# תוכן עניינים

[תוכן עניינים 1](#_Toc1734907)

[תהליך קומפילציה 4](#_Toc1734908)

[קדם מעבד (Preprocessor) 4](#_Toc1734909)

[מהדר (Compiler) 4](#_Toc1734910)

[מחבר (linkage) 4](#_Toc1734911)

[זיכרון 4](#_Toc1734912)

[מחסנית (stack) 4](#_Toc1734913)

[Global area (static heap) 4](#_Toc1734914)

[Dynamic heap 5](#_Toc1734915)

[Code segment 5](#_Toc1734916)

[זיכרון משתנים 5](#_Toc1734917)

[כתיבת קוד 6](#_Toc1734918)

[הערות 6](#_Toc1734919)

[בלוקים 6](#_Toc1734920)

[שמות משתנים ופונקציות 6](#_Toc1734921)

[דגשים 6](#_Toc1734922)

[ספריות וקבצים 6](#_Toc1734923)

[הוספת ספריות וקבצים 6](#_Toc1734924)

[ספריות שימושיות 7](#_Toc1734925)

[מודולים 7](#_Toc1734926)

[יבוא משתנים - extern 7](#_Toc1734927)

[משתנים וטיפוסים 7](#_Toc1734928)

[הצהרה ואתחול 7](#_Toc1734929)

[סוגי משתנים 7](#_Toc1734930)

[משתנים קבועים - const 8](#_Toc1734931)

[סוגי טיפוסים 8](#_Toc1734932)

[בין signed ל-unsigned 8](#_Toc1734933)

[הסבה - Casting 8](#_Toc1734934)

[אופרטורים 9](#_Toc1734935)

[enum 10](#_Toc1734936)

[תנאים ולולאות 10](#_Toc1734937)

[משפטי תנאי: 10](#_Toc1734938)

[switch 10](#_Toc1734939)

[לולאת for: 10](#_Toc1734940)

[לולאת while: 10](#_Toc1734941)

[לולאת do while: 11](#_Toc1734942)

[פונקציות 11](#_Toc1734943)

[כיצד מצהירים: 11](#_Toc1734944)

[פרמטרים 11](#_Toc1734945)

[פונקציה ריקה void 11](#_Toc1734946)

[שימוש בפונקציות בתוך פונקציות: 12](#_Toc1734947)

[בדיקת שגיאות 12](#_Toc1734948)

[assert 12](#_Toc1734949)

[DEBUG ו-NDUBG 12](#_Toc1734950)

[#Ifdef ו-#ifndef 12](#_Toc1734951)

[קלט ופלט 13](#_Toc1734952)

[פלט – דרוש בדיקה 13](#_Toc1734953)

[קלט 13](#_Toc1734954)

[חוצץ הקלט – Input Buffer 13](#_Toc1734955)

[קלטים מיוחדים 14](#_Toc1734956)

[מצביעים 14](#_Toc1734957)

[הגדרה והצהרה 14](#_Toc1734958)

[השמה של פוינטרים 14](#_Toc1734959)

[אריתמטיקה של מצביעים 14](#_Toc1734960)

[שליחת מצביע לפונקציה 15](#_Toc1734961)

[זיכרון מצביעים 15](#_Toc1734962)

[טיפוס void\* 15](#_Toc1734963)

[NULL 15](#_Toc1734964)

[מצביע קבוע const 16](#_Toc1734965)

[מצביע למצביע 16](#_Toc1734966)

[מצביע לפונקציה 16](#_Toc1734967)

[מערכים 17](#_Toc1734968)

[הגדרה והצהרה 17](#_Toc1734969)

[כיצד עובדים מערכים 18](#_Toc1734970)

[מערכים ומצביעים 18](#_Toc1734971)

[שליחת מערך לפונקציה 18](#_Toc1734972)

[פעולות בין מערכים 19](#_Toc1734973)

[מערך דו-מימדי 19](#_Toc1734974)

[העברה לפונקציה מערך דו- מימדי 20](#_Toc1734975)

[תווים ומחרוזות 20](#_Toc1734976)

[תו char 20](#_Toc1734977)

[מחרוזת 20](#_Toc1734978)

[העברת מחרוזת לפונקציה 21](#_Toc1734979)

[מערך של מחרוזות 21](#_Toc1734980)

[פונקציות על מחרוזות 21](#_Toc1734981)

[מבנה - struct 22](#_Toc1734982)

[הגדרה 22](#_Toc1734983)

[typedef 22](#_Toc1734984)

[גישה לשדות במבנה 22](#_Toc1734985)

[השמה בין מבנים 22](#_Toc1734986)

[זיכרון 23](#_Toc1734987)

[איגוד – union 23](#_Toc1734988)

[הקצאת זיכרון 23](#_Toc1734989)

[הקצאת זיכרון - malloc ו-calloc 23](#_Toc1734990)

[שחרור הזיכרון - free 24](#_Toc1734991)

[הקצאה מחדש – realloc 24](#_Toc1734992)

[Vla 24](#_Toc1734993)

[פונקציות נוספות 25](#_Toc1734994)

[הגדר – #define 25](#_Toc1734995)

[פונקציית go to 25](#_Toc1734996)

[פונקציית exit 25](#_Toc1734997)

[פונקציית typedef 25](#_Toc1734998)

[פונקציית time 25](#_Toc1734999)

[פונקציות srand ו-rand 25](#_Toc1735000)

[לינוקס 26](#_Toc1735001)

[פקודות על קבצים ותיקיות 26](#_Toc1735002)

[הרשאות 26](#_Toc1735003)

[הפעלת קובץ 26](#_Toc1735004)

[Piping 26](#_Toc1735005)

[שלבי קימפול תוכנית 27](#_Toc1735006)

[קבלת ערכים בשורת הפקודה 27](#_Toc1735007)

[הגדרת קבוע 27](#_Toc1735008)

# תהליך קומפילציה

תהליך הקומפילציה של קוד ב-C מורכב משלושה שלבים:

## קדם מעבד (Preprocessor)

מבצע מספר פעולות:

1. מחבר את הקוד שכתבנו עם כל הספריות שהוספנו אליו (header files) אל קובץ אחד.
2. מחליף את כל הערכים שהגדרנו אותם קבועים, באמצעות הפקודה "define".
3. מסיר הערות ורווחים שכתבנו.

פעולות אלו מתבצעות בנפרד, על קובץ התוכנית המכיל פונקציית main, ועל קבצים שהוספנו המכילים פונקציות. כל קבצים אלו נשלחים לקומפיילר.

כדי לתת פקודה לקדם מעבד נשתמש בסימון #. כל הפקודות לקדם מעבד אין להם סיומת, ולכן הפקודה נגמרת בירידת שורה. במקרה שרוצים לרדת שורה תוך כדי פקודה לקדם מעבד נשתמש בסימון \.

#define x (5 + **\**  
 5)

בסביבת עבודה של לינוקס עם gcc ניתן לראות את הקוד לאחר פעולת הקדם מעבד וקודם הקומפילציה, באמצעות הפקודה הבאה בשורת הפעלה: gcc –E filename.c.

## מהדר (Compiler)

מתרגם את הטקסט בשפת C שנוצר מהקדם מעבד אל שפת מכונה. אמנם לא מתרגם את כל הטקסט, אלא חלקים מהטקסט שנקראים "הפניות חיצוניות", כמו פקודת הדפסה - printf, הוא אינו מתרגם לשפת מכונה אלא משאיר אותם כמו שהם. הקובץ שנוצר לאחר פעולת הקומפיילר הוא object file עם סיומת filename.o.

## מחבר (linkage)

תפקידו הוא לחבר את כל קבצי object שנוצרו מהקומפיילר שהם קובץ התוכנית שלנו, קבצים שהוספנו המכילים פונקציות, וגם קבצים נוספים כמו לדוגמא libc.a. כך שנוצר קובץ אחד שכל הקוד בו מתורגם אל שפת מכונה, כולל ההפניות החיצוניות.

# זיכרון

במהלך הרצת התוכנית משתנים יכולים להישמר בשלושה אזורים בזיכרון: stack, global area, dynamic heap.

## מחסנית (stack)

תפקידה לשמור זיכרון במהלך קריאה לפונקציה. חשוב לזכור שגם main היא פונקציה.

1. לשמור את ערכי הפרמטרים שהועברו לפונקציה. בפונקציה רקורסיבית כל פעם שמפעילים את הפונקציה ישמרו פרמטרים אחרים במחסנית.
2. לשמור את כל המשתנים המקומיים של הפונקציה, כלומר משתנים שהוגדרו בתוך בלוק מסוים.
3. לשמור את כתובת החזרה ב-code segment לאחר סיום הפונקציה.

בתחילת התוכנית המחסנית ריקה. בכל פעם שנכנסים לתוך בלוק או פונקציה, התוכנית מקצה את המשתנים המקומיים של הבלוק או הפונקציה בראש המחסנית וכך היא גדלה. ברגע שיוצאים מבלוק, התוכנית מפנה ומוחקת את כל המשתנים שהוקצו עבור בלוק זה, והמחסנית קטנה בחזרה. גודל הזיכרון במחסנית הוא סופי וחייב להיקבע עוד בזמן קומפילציה. כאשר לא ניתן להגדיר גודל סופי למחסנית, כמו לדוגמא בפונקציה אינסופית, נקבל שגיאת זמן ריצה - "out of stack space".

המחסנית עובדת בדרך כלל בצורה של "מחסנית יורדת", כלומר כל המשתנים המקומיים נשמרים בתאים זה אחר זה לפי סדר הופעתם בתוכנית. אם אחד ממשתנים אלו הוא פוינטר, ונעשה עליו פעולה אריתמטית שמקדמת אותו באחד, בעצם נגיע לכתובת זיכרון של המשתנה שהוגדר קודם בזיכרון.

## Global area (static heap)

אזור זיכרון בו נשמרים כל המשתנים הגלובליים והסטטיים, כלומר משתנים שהוגדרו מחוץ לבלוקים או משתנים מסוג static. משתנים אלו נשמרים ב-global area לאורך כל התכנית. גודל הזיכרון המוקצה ל- global area חייב להיות סופי ומוגדר עוד בזמן קומפילציה.

## Dynamic heap

חלק בזיכרון שגודלו אינו ידוע מראש, אלא ניתן להקצות ולשחרר אותו במהלך ריצת התוכנית באמצעות פונקציות המיועדות לכך. בעזרת פונקציות malloc או calloc נוכל לבקש ממערכת ההפעלה להקצות לנו זיכרון רציף בגודל המבוקש. מערכת ההפעלה תחפש האם קיים קטע זיכרון מתאים בערימה, ואם כן, תסמן אותו כתחת שימוש ותחזיר לנו את כתובתו. אם תגיע עוד בקשה להקצאת זיכרון, מערכת ההפעלה לא תחזיר כתובת של קטע החופף לקטע הזה. במידה והפונקציות נכשלו בהקצאה, הכתובת שיחזירו תהיה NULL. לאחר שנסיים להשתמש בקטע הזיכרון, נוכל להשתמש בפונקציה free כדי לשחרר אותו. הדבר יודיע למערכת ההפעלה שקטע זה אינו בשימוש יותר, ואפשר להקצותו לבקשות אחרות. גודל הזיכרון המקסימלי שניתן להקצות ב- Dynamic heapהוא גודל הזיכרון הפנוי במחשב.

הסיבה שנרצה להשתמש בחלק זה של הזיכרון, הוא במקרים שאנו משתמשים בקוד עם מערך שהמשתמש מגדיר מהו גודלו או במקרים שגודל המערך משתנה במהלך הקוד. מכיוון שבשני חלקי הזיכרון הקודמים גודל הזיכרון המוקצה הוא קבוע בזמן קומפילציה, וכאן אנו צריכים להגדירו בזמן ריצה, אנו חייבים שיהיה אזור זיכרון דינאמי שניתן להקצות ממנו זיכרון בזמן ריצה. עוד על נושא זה בפרק "הקצאת זיכרון".

