה שהצגנו עושה את מלאכת ההחלפה בין הערכים. נתחיל בהצגת המחסנית הכוללת את ארבעת משתני התכנית הראשית. וננִיח כי למשתנים a, b הוכנסו ערכים כמתואר בציור:



הפקודות: pa = &a; pb = &b; תְצורנה את המצב הבא:



כלומר כל מצביע מורה על המשתנה המתאים לו. עתה אנו מזמנים את הפונקציה new_swap(pa, pb); אשר מקבלת את שני המצביעים כפרמטרי ערך, ולכן ערכם של הארגומנטים pa, pb מועתק על הפרמטרים pvar1, pvar2 בהתאמה; מבחינה ציורית, pvar1, pvar2 יצביעו גם הם על אותם מקומות כמו pa, pb, כלומר על a, b. נציג את מצב המחסנית:



עתה הפונקציה מתחילה להתבצע: הפקודה: temp = *pvar1; מכניסה ל-temp את הערך המצוי בתא עליו מצביע pvar1, כלומר את הערך 17. הפקודה: *pvar1 = *pvar2; מכניסה לתא הזיכרון עליו מצביע pvar1 את הערך המצוי בתא הזיכרון עליו מצביע pvar2, לכן לתוך תא הזיכרון a מוכנס ערכו של התא הזיכרון b, כלומר 3879. לבסוף הפקודה: *pvar2 = temp; מכניסה לתוך תא הזיכרון עליו מצביע pvar2, כלומר לתוך תא הזיכרון b, את הערך 17 השמור ב-temp. בכך השלימה הפונקציה את פעולתה בהצלחה.

עתה נשאל מדוע new_swap עדיפה על-פני swap? התשובה היא שעת אנו מעבירים מצביע לְמשתנה הדבר ניכר לעין גם בקריאה לַפונקציה, ומרמז על כך שהפונקציה עתידה לשנות את ערכו של המשתנה, שהרי אחרת היינו מעבירים לה את ערכו של המשתנה (כלומר היה לה פרמטר ערך), ולא מצביע לַמשתנה. מטעמים דידקטיים אנו קראנו לפונקציה new_swap בשני שלבים: ראשית הפנינו את המצביעים: pa, pb להצביע על a, b, ושנית קראנו לפונקציה עם pa, pb. למעשה ניתן לקרוא לפונקציה בצעד אחד: new_swap(&a, &b); . בצורת קריאה זאת אנו מוותרים על משתני העזר pa, pb, הקריאה לפונקציה ראשית מחשבת את כתובתם של a ושל b, ושנית משימה בפרמטרים pvar1, pvar2 את הכתובות הללו, באופן שהפרמטרים מצביעים על הארגומנטים המתאימים כנדרש. איחוד שני הצעדים לכדי צעד אחד מדגיש יותר את העובדה כי new_swap מקבלת מצביעים לַמשתנים a,b (ולכן סביר להניח שהיא עתידה לשנות את ערכם).

11.10 מצביע מדרגה שנייה (int **) כפרמט קבוע (const)

פוני שאמורה לקבל int ** לקריאה בלבד תקבלו: const int * const *
הסיבה: נניח תסריט:

```
const int one = 1 ;
int *p = NULL ;
const int **p2p = &p ;    // ← assume this is legel
                          // (actually, fortunately, it is not).

*p2p = &one               // legal as both have the same type:
                          // const int *

                          // but now p points to one
*p = 2 ;                  // BAD, but legal, p's type is int *
```

11.11 פרמטרים המועברים ל- main: argc ו- argv

בפרק תשע ראינו כי כדי לשמור מערך של N סטרינגים כל אחד באורך של לכל היותר M-1 תווים אנו מגדירים מערך דו-ממדי של תווים: `char s[N][M];` .
בפרק הנוכחי ראינו כי במקום להגדיר מערך סטטי כנ"ל אנו יכולים להגדיר מערך של מצביעים: `char *s[N];` . מערך זה יוכל לשמור לכל היותר N סטרינגים, כל אחד מהם באורך שייקבע במהלך ריצת התכנית. אחר ראינו כי במקום להגדיר את גודלו של המערך להיות בן N מצביעים אנו יכולים להגדיר מצביע למצביע: `char **s;` . באופן זה נוכל תוך כדי ריצת התכנית לקבוע (ולשנות) הן את גודלו של המערך, והן את אורכו של כל סטרינג וסטרינג שישמֵר במערך.

בעבר העלנו את השאלה האם גם לתכנית הראשית, `main`, ניתן להעביר ארגומנטים? ענינו אז על השאלה בחיוב, אך לא יכולנו להסביר כיצד בדיוק הדבר מתבצע. עתה חכמנו מספיק כדי ללמוד כיצד הדבר מתבצע. נניח כי ברצוננו לכתוב תכנית אשר מקבלת מספר כלשהו של מספרים שלמים, ומציגה את סכומם. נוכל לעשות זאת באופן הבא:

א. נכתוב תכנית, אותה נציג מייד, ונשמור את התכנית בקובץ בשם `sum.cpp` . (הסיומת `cpp` מעידה כי זהו קובץ המכיל תכנית מקור בשפת c plus plus, התחילית sum היא שם שנתנו לקובץ כדי להעיד על מהות התכנית המצויה בו).
ב. נקמפל את הקובץ sum.cpp, ונייצר קובץ בשפת מכונה שיקרא sum.exe . (הסיומת exe מעידה כי קובץ זה מכיל תכנית ניתנת להרצה, executable code).
ג. אם עבדנו בסביבת עבודה אזי עתה נצא מסביבת העבודה, ונעבור לעבוד אל מול מערכת ההפעלה באופנות הקרויה command line, כלומר לאופנות בה אנו מקלידים (בשורה, אות אחר אות) את הפקודות שברצוננו לבצע. בסביבת חלונות, למשל, אנו עוברים לעבוד בצורת עבודה של MS-DOS prompt.
ד. נתמקם במחיצה בה מצוי הקובץ sum.exe .
ה. נריץ את התכנית באופן הבא: sum 3879 17 9 או באופן הבא: sum –6 4 1 1 , כלומר תוך שאנו כותבים את שם התכנית (sum) ואחר את סדרת

המספרים השלמים שברצוננו שהתכנית תסְכום. לשם הפשטות נניח כי התכנית מורצת עם סדרת מספרים תקינה.

עתה נציג את התכנית, ואחר נסבירה:

```
#include <iostream.h>
#include <stdlib.h>                    // needed for atoi()

int main(int argc, char **argv) {
   int sum = 0 ;

   for (int i = 1; i < argc; i++)
      sum += atoi(argv[i]) ;

   cout << sum << endl ;

   return EXIT_SUCCESS ;
}
```

אנו רואים כי לתכנית הראשית יש שני פרמטרים:

א. הפרמטר `argc` (קיצור של argument counter) מעיד כמה סטרינגים כוללת הפקודה באמצעותה הורצה התכנית. מספר הסטרינגים יהיה לכל הפחות אחד: שם התכנית. במידה והמשתמש הקליד בנוסף לשם התכנית סטרינגים נוספים יהיה ערכו של `argc` כמספר הסטרינגים שהוקלדו (כולל שם התכנית). לדוגמה אם התכנית מורצת באופן: sum 643 –49 –3 1 אזי ערכו של `argc` הוא חמש, שכן הוזנו חמישה סטרינגים: (א) sum, (ב) 643, (ג) -49, (ד) -3, (ה) 1.

ב. הפרמטר `argv` (קיצור של argument vector) הוא מערך של סטרינגים. התא מספר אפס בַמערך כולל את שם התכנית (בדוגמה שלנו sum). התאים הבאים כוללים את הסטרינגים הנוספים, זה אחר זה.

