**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 התוכנית, 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 התוכנית, 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 התוכנית 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". בחלק מהקומפיילרים משתנה קבוע אינו תופס מקום בזיכרון. הקומפיילר לוקח את הערך שהכנסנו למשתנה הקבוע, ובכל מקום שכתוב שם המשתנה מדביק את הערך שקיבל.

משתנים אלו נראים רק בבלוק שהם מוצהרים. הרחבה נוספת בפרק "מצביעים".

const int arr[] = {1,2};   
arr[0] = 3; // 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 שלמים** - 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 a, ואילו ++a יקדם קודם את a 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 .

## lvalue ו- rvalue

אחד המושגים שמופיעים הרבה בתיעוד של ++C ובהודעות-השגיאה של קומפיילרים הם: lvalue, rvalue. שהם שני החלקים של משוואה בC++.

* **lvalue** - מציין דבר שיש לו כתובת בזיכרון, דבר שמתקיים גם אחרי שהביטוי מסתיים. למשל: משתנה, רפרנס, פוינטר, וכו'. הוא נקרא **l**value כי יש לו **l**ocation וכי אפשר לשים אותו בצד השמאלי (**l**eft) של סימן '=' .
* **rvalue** - מציין דבר שלא ממשיך להתקיים אחרי שהביטוי מסתיים. למשל: מספר, מחרוזת קבועה, ביטוי מתמטי (x+y), קריאה לפונקציה f(123). הוא נקרא **r**value כי הוא יכול להופיע רק בצד הימני (**r**ight) של סימן '=' .

משוואה בC++ יכולה להיות אחת משני אפשרויות:

lvalue = rvalue;

lvalue = lvalue;

## 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;

**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 בדרך הבאה:

<type> \*p;

אפשר גם להצהיר שה-\* נמצאת מימין לטיפוס ולא משמאל למשתנה. ההבדל בין ההצהרות 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.

<type>\* p;

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 is pointing to equals 3, that is 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. 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.

int\* p;

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 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

}

# משתנה מיוחס - Reference

## הגדרה והצהרה

משתנה רפרנס הוא שם אחר (כינוי) למשתנה שכבר קיים. אם נחשוב על שם משתנה רגיל כתווית המחוברת למיקום המשתנה בזיכרון, אזי רפרנס הוא כמו תווית שנייה המחוברת למיקום הזיכרון ההוא. לכן ברגע שמאתחלים רפרנס עם משתנה אחר, ניתן לגשת לתוכן המשתנה, ואף לשנות אותו, דרך שם המשתנה המקורי או דרך הרפרנס. מבחינת הקומפיילר רפרנס הוא פוינטר אך אנו יכולים להתייחס אליו כמשתנה רגיל.

משתנה רפרנס פועל בדומה למצביע, ואכן בא במקרים מסוימים להחליף אותו, ולכן יש ביניהם בלבול. אך ישנם הבדלים עיקריים בין משתני רפרנס למצביעים והם:

* רפרנס חייב להיות מאותחל כבר כשהוא נוצר. לעומת זאת מצביעים אפשר לאתחל מתי שרוצים.
* רפרנס לא יכול להפנות לערך NULL, חייב תמיד שהוא יכיל ערך חוקי בזיכרון.
* לאחר אתחול של הרפרנס למשתנה לא ניתן לשנות אותו להתייחס למשתנה אחר, הם נעולים ביחד. מצביעים לעומת זאת ניתן לשנות אותם שיצביעו אל משתנה אחר בכל זמן.
* כמו במצביע גם בעזרת רפרנס ניתן להשפיע בעקיפין על משתנה אחר, אלא שהוא חוסך את הכתיבה המסורבלת של אופרטורים נוספים. במצביעים צריך להשתמש ב-\* במקרה של משתנה רגיל וחץ (<-) במקרה של אובייקט או מבנה כדי להגיע אל הערך השמור במצביע. אך ברפרנס ניתן לפנות למשתנה באופן טבעי הרבה יותר באמצעות שם הרפרנס בלבד במקרה של משתנה רגיל, ואופרטור נקודה במקרה של אובייקט או מבנה.
* לא ניתן לבצע אריתמטיקה על רפרנס, וכן לא ניתן להכניס אותו למערך או container. לעומת זאת מצביעים כן אפשר.

בהצהרה על רפרנס יש לרשום את שם הטיפוס שהוא מפנה אליו, ולאחר מכן מוסיפים & (בצמוד לטיפוס או לשם המשתנה) המסמל שזהו משתנה רפרנס. כאמור לעיל, חייבים לאתחל אותו מיד עם ההצהרה עליו.

int i = 5;

int& r = i;

i = 10; // r = 10

r = 15; // i = 15

השימוש העיקרי ברפרנס הוא בהעברתו כארגומנט לפונקציה, במקרה שאנו רוצים שהוא יושפע ממנה, או כערך מוחזר מפונקציה, במקרה שאנו רוצים לבצע פעולות על הערך במוחזר.

## משתני רפרנס כפרמטרים

בשפת C אם רצינו לשלוח משתנים לפונקציה, כך שבתוך הפונקציה נוכל לשנות את הערכים בהם, היינו עושים זאת באמצעות מצביעים, אך זה היה דורש מאיתנו להפוך את המשתנים למצביעים ולהשתמש באופרטורים רבים. בC++ נוכל להגדיר בפונקציה את הפרמטרים כרפרנסים ואז שליחת הפרמטרים לפונקציה והשימוש במשתנים בתוך הפונקציה יהיה עם שמות המשתנים בלבד. לדוגמה, נכתוב פונקציה שמחליפה בין ערכיהם של שני משתנים בשם swap.

void swap(**int& x, int& y**) {

int temp = x; // save the value at address x

x = y; // put y into x

y = temp; // put x into y

}

int main () {

int x = 100;

int y = 200;

swap(**x, y**); // x = 200, y