## Code segment

code segment הוא אזור בזיכרון המכיל העתק של אוסף הפקודות של התוכנית. בזמן הרצת התהליך, כתובת הפקודה הבאה לביצוע מצויה באוגר הנקרא program counter. בדרך כלל מקטע זה משמש לקריאה בלבד וניסיון לבצע כתיבה לכתובת זיכרון השייכת אליו יסתיים בשגיאת זמן ריצה.

## זיכרון משתנים

הזיכרון במחשב מחולק לתאים, כל תא מחזיק בזיכרון של 1 byte שהוא 8 bits. ולכל תא יש מספר שנקרא "הכתובת" של התא. כל משתנה בתוכנית תופס מספר מסוים של תאים בזיכרון לפי הטיפוס שלו. הכתובת של כל משתנה הוא הכתובת של התא הראשון שמכיל אותו. בשונה משפות אחרות, גודל הזיכרון שתופס כל טיפוס משתנה בהתאם למכונה שמפעילה את הקוד, אמנם חייבים להישמר שני כללים:

1. הגודל של Char הוא 1 byte, כלומר תא אחד.
2. Size of short ≤ size of int ≤ size of long.

כדי לדעת מה הגודל של טיפוס מסוים, כלומר כמה תאים (bytes) הוא מכיל, נשתמש באופרטור sizeof(type\_name), הערך שמוחזר מפעולה זו הוא משתנה מטיפוס size\_t שמייצג את גודל הטיפוס שהכנסנו לפונקציה.

# כתיבת קוד

## הערות

יש שתי שיטות לרשום הערות:

// This line is a comment,

/\* and this line also \*/

מטרת ההערות להעביר מידע לקורא שאינו ברור מאליו מקריאת הקוד, אין צורך לרשום בהערות מידע המובן מאליו.

## בלוקים

בלוק הוא קטע קוד שתחום על ידי זוג סוגריים מסולסלים { }. בלוקים יכולים להיות זרים או מקוננים (אחד מוגדר בתוך השני, כמו במקרה של לולאה בתוך לולאה).

דוגמאות לבלוקים:

- כל פונקציה נכתבת בתוך בלוק.

- ניתן להשתמש בבלוקים בלולאות ובפקודות התניה.

- ניתן גם לשים בלוק סתם כך באמצע קוד רציף!

## שמות משתנים ופונקציות

שמות משתנים צריכים להיות קצרים ובעלי משמעות, כך שפעולות השמה על המשתנים יהיו חלק "מהסיפור" של הקוד ולא פעולה הדורשת פיענוח. יש שתי שיטות נפוצות בנתינת שמות עם כמה מילים למשתנים, אחת עם אות גדולה בתחילת מילה numPending, והשנייה עם מקף תחתון num\_pending. לא משנה באיזו שיטה נבחר העיקר שנהיה עקביים בשיטה אחת למשך הקוד.

לפונקציות ניתן שמות שמייצגות את הפעולה שהן עושות, לדוגמא: isDigit, getData וכו'. בדרך זו הקוד יהיה ברור יותר וקל למצוא טעויות.

## דגשים

* קוד צריך לספר סיפור, לכן קוד שקשה להסביר אותו הוא כנראה לא קוד טוב. לפני שכותבים קוד מומלץ לחשוב ולספר לעצמך אותו בראש בצורה הכי קלה ומובנת, ולאחר מכן לכתוב אותו בצורה זו.
* חשוב להוסיף סוגריים במקרה של אופרטורים מרובים במשוואה אחת כדי להימנע מטעויות של סדר פעולות חשבון.
* לא מומלץ להשתמש בתנאי if מקוננים, אלא פשוט להשתמש ב-else if.

# ספריות וקבצים

ספרייה היא אוסף של פונקציות המשמשות את המתכנת על מנת שיוכל לבצע פעולות גם בלי לממש אותם.

## **הוספת ספריות וקבצים**

להוספת ספריות נשתמש בפקודה:

#include <libraryname.h>

להוספת קבצים נשתמש בפקודה:

#include  "filename.h"

הערה: במקרים בהם מצרפים שני קבצים לקובץ C שבשניהם מוגדרים פונקציה או מבנה עם שם זהה, נקבל שגיאה כי הם מוגדרים בקובץ אחד פעמיים. שני פתרונות לבעיה זו:

1. נשתמש בקובץ שצורף אחרון בפקודה #ifndef לקדם מעבד. משמעות הפקודה היא שלאחר שהקדם מעבד יצרף את שלושת הקבצים לקובץ אחד, הוא יבדוק אם מוגדרת פונקציה זו, אם לא אז יגדיר, ואם כן ימחק שורות אלו. בכך נפתרה הבעיה.

#ifndef COMPLEX\_H   
#define COMPLEX\_H   
struct Complex {..)

#endif

1. נשתמש בקובץ שצורף אחרון בפקודה #pragma once מעל הפונקציה, כך שלא ניתן להגדיר אותה יותר מפעם אחת.

## ספריות שימושיות

stdio.h: ספרייה סטנדרטית הכוללת בתוכה פעולות קלט ופלט.

stdlib.h: ספרייה סטנדרטית הכוללת בתוכה תכונות רבות כמו: טיפוס size\_t, מצביע NULL, ופונקציות להקצאה דינמית.

math.h: כוללת פעולות מתמטיות כמו שורש, חזקה ופונקציות טריגונומטריות וכו'.

stdbool.h: מוגדר בה הטיפוס bool. בטיפוס זה הקבוע 0 מוגדר כ-false, והקבוע 1 מוגדר כ-true.

string.h: מוגדרים בה כל מיני פונקציות על מחרוזות.

## מודולים

כאשר כותבים קוד מסובך וארוך מאוד בקובץ אחד יכולים להיווצר בעיות, כמו קושי ביכולת לאתר בעיות ולהבין את הקוד. גם במקרים בהם יותר מאדם אחד עובד על הקוד באותו פרק זמן, קובץ יחיד אינו פתרון טוב במצב כזה, כי רק אדם יחיד יכול לעבוד עליו בכל פרק זמן. בנוסף, אם כל הקוד היה מרוכז בקובץ יחיד, אז כל שינוי בקובץ היה מצריך את הידור כל הקוד מחדש. שהו בזבוז משאבי מחשב. לכן מומלץ לחלק את הקוד למודולים, או קבצי קוד שכל אחד מהם מכיל את חלקי הקוד המתאימים לנושא אחד בלבד.

## יבוא משתנים - extern

במקרה והקובץ שלנו מחולק לכמה מודולים, לדוגמא file1.c ו-file2.c, ניתן להגדיר משתנים בקובץ file2 שהם בעצם משתנים גלובליים המיובאים מ-file1. לשם כך נגדיר בקובץ file2, משתנה עם אותו שם כמו ב-file1 ונרשום בתחילת השורה "extern". אמנם אם המשתנה שאנו רוצים לייבא מסוג static נקבל שגיאת קישור linage error. כמו כן אם ננסה לייבא קובץ ששמור בו פונקציה static נקבל שגיאה זו.

File1.c

Int x;

File2.c

Extern int x; // x should be imported

הדרך הנכונה ביותר לייבא משתנים גלובליים, היא ליצור קובץ מיוחד ובו להגדיר אותם, ואז בכל מודול לייבא קובץ זה.

# משתנים וטיפוסים

## הצהרה ואתחול

רושמים את טיפוס המשתנה ואז את שמו. לא כל מילה יכולה להיות שם של משתנה, לדוגמא שם משתנה אינו יכול להתחיל בספרה. נהוג לתת למשתנים שם המלמד על המשמעות שלהם בתוכנית.

כל משתנה שאינו מאותחל מקבל ערך זבל, ובנוסף מאפשר פריצה לתוכנית על ידי האקרים, לכן חשוב לאתחל.

int x=0;

## סוגי משתנים

משתנים לוקליים: משתנים שהוגדרו בתוך בלוק מסוים. נראים רק בתוך בלוק זה, כולל בתוך בלוק פנימי. ניתן להצהיר בתוך בלוק על משתנה עם אותו שם כמו משתנה שהוצהר מחוץ לבלוק. במקרה זה ההצהרה החדשה "מחביאה" את ההצהרה הקודמת. לאחר הבלוק ערך המשתנה יחזור להיות כמו לפני הבלוק.

int x=1;

{   
   int x=2; //x is 2   
} //x is 1 again!

משתנים סטטיים: משתנים החיים מרגע תחילת התוכנית ועד סופה. תמיד מוקצים בתחילת ריצת התכנית, לא משנה היכן הם הוגדרו. נגישים בבלוק בו הם הוגדרו ובכל בלוק פנימי (כמו משתנים מקומיים). מוגדרים ע"י כתיבת המילה השמורה static לפני שם הטיפוס בהגדרת המשתנה.

static int count;

מכיוון שמשתנים אלו חיים לכל אורך התכנית, בפונקציה שמוגדר בה משתנה static, אם נפעילה מספר פעמים, בכל פעם אנו בעצם חוזרים לאותו משתנה, ואם נבצע השמה על משתנה זה השמה זו תישמר בכל פעם שנפעיל את הפונקציה.

אתחול משתנים סטטיים ניתן ע"י ערכים ידועים בזמן קומפילציה בלבד. אם לא צוין ערך אתחול, משתנה סטטי יאותחל ל-0, בניגוד למשתנה מקומי שאם לא יאותחל לא ניתן לצפות את ערכו.

משתנים גלובליים: משתנים שהוגדרו מחוץ לכל הבלוקים, כולל main. נגישים מכל נקודה בתוכנית **החל מרגע הגדרתם**. גם הם חיים מרגע תחילת התכנית ועד סופה. מאותחלים ל-0 אם לא צוין ערך אתחול אחר. ניתן לאתחל רק לערכים קבועים הידועים בזמן קומפילציה.

## משתנים קבועים - const

Const היא מילה שמורה שניתן להפעיל על כל סוג משתנה, מערך או מצביע, כך שאם ננסה לשנות את ערכם נקבל שגיאת קומפילציה. מצהירים על משתנים אלו עם המילה השמורה "const". משתנים אלו נראים רק בבלוק שהם מוצהרים. הרחבה נוספת בפרק "מצביעים".

const int arr[] = {1,2};   
arr[0] = 1; // compile error!

אמנם ניתן לעקוף אותו בדרכים מסוימות. בהמשך לדוגמא הקודמת:

int\* arr\_ptr = (int\*)arr;   
arr\_ptr[0] = 3; // compile ok, but might give a run-time error

## סוגי טיפוסים

ישנם טיפוסים רבים בשפת C:

**שלמים** - char, short, int, long, long long.

**חיוביים** - unsigned char, unsigned short, etc..

**שברים** - float, double, long double.

**Boolean** - אין טיפוסים בוליאנים ב-C, אמנם יש משהו דומה אך מסורבל יותר. ב-C הערך 0 מקבל את הערך false, וכל מספר אחר מקבל את הערך true.

לולאה אינסופיתwhile (1) //

ניכנס לתנאי רק אם If ( !(a-3) ) // a=3

המשמעות היאi=(3==4); // i=0

**Size\_t** - הוא טיפוס מסוג unsigned, שמטרתו היא לייצג גודל של משתנים ומערכים. זהו הטיפוס המוחזר מהפונקציה sizeof(type\_name). כדי להשתמש בטיפוס זה יש לכלול את הספרייה stddef.h, שכלולה בתוך הספרייה stdlib.h.

## בין signed ל-unsigned

כל טיפוס ב-C ניתן להצהיר בתור signed או unsigned. נקדים ונאמר כי כל ערך של משתנה הקומפיילר הופך למספר בינארי כך שהמחשב יוכל לעבוד איתו. ב-C לכל טיפוס אין גודל קבוע של ביטים (מלבד (char אלא משתנה בהתאם למכונה. בטיפוס מסוג signed המספר הבינארי שמייצג אותו עובד בשיטת משלים ל-2, כלומר לפי מספר הביטים המקסימלי x שהוא יכול להכיל נקבע המספר הבינארי המקסימלי (שהוא בגודל ), חצי ממספר זה ומטה משמש למספרים חיוביים וחצי ממספר זה ומעלה משמש למספרים שליליים.