בתכנית שלנו אנו מגדירים משתנה בשם `sum` שיכיל את סכום הארגומנטים. אחר אנו מבצעים לולאה שעוברת על הארגומנטים, החל בארגומנט מספר אחד (שכן `argv[0]` כולל, כזכור, את שם התכנית), עבוּר כל אחד ואחד מארגומנטים הלולאה ממירה את הסטרינג המצוי בתא המתאים במערך `argv` לכדי ערך שלם (באמצעות הפונקציה `atoi`, ששמה הוא קיצור של `ascii to integer`), ומוסיפה את הערך השלם לַמשתנה `sum`. לבסוף הפונקציה מציגה את ערכו של `sum`.

נרְאה דוגמה נוספת: נניח כי ברצוננו לכתוב תכנית בשם `prog` אשר מקבלת שני סטרינגים ומודיעה האם כל אחד ואחד מתווי הסטרינג הראשון מצוי בסטרינג השני. לדוגמה עבור ההרצה: `prog abbca cba` תציג התכנית את הפלט 'כן', שכן כל אחד ואחד מתווי הסטרינג `abbca` מצוי סטרינג `cba`. לעומת זאת עבור ההרצה: `prog xyz xy` יוצג הפלט 'לא', שכן התו 'z' מופיע בסטרינג הראשון אך לא בשני.

התכנית תיכתב באופן הבא:

```
int main(int argc, char **argv) {
   if (argc != 3) {
      cout << "Program should be run:  prog "
           << " <string> <string> \n" ;
```

```
      return( 1 );
   }

   for (int i1 = 0; i1 < strlen( argv[1] ); i1++)
   {
      for (int i2 = 0; i2 < strlen( argv[2] ); i2++)
         if (arvg[2][i2] == argv[1][i1])
            break ;
      if (i2 == strlen( argv[2] ) )
      {   cout << "no\n" ; return(0) ; }
   }
   cout << "yes\n" ;
   return(0) ;
}
```

נסביר: ראשית, אנו יודעים כי בתכנית זאת ערכו של `argc` צריך להיות בדיוק שלוש (שם התכנית פלוס שני הסטרינגים המועברים כארגומנטים לתכנית), לכן אם ערכו של `argc` שונה משלוש סימן שהמשתמש לא הריץ את התכנית כהלכה; אנו משגרים לו הודעת שגיאה, ועוצרים את ביצוע התכנית.

בַהמשך אנו נכנסים ללולאה כפולה. הלולאה החיצונית עוברת על כל תווי `argv[1]`. עבור כל תו ותו הלולאה בודקת האם הוא מצוי ב- `argv[2]`. עבור כל ערך קבוע של `i1`, כלומר עבור כל תו של `argv[1]` הלולאה הפנימית בודקת האם תא זה מצוי ב- `argv[2]`. הלולאה הפנימית רצה על כל תאי `argv[2]` ובמידה והיא מוצאת תא `argv[2][i2]` שערכו שווה ל- `argv[1][i1]` היא מסיקה שהתו המתאים ב- `argv[1]` מצוי ב- `argv[2]`. במקרה שכזה אנו שוברים את ביצוע הלולאה הפנימית. אחרי הרצת הלולאה הפנימית, אם ערכו של `i2` הוא כאורכו של הסטרינג `argv[2]` סימן שהתו לא נמצא. במקרה שכזה אנו מציגים הודעה מתאימה למשתמש, ועוצרים את ביצוע התכנית. מנֻגד, אם מיצינו את הלולאה החיצונית, בלי שהודענו כי תו כלשהו מ- `argv[1]` אינו מצוי ב- `arv[2]`, סימן שכל תווי `argv[1]` מצויים ב- `argv[2]`. אנו מודיעים על כך, ועוצרים.

11.12 תרגילים

11.12.1 תרגיל מספר אחד: איתור תת-מערך במערך

ממשו את הפונקציה הבאה:

```
int *subarray_in_range(const int array[],
                       const int array_size,
                       const int lower_bound,
                       const int upper_bound,
                        =int *subarray_sizep)
```

בהינתן המערך `array` שגודלו `array_size` עליכם למצוא את תת המערך הראשון (קרי הקרוב ביותר לתחילת המערך array) מגודל לא טריביאלי (עם יותר מאבר יחיד) שכל אבריו נמצאים בין `lower_bound` לבין `upper_bound`, כולל שני הגבולות.
אם מצאתם תת מערך כזה ערך החזרה של הפונקציה יהיה כתובתו של האבר הראשון בתת המערך, ו- `subarray_sizep*` יעודכן לגודלו של תת מערך זה. אחרת ערך החזרה ו- `subarray_sizep*` יעודכנו שניהם לאפס.

שימו לב: על הפונקציה שלכם להיכתב תוך שימוש במצביעים בלבד.
אל תגדירו משתנים מכל סוג אחר, ואל תשתמשו ב- subarray_sizep
לאף מטרה למעט זו שלשמה נועד.

11.12.2 תרגיל מספר שתיים: מסד נתוני משפחות

תרגיל 11.6.4 מתרגל גם מערכים ומצביעים באופן אינטנסיבי.

11.12.3 תרגיל מספר שלוש: טיפול בסיסי במערכים ומצביעים

בתכנית מוגדרים:

```
struct a_nums_list {
   int *nums ;
   unsigned int list_len ;
} ;

struct many_nums_lists {
   a_nums_list *lists ;
   unsigend int num_of_lists ;
} ;

many_nums_lists arr ;
```

המשתנה arr מסוגל (אם יעשה בו שימוש מתאים) להיהפך למערך שיכיל שורות של מספרים שלמים, כל-אחת באורך שונה. לצד כל שורה נחזיק את אורכה, וכן נחזיק את מספר השורות שהמשתנה arr מחזיק בכל נקודת זמן. (ראשית בררו לעצמכם מדוע וכיצד מתקיימים כל ההיגדים שנזכרו עד כה). בתכנית שנכתוב נשאף שלא להחזיק שורות יותר מהמינימום ההכרחי, ואורכה של כל שורה לא יהיה ארוך מהמינימום ההכרחי.

כתבו את הפונקציה add_val. לַפונקציה ארבעה פרמטרים: מערך (בשם arr) מטיפוס many_nums_lists, ערך שלם (val) אותו יש להכניס לַמערך, מספר שורה (line) לתוכה יש להכניס את הערך, ומספר תא רצוי (cell) באותה שורה. הפונקציה תכניס את הערך val למקום cell בשורה line במערך arr תוך שהיא דואגת להיבטים הבאים:

1. אם ב- arr אין די שורות אזי הוא יוגדל כך שתהייה בו שורה מספר line.
2. אם בשורה הרצויה אין די מקום אזי היא תוגדל כך שיהיה בה תא מספר place. בַתאים הריקים בשורה יושם הערך הקבוע EMPTY.

12 struct

מבנים (structures) הם כלי נוסף אשר יסייע לנו לשפר את סגנונן של התכניות שאנו כותבים.

12.1 מוטיבציה

נניח כי ברצוננו לכתוב תכנית אשר מחזיקה נתונים אודות המשתמש בה. בתכנית נרצה לשמור את שמו הפרטי, מינו, ומספר הנעליים של המשתמש. לשם כך נוכל להגדיר משתנים:

```
char name[N] ;
bool gender ;
unsigned int shoe_num ;
```

כדי לקרוא את הנתונים הדרושים נכתוב את הפונקציה:

```
void read_data(char name[], bool &gender,
               unsigned int &shoe_num);
```

הפונקציה מקבלת את הפרמטרים המשתנים הדרושים.

צורת הכתיבה שהצגנו היא לגיטימית, אך היא אינה די מדגישה כי שלושת המשתנים שהגדרנו מתארים שלושה היבטים של אותו אדם, ועל כן הם אינם בלתי קשורים זה לזה כפי ששלושה משתנים כלשהם עשויים להיות.

נרצה, על-כן, כלי באמצעותו נוכל 'לארוז' יחד לכדי 'חבילה אחת' מספר משתנים המתארים מספר היבטים שונים של אותו אובייקט. הכלי שמעמידה לרשותנו שפת C לשם השגת המטרה הוא ה- `struct`.