טיפוס מסוג unsigned הוא טיפוס שכל המספר הבינארי מוקדש אך ורק למספרים חיוביים, כך שלא ניתן לייצג בו מספרים שליליים אך המספר החיובי המקסימלי שניתן להגיע אליו הוא פי 2 מטיפוס מסוג signed. אם ננסה לעשות המרה של מספר שלילי לטיפוס unsigned לא נקבל שגיאה, אלא נקבל את הערך החיובי של המספר הבינארי המייצג מספר זה.

Unsigned short num = -1 //Will be interpreted as num=65535

Unsigned int num = -1 //Will be interpreted as num=4,294,967,295

## הסבה - Casting

כשמבצעים אריתמטיקה בין אופרנדים (משתנים או מספרים) מטיפוסים שונים, אזי צריך לבצע פעולה שנקראת "הסבה" (casting), יש כמה אפשרויות לביצוע פעולה זו:

1. השמה של אופרנד עם פחות מידע לאופרנד עם יותר מידע. ההסבה מתבצעת באופן אוטומטי על ידי המהדר ואין איבוד מידע. הכלל הוא:

int a = 1;

float b = a; // legal

הערה: הסבה אוטומטית אינה מתבצעת כשמשתמשים ב-printf(), במקרה זה יש לבצע המרה מכוונת.

1. השמה של אופרנד עם יותר מידע לאופרנד עם פחות מידע. כאן ההסבה לא מתבצעת באופן אוטומטי אלא עלינו "להכריח" אותו. נעשה זאת על ידי שלפני האופרנד שאותו אנו רוצים להמיר לטיפוס קטן יותר, נרשום את הטיפוס הרצוי בסוגריים. מכיוון שבפעולה זו יש איבוד מידע יש להימנע מפעולה זו כמה שניתן, והיא נחשבת ללא אסתטית.

double a = 7.654321;

int b = (int) a;

1. הסבה היא חוקית בין משתנים פשוטים, אך בדרך כלל איננה חוקית עם סוגים אחרים. לדוגמא לא ניתן לעשות הסבה בין משתנה מסוג int ל-struct.

## אופרטורים

חילוק (/) - כשמבצעים חלוקה על מספרים שלמים התוצאה תהיה מספר שלם וזריקת השארית, לכן חשוב לעשות casting או לשנות את אחד השלמים.

float f=1/3; //0   
float f=(float)1/3; //0.3333..

float f=1/3.0; //0.3333..

מודולו (%) - פעולת השארית מוגדרת על שלמים בלבד, ומחזירה את השארית של פעולת החילוק בין שני הפרמטרים שלה.

11%-5 = 1

-11%-5 = -1

קיצורים לאופרטורים אונאריים - האופרטור a++ וכן ++a מבצעים את הפעולה a=a+1, אלא שההבדל ביניהם הוא כאשר יש עוד אופרטור במשוואה. a++ יבצע את הפעולה ואז יקדם את x, ואילו ++a יקדם קודם את x ואז יבצע את הפעולה הנוספת. אותם אופרטורים יש גם במינוס: a-- וגם --a.

Int x=3, y=0;

y = ++x + 5; // y=9, x=4

y = x++ + 5; // y=8, x=4

קיצורים לאופרטורים בינאריים - האופרטור a+=b מבצע את הפעולה a=a+b. אופרטורים דומים יש בין כל ארבעת הפעולות האריתמטיות: +, -, \*, /.

אופרטורי השוואה – הם אופרטורים בינאריים כמו: ==, =!, >, <, =>, +<, שמחזירים 0 או 1 בהתאם לנכונות ההשוואה.

אופרטורים לוגיים – מתייחסים לאופרנדים עליהם מופעלים כערכי אמת, כאשר אופרנד 0 משמעו false, וכל אופרנד השונה מ-0 משמעו true. מחזירים 0 או 1 לפי הכללים הבאים:

!A – הופך את ערך האמת של A. אם A שונה מ-0 אז ביטוי זה שווה 0, ואם שווה ל-0 ביטוי זה שווה 1.

B&&A – מחזיר 1 רק אם גם A וגם B אמיתיים.

B| |A – מחזיר 1 רק אם לפחות אחד מ-A או B אמיתיים.

אופרטורים נוספים – ניתן להפעיל על כל הטיפוסים מסוג מספרים שלמים.

& - כאשר מפעילים בין שני אופרנדים את הסימון &, פעולה זו לוקחת את שני המספרים הבינאריים המייצגים שני אופרנדים אלו ויוצרת מספר בינארי חדש, שבו בכל ספרה יהיה 1 רק אם יש בשני האופרנדים 1 במיקום זה, אחרת יהיה 0.

| - כמו ב-&, רק שבמספר הבינארי החדש יהיה 1 רק אם לפחות באחד מהאופרנדים יש 1 במיקום זה.

^ - כאשר מפעילים בין שני אופרנדים את הסימון ^, פעולה זו לוקחת את שני המספרים הבינאריים המייצגים שני אופרנדים אלו ויוצרת מספר בינארי חדש, שבו בכל ספרה יהיה 1 רק אם בשני האופרנדים הספרה שונה במיקום זה, אם זהה יהיה 0.

~ - כאשר מפעילים אופרטור זה על אופרנד, הוא מחליף את כל הספרות הבינאריות, 1 ב-0, ו-0 ב-1.

Int a = 5, b = 9; // a = 00000101, b = 00001001

c = a&b ; // c = 00000001

c = a | b ; // c = 00001101

c = a ^ b ; // c = 00001100

c = ~a ; // c = 11111010

נרשום 0 לפני בהשמה של מספר כדי לציין שהוא בסיס אוקטלי, ו-0x כדי לציין בסיס הקסדצימלי.

הסימון a<< פירושו הכפלת a ב-2. ו-a>> חילוק a ב-2. הסימון a<<n משמעו .

## enum

(קיצור של enumerated type) הוא כלי בשפת C, שניתן להגדיר בו טיפוס המכיל רשימה של קבועים מסוג int בעלי שם, ואפשר להשתמש בהם בכל מקום שבו משתמשים ב-int-ים. הצורך בכלי זה עולה כאשר אנו רוצים לבחור בין מספר אפשרויות, לדוגמא סוגי כלי רכב, נוכל לעשות זאת על ידי הגדרת משתנה מסוג int כאשר כל ערך מייצג סוג רכב שונה. ב-enum אפשר לעשות זאת בקלות על ידי שנגדיר טיפוס בשם vehicle\_type"", ונגדיר בו קבועים עם ערכים לפי רצוננו.

enum עובד בצורה כזו שאם לא מגדירים את ערך הקבוע, הערך הראשון ברשימה מקבל ערך 0, וכל ערך אחריו מקבל את הערך הקודם + 1.

enum vehicle\_type {

 car, // = 0 by default   
  truck, // = car+1  
  motorcycle, // = car+2  
};

לאחר הגדרת המשתנה באמצעות טיפוס ה-enum שכתבנו, ניתן להתייחס אליו כאל משתנה רגיל היכול לקבל את אחד מהערכים שהגדרנו ברשימה.

vehicle\_type type;  
type = truck;

# תנאים ולולאות

## משפטי תנאי:

if (expression)   
//statement or block  
else if (expression)  
//statement or block

else (expression)  
//statement or block

במקרה הפרטי שבו יש חישוב ערך בודד בבלוק ה- if וה- else ניתן להשתמש באופרטור המקוצר - ?. נשאל האם התנאי מתקיים, אם כן בצע מה שמשמאל הנקודותיים, אחרת בצע מה שמימינם.

y=(x+2 < 0) ? x+2 : -(x+2);

printf(“%d”, (x>y) ? x : y);

## switch

במקרה הפרטי שבו יש לבחור בין פעולות שונות לפי ערכים קבועים של ביטוי כלשהו, נעדיף להשתמש ב-switch. בתנאי מסוג זה רושמים למעלה את הביטוי, ואז לכל ערך שנרצה של הביטוי נוכל לבצע פעולה שונה על ידי שנרשום את הערך הרצוי לאחר המילה "case". מומלץ תמיד לציין מקרה ברירת מחדל default, למקרה שאף אחד מהערכים שהצגנו לא מתקיים.

switch (*expression*) {

case *1*: do something; break;

case *2*: do something; break;

…

default: do something; break

}

## לולאת for:

בנויה משלושה חלקים: אתחול, בדיקת תנאי, קידום. ניתן להשאיר חלקים ריקים בהגדרת לולאה.

הערה: ב-C אין אפשרות להצהיר על משתנה בתוך לולאת ה-for!

int x,y;   
for (x=0,y=0;x<10 && y<5; x++,y+=2)   
//statement or block

## לולאת while:

while (condition)   
//statement or block

## לולאת do while:

do   
//statement or block   
while (condition)

# פונקציות

הפונקציה הראשית main היא פונקציה שחייבת להופיע בכל קובץ C. גם לפונקציה זו יש ערך החזרה return. בדרך כלל נהוג להחזיר 0 בסיום תקין, ולהחזיר כל ערך אחר בשגיאה.

int main()

{

return 0;

}

## כיצד מצהירים:

מצהירים על פונקציה בדיוק כמו שמצהירים על משתנה <type> <name>, כלומר שרושמים קודם את טיפוס המשתנה המוחזר מהפונקציה, ולאחר מכן את שם הפונקציה. ההבדל ביניהם הוא שבהצהרה על פונקציה לאחר מכן צריך להוסיף סוגריים () בהן נרשום את טיפוס הפרמטרים שהפונקציה מקבלת ומהו שמם. בסוף הפונקציה נחזיר את הערך הרצוי באמצעות פקודת return.

int power( int a, int b )

{

…

return ; //the return type must be int

}

כל פונקציה שבהגדרתה מחזירה ערך, כלומר שאינה void, חייב שיהיה בסיומה פקודת return עם ערך מהטיפוס שהגדרנו שהפונקציה מחזירה, אחרת התוכנית לא תתקמפל. כל פונקציה יכולה להחזיר ערך אחד בלבד. במקרה שרוצים שהפונקציה תחזיר ערכים לפי מקרים, ניתן לעשות תנאי כך שבכל מקרה יהיה פקודת return שונה. אמנם צריך לשים לב שבחלוקה למקרים בכל מצב אפשרי רק מקרה **אחד** מתקיים ולא יותר או פחות!

if (x > y){

return x;

}

else{

return y;

}

## פרמטרים

הפרמטרים שהפונקציה מקבלת קיימים רק בתוך הפונקציה ונעלמים לאחר סיום הפונקציה. ניתן להכניס פרמטרים מטיפוס שאינו תואם לטיפוס הפרמטר שהגדרנו, אך שתתבצע הסבה אוטומטית כמו שהסברנו לעיל.

פרמטרים בשפת C מועברים by value, כלומר שכאשר בקריאה לפונקציה מוכנס משתנה בתור פרמטר, לא מועברת הכתובת של המשתנה אלא רק הערך. לכן שינוי ערך הפרמטר בתוך הפונקציה לא ישפיע על ערך המשתנה שמחוץ לפונקציה. הפרמטרים הם בעצם עותקים של הערכים שחושבו בקריאה לפונקציה.

## פונקציה ריקה void

פונקציה שמקבלת ערכים אך לא מחזירה דבר. משמשת כדי לשנות ערכים במשתנים או מערכים בצורה נוחה ומאורגנת. לא הכרחי לכתוב בסוף הפונקציה return.

void power( int a, int b )

{

…

return;

}

## שימוש בפונקציות בתוך פונקציות:

כל פונקציה רואה רק את הפונקציות שהצהירו עליהן מעליה, לכן לא ניתן להשתמש בפונקציה שעדיין לא הצהרנו עליה. ניתן לפתור בעיה זו בשני דרכים:

1. להצהיר בראש התוכנית על הכותרת של הפונקציה בלבד, בלי הבלוק והתוכן. הכלל הוא שניתן להצהיר על פונקציה כמה פעמים אך להגדיר אותה מותר פעם אחת. נסיים כותרת זו עם ";".

int power( int a, int b) ;

1. ליצור קובץ שבו נמצאות כל הפונקציות שאנו נשתמש בהן, נקרא לו "foo". כדי לצרף קובץ זה לקוד שלנו נשתמש בפקודה:

#include "foo.h"

# בדיקת שגיאות

## assert

במקרה שיש איזו פעולה שמחייבת תנאי מסוים בזמן ריצה, נוכל להשתמש בפונקציית assert כדי לוודא שתנאי זה מתקיים, ובמקרה שאינו מתקיים הפונקציה תחסל את המשכיות התוכנית ותחזיר הודעה מתאימה ומיקום הטעות. המוטיבציה להשתמש ב-assert ולא בתנאי רגיל היא מכמה תנאים:

1. פונקציה זו מחזירה הודעה מתאימה כך שאנו יודעים בדיוק היכן הייתה הטעות.
2. מסיימת את התהליך וחוסכת מאתנו בדיקה בכל פעם שנפעיל פעולה זו האם היא הצליחה.
3. קוד קצר וברור יותר.

כדי להשתמש ב-assert יש להוסיף את הספרייה assert.h. לא שולחים לתוך assert פונקציה אלא רק משתנים.

#include <assert.h>   
double Sqrt(double x ) { // compute square root of x. Assumption: x non-negative  
    assert( x >= 0 ); // aborts if x < 0

## DEBUG ו-NDUBG

הם קבועים בעלי משמעות מקובלת שמוגדרים בעזרת הפקודה לקדם מעבד #define, או באמצעות פקודה ב-gcc.

הקבוע DEBUG מציין שמהדרים קוד בגרסה לתיקון שגיאות. והקבוע NDEBUG מציין שמהדרים קוד בגרסה שאיננה כוללת תיקון שגיאות.

## #Ifdef ו-#ifndef

בעזרת הפקודות לקדם מעבד, אפשר לקבוע שקטע קוד יהודר אך ורק בהתאם לשאלה האם קבוע הוגדר או לא הוגדר. בסוף קטע קוד זה נסיים את ההתניה בפקודה #endif. ב-#ifdef אם הקבוע אכן מוגדר קטע הקוד יהודר, ואם אינו מוגדר הקדם מעבד ימחק קטע קוד זה. ב-#ifndef בדיוק ההיפך, אם הקבוע מוגדר ימחק קטע הקוד, ואם אינו מוגדר הקוד יהודר.

#ifdef DEBUG

int is\_sorted(const int \*a, unsigned int length) {

for(i = 1; i < length; ++i)

if(a[i] < a[i - 1])

return 0;

return 1;

}

#endif // ifdef DEBUG

# קלט ופלט

כדי לאפשר פעולות של קלט/פלט יש להוסיף את הספרייה הסטנדרטית - #include <stdio.h>.

## פלט – דרוש בדיקה

כדי להדפיס טקסט יש להשתמש בפונקציה printf(" ");, כל טקסט צריך לרשום עם גרשיים – " ". כדי להדפיס ערכי משתנים צריך להציב "דגלים" במקומות הרצויים בטקסט, על ידי הסימון %, ביחד עם הסימון צריך לציין את סוג הטיפוס שהמערכת מצפה להדפיס. לאחר כתיבת כל הטקסט בסיום הגרשיים, נרשום פסיק ואז נרשום את כל המשתנים שאנו רוצים להדפיס לפי סדר הדגלים שהצבנו. סוגי הדגלים הם:

%d – טיפוס מסוג int.

%f – טיפוס מסוג double. אם רוצים מספר מוגבל של ספרות אחרי הנקודה, נרשום "נקודה" ואז את מספר הספרות הרצוי קודם ה-f, לדוגמא: %.3f.

%c – טיפוס מסוג char.

%u – טיפוס מסוג unsigned int.

%p – טיפוס מסוג מצביע pointer.

%s – להדפסת מחרוזת. יש להעביר כפרמטר ל-printf מצביע לתחילת המחרוזת, ו-printf תדפיס את כל התווים עד שתמצא ‘\0’

int n; float q; double w;   
printf("the variables are: n=%d, q=%f, w=%lf", n, q, w);

בנוסף יש כל מיני תווים מיוחדים:

\n – יורד שורה. ללא הוספת התו הזה מפורשות ההדפסות ימשיכו להיות באותה השורה.

\t – יוצר רווח כמו לחיצה על מקש tab.

\” – מדפיס את התו " .

\\ - מדפיס את התו ' \ '.

%% - מדפיס את התו ' % '.

## קלט

כדי לקלוט ערכים נשתמש בפונקציה scanf();. בתוך הפונקציה נרשום בתוך גרשיים את כל "הדגלים" עם סוג הטיפוס שאנו מצפים לקלוט מהמשתמש כמו שפירטנו לעיל, לאחר מכן נרשום פסיק, ולבסוף את שמות המשתנים שאליהם יקלטו ערכים אלו עם הסימון & לפניהם. יש לשים לב שסדר המשתנים מתאים לטיפוסים שאנו מצפים לקלוט.

scanf("%d %f %lf", &n, &q, &w);

במקרה שהפונקציה scanf לא קלטה מספיק ערכים כמו שביקשנו, התוכנית תיעצר עד שיוכנסו מספר הקלטים הרצוי. במקרה שטיפוס אחד מהמשתנים אינו מתאים לערך שביקשנו להכניס, הפונקציה לא תעשה דבר והערך של המשתנה שאליו ניסינו להכניס קלט לא ישתנה. scanf היא פונקציה שמטרתה לקלוט, אך היא גם מחזירה את כמות הקלטים שהיא הצליחה לקבל, לכן ניתן לוודא שהפונקציה קיבלה את מספר הקלטים שרצינו בדרך הבאה:

do {

res = scanf("%d", &x);

} while (res!=1);

if (scanf("%d", &x)<1)

return(1);

בקלט של מחרוזת עם הדגל %s, scanf מדלג על רווחים, וקולט את המילה עד הרווח הבא.

## חוצץ הקלט – Input Buffer

חוצץ הקלט הוא אזור זיכרון המתוחזק ע"י מערכת ההפעלה. חוצץ הקלט שומר את כל הקלטים שהכניס המשתמש או שנקראו מקובץ, והוא משחרר אותם בזה אחר זה לפי בקשת התוכנית. רק כאשר החוצץ ריק המשתמש יתבקש להכניס קלט. בתחילת התוכנית החוצץ תמיד ריק.

כאשר המשתמש התבקש להכניס ערך אחד, והוא הכניס מספר קלטים עם רווח ולחץ "אנטר", אזי הקלט הראשון יכנס למשתנה וכל השאר ישמרו בחוצץ הקלט עד הפעם הבאה שהתוכנית תבקש קלט. כאשר הקלט אינו מתאים לטיפוס שהתכונית ביקשה, לא יקרה דבר, כלומר התוכן של המשתנה נשאר כשהיה והקלט נשאר בחוצץ.

## קלטים מיוחדים

EOF – בכל קובץ התו האחרון שמציין שהקובץ נגמר נקרא EOF (End Of File), לכן אם נרצה לקלוט קובץ שלם עד סופו נשתמש בפקודה:

While (scanf("%c", ch)!=EOF)

# מצביעים

## הגדרה והצהרה

הזיכרון במחשב מחולק לתאים, ולכל תא יש מספר שנקרא "הכתובת" של התא. כל משתנה תופס מספר מסוים של תאים בזיכרון לפי הטיפוס שלו. הכתובת של כל משתנה היא הכתובת של התא הראשון שמכיל אותו. מצביע או פוינטר (pointer) הוא משתנה שהערך שהוא מכיל הוא כתובת של משתנה אחר. נצהיר על פוינטר בשתי שיטות:

1. בשיטה הראשונה רושמים את סוג הטיפוס של המשתנה שאנו רוצים להצביע עליו, ולאחר מכן כוכבית \* משמאל לשם המשתנה.

<type> \*p;

1. השיטה השנייה היא זהה לשיטה הראשונה, אלא שהכוכבית \* צמודה לטיפוס ולא לשם המשתנה.

<type>\* p;

ההבדל בין שתי השיטות במקרה שנגדיר עוד משתנים בשורת ההצהרה. בשיטה הראשונה הפוינטר הוא רק המשתנים שיש משמאלם כוכבית, ואילו בשיטה השנייה מכיוון שהכוכבית על הטיפוס, כל המשתנים בשורה זו יהיו פוינטרים.

אם לאחר ההצהרה על מצביע לא הגדרנו לאיזה כתובת הוא מצביע בזיכרון, הוא יצביע למקום שרירותי כלשהו בזיכרון.

## השמה של פוינטרים

יש שני סימונים שאנו משתמשים בהם:

\* - סימון שנשתמש בו רק על פוינטרים, \*p משמעו המשתנה שעליו מצביע הפוינטר p.

& - סימון שמשמעו "הכתובת של", ניתן להשתמש בסימון זה גם על פוינטרים וגם על משתנים רגילים, שהרי לכל סוג משתנה יש כתובת בזיכרון. נשים לב כי לכל משתנה רגיל a, הסימון &a הוא פוינטר שמצביע על הכתובת של a.

חשוב לשים לב שבהשמה על פוינטרים אנו מכניסים לפוינטר כתובת ולא ערך של משתנה. קל מאוד להתבלבל בתחום זה וצריך להיות מאוד חדים במשמעויות של כל השמה.

int i=1, j, \*p; // p points to an integer   
   p = &i; // p points to i address  
   j = \*p; // j equals what p point to, j=1  
   (\*p) = 3; // the variable p point to equals 3, i=3

## אריתמטיקה של מצביעים

הערך שנשמר בכל מצביע הוא כתובת של תא כלשהו בזיכרון. לכן פעולות אריתמטיות על מצביעים הם בעצם פעולות על כתובות שהם מספרים. אריתמטיקת מצביעים מאפשרת לקדם מצביעים, וכן לחבר אליהם מספרים שלמים. ניתן לעשות שימוש בכל האופרטורים החיבוריים. כל יחידה מייצגת את גודל התאים בזיכרון של טיפוס המצביע. לדוגמא, אם קיים מצביע p מסוג int\*, הפעולה p++ תבצע p = p+(1\*sizeof(int)). ואילו אם p מסוג char\* הפעולה שהייתה מתבצעת היא p = p+(1\*sizeof(char)), כלומר p היה מצביע לתא הבא.

int a[3];   
int \*p = a;   
char \*q = (char \*)a; // p and q point to the same location   
p++; // increment p by 1 int (4 bytes)   
q++; // increment q by 1 char (1 byte)

ניתן לחסר שני מצביעים זה מזה, רק אם הם מאותו טיפוס. התוצאה מומרת למספר שלם (int) המחזיק את המרחק ביניהם בזיכרון. אסור לחבר/להכפיל/לחלק שני מצביעים, או להכפיל/לחלק מצביע בקבוע.

בשפת C, ניתן להשוות בין שני מצביעים על ידי השוואת הכתובות שהם מכילים. מצביע ptr1 הוא קטן ממצביע ptr2 אם הוא מכיל כתובת מוקדמת יותר בזיכרון. השוואה בין מצביעים מועילה, למשל, כאשר יש לנו שני מצביעים לאותו המערך, ואנו רוצים לדעת איזה מהם מצביע למקום קודם במערך. במקרה זה, אם מצביע אחד קטן מהשני - סימן שהוא מצביע לתא מוקדם יותר במערך.

## שליחת מצביע לפונקציה

מתי נעביר מצביעים כפרמטר לפונקציה?

* 1. כאשר נרצה לשנות את ערכו של המשתנה בתוך הפונקציה נשלח מצביע למשתנה זה, המצביע מקנה לפונקציה גישה ישירה לזיכרון המשתנה.

void swap(int\* x, int\* y)

* 1. ב-C לכל פונקציה ערך החזרה אחד, לכן אם נרצה פונקציה שמחזירה יותר מערך החזרה אחד נממש זאת באמצעות מצביעים.

int foo(int x, int\* res)

פונקציה זו יכולה להחזיר 2 ערכים. ערכו של אחד מהם יוחזר כערכו של המשתנה res.