בַּתכנית בה אנו דנים, נוכל לכתוב מתחת להגדרת הקבועים:

```
struct User {
   char _name[N] ;
   bool _gender ;
   unsigned int _shoe_num ;
} ;
```

שימו לב לכמה הערות קטנות:

א. שם של טיפוס\סוג מבנה נהוג שיתחיל באות גדולה.

ב. שם של חברים במבנה נהוג להתחיל עם קו תחתון.

ג. אחרי הסוגר הימני מופיעה נקודה-פסיק (כמו ב- enum).

מה קבלנו? אמרנו בכך למחשב כי אנו מגדירים בזאת 'סוג חבילה' שנקרא `User` (המילה השמורה `struct` היא שֶׁמורה שאנו מגדירים כאן 'סוג חבילה', והמילה שמופיעה אחריה מתארת את שם 'החבילה', במקרה שלנו: `User`). בתוך הסוגריים המסולסלים אנו מתארים אילו מרכיבים יהיו בכל חבילה מסוג `User`.

אם אנו מעוניינים בכך אנו יכולים להגדיר בתכנית גם סוגים נוספים של חבילות. לדוגמה:

```
struct Course {
   char _name[N] ;
   unsigned int _weight ;
   char _for_year ;
};
```

'חבילה' מסוג Course מיועדת לתיאור נתונים אודות קורסים: שם הקורס, מספר נקודות הזכות שהוא מקנה, לאיזה שנה אקדמית הוא מיועד (a,b,c,...).

נדגיש כי הגדרת 'חבילה' אינה מקצה זיכרון בו מאוחסן משתנה כלשהו; היא רק מתארת סוג 'מארז' המכיל היבטים שונים המתארים נושא כלשהו אותו ברצוננו לייצג בתכניתנו. כלומר, לעת עתה לא ניתן לשמור בתכנית שלנו מידע אודות משתמשים (או אודות קורסים).

אחרי שהגדרנו struct-ים כפי צרכינו (בַּמקום שבין הגדרת הקבועים להצהרה על הפרוטוטיפים) אנו יכולים בגוף התכנית להגדיר משתנים:

```
struct User user1, user2 ;
struct Course c1 ;
```

בכך הגדרנו זוג משתנים user1, user2 שהם חבילות, כלומר כל-אחד מהם כולל שלושה מרכיבים. המרכיבים הם user1._name, user1._gender, user1._shoe_num. משמע כדי לפנות למרכיב כלשהו עלינו לציין את שם המשתנה, אחר-כך את התו נקודה, ואחר-כך את שם המרכיב. בשפת C אנו קוראים לכל מרכיב ב- struct בשם **חבר** (member). בשפות אחרות המרכיבים נקראים **שדות** (fields). חבר בְּמבנה הוא משתנה ככל משתנה אחר מהטיפוס המתאים. לכן אנו רשאים לכתוב פקודות כגון הבאות:

```
cin >> setw(N) >> user1._name ;
cout << strlen(user1._name) ;

cin >> user1._shoe_num ;
if (user1._shoe_num < 19 || user1._shoe_num > 55)
   cout << "Are u shure?\n" ;

cin >> i ;
if (i == 1)
   user1._gender = FEMALE ;
```

אנו רואים, אם כן, כי struct User מאפשר לנו לאגד את ההיבטים השונים של המשתמש בתכניתנו לכדי משתנה יחיד.

אעיר כי בשפת סי חובה להגדיר את המשתנה באופן: struct User user1 ; בעוד בשפת C++ ניתן לכתוב גם: User usr1, כלומר להשמיט את המילה struct. לטעמי, גם בשפת C++ לא כדאי להשמיטה וזאת כדי שלקורא יהיה ברור שהטיפוס User הוא טיפוס של מבנים.

12.2 העברת struct כפרמטר לפונקציה

נניח כי בתכנית כלשהי הגדרנו את struct User, ואת המשתנים user1, user2. עתה נרצה לכתוב פונקציה read_data אשר תקרא נתונים לתוך משתנה מטיפוס struct user. הפונקציה תיכתב באופן הבא:

```
void read_data(struct User &an_user) {
   int temp ;

   cin >> setw(N) >> an_user._name ;
   cin >> an_user._shoe_num ;
   cin >> temp ;
   an_user._gender = (temp == 1) ? FEMALE : MALE ;
}
```

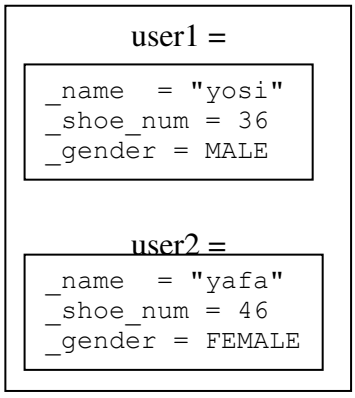
זימון הפונקציה יעשה באופן הבא: `read_data(user1);` . מכיוון שהפרמטר של הפונקציה הוא פרמטר הפניה אזי שינויים שהפונקציה תכניס לתוך הפרמטר `an_user` יישארו בתום ביצוע הפונקציה בארגומנט המתאים (בדוגמה שלנו: `user1`).

נבחן עתה פונקציה מעט שונה שתדגים לנו מה קורה עת מבנה מועבר כפרמטר ערך. ראשית נציג את הפונקציה, ואחר נדון בה:

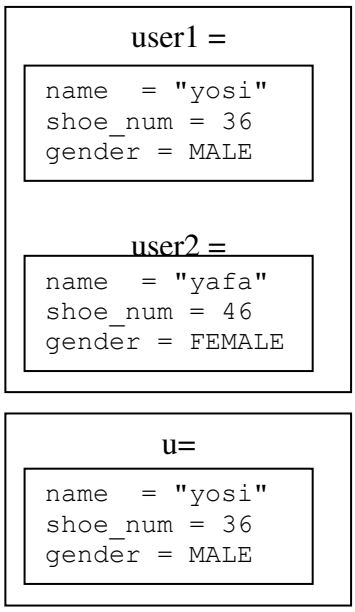
```
void f(struct User u) {
   strcpy(u._name, "dana") ;
}
```

נניח כי אנו מזַמְנים את הפונקציה `f` באופן הבא: `f(user1);` וזאת אחרי שלְ-`user1` הוזנו פרטי המשתמש יוסי. האם בתום ביצוע הפונקציה יכיל `user1.name` את הערך דנה? בעבר ראינו כי לפונקציה יש את הכוח לשנות את ערכם של תאי מערך המועבר לה כפרמטר. בפרק 10 גם הבנו מדוע הדבר קורה: שכן לַפונקציה מועבר מצביע לתא הראשון של המערך, והשינויים שהפונקציה מבצעת קורים על המערך המקורי. עתה נלמד כי אם המערך הוא חלק ממבנה המועבר לפונקציה כפרמטר *ערך* אזי אין בכוחה של הפונקציה לשנות את ערכם של חברי המבנה בארגומנט עִמו היא נקראה, בפרט לא את ערכו של החבר `name`; ועל-כן בתום ביצועה של הפונקציה `f` ימשיך `user1.name` להכיל את הערך "`yosi`". הסיבה לכך היא שעת מבנה מועבר כפרמטר ערך יוצר המחשב עותק חדש של המבנה במסגרת המחסנית של הפונקציה (הנבנית על-גבי המחסנית). השינויים שהפונקציה עורכת מתבצעים על העותק שבַּפרמטר (ולא על הארגומנט).

נדגים זאת בעזרת ציור המחסנית: נתחיל מכך שבתכנית הראשית הוגדרו שני משתנים `user1, user2` המכילים ערכים כמתואר (בציור הצגנו כל מבנה כמלבן נפרד בתוך המלבן המייצג את המחסנית):



עתה נניח כי אנו קוראים: `f(user1);` . נציג את מצב המחסנית:



עת `f` מבצעת את הפעולה: `strcpy(u._name, "dana");` מועתק הסטרינג `"dana"` על החבר `_name` בפרמטר `u` של `f`. ערכו של `user1._name` אינו משתנה.