## זיכרון מצביעים

הגודל של כל מצביע אינו תלוי בטיפוס שאליו הוא מצביע, אלא יש גודל קבוע בזיכרון לכל טיפוסי המצביעים, ברוב המחשבים כיום מצביע תופס 8 בתים (64 ביט). ניתן לחשב אותו באמצעות הפונקציה:

int \*p;   
sizeof (p); // == sizeof (void\*)

## טיפוס void\*

הוא טיפוס של מצביע שלא נקבע על איזה טיפוס הוא מצביע. הוא מדלג בקפיצות של 1 byte כמו טיפוס מסוג char\*. ניתן לגרום לו להצביע לכל משתנה שנרצה בלי להשתמש ב-casting, אמנם אם נרצה שמצביע מסוג שאינו void\* יצביע על אותו כתובת כמו טיפוס מסוג void\* נהיה חייבים להשתמש ב-casting.

void\* q= p; // p is pointer from type int\*. no cast needed   
p = (int\*)q ; // cast is needed

בנוסף, לא ניתן לגשת אל משתנה דרך מצביע מסוג void\* שמצביע עליו, ללא casting. וכן לא ניתן להדפיס ערך שמצביע \*void מצביע עליו.

int j;   
void \*p = &j;   
int k = \*p; // illegal   
int k = (int)\*p ; // still illegal   
int k = \*(int\*)p; // legal

printf("%f\n",\*p)); // illegal

printf("%f\n",\*((double\*)p)) // legal

## NULL

הכתובת 0 הינה כתובת מיוחדת שאינה כתובת חוקית בזיכרון, ומשמשת לאתחול מצביע כאשר רוצים לציין שהוא אינו מאותחל לזיכרון חוקי כלשהו. במקום הכתובת 0, יש להשתמש בקבוע NULL שהוא מצביע מוגן מסוג void\*, שבאמצעות הפקודה #define מוגדר אוטומטית כמצביע לכתובת 0. הוא מוגדר בספרייה stdlib.h. מנגנון זה מאפשר לבדוק את חוקיותו של כל מצביע לפני השימוש בו:

if (p != NULL)

\*p = …

else

printf("p is unallocated!\n");

ניסיון לגשת לתוכן של מצביע NULL גורר מיידית שגיאת זמן ריצה:

int \*p = NULL;

\*p = 3; // run-time error

## מצביע קבוע const

כמו שפירטנו לעיל המילה const בהצהרה על משתנה גורמת לו להיות קבוע בלי שניתן לשנותו. אמנם בהצהרת מצביעים, ניתן להשתמש ב-const כדי להגן על הכתובת שאליה מצביעים או כדי להגן על הערך של המשתנה שאליו מצביעים. המילה השמורה const פועלת כך שהיא מגינה על מה שנמצא משמאלה, אך אם אין שם כלום מגינה על מה שמימינה. חשוב מאוד לשים לב להדרכה למטה, ולא להתבלבל במבחן על מה מגן const, על הכתובת או על ערך המשתנה.

בשני מקרים אלו לא ניתן לשנות את הערך של המשתנה שאליו מצביע p, אמנם כן ניתן לשנות את הכתובת השמורה ב-p.

const int \* p = arr;   
int const \* p = arr;

במקרה זה לא ניתן לשנות את הכתובת השמורה ב-p, אך ניתן לשנות את ערך המשתנה שאליו p מצביע.

int \* const p = arr;

גם הכתובת וגם המשתנה מוגנים.

int const \* const p = arr;

## מצביע למצביע

מצביע הוא סוג של משתנה שמלבד זה שהערך שהוא שומר בתוכו הוא כתובת זיכרון של משתנה מסוים, גם לו עצמו יש כתובת בזיכרון. ולכן ניתן להגדיר מצביע אל מצביע. בדוגמא למטה נוכל לראות את המידע שמכיל מצביע מסוג כזה.

int n = 10**;**  
int \*p = &n;   
int \*\*p2 = &p;   
printf("the address of p2 is %p \n",&p2);   
printf("the address of p is %p \n",p2);   
printf("the address of n is %p \n",\*p2);   
printf("the value of n is %d \n",\*\*p2);

## מצביע לפונקציה

פונקציות הן למעשה רצף של פקודות למעבד המאוחסנות אף הן בזיכרון המחשב, באזור הנקרא code segment. לכן, ניתן גם להגדיר משתנים שמצביעים על כתובות זיכרון בהן יושבות פונקציות. מגדירים מצביע לפונקציה בצורה:

<return\_type> (\*<ptr\_name>)(args);

כאשר return\_type הוא סוג הערך המוחזר מהפונקציה. ptr\_name הוא שם המצביע ו-args הם הארגומנטים שאותה מקבלת הפונקציה. פונקציות כידוע אינם חייבות לקבל ארגומנטים, אמנם כאשר כן מגדירים מצביע לפונקציה שמקבלת ארגומנטים, אין חשיבות לשם המשתנים ואפילו לא חייב לרשום אותם. לדוגמא, כל שלושת שורות אלו מגדירים מצביע לאותו סוג פונקציה:

float (\*p\_fn)(float a, float b);

float (\*p\_fn)(float foo, float bar);

float (\*p\_fn)(float, float);

כאשר רוצים להגדיר כמה מצביעים לפונקציות עם אותו מבנה, מומלץ להשתמש ב-typedef כדי להגדיר שם נרדף לטיפוס של מצביעים לפונקציות מסוג זה, כך הקוד יהיה קצר וקריא יותר. עושים זאת בצורה הבאה:

**typedef** <return\_type> (\*<**type\_name**>)(args);

כאשר type\_name הוא שם הטיפוס החדש. נחליף את הדוגמא לעיל בפקודה המשתמשת ב-typedef, ונצהיר על מצביע p\_fn:

typedef float (\*twoFloatFunc)(float, float);

twoFloatFunc p\_fn;

לאחר שהגדרנו פונקציה foo כלשהי ומצביע לפונקציה מהסוג של foo, נבצע השמה בין המצביע לכתובת של הפונקציה foo. ניתן לרשום בהשמה &foo או פשוט foo, מכיוון שם הפונקציה, כמו מערך, הוא עצמו כבר מצביע לקוד.

p\_fn = &foo // same as p\_fn = foo

קוראים למצביע לפונקציה בצורה הבאה: (\*<p\_fn>)(args) או פשוט p\_fn(args), כאשר p\_fn הוא שם המצביע לפונקציה, ו-args הם ערכי הארגומנטים שאנו שולחים לפונקציה. הקריאה היא קריאה לפונקציה לכל דבר. בפרט, אם הפונקציה מחזירה ערך, אז גם קריאה דרך מצביע תחזיר ערך.

float add(float x, float y) {

return x + y;

}

int main(){

float (\*p\_fn)(float x, float y) = &add;

printf("%f", (\*p\_fn)(2, 3) ); // will print 5

}

ניתן בנוסף לשלוח מצביע לפונקציה כארגומנט לתוך פונקציה אחרת, כך שבתוך פונקציה זו נפעיל את המצביע שגם הוא מפעיל פונקציה.

float add(float x, float y) {

return x + y;

}

void foo(float (\*p\_fn)(float, float)){

(\*p\_fn)(2, 3);

}

int main(){

printf("%f", foo(&add) ); // will print 5

}

# מערכים

## הגדרה והצהרה

מערך הוא מבנה נתונים שמאפשר שמירה של משתנים רבים מאותו טיפוס תחת אותו השם, כאשר כל אחד מהמשתנים מקבל מספר מזהה ייחודי.

כדי להצהיר על מערך נרשום קודם את הטיפוס של המשתנים, לאחר מכן את שם המערך, ובסוגריים מרובעות את מספר המשתנים שאנו רוצים.

<type> <name>[<size>];

Int numElements[5];

כדי לגשת אל משתנה מסוים במערך נרשום את שם המערך, ובסוגריים המרובעות את האינדקס של משתנה זה במערך. חשוב לזכור שהאינדקסים מתחילים מ-0, ולכן בדוגמא לעיל המשתנה numElements[4], הוא המשתנה החמישי והאחרון במערך.

יש שתי שיטות להגדיר את ערכי המשתנים במערך באופן פשוט.

1. בשיטה הראשונה נבנה לולאה שתרוץ על אינדקס המערך, ובכל איטרציה נכניס למערך את הערך שנרצה לפי תבנית מסוימת.
2. השיטה השנייה היא שיטה מקוצרת שנקראת "רשימת אתחול", בה לאחר שמצהירים על המערך רושמים '=' ואז בסוגריים מסולסלות את כל הערכים של המשתנים לפי הסדר הרצוי. אין צורך לרשום את גודל המערך בסוגריים המרובעות, הוא יחושב אוטומטית לפי מספר הערכים בסוגרים המסולסלות. ניתן להשתמש בשיטה זו רק בשורת ההצהרה ולא לאחר מכן! החיסרון בשיטה זו הוא במקרים בהם המערך גדול מאוד, או כאשר לא ידוע למתכנת מהו גודל המערך.

int arr[] = {1, 5, 2, 1, 3};

כאשר הגדרנו גודל מערך, אך לא כתבנו מספיק ערכים בסוגריים המסולסלות, המשתנים בתחילת המערך יקבלו ערכים אלו ואילו המשתנים הנותרים יאותחלו ל-0. לחילופין אם כתבנו רק ערך אחד, כל המשתנים יאותחלו לערך זה.

int arr[3] = {0}; // Initialize all items to 0  
int arr[4] = {3, 4, 5}; // The last is 0

אמנם אם כתבנו יותר ערכים מגודל המערך נקבל שגיאת קומפילציה.

int arr[2] = {3, 4, 5}; // does not compile

## כיצד עובדים מערכים

כאשר הגדרנו מערך, לדוגמא: int arr[5], מערכת ההפעלה מקצה גודל בזיכרון המתאים לגודל המערך לפי הטיפוס שהגדרנו. שם המערך arr הוא בעצם מצביע מוגן אל המשתנה הראשון בהקצאה זו: arr=&arr[0]. כדי לגשת אל משתנה מסוים במערך, לדוגמא arr[3], המחשב מבצע את הפעולה הבאה: \*(arr+3). כך המערכת יודעת על כמה תאים לדלג כדי להגיע אל המשתנה הרצוי.

ניתן ללמוד מכך, שנוכל לגשת לכל משתנה במערך שהאינדקס שלו i בכמה דרכים:

1. הדרך הרגילה והמקובלת - arr[i].
2. דרך נוספת היא - \*(arr+i), המתייחסת לכך ששם המערך הוא פוינטר למשתנה הראשון.
3. נוכל גם להשתמש בביטוי - i[arr], מפני שהמחשב יראה זאת \*(i+arr) כך שנגיע לאותה תוצאה. זו דרך תקינה אך אינה נוחה לקריאה.

בנוסף, למשתנה הראשון arr[0] ניתן לגשת גם באמצעות שם המערך arr\*. מכיוון ששם המערך הינו מצביע מוגן לא ניתן לבצע קידום אריתמטי או השמה למצביע אחר.

a = p; // illegal   
a++; // illegal

**הערה חשובה:** C אינה מבצעת בדיקות חריגה ממערך, ולכן ניתן גם לגשת אל התא arr[-1] או אל התא arr[5] למרות שהם חורגים מגבולות המערך שהצהרנו. המהדר יריץ שגיאה זו אך התוצאות בלתי צפויות. זו אחריותו של המתכנת לוודא שהוא אינו חורג מגבולות המערך.