ראינו כי עת מבנה מועבר כפרמטר ערך יוצר המחשב עותק נוסף של המבנה. עבור מבנה גדול, הכולל חברים רבים, ביניהם מערכים, תחייב עבודת ההעתקה השקעה של זמן ושל זיכרון. על-כן נהוג להיזהר מלהעביר מבנים גדולים כפרמטרי ערך. במקום זאת נוהגים להעבירם כפרמטרי הפניה, ואז נשלח חץ (עם ראש משולש שחור) מהפרמטר לארגומנט המתאים, ולא מתבצעת ההעתקה של המבנה. לחילופין, ניתן להעביר מצביע למבנה, ובכך שוב לחסוך את עבודת ההעתקה.

אם ברצוננו למנוע מהפונ' לשנות את המבנה אזי נעבירו כפרמטר הפניה קבוע: `const User &u` . באופן זה, מצד אחד תודות לכך שזהו פרמטר הפניה לא מבוצעת עבודת העתקה (הדורשת זמן וזיכרון), ומצד שני, תודות לכך שהפרמטר הוא קבוע, לא ניתן לשנותו. שימו לב שפרמטר שהוגדר כך אינו זהה לפרמטר ערך, שכן פרמטר ערך פונ' רשאית לשנות (רק שהשינוי לא ייוותר בארגומנט), כאן הפונ' אינה רשאית לשנות כלל את הפרמטר.

12.3 מערך של `struct`

בסעיף הקודם ראינו כיצד מגדירים מבנה יחיד. לעיתים אנו זקוקים למערך של מבנים. לדומה: נניח כי ברצוננו לשמור נתונים אודות כל תלמיד ותלמיד הלומד את הקורס 'מבוא לאגיפטולוגיה'. עבור כל תלמיד נרצה לשמור את שמו הפרטי, את מינו, מספר הנעליים שלו, ומערך שיכיל את ציוניו בתרגילים השונים שניתנו בקורס. נגדיר לכן:

```
struct Stud {
   char _name[N] ;
   bool _gender ;
   unsigned _int shoe_num ;
   int _grades[MAX_EX] ;
} ;
```

אחר, בתכנית הראשית, נגדיר: `struct Stud egypt[MAX_STUD];`. מה קיבלנו? מערך שכל תא בו הוא מבנה מטיפוס `struct Stud`.

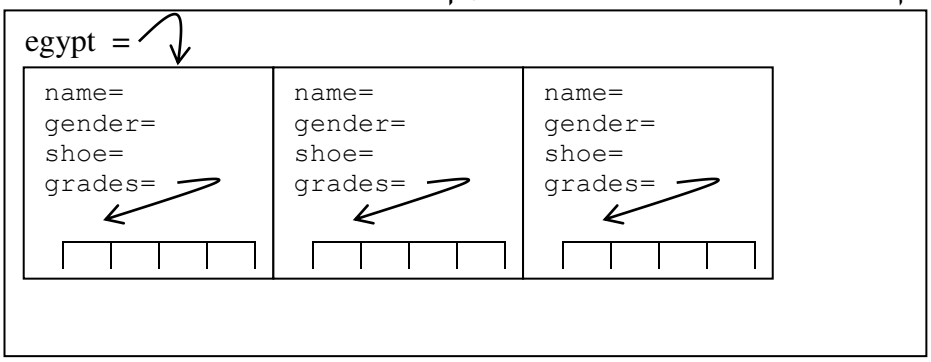
עתה נוכל לקרוא לפונקציה: `read_data(egypt);`. אנו מעבירים לפונקציה את המערך, על-מנת שהיא תקרא לתוכו נתונים. כיצד תראה הגדרת הפונקציה? בפרט: האם את הפרמטר שלה נצטרך להגדיר כפרמטר משתנה? לא, ולא! אנו יודעים כי עת מערך מועבר כפרמטר לפונקציה בכוחה של הפונקציה לשנות את ערכם של תאי המערך, שכן הפונקציה למעשה מקבלת מצביע לתא מספר אפס במערך, וכל השינויים שהפונקציה עורכת מתבצעים על המערך המקורי. נציג את הפונקציה `read_data`, נסבירה, ואחר גם נציג את מצב המחסנית בעקבות הקריאה:

```
void read_data( struct Stud egypt[] ) {
   int temp ;

   for (int i= 0; i < MAX_STUD; i++) {
      cin >> setw(N) >> egypt[i]._name ;
      cin >> egypt[i]._shoe_num ;
      for (int ex = 0; ex < MAX_EX; ex++)
         cin >> egypt[i]._grades[ex] ;
      cin >> temp ;
      egypt[i]._gender = (temp == 1) ? FEMALE : MALE ;
   }
}
```

הסבר הפונקציה: הפונקציה מקבלת מערך של `Stud`-ים, ולכן טיפוס הפרמטר שלה הוא `struct Stud egypt[]`. הפונקציה מתנהלת כלולאה אשר קוראת נתונים לתוך תאי המערך. נדון למשל בפקודה: `cin >> setw(N) >> egypt[i]._name;`. הפרמטר `egypt` הוא מערך, לכן כדי לפנות לתא רצוי בו עלינו לכתוב ביטוי עם סוגריים מרובעים, כדוגמת `egypt[i]`. התא `egypt[i]` הוא מבנה מטיפוס `Stud`; כדי לפנות לחבר רצוי במבנה אנו משתמשים בצורת הכתיבה: שם-המבנה.שם-החבר, ובמקרה שלנו: `egypt[i]._name`. זהו מערך של תווים, ולכן אנו יכולים לקרוא לתוכו מחרוזת.

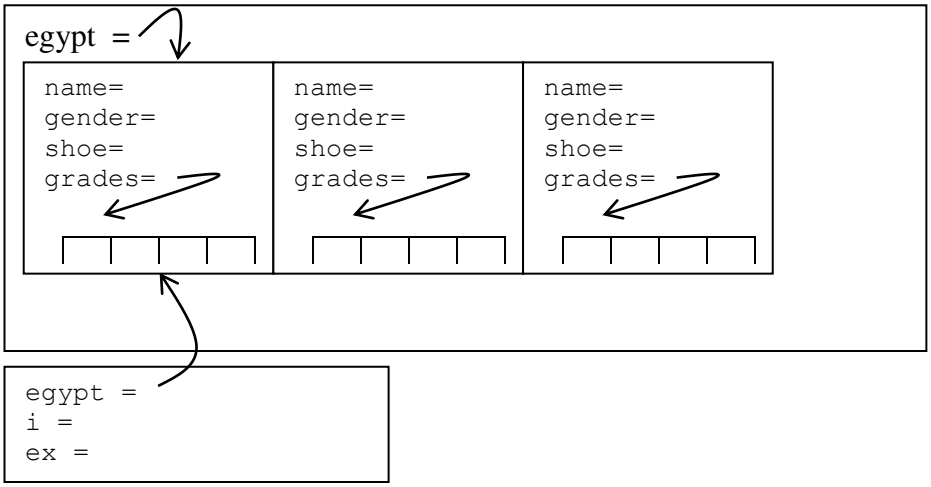
עתה נבחן את לולאת קריאת הציונים. בלולאה זאת המשתנה `ex` עובר על מספרי התרגילים השונים. עת ערכו הוא, למשל ,שלוש אנו קוראים ערך לתוך `egypt[i]._grades[3]`. נסביר: `egypt` הוא מערך, ולכן כדי לפנות לַתא של התלמיד מספר `i` עלינו לכתוב: `egypt[i]`. התא `egypt[i]` הוא תא בְּמערך של מבנים, ולכן משתנה מטיפוס `Stud`; לכן כדי לפנות לחבר `grades` במבנה עלינו לכתוב: `egypt[i]._grades`. החבר `egypt[i]._grades` הוא מערך, ולכן כדי לפנות לתא מספר שלוש בו עלינו לציין אינדקס רצוי: `egypt[i]._grades[3]`.

עתה נבחן את מצב המחסנית. נתחיל במערך שהוגדר בתכנית הראשית:



המערך egypt כולל שלושה תאים. כל תא הוא מטיפוס struct Stud, ולכן מכיל ארבעה חברים: החבר _name הוא למעשה מערך, אולם מכיוון שאנו חושבים עליו כעל מחרוזת לא ציירנו אותו כמערך; החברים _shoe_num ו- _gender הם חברים פרימיטיביים, וציירנו אותם כפי שאנו מציירים כל משתנה פרימיטיבי; החבר _grades הוא מערך סטטי בן ארבעה תאים, ולכן ציירנו את תאיו, וכן מצביע משם החבר לתא מספר אפס במערך. אנו זוכרים כי egypt הוא למעשה מצביע לתא מספר אפס במערך המבנים, ולכן לצד שם המשתנה ציירנו מצביע לתא הראשון במערך.

עת מתבצעת קריאה לפונקציה read_data, אשר מקבלת את המערך egypt, כלומר מצביע לתא מספר אפס של המערך, נראית המחסנית באופן הבא:



אנו רואים כי מהפרמטר egypt של הפונקציה נשלח מצביע לתא מספר אפס במערך של התכנית הראשית. (בדיוק כפי שנשלח מצביע מהמשתנה egypt של התכנית הראשית לאותו מערך). לכן עת הפונקציה פונה ל-egypt[1]._shoe_num היא מעדכנת ישירות את החבר _shoe_num בתא מספר אחד במערך המקורי (והיחיד הקיים).

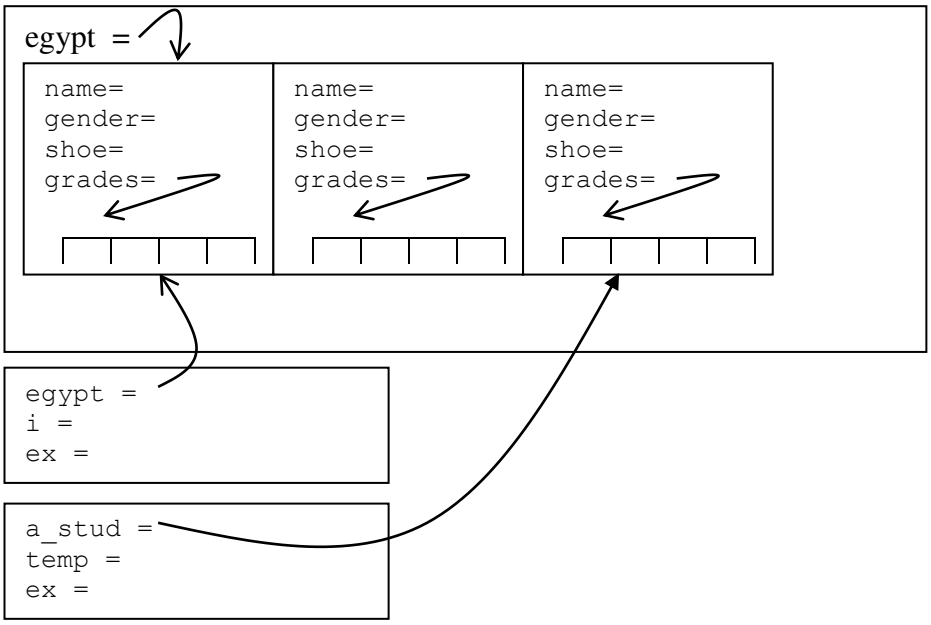
עתה נניח כי ברצוננו לכתוב את הפונקציה read_data בצורה שונה:

```
void read_data( struct tud egypt[] ) {
   for (int i= 0; i < MAX_STUD; i++)
      read_1_stud( egypt[i] ) ;
}
```

כלומר הפונקציה read_data קוראת בלולאה לפונקציה נוספת read_1_stud. הפונקציה הנוספת תקרא נתונים של תלמיד יחיד, ולכן הפונקציה read_data מעבירה לפונקציה read_1_stud כארגומנט תא יחיד במערך egypt, כלומר מבנה יחיד מטיפוס stud. עתה נשאל כיצד יראה הפרוטוטיפ של הפונקציה read_1_stud? האם הפרמטר המועבר לה הוא פרמטר ערך או פרמטר הפניה? התשובה היא שעל-מנת שלפונקציה read_1_stud יהיה את הכוח לשנות את ערכו של הארגומנט (מטיפוס stud) המועבר לה עליה לקבלו כפרמטר הפניה. שהרי הפונקציה read_1_stud מקבלת מבנה יחיד, וכבר מיצינו כי על-מנת שלפונקציה המקבלת מבנה יהיה את הכוח לשנות את הארגומנט המועבר לה, היא חייבת לקבל את המבנה כפרמטר הפניה. נציג ראשית את הפונקציה:

```
void read_1_stud(sruct Stud &a_stud) {
   int temp ;

   cin >> a__stud.name ;
   cin >> a_stud._shoe_num ;
   for (int ex = 0; ex < MAX_EX; ex++)
      cin >> a_stud._grades[ex] ;
   cin >> temp ;
   a_stud._gender = (temp == 1) ? FEMALE : MALE ;
}
```

בקוד של הפונקציה אין רבותא. אך נציג את מצב המחסנית עת אנו קוראים לפונקציה באופן הבא: `read_1_stud(egypt[2]);`



אנו רואים כי מהפרמטר `a_stud` של הפונקציה נשלח חץ (עם ראש משולש שחור) לתא מספר שתיים במערך של התכנית הראשית. ועל כן כל שינוי שהפונקציה מכניסה לפרמטר שלה מתחולל למעשה על התא המתאים במערך המקורי.

מה היה קורה לו הפונקציה `read_1_stud` הייתה מוגדרת באופן הבא:

```
void read_1_stud(struct Stud a_stud) ;
```

כלומר לו הפרמטר של הפונקציה היה פרמטר ערך? אזי הקריאה `read_1_stud(egypt[2])` הייתה *מעתיקה* את תוכנו של התא מספר שתיים במערך `egypt` על מבנה בשם `a_stud` שהיה מוקצה ברשומת ההפעלה של `read_1_stud` . הפרמטר `a_stud` כבר לא היה מורה על התא מספר שתיים במערך המקורי. כל שינויי ש- `read_1_stud` הייתה מכניסה לפרמטר שלה היה מתחולל על העותק השמור ברשומת ההפעלה שלה במחסנית. כמובן שבתום ביצוע הפונקציה ערכו של `egypt[2]` היה נותר כפי שהוא היה טרם הקריאה.

12.4 מצביע ל- `struct` כפרמטר ערך

את הפונקציה `read_1_stud` אנו יכולים לכתוב גם בדרך שונה. במקום שהפונקציה תקבל פרמטר הפניה מטיפוס `stud`, היא תקבל מצביע למבנה מטיפוס

stud.האפקט שיתקבל מבחינה תכנותית יהיה אותו אפקט. את הפונקציה נכתוב באופן הבא:

```
void read_1_stud( stud *stud_p) {
   int temp ;

   cin >> setw(N) >> (*stud_p)._name ;
   cin >> (*stud_p)._shoe_num ;
   for (int ex = 0; ex < MAX_EX; ex++)
      cin >> (*stud_p)._grades[ex] ;
   cin >> temp ;
   (*stud_p)._gender = (temp == 1) ? FEMALE : MALE ;
}
```

נסביר: עתה הפרמטר של הפונקציה מוגדר להיות sruct Stud *stud_p, כלומר מצביע למבנה מטיפוס Stud. על כן בגוף הפונקציה, כדי לפנות למבנה עליו stud_p מצביע עלינו להשתמש בכתיבה: stud_p*. 'היצור': *stud_p הוא מבנה, ועל כן בו עלינו לפנות לחבר המתאים, למשל: (*stup_p).gender .

הקריאה לפונקציה בגרסתה הנוכחית תהיה: read_1_stud(&(egypt[i]));, כלומר אנו מעבירים לפונקציה מצביע לתא מספר i במערך.