## מערכים ומצביעים

מכיוון ששם המערך הוא בעצם מצביע אל התא הראשון במערך, נוכל ליצור מצביע נוסף \*p שיצביע גם הוא אל תא זה. התוצאה תהיה שניתן להגיע לכל תא במערך גם דרך מצביע זה ולא רק דרך שם המערך.

int \*p;   
int arr[3];   
p = arr; // same as p = &arr[0]   
\*(p+1) = 2; // same as arr[1]=2

p[2] = 3; // same as arr[1]=3

מכיוון ש-arr הינו מצביע מוגן ואילו p לא, ניתן לבצע עליו קידום אריתמטי p++, או השמה לכתובת אחרת. שוני נוסף הוא גם בגודל הזיכרון אותו הם תופסים.

int \*p;   
int a[4];   
sizeof (p) == sizeof (void\*)   
sizeof (a) == 4 \* sizeof (int)

## שליחת מערך לפונקציה

מכיוון שפרמטרים בשפת C מועברים by value ולא by reference, כאשר שולחים מערך לפונקציה מה שאנו שולחים הוא בעצם פוינטר פשוט שמצביע על המשתנה הראשון במערך. על כן, מכיוון שפונקציה זו צריכה לדעת להתמודד עם כל סוג גודל של מערך, לא נוכל לרוץ בלולאה על אינדקסי המערך, מפני שאיננו יודעים מהו גודל המערך שישלח לפונקציה. גם אם נשתמש בפונקציה sizeof (arr) כמו שפירטנו בסעיף קודם, לא נקבל את גודל המערך אלא גודל מצביע פשוט שהוא תמיד sizeof (void\*).

מגבלה זו מכריחה אותנו כך שבכל פעם שאנו בונים פונקציה שאחד מהפרמטרים שלה הוא מערך, צריך להיות פרמטר נוסף המציין את גודל המערך. מומלץ שגודל המערך ייוצג בטיפוס size\_t מפני שמיועד לכך.

int sum (int arr[], size\_t **n**) {   
   size\_t i;

int sum = 0;  
   for (i=0; i<n; ++i) {  
    sum += arr[i];

}  
 return sum;

}

כששולחים מערך לפונקציה שולחים רק את שם המערך, ללא סוגריים מרובעות. בנוסף, להבדיל משליחת משתנים רגילים, כאשר שולחים מערך לפונקציה שינוי תוכן מערך בתוך פונקציה משנה את תוכן המערך גם מחוץ לפונקציה.

Int arr[] = {2, 2};

sum(arr);

## פעולות בין מערכים

אם רוצים להשוות ערכים של שני מערכים לא ניתן לעשות זאת בדרך הבאה:

int a[4], b[4];   
a = b; // illegal

מפני שכבר אמרנו ששם המערך הוא מצביע **מוגן** אל המשתנה הראשון במערך, ולכן פעולה זו אינה חוקית. אולם כן ניתן להפעיל בתנאי את האופרטור '==' בין שני מערכים, כלומר לשאול האם שנים מצביעים על אותה הכתובת? התשובה כמובן היא תמיד – לא.

if( a == b ) // legal. Always false

## מערך דו-מימדי

ניתן לחשוב על מערך דו מימדי כעל מטריצה, כאשר המערך הראשון מייצג את השורות והמערך השני מייצג את העמודות.

יש שלושה סוגים של מערך דו מימדי:

1. אוטומטי – מערך דו מימדי שתאי הזיכרון שבו הם ברצף ארוך, כאשר סדר התאים הוא קודם כל השורה הראשונה, ואז השורה השנייה וכן הלאה. הצהרה על מערך דו מימדי מסוג זה זהה למערך חד מימדי, אלא שיש להוסיף עוד סוגריים מרובעים, כך שבסוגריים הראשונים נכתוב את גודל המערך הראשון, ובסוגריים השניים את גודל המערך השני.

int arr[2][3];

יש שיטה מקוצרת להגדיר ערך של כל משתנה במערך דו מימדי מסוג זה באמצעות רשימת אתחול. אם משתמשים בה לא חייבים להגדיר את גודל המערך הראשון בהצהרה, המערכת תעשה זאת לבד, אמנם תמיד חייבים להגדיר את גודל המערך השני.

int arr[][3] = {{2,5,7},{4,6,7}}; // Good, works same as arr[2][3]

int arr[2][3] = {2,5,7,4,6,7}; // Bad style, but same as above  
int arr[3][2] = {{2,5,7},{4,6,7}}; // Does not compile

1. דינמי למחצה – הגדרת מערך של פוינטרים, כך שלכל פוינטר הקצינו מספר תאים בזיכרון לפי גודל טיפוס המערך. ניתן גם להגדיר לכל פוינטר כמות שונה של תאים, כך שאם נסתכל שוב על ההשוואה למטריצה, במערך דו-מימדי מסוג זה מספר המשתנים בכל שורה אינו אחיד, בשונה מהסוג הקודם. הבדל נוסף הוא שהתאים בזיכרון אינם מסודרים ברצף אחיד. חשוב לזכור לפנות את הזיכרון שהקצינו בסוף השימוש בו באמצעות free().

int\* m[5]; // allocates memory for 5 pointers   
for (i=0; i<5; ++i)   
   m[i] = (int\*) malloc( 7\*sizeof(int) );  // m[i] now points to a memory for 7 ints.

for (i=0; i<5; i++ )   
   free( m[i] );

1. דינמי – נגדיר מצביע אל מצביע, ונקצה לו בזיכרון מספר תאים לפי גודל כללי של מצביע. מכיוון שכל מספר תאים בגודל זה הם בעצמם כתובת של מצביע, נקצה לכל מצביע כזה מספר תאים בזיכרון לפי טיפוס המצביע הראשון. סוג זה של מערך דו-מימדי דומה לסוג השני בכך שאפשר שמספר המשתנים בכל שורה אינו יהיה אחיד, ובנוסף התאים בזיכרון אינם מסודרים ברצף אחיד. אך שונה ממנו בכך שניתן בכל רגע באמצעות realloc לשנות את גודל המערך הראשון. חשוב לזכור לפנות את הזיכרון שהקצינו בסוף השימוש בו באמצעות free().

int \*\*m;   
m = (int\*\*) malloc( 5\*sizeof(int\*) );   
for (i=0; i<5; ++i)   
   m[i] = (int\*)malloc( 7\*sizeof(int) );

free (m);

הערה: דרך נוספת לטפל במערך דו-מימדי, היא להגדיר מצביע ולהקצות לו זיכרון בגודל סך כל המשתנים במערך הדו-מימדי שאנו רוצים לבנות. כלומר אם מספר השורות הוא R ומספר הטורים הוא C, הגדרת הזיכרון תיראה כך:

int\* A= (int\*) malloc(R\*C\*sizeof(int));

אם נרצה לגשת לתא מסוים A[i][j] במערך, נשתמש בנוסחה: A[ i\*C + j ].

## העברה לפונקציה מערך דו- מימדי

כאשר אנו שולחים מערך דו מימדי לפונקציה, לדוגמא matrix[M][N], אנו שולחים כמובן רק מצביע לאיבר הראשון במערך matrix. אם נרצה להגיע למצביע אל האיבר matrix[i][j], נצטרך לחשב באופן הבא: &matrix[i][j] == &matrix[0][0] + N\*i + j. המסקנה היא שעל מנת לאתר איבר כלשהו במערך דו-ממדי, הכרחי לדעת את אורך השורה שלו, כלומר את מספר הטורים במערך N. זו הסיבה אגב, מדוע כאשר מאתחלים מערך דו-ממדי חייבים לציין את אורך השורה שלו, גם כאשר מציינים מפורשות רשימת אתחול.

אם רוצים להעביר לפונקציה מערך דו מימדי, נהיה חייבים לציין באופן מפורש בחתימת הפונקציה מה אורך השורות במערך במספר קבוע וידוע ולא משתנה. בדר"כ נשלח כפרמטר את מספר השורות r.

void func( int matrix[][], int r) //does not compile

void func (int matrix[][4], int r) // ok

חשוב לשים לב שהפונקציה שהגדרנו זה עתה יכולה לקבל כפרמטר אך ורק מערכים שאורך השורה שלהם הוא 4. כל ניסיון להעביר לה מערך בעל אורך שורה אחר יגרור שגיאת קומפילציה.

# תווים ומחרוזות

## תו char

Char הוא טיפוס שגודלו 1 byte (8 bit) ולכן יכול לשמור רק אחד מ-256 ערכים שונים. לפי טבלת ASCII ממירים בין כל ערך מספרי לתו מסוים. משתנה char יכול לקבל את ערכו על ידי שנציב בו את התו הרצוי עם גרשיים משני צדי התו, או על ידי שנציב את ערך הASCII- של התו.

char c = 'A';

char c = 65;

בנוסף, ניתן להתייחס למשתנה char כאל התו עצמו או כאל ערך הASCII- שלו. לדוגמא:

printf("The character %c has the ASCII code %u.\n", ch, ch);

for (char ch= 'A'; ch <= 'Z'; ++ch)

## מחרוזת

ב-C מחרוזת היא מערך של תווים מסוג char. כל מחרוזת חייבת להיגמר בתו הריק - '/0', שמציין את סיום המחרוזת. שהערך המספרי של התו הריק לפי טבלת ASCII הוא 0. אם נשים באמצע המחרוזת את התו הריק, המערכת תבין ששם נגמרה המחרוזת.

יש שלושה דרכים שונות להצהיר על אותה מחרוזת:

1. בדרך הרגילה להצהרת מערך:

char txt1[] = {’t’, ’e’, ’x’, ’t’, ’\0’};

1. דרך מקוצרת להגדרת מחרוזת. יוגדר מערך בגודל 5 תאים. התא האחרון מכיל את התו הריק.

char txt2[] = "text";

1. דרך מקוצרת באמצעות מצביע.

const char\* txt3 = "text"; //means txt3[5] = g, txt3[6] = \0.

**הערה חשובה**: ההבדל שבין השיטות הראשונות לשיטה השלישית, הוא שבשיטות הראשונות המחרוזת נשמרת במחסנית stack, ולכן ניתן לשנות את ערכי המחרוזת. ואילו בשיטה השלישית המחרוזת נשמרת באזור זיכרון שנקרא "זיכרון הקבועים" (יכול להיות שנמצא ב-code-segment) ולא במחסנית לכן לא ניתן לשנות ערכים במחרוזת. אם בכל זאת ננסה לשנות ערכים בשיטה השלישית נקבל שגיאת זמן ריצה ולא שגיאת קומפילציה. לכן כאשר נגדיר מחרוזת עם מצביע, נשתמש ב-const, כך שאם ננסה לשנות ערכים במחרוזת נקבל שגיאת קומפילציה ולא שגיאת זמן ריצה.

char txt1[] = "text";  
const char\* txt2 = "text";

txt1[0] = ’n’; // now txt1="next“

\*txt2 = ’n’; // illegal! “text” is in the code segment  
txt2 = txt1; // legal. now txt2 points to the same string.

txt1 = txt2; // illegal! txt1 is protected pointer

## העברת מחרוזת לפונקציה

כמו במערכים, אפשר להעביר לפונקציה מצביע לתו הראשון של המחרוזת. אבל בגלל שסוף מחרוזת תמיד מסומן ע"י '\0', אין צורך להעביר פרמטר של אורך.

## מערך של מחרוזות

מערך של מחרוזות הוא בעצם מערך של מצביעים מטיפוס char\*, שכל אחד מהם מצביע למחרוזת אחרת. שימו לב שזהו מערך חד-ממדי רגיל, ולא מערך דו ממדי. כל מצביע יכול להצביע למחרוזת באורך אחר, או אפילו ל-null.

char s[] = "I'm a normal string";

char\* ptrs[3];

ptrs[0] = "I am a happy pointer";

ptrs[1] = s;

ptrs[2] = 0;

ניתן לאתחל מערך של מצביעים למחרוזות באמצעות רשימת אתחול, כמו כל מערך. שימו לב שהכוכבית שהוצמדה קודם לשם הטיפוס char מוצמדת כאן לשם המשתנה. שתי צורות הכתיבה שקולות לחלוטין.

char \*beatles[4] = {"John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr"};

בקוד זה, כל אחת מארבע המחרוזות בתוך ה-{ } הינה מחרוזת קבועה. בתחילת ריצת התוכנית, כל אחת מהן נכתבת לזיכרון המוגן של התוכנית, וכאשר מערך המצביעים מוקצה, כל מצביע במערך מאותחל לכתובת בה נמצאת המחרוזת המתאימה בזיכרון הקבועים.

## פונקציות על מחרוזות

כדי להשתמש בפונקציות אלו יש לצרף את הספרייה string.h.

Strlen – פונקציה המחזירה את מספר התווים במחרוזת ללא התו הריק.