ציור המחסנית יישאר כפי שהוא היה בגרסה הקודמת פרט לשינוי יחיד: החץ (עם הראש השחור) שנשלח בגרסה הקודמת מהפרמטר a_stud לתא המתאים במערך, מומר עתה במצביע (עם ראש בצורת v) אשר נשלח מאותו מקום ולאותו מקום. הסיבה לכך היא שעתה הפונקציה מקבלת מצביע (ולא פרמטר הפניה). ובקריאה לפונקציה מעבירים בצורה מפורשת מצביע לתא המתאים (באמצעות הכתיבה &...), ולא את התא המתאים כפרמטר הפניה (כפי שהיה בגרסה הראשונית).

מכיוון שבשפת C מקובל מאוד להשתמש במצביע למבנה, מאפשרת לנו השפה לכתוב במקום ביטוי כגון: (*stup_p)._gender ביטוי כדוגמת: stup_p -> _gender . כלומר במקום ראשית לפנות למבנה עליו מצביע stud_p , (באמצעות *stud_p) ושנית לַחבר המתאים במבנה זה (באמצעות (*stud_p)._gender), אנו כותבים ישירות: פְּנֵה לחבר בַמבנה עליו מצביע stud_p. לכן את הפונקציה שכתבנו לאחרונה נוכל לכתוב גם בצורת הכתיבה הבאה, תוך שימוש בנוטציה שהצגנו זה עתה:

```
void read_1_stud( struct Stud *stud_p) {
   int temp ;

   cin >> setw(N) >> stud_p-> _name ;
   cin >> stud_p-> _shoe_num ;
   for (int ex = 0; ex < MAX_EX; ex++)
      cin >> stud_p-> _grades[ex] ;
   cin >> temp ;
   stud_p-> _gender = (temp == 1) ? FEMALE : MALE ;
}
```

12.5 מצביע ל- struct כפרמטר הפניה

בעבר ראינו כי במקום להגדיר מערך סטאטי: int a[N]; , אנו יכולים להגדיר פוטנציאל למערך בדמות מצביע, ובהמשך לממש את הפוטנציאל על-פי צרכי התכנית בעת ריצתה.

מצב דומה חל עם מבנים, ואין כל הבדל עקרוני בין מבנים למשתנים פרימיטיביים. נציג את הנושא עבור מבנים, ולו בשם החזרה.

נניח כי בתכנית הראשית הגדרנו: struct Stud *egypt_p; ועתה ברצוננו לקרוא לפונקציה alloc_arr_n_read_data אשר תקצה מערך בגודל המתאים, ותקרא לתוכו נתונים. ניתן לכתוב את הפונקציה בדרכים שונות (למשל הפונקציה עשויה לקבל פרמטר הפניה מטיפוס מצביע ל- stud), אנו נציג את הגרסה הבאה, (בה הפונקציה מחזירה באמצעות פקודת return מצביע למערך שהוקצה על-ידה):

```
sruct Stud *alloc_arr_n_read_data() {
   sruct Stud *arr_p ;
   int size ;

   cin >> size ;
   arr_p = new stud[size +1] ;
   if (arr_p == NULL)
      terminate("Memory allocation error\n", 1) ;

   strcpy(arr_p[size]._name, "") ;  // end of data sign

   read_data(arr_p, size) ;
   return( arr_p )
}
```

הקריאה לפונקציה תהיה: egypt_p = alloc_arr_n_read_data(); .

הסבר הפונקציה: לפונקציה יש משתנה לוקלי: stud *arr_p . הפונקציה קוראת מהמשתמש את גודלו של המערך הדרוש לו ואחר מקצה מערך הכולל תא אחד יותר מהדרוש. (במידה וההקצאה נכשלת אנו עוצרים את ביצוע התכנית.) לתוך המרכיב _name בתא האחרון שבמערך משימה הפונקציה את הסטרינג הריק, וכך תדענה פונקציות אחרות לזהות את סוף המערך. אחר הפונקציה קוראת לפונקציה read_data אשר תקרא נתונים לתאי המערך. הפונקציה read_data דומה לפונקציה שכתבנו בעבר, פרט לכך שאנו מעבירים לה גם את מספר התאים לתוכם read_data אמורה לקרוא נתונים (read_data כפי שנכתבה בעבר קראה ערכים לתוך MAX_STUD תאים במערך).

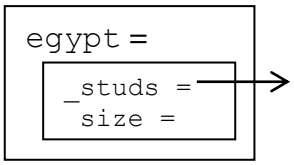
הדוגמה האחרונה תשרת אותנו כדי להכיר מצב נוסף בו אנו עשויים לרצות להשתמש במבנה: בתכנית שראינו לאחרונה הוגדר מצביע: struct Stud *egypt_p , ואחר למצביע הוקצה מערך. את התא האחרון במערך סימנו באמצעות שדה name שכלל את הסטרינג הריק (""). במקום להשתמש בשיטת הסטרינג הריק לסימון סוף המערך נוכל לחשוב על אפשרות אחרת, נקייה יותר. ראשית, בנוסף למבנה stud, נגדיר מבנה:

```
struct Stud_arr {
   struct Stud *_studs ;
   int _size ;
```

```
} ;
```

כפי שאנו רואים, מבנה מסוג `Stud_arr` מכיל שני מרכיבים: המרכיב `_studs` הוא פוטנציאל למערך של נתוני סטודנטים, המרכיב `_size` מציין את גודלו של המערך עליו מצביע `_studs`.

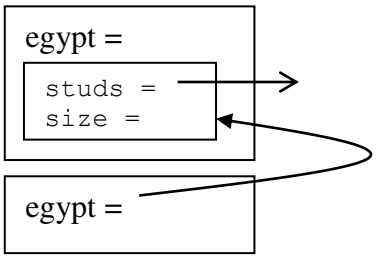
עתה נגדיר בתכנית הראשית משתנה: `stud_arr egypt;`. שימו לב כי איננו מגדירים מצביע כי אם מבנה (הכולל בתוכו מצביע). נתאר את מצב המחסנית:



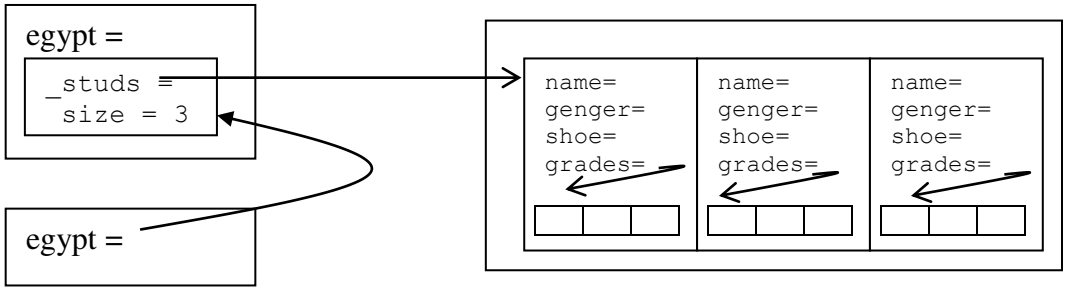
עתה נקרא לפונקציה: `alloc_arr_n_read_data2(egypt);` הפונקציה מקבלת את הארגומנט `egypt` כפרמטר משתנה. נבחן את הגדרת הפונקציה:

```
void alloc_arr_n_read_data2(struct Stud_arr &egypt) {

   cin >> egypt._size ;
   egypt._studs = new stud[ egypt._size ] ;
   if (egypt._size == NULL)
        terminate("Memory allocation error\n", 1) ;

   read_data2( egypt ) ;
}
```

הסבר: הפונקציה ראשית קוראת מהמשתמש את גודלו של המערך הדרוש לו לתוך החבר `_size` בפרמטר המשתנה `egypt`. אחר הפונקציה מקצה מערך ל-`egypt._studs` שהוא מרכיב מטיפוס `struct Stud *` (כפי שאנו רואים בהגדרת הטיפוס `Stud_arr`, שהוא טיפוסו של הפרמטר `egypt`). לבסוף הפונקציה מזמנת את: `read_data2( egypt );`. נבחן את מצב הזיכרון עם הכניסה לפונקציה `alloc_arr_n_read_data2` (וטרם שהיא הקצתה את המערך):