Int strlen(const char \* str);

strcmp – פונקציה זו משווה שתי מחרוזות באופן לקסיקוגרפי (לפי סדר מילוני, על פי טבלת ASCII). היא מחזירה 0 אם שתי המחרוזות זהות, מספר שלילי אם str1 "קטנה יותר" (קודמת במילון), ומספר חיובי אם str1 "גדולה יותר" (מאוחרת יותר במילון).

int strcmp(char \* str1, char\* str2);

בהשוואת מחרוזות ללא הפונקציה אלא בסימון '==', יש להבחין האם המחרוזת הוגדרה דרך מצביע או דרך מערך, מפני שנמצאות באזור אחר בזיכרון. רק במידה ומצביעות לאותה כתובת נקבל true.

strcpy – פונקציה המקבלת שתי מחרוזות, ומעתיקה את השנייה לתוך הראשונה, כולל תו סיום המחרוזת. תוך דריסת התוכן הנוכחי של המחרוזת הראשונה. ערך ההחזרה הוא המצביע של המחרוזת הראשונה.

char\* strcpy(char\* dest, char\* src);

strcat - פונקציה המשרשרת מחרוזת אחת לסוף של מחרוזת אחרת. הפונקציה מניחה ש-dest הוא מערך בגודל שמספיק להכיל את שתי המחרוזות ו-‘\0’ בסוף.

char\* strcat(char\* dest, char\* src);

strdup – פונקציה שמקבלת מחרוזת ומחזירה שכפול של המחרוזת.

char\* strdup(char const \*p );

הפונקציה פועלת עם הקצאת זיכרון ולא משחררת אותו, לכן זה התפקיד שלנו לדאוג שהזיכרון משתחרר בסוף התהליך.

char \*q = strdup( p );

// do something with q

free(q);

# מבנה - struct

## הגדרה

מבנה הוא כלי בשפת C בו אנו יכולים להגדיר טיפוס חדש לפי שם שנבחר. אמנם טיפוס זה אינו טיפוס רגיל אלא ניתן להגדיר בתוכו משתנים פנימיים לפי טיפוס ושם שנבחר. כך שכל משתנה שניצור מהטיפוס החדש הוא בעצם סוג של מערך שמכיל בתוכו משתנים, אמנם בניגוד למערך שבו המשתנים הם מסוג אחיד, בתוך משתנה זה נוכל להגדיר משתנים מאיזה סוג שנרצה ואפילו מצביעים ומערכים. המשתנים הפנימיים נקראים גם "שדות" (fields). אין חשיבות מיוחדת לסדר השדות בתוך המבנה.

לדוגמא, אם נרצה להגדיר טיפוס שנקרא "item", כך שכל משתנה שניצור ממנו יכיל משתנה int, מערך בגודל 3, ומצביע ל-int, נשתמש בהצהרה הבאה: דוגמא זו תלווה אותנו גם בדוגמאות הבאות.

struct item {

   int i;   
   int a[3];   
   int \*p;

};

לאחר הגדרת הטיפוס החדש, נתייחס אליו בדיוק כאל טיפוס רגיל כך שנוכל ליצור משתנה, מערך, ואפילו מצביע אל טיפוס זה. כדי לבנות משתנה מהטיפוס החדש נצהיר בדיוק כמו בכל טיפוס אחר.

struct item milk;

struct item dairy\_products[20];

struct item \*ptr;

## typedef

מכיוון שלטיפוס החדש שאנחנו מגדירים ב-struct יש שם יחסית ארוך, "struct item" בדוגמא לעיל, נוכל לקצר אותו על ידי פונקציה typedef, שניתן באמצעותה להגדיר שמות שבחרנו בתור טיפוסים.

typedef struct item {

   int i;   
   int a[3];   
   int \*p;

} item;

Item milk; // same as – "struct item milk;"

## גישה לשדות במבנה

כדי לגשת אל אחד השדות במשתנה זה נרשום "נקודה" בין שם המשתנה לשדה. נוכל גם להגדיר את כל המשתנה בדרך מקוצרת ביחד עם ההצהרה כמו מערך.

milk.i = 3;

int k;   
struct item milk = {5, {0, 1, 2}, &k};

כמו שכבר אמרנו ניתן להגדיר מצביע אל מבנה כלשהו. כדי לגשת לשדה של מבנה דרך מצביע אליו נבצע בדיוק את אותה הפעולה, אלא שבמקום שם המבנה נשתמש בשם המצביע עם כוכבית בסוגריים עליהם, אחרת המערכת תבין שהכוכבית היא על כל הביטוי. בנוסף, מכיוון שגישה למבנה על ידי מצביע היא מהפעולות השכיחות בשפת C, ישנה דרך מקוצרת לפעולה זו.

Struct item milk;

Struct item \*p = &milk;

milk.a[0] = 3;

(\*p).a[1] = 4; // regular form

p -> a[2] = 5; // Shortened form

## השמה בין מבנים

כאשר יש לנו שני משתנים מאותו הטיפוס של מבנה, ועשינו השוואה ביניהם, אזי התוצאה של פעולה זו היא שכל ערכי השדות שלהם שווה. אמנם אם אחד מהשדות הוא פוינטר, בשני המשתנים השדה של הפוינטר יצביע לאותה כתובת בזיכרון, מפני שהשמה בין פוינטרים היא by value.

typedef struct \_Complex {   
   double \_real, \_imag;   
} Complex;

complex a, b;   
a.\_real = 5;   
a.\_imag = 3;   
b = a; // means that b.\_real=5, b.\_imag=3

## זיכרון

גודל הזיכרון שדורש מבנה נקבע בזמן קומפילציה, אמנם לא בהכרח שהוא רציף אלא יכולים להיות קפיצות בתאי הזיכרון בין שדות של אותו מבנה. חשוב לזכור ששולחים מבנה לפונקציה המבנה מועבר by value, לכן אם יש שדה שהוא מערך צריך לשלוח גם את מספר המשתנים במערך.

## איגוד – union

איגוד הוא דומה מאוד למבנה, מלבד שני הבדלים:

1. ההבדל הראשון הוא טכני, כשמצהירים על איגוד או בהגדרת פונקציה נשתמש במילה union במקום struct.
2. השדות של איגוד תופסים את אותם התאים בזיכרון, כלומר כל פעם שאנו משימים ערך באחד משדות של איגוד אנו דורסים את שאר השדות. התועלת היא חיסכון מקום בזיכרון. הגודל הכללי של איגוד הוא לפי המשתנה הכי גדול.

**union** foo{

int a\_number;

char a\_string[20];

};

int main() {

**union** foo f;

f.a\_number = 3;

printf("%d\n", f.a\_number);

strcpy(f.a\_string, "hello");

printf("%s\n", f.a\_string);

{

# מבנים נוספים

## רשימה מקושרת

רשימה מקושרת היא אחד ממבני הנתונים הדינמיים הבסיסיים וגם השימושיים ביותר. רשימה משורשרת היא סדרה של מבנים שלכל אחד מהם יש ערך ומצביע המצביע למבנה הבא. המבנה האחרון מצביע ל-NULL. כל קודקוד נקרא: NODE, בעברית "צומת".

typedef struct Node {

int data;

struct Node\* next;

} Node;

## עצים

עץ הוא מבנה נתונים הבנוי מקודקודים כך שכל קודקוד מכיל ערך מסוים ומספר מצביעים לקודקודים המשמשים לו בנים. לרוב ההתעסקות היא בעיקר בעצים בינאריים, כלומר עצים שבהם לכל קודקוד יש לכל היותר 2 בנים – בן ימני ובן שמאלי.

typedef struct Node {

int data;

struct Node \*ls, \*rs;

} Node;

הקודקוד הראשון בעץ, זה שאף קודקוד אחר לא מצביע עליו – נקרא ה"שורש" של העץ (root). הקודקודים שאינם מצביעים על שום קודקוד אחר נקראים עלים.

השימוש הנפוץ ביותר לעצים דינאמיים הוא עץ חיפוש בינארי. עץ חיפוש בינארי הוא עץ בינארי המקיים עבור כל קודקוד: תת העץ הימני מכיל קודקודים המכילים ערכים הגדולים (או שווים) מהערך של הקודקוד עצמו. תת העץ השמאלי מכיל קודקודים המכילים ערכים הקטנים מהערך של הקודקוד עצמו.

ערימה היא עץ המקיים תנאי אחד, תנאי הערימה - כל קודקוד בעץ מכיל ערך הגדול או שווה לכל אחד מערכי בניו. (בערימת מקסימום)

# הקצאת זיכרון

## הקצאת זיכרון - malloc ו-calloc

אלו שתי פונקציות שצריך להוסיף את הספרייה stdlib.h להשתמש בהן. באמצעות פונקציות אלו ניתן להקצות זיכרון מה-dynamic heap. כדי להקצות זיכרון, צריך לדעת ראשית את גודל הזיכרון הרצוי, כלומר מהו מספר התאים (bytes) שברצוננו להקצות. לשם כך אנו צריכים לדעת מהו טיפוס המשתנים הרצוי ואת מספר המשתנים הרצויים ברצף. נשתמש בפונקציית sizeof(type\_name) כדי לדעת את גודל הטיפוס, ונכפיל ערך זה במספר המשתנים כדי לקבל את סך הזיכרון.

total size = num\_elem\*sizeof(type\_name)

פונקציות malloc ו-calloc מבצעות את אותה הפעולה, אלא שהפרמטרים שהם מקבלות שונה במקצת. חתימות הפונקציות הן:

void\* malloc( size\_t total\_size );

void\* calloc(size\_t num\_elem, size\_t type\_size);

ניתן לראות ש-malloc מקבלת את כל גודל הזיכרון הרצוי כמו שפירטנו לעיל, ואילו calloc מקבלת את שני הערכים בנפרד לפני הכפל ביניהם. הבדל נוסף הוא בכך ש-calloc גם מאפסת את קטע הזיכרון אותו היא מקצה, ואילו malloc לא, לכן לפני שנגדיר מה יש בכל משתנה במערך המידע שיהיה בו הוא לא מוגדר.

הערך החוזר מפונקציות אלו יכול להיות אחת משתי אפשרויות:

1. אם מערכת ההפעלה הצליחה להקצות את גודל הזיכרון המבוקש, יוחזר מצביע מטיפוס void\* שמצביע על התא הראשון בזיכרון שהוקצה. לכן תמיד נשתמש בהם במשפט השמה אל מצביע שמייצג את התא הראשון במערך. ובנוסף, מכיוון שהערך המוחזר הוא מצביע מטיפוס void\*, נצטרך לעשות casting על הפונקציה כולה אל טיפוס מהסוג של המערך, כדי שהם יוכלו להצביע על אותה כתובת.

size\_t arraySize = readInputFromUser(…);

int\* arr = (int\*)malloc(sizeof(int)\*arraySize);

1. אם לא הצליחה להקצות זיכרון בגודל המבוקש, יוחזר המצביע NULL. לכן תמיד לאחר שימוש בפונקציות אלו נהיה חייבים לבדוק האם הערך המוחזר הוא NULL, ובמקרה שכן, להודיע על השגיאה או לסיים את התהליך הנוכחי. אפשרות נוספת היא להמשיך את ביצוע התוכנית תחת התנאי כי המצביע שונה מ-NULL, כלומר מצביע שונה מ-0.

char \*str = (char \*)malloc(5\*sizeof(char));   
   if (str==NULL) {   
     printf("Error: could not allocate memory!\n");

return 1; // or terminate the calling process

if(!str) //another option

## שחרור הזיכרון - free

לאחר שסיימנו להשתמש בזיכרון שהקצנו, נהיה חייבים לשחרר אותו אחרת הוא ישמר בזיכרון המחשב.