הסבר: הפונקציה מקבלת פרמטר משתנה בשם egypt. מכיוון שזה פרמטר משתנה אנו שולחים חץ מהפרמטר לארגומנט המתאים. לכן עת הפונקציה פונה ל-egypt._size אנו הולכים בעקבות החץ, ומעדכנים את המרכיב _size בארגומנט שהועבר לפונקציה בקריאה לה. באפן דומה גם הקצאת זיכרון תַפנה את המצביע studs בארגומנט של הפונקציה להצביע על המערך שהוקצה. נציג זאת בציור:



מטרתה של הדוגמה האחרונה הייתה להראות כי לעיתים מבנה עשוי להכיל: (א) מערך כלשהו, או מבנה אחר, אשר שומרים את הנתונים בהם מעוניין המשתמש, (ב) נתונים אודות המערך (או המבנה) כגון גודלו, מתי הוא עודכן לאחרונה, לאילו מתאי המערך יש להתייחס כריקים וכו'.

12.6 תרגילים

12.6.1 תרגיל מספר אחד: פולינומים

נחזור לתרגיל הפולינומים שהוצג בפרק שש. הפעם נייצג מונום באופן הבא:

```
struct monom {
   float coef, degree ;
} ;
```

כלומר עבור כל מונום נשמור את המקדם והחזקה.

פולינום ייוצג באופן הבא:

```
polynom {
   monom *the_poly ;
   unsiged int num_of_monoms ;
} ;
```

כלומר נחזיק מערך של מונומים, ושדה המציין כמה מונומים מרכיבים את הפולינום.

סדרה של פולינומים תיוצג, לפיכך, כמערך של מבנים מטיפוס polynom. את המערך נוכל להגדיר סטטית באופן:

```
polynom polynoms[MAX_POLY] ;
```

או שנקצה את המערך דינמית, על הערמה, ואז נוכל גם לשנות את גודלו במידת הצורך; אם נבחר באפשרות זאת נגדיר את המצביע:

```
polynom *polynoms ;
```

עתה כתבו שנית את תכנית הפולינומים כפי שתוארה בפרק שש, תוך שאתם משתמשים במבנה הנתונים שתיארנו בפרק זה.

12.6.2 תרגיל מספר שתיים: סידור כרטיסים בתבנית רצויה

בפרק שמונה הצגנו תרגיל אשר דן במשחק בו על המשחק לסדר תשעה קלפים בתבנית רצויה. ראשית חיזרו לתיאור המשחק כפי שמופיע שם. עתה נרצה לכתוב

שוב את התכנית הדרושה תוך שאנו עושים שימוש במבני נתונים חכמים יותר, אשר יגבירו את קריאותה של התכנית.
מבנה הנתונים הראשון שנגדיר ישמש לתיאור כל אחד מארבעת צדדיו של כל כרטיס:

```
enum color { RED, YELLOW, GREEN, BLUE } ;
stuct side {
   bool head_tail ;
   color a_color ;
} ;
```

כרטיס שלם מורכב מארבעה צדדים:

```
enum urdl { UP=0, RIGHT=1, DOWN=2, LEFT=3 } ;
struct card {
   side sides[4] ;
   urdl direction ;
} ;
```

הטיפוס `urdl` ישמש אותנו הן כאינדקס למערך `sides` כדי להגביר את הקריאות (`sides[UP]` הוא ביטוי קריא יותר מ- `sides[0]`), והן כדי לציין האם הכרטיס מוצב כאשר הצד המתואר כ- `UP` אכן למעלה, או שמא הוא מסובב ימינה, הפוך (כשצדו התחתון הוא שפונה כלפי מעלה), או שמאלה.

מאגר הכרטיסים אותם יש לסדר במהלך המשחק יתואר באופן הבא:

```
card cards[9] ;
```

ולתוכו יהיה על התכנית לקרוא את הנתונים מהמשתמש.

לבסוף, על התכנית לסדר את תשעת הכרטיסים בתבנית הרצויה (תוך שימוש באלגוריתם שתואר בפרק שמונה), ואת הסידור הנ"ל יתאר המערך:

```
unsigned int arrange[3][3] ;
```

כל תא במערך יכיל אינדקס של כרטיס במערך `cards`.

12.6.3 תרגיל מספר שלוש: בעיית מסעי הפרש

נחזור לבעיית מסעי הפרש, כפי שהוצגה בפרק שמונה (תרגיל 8.7.10). להזכירכם, המשימה אותה היה עלינו לממש היא מלוי לוח שחמט בגודל NxN בצעדי פרש, תוך שהפרש מבקר פעם אחת בדיוק בכל משבצת בלוח.

הפעם נרצה להשלים את המשימה תוך שימוש במבנים. נגדיר על-כן את המבנה:

```
stuct square {
   bool visited ;
   unsigned int next_x, next_y ;
} ;
```

ונגדיר את המשתנה: `square board[N][N];` משתנה זה ייצג את הלוח.

התכנית הראשית תאתחל את המשתנה כך שערכי השדה `visited` בכל תאי המערך יכילו את הערך `false`.

אחר, התכנית הראשית תקרא מהמשתמש את מקומה של המשבצת ממנה על הפרש להתחיל את מסעו.

הפונקציה הרקורסיבית שתהווה את לב ליבה של התכנית תמלא את הלוח בצעדי הפרש תוך שהיא מסמנת בכל תא בו הפרש מבקר, בשדה `visited`, כי הפרש ביקר בתא זה. השדות `next_x, next_y` יתארו את המשבצת אליה התקדם הפרש מהמשבצת הנוכחית.

בתום תהליך החיפוש הציגו פלט שיכלול שני מרכיבים:

א. רישום של סדרת המשבצות בהן ביקר הפרש, בסדר בו בוקרו המשבצות השונות.

ב. תיאור של הלוח (כטבלה דו-ממדית), כאשר עבור כל משבצת מופיע מספר הצעד בו בוקרה משבצת זאת.

12.6.4 תרגיל מספר ארבע: נתוני משפחות

בתרגיל זה נחזיק מסד נתונים הכולל נתוני משפחות.
כל משפחה בתרגיל תהיה משפחה גרעינית הכוללת לכל היותר אם, אב ומספר כלשהו של ילדים. נתוני משפחה יחידה ישמרו במבנה מהטיפוס:

```
struct family
{
  char *family_name ;
  char *mother_name, *father_name ;   // parents' names
  child *children_list ;              // list of the children in this family
  unsigned child_list_size ;          // size of child_list array

};
```

נתוני כל ילד ישמרו ב:

```
struct child
{
  char *child_name ;
  bool sex ;
} ;
```

מבנה הנתונים המרכזי של התכנית יהיה: family *families

התכנית תאפשר למשתמש לבצע את הפעולות הבאות:

1. הזנת נתוני משפחה נוספת. לשם כך יזין המשתמש את שם המשפחה (וזהו מרכיב חובה), ואת שמות בני המשפחה. במידה והמשפחה אינה כוללת אם יזין המשתמש במקום שם האם את התו מקף (-) (המרכיב mother_name במבנה המתאים יקבע אז, כמובן, להיות NULL), באופן דומה יצוין שאין אב במשפחה. לפני הזנת נתוני הילדים יוזן מספר הילדים במשפחה. התכנית תקצה מערך הכולל שני תאים נוספים, מעבר למספר הילדים הנוכחי במשפחה, וזאת לשימוש עתידי. שמות הילדים בתאים הריקים ייקבעו להיות NULL וכך נסמן כי התאים ריקים. תיתכן משפחה ללא ילדים. לא תיתכן משפחה בה אין לא אם ולא אב (יש לתת הודעת שגיאה במידה וזה הקלט מוזן, ולחזור על תהליך קריאת שמות ההורים). אתם רשאים להגדיר משתנה סטטי יחיד מסוג מערך של תווים לתוכו תקראו כל סטרינג מהקלט. את רשימת המשפחות יש להחזיק ממוינת על-פי שם משפחה. את רשימת ילדי המשפחה החזיקו בסדר בה הילדים מוזנים. הניחו כי שם המשפחה הוא **מפתח** קבוצת המשפחות, כלומר לא תתכנה שתי משפחות להן אותו שם משפחה (משמע ניסיון להוסיף משפחה בשם כהן, עת במאגר כבר קיימת משפחת כהן מהווה שגיאה). אין צורך לבדוק כי לא

מוזנים באותה משפחה שני ילדים בעלי אותו שם. במידה ומערך המשפחות התמלא יש להגדילו.
דוגמה לאופן קריאת הנתונים:

Enter family name: *Cohen*
Enter father name: *Yosi*
Enter mother name: -
Enter children's names: *Dani Dana* -

2. הוספת ילד למשפחה. יוזן שם המשפחה ושמו של הילד. הילד יוסף בסוף רשימת הילדים. במידה ומשפחה בשם הנ"ל אינה קיימת יש להודיע על-כך למשתמש (ולחזור לתפריט הראשי). במידה ומערך הילדים מלא יש להגדילו בשיעור שלושה תאים נוספים.
3. פטירת בן משפחה. יש להזין: (א) את שם המשפחה, (ב) האם מדובר באם (ולשם כך יוזן הסטרינג mother), באב (יוזן father), או באחד הילדים (יוזן child) (ג) במידה ומדובר בילד יוזן גם שם הילד שנפטר. במידה ואין במאגר משפחה כמצוין, או שבמשפחה אין בן-משפחה כמתואר (למשל אין אב והתבקשתם לעדכן פטירת אב, או אין ילד בשם שהוזן) יש לדווח על שגיאה (ולחזור לתפריט הראשי). כמו כן לא ניתן לדווח על פטירת הורה במשפחה חד-הורית (ראשית יש להשיא את המשפחה למשפחה חד-הורית אחרת, כפי שמתואר בהמשך, ורק אז יוכל ההורה המתאים לעבור לעולם שכולו טוב).
4. נשואי שתי משפחות קיימות. יש לציין את שמות המשפחות הנישאות, ואת שם המשפחה החדש שתישא המשפחה המאוחדת. פעולה זאת לגיטימית רק אם אחת משתי המשפחות אינה כוללת אם בעוד השניה אינה כוללת אב. בעקבות ביצוע הפעולה תאוחדנה רשימות הילדים והתא שהחזיק את נתוני אחת משתי המשפחות ייחשב כריק, ולכן תוזנה הרשומות בהמשך המערך, כדי לצמצם הפער.
5. הצגת נתוני המשפחות בסדר עולה של שמות משפחה. (זהו, כזכור, הסדר בו מוחזקת רשימת המשפחות). עבור כל משפחה ומשפחה יש להציג את שם המשפחה, ואת שמות בני המשפחה.
6. הצגת נתוני משפחה בודדת רצויה. יש להזין את שם המשפחה ויוצגו פרטי בני המשפחה. (במידה והמאגר אינו כולל משפחה כמבוקש תוצג הודעת שגיאה).
7. הצגת נתוני המשפחות בסדר יורד של מספר בני המשפחה. יש להציג את שם המשפחה ואת מספר בני המשפחה. לשם סעיף זה החזיקו מערך נוסף. כל תא במערך יהיה מטיפוס:

```
struct family_size
{   unsigned the_family ;  // index of a family in the families array
  int size ;                  // number of children in this family ;
};
```

המצביע למערך זה יהיה family_size *families_sizes. המערך ימוין בסדר יורד של המרכיב size. על-ידי סריקה של מערך זה ופניה ממנו בכל פעם למשפחה המתאימה (תוך שימוש באינדקס the_family) תוכלו להציג את המשפחות בסדר יורד של גודל המשפחות. שימו לב כי יש גם לתחזק רשימה זאת. בין כל המשפחות להן אותו גודל אין חשיבות לסדר ההצגה
8. הצגת נתוני שכיחות שמות ילדים. יש להציג עבור כל שם ילד המופיע במאגר, כמה פעמים הוא מופיע במאגר. אין צורך להציג את הרשימה ממוינת. (רמז\הצעה: סרקו את מאגר המשפחות משפחה אחר משפחה, עבור כל משפחה עיברו ילד אחר ילד, במידה ושמו של הילד הנוכחי עדיין לא מופיע ברשימת הילדים ששמם הודפס סיפרו כמה פעמים מופיע שם הילד במאגר הנתונים, ואחר הוסיפו את שמו של הילד לרשימת שמות הילדים שהוצגו).

9. סיום. בשלב זב עליכם לשחרר את **כל** הזיכרון שהוקצה על-ידכם דינמית.

<u>הערות כלליות:</u>

א. התכנית תנהל כלולאה בה בכל שלב המשתמש בוחר בפעולה הרצויה לו, הפעולה מתבצעת (או שמוצגת הודעת שגיאה), והתכנית חוזרת לתפריט הראשי.

ב. הקפידו שלא לשכפל קוד, למשל כתבו פונקציה יחידה אשר מציגה נתוני משפחה, והשתמשו בה במקומות השונים בהם הדבר נדרש. וכו'.
Needless to say שיש להקפיד על כללי התכנות המקובלים, בפרט ובמיוחד מודולריות, פרמטרים מסוגים מתאימים (פרמטרי ערך vs פרמטרים משתנים), ערכי החזרה של פונקציות ושאר ירקות. תכנית רצה אינה בהכרח גם תכנית טובה! (קל וחומר לגבי תכנית מקרטעת)...

12.6.5 <u>תרגיל מספר חמש: סימולציה של רשימה משורשרת</u>

נתונות ההגדרות הבאות:

```
const int N = … ;

struct node {
   char ch ;
   unsigned int next ;
} ;

struct linked_list {
   node list[N] ;
   unsigned int head ;
} ;
```

משתנה מטיפוס `linked_list` מכיל שני מרכיבים:

א. מערך `list` המכיל בכל תא תו, וכן הפְּניה לתא נוסף במערך. דוגמה אפשרית למרכיב `list` ברשומה כלשהי:

s	x	y	x	o	a	n	a	i	d
8	3	4	1	0	1-	5	6	17	7
0#	1#	2#	3#	4#	5#	6#	7#	8#	9#

ב. מספר של תא כלשהו במערך, השמור בשדה `head`.

נוכל להתייחס למרכיב המערך כמכיל מִספר מחרוזות, את כתובת ההתחלה של *אחת* מהן מחזיק המרכיב `head`. לדוגמה: אם ערכו של `head` הוא 2, אזי המחרוזת בה מתמקדים מורכבת מהתווים הבאים: התו בתא מספר שתיים במערך (`y`), מתו זה אנו מתקדמים על פי השדה `next` לתא מספר ארבע במערך, בו שמור התו `o`, משם לתא מספר אפס המכיל את `s`, ומשם לתא מספר שמונה המכיל את `i`. מכיוון שבתא מספר שמונה מצוי ערך שאינו בתחום `0..N-1` אנו מסיקים כי בכך תם הסטרינג, ולפיכך הסטרינג בו עסקינן הוא `yosi`. באופן דומה בתא מספר תשע במערך מתחיל הסטרינג `dana`.

לעומת זאת בתא מספר שלוש במערך אנו מוצאים הַפְּניה לתא מספר חמש, בתא מספר חמש קיימת הפניה לתא מספר שלוש, וחוזר חלילה. אנו אומרים כי סטרינג זה אינו נגמר, ולכן הוא בגדר שגיאה.

כתבו פונקציה המקבלת מבנה מסוג `linked_list` ומציגה את הסטרינג שעל התו הראשון שלו מורה המרכיב `head` אלא אם הסטרינג הינו שגוי, ואז תוצג הודעת שגיאה, ולא יוצג כל תו מהסטרינג.