נעשה זאת באמצעות פונקציית free(), שגם היא נמצאת בספרייה stdlib.h. נשלח לפונקציה free מצביע שמצביע על כתובת זיכרון שהוחזרה על ידי אחת מפונקציות ההקצאה, שהיא הכתובת של התא הראשון בזיכרון שהקצינו. הפונקציה תשחרר את הזיכרון ותאפשר הקצאה שלו מחדש.

void free(void \*p); //function signature

free(str);

לפונקציה זו אין מנגנון בדיקת טעויות, לכן אם המצביע הוא NULL, או שהמצביע כבר שוחרר ולא הוקצה מחדש, או שהוא מעולם לא הוקצה ע"י malloc או calloc, נקבל שגיאת זמן ריצה. זהו תפקידנו לוודא שטעויות אלו לא קורות. חשוב מאוד לשים לב שבכל פעם שהפעלנו malloc זכרנו לשחרר את הזיכרון. כמספר הקצאות הזיכרון כך מספר השחרורים.

## הקצאה מחדש – realloc

לעיתים לאחר שהקצינו רצף זיכרון בעל מספר בתים כלשהו אל מצביע, מתברר שיש צורך להגדיל או להקטין את רצף הזיכרון המוקצה. שינויי הקצאות אפשר לעשות בעזרת הפונקציה realloc, המוגדרת בספרייה stdlib.h, על ידי קריאה מהצורה הבאה:

realloc(<old\_ptr>, <total\_size>)

כאשר old\_ptr היא כתובת הזיכרון המוקצית הנוכחית, וtotal\_size הוא הגודל החדש המבוקש (בבתים).

מערכת ההפעלה תנסה לראות האם אפשר לשנות את רצף הזיכרון הנוכחי לגודל המבוקש. אם הדבר אפשרי, הפונקציה תחזיר את כתובת הזיכרון של הרצף הנוכחי מטיפוס void\*, בדיוק כפי שראינו מקודם בהקצאה. אם הדבר אינו אפשרי, היא תבדוק האם יש רצף אחר מתאים בזיכרון. אם היא הצליחה, היא תעתיק את תוכן הרצף הנוכחי לרצף החדש, תשחרר את הרצף הנוכחי, ותחזיר את כתובת הרצף החדש, שוב מטיפוס \*void. אם אין רצף אחר מתאים בזיכרון, היא לא תשנה כלום, לא תשחרר את הרצף הנוכחי, ותחזיר NULL כדי לסמן שלא הצליחה.

על כן, השימוש בפונקציית realloc זהה לפונקציות malloc ו-calloc, שמשתמשים בה רק במשפט השמה עם casting לטיפוס המצביע, וצריך לוודא שהערך המוחזר אינו מסוג NULL. במקרה שכן, יש להודיע על השגיאה או לסיים את התהליך הנוכחי. אפשרות נוספת היא להמשיך את ביצוע התוכנית תחת התנאי כי המצביע שונה מ-NULL, כלומר מצביע שונה מ-0.

int\* arr = (int\*)malloc(sizeof(int)\*oldSize);

arr = (int\*)realloc(arr, sizeof(int)\*newSize);

התאים החדשים שהתווספו למערך שלנו לאחר פעולה זו אינם מאופסים ומכילים מידע לא מוגדר.

## Vla

ב-C99 ישנו כלי שנקרא vla ובאמצעותו ניתן להגדיר מערך בגודל שהוחלט בזמן ריצה בדרך הרגילה של הגדרת מערך.

size\_t length = get\_user\_length();  
int arr[length];

# פונקציות נוספות

## הגדר – #define

פקודת define# נותנת פקודה לקדם מעבד להמיר את כל הערכים של קבוע שהגדרנו אל ערך מסוים. כלומר הקדם מעבד ימחק את שם הקבוע וירשום את הערך שבחרנו במקום. נכתוב את שם הקבוע באותיות גדולות. צריך להשתמש בפקודה זו בזהירות לוודא שאנחנו אכן מקבלים מה שרצינו.

#define FOO 1   
המשמעות היאint x = FOO; // x=1

#define SQUARE(x) x\*x   
המשמעות היא b = SQUARE (a); // b=a\*a

## פונקציית go to

פונקציה שמאפשרת לדלג על קטע קוד מסוים. הפונקציה עובדת בצורה הבאה:

goto bla;   
 //code here   
bla:   
 //code there

לא מומלץ להשתמש בפונקציה זו אלא בשני מקרים מיוחדים כדי שלא ליצור קוד מבולגן. שני מקרים אלו הם או כאשר רוצים לצאת מכמה לולאות מקוננות, או כדי לקפוץ לקטע קוד שמטפל בבעיות.

## פונקציית exit

מסיימת את התהליך הנוכחי, כלומר אם במהלך הרצת פונקציה או בלוק נתקלים בפקודה זו, תסתיים מידית תהליך הפונקציה או הבלוק.

## פונקציית typedef

פונקציה שניתן באמצעותה להגדיר שמות שבחרנו בתור טיפוסים כלשהם. נשתמש בפונקציה זו כדי שהקוד יהיה קריא, קצר וברור יותר.

typedef <existing\_type\_name> <new\_type\_name>;

typedef unsigned int size\_t;

## פונקציית time

הפונקציה time() מחזירה את הזמן הנוכחי כמספר שלם, בד"כ מספר השניות. מוגדרת בספרייה <time.h>.

## פונקציות srand ו-rand

בשפת C מזינים ל srand()-את המספר האקראי ההתחלתי, בדר"כ השעה. זה נעשה פעם אחת בלבד, בתחילת התוכנית. בכל פעם שצריך מספר אקראי נוסף, קוראים לפונקציה rand() המחזירה int אקראי. הפונקציות rand() ו- srand() מוגדרות בספרייה <stdlib.h>.

srand(time(0));

x = rand();

…

x = rand();

# לינוקס

## פקודות על קבצים ותיקיות

1. כשרוצים עזרה עם פקודה מסוימת נרשום "man"ולאחר מכן את שם הפקודה הרצויה. לדוגמא:

man echo

1. כדי לגלות באיזה תיקיה (path) אני נמצא נרשום "pwd".
2. כדי לשנות תיקייה נרשום "cd" ואז את כתובת התיקייה הרצויה.

להגיע לדף הבית נרשום "cd ~"

לחזור לתיקייה הקודמת נרשום "cd ..".

1. כדי למנות את כל הקבצים בתיקייה נוכחית נרשום "ls" (קיצור של list). ניתן להרחיב את הפקודה ls באמצעות הפקודות הבאות. ניתן גם לחבר פקודות אלו.

ls -l – נותן עוד פירוט על כל קובץ.

ls -t – ממיין לפי תאריך שינוי.

ls -s – ממיין לפי גודל.

ls -h – מציין את גודל הקובץ בצורה קריאה.

ls -r – הופך את הסדר.

ls -\*.filesignture – כדי לקבל את כל הקבצים שיש להם סיומת מסוג מסוים.

man ls – לעוד אפשרויות.

1. נשתמש בפקודת "mkdir filename" כדי ליצור תיקייה חדשה בשם שכתבנו.
2. בפקודת "rmdir filename" נמחק תיקייה עם שם שכתבנו. וכדי למחוק קובץ נשתמש בפקודה rm.
3. כדי לשכפל קובץ נשתמש בפקודה "cp", נרשום את השם והסיומת של הקובץ שאנו רוצים לשכפל ואז את השם שאנו רוצים שיהיה של הקובץ המשוכפל.

Cp filename.txt filename1.txt

1. כדי להעביר קובץ לתיקייה אחרת נשתמש בפקודה "mv", נרשום את השם והסיומת של הקובץ, ואת את כתובת התיקייה הרצויה.

mv filename.txt ./new\_directory/

ניתן להשתמש בפקודה זו גם לשנות שם קובץ

mv filename.txt newname.txt

## הרשאות

לכל קובץ בלינוקס יש רמת הרשאה. נוכל להשתמש בפקודה ls -l filename"" כדי גלות את רמת ההרשאה.

"r" – פירושו ניתן לקריאה.

"w" – פירושו ניתן לכתיבה.

"x" – חל רק על תיקיות, פירושו שניתן למנות את הקבצים בתיקייה.

ההרשאה מתחלקת ל-3 חלקים. השמאלי פונה למשתמש הראשי, האמצעי לקבוצה, והימני לעולם.

לדוגמא: -rw-rw-r- פירושו ניתן לקריאה וכתיבה למשתמש ולקבוצה, ורק לקריאה לעולם.

ניתן לשנות הרשאה באמצעות הפקודה "chmod":

להורדת הרשאה:

chmod [**u**ser**/g**roup**/o**thers**/a**ll]-[permission] [file(s)]

להוספת הרשאה:

chmod [**u**ser**/g**roup**/o**thers**/a**ll]+[permission] [file(s)]

## הפעלת קובץ

כדי להפעיל קובץ נרשום "./" ואז נרשום את שם הקובץ והסיומת שלו. וכדי לסיים אותו נלחץ ctrl+c.

./ filename.txt

כדי לספור כמה שורות, מילים ואותיות יש בקובץ טקסט, נשתמש בפקודה "wc". בתוצאה הטור השמאלי מייצג מספר שורות, האמצעי מספר מילים, והימני מספר אותיות.

## Piping

ניתן לחבר בין תוכניות בדרכים הבאות:

1. a | b – הפלט של תוכנית a הופכת להיות הקלט של תוכנית b.
2. a > file.txt – הפלט של תוכנית a נכתב בקובץ ששמו file. אמנם כל מה שהיה בקובץ file נדרס.
3. a >> file.txt – הפלט של תוכנית a נכתב בקובץ ששמו file. אך מתווסף ולא נדרס.
4. a < input.txt – הקלט של תוכנית a נקרא מקובץ בשם input.

## שלבי קימפול תוכנית

בפרק הראשון הסברנו שתהליך הקימפול ב-C מחולק לשלושה חלקים: קדם מעבד (preprocessor), מהדר (compile), ומחבר (linkage). הקדם מעבד והמהדר לוקחים את כל קבצי הקוד (file.c) בתכנית שלנו, ומחזירים במקומם קבצי object (file.o). המחבר לוקח את כל קבצי object ומחבר אותם לקובץ אחד (בשם של קובץ main) המוכן להרצת המחשב.

באמצעות gcc ניתן לבצע שתי פעולות אלו בנפרד. אם לדוגמא יש לנו שני קבצי קוד, אחד בשם main.c והשני בשם square.c. השורה הראשונה תיצור שני קבצי object – main.o ו-square.o. השורה השנייה תיצור את הקובץ המאוחד המוכן להרצה.

$ **gcc** **–c** Square.c –o Square.o // preprocessor and compile only

$ **gcc** **–c** Main.c –o Main.o

$ **gcc** Square.o Main.o –o Main // linkage only

אם נעדכן קובץ אחד בתוכנית שלנו, באמצעות פקודות אלו לא נצטרך לקמפל את כל הקבצים שוב אלא רק את אותו קובץ.

## קבלת ערכים בשורת הפקודה

כדי שנוכל להכניס ערכים לתוכנית דרך שורת הפקודה נגדיר את פונקציית main בדרך הבאה:

Int main (int argc, char\* argv[]) { ...

לאחר הגדרה זו אם נרשום בשורת הפקודה:

$ prog1 –u danny –p 1234

כל ביטוי שיש אחריו רווח, יקלט בתוכנית בתור איבר במערך argv:

argc == 5   
argv[0] == “prog1”   
argv[1] == “-u”  
 ...   
argv[4] == “1234"

ניתן להבחין שיש דמיון ל-JAVA מבחינת הגדרת הפונקציה main. אמנם בניגוד ל-JAVA ב-C האיבר הראשון הוא שם התוכנית עצמה.

## הגדרת קבוע

ניתן להגדיר שקבוע מוגדר לערך כלשהו דרך פקודת gcc, במקום להשתמש ב-#define. כותבים:

gcc -D <**const**> = <val> <source\_file> -o <executable>

כאשר const הוא הקבוע, val הוא ערכו, source\_fileהוא שם קובץ הקוד, ו executable-הוא שם קובץ התוכנית המתקבלת. לדוגמה, הפקודה:

gcc -D RED=1 red\_test.c -o red\_test.out

תהדר את הקובץ red\_test.c, ובכל מקום בו תזהה בו את הרצף RED היא תחליפו ל-1.

כדי להגדיר שקבוע כלשהו סתם מוגדר (לא לערך כלשהו), נבצע את אותה הפעולה בלי val.

gcc -D DEBUG red\_test.c -o red\_test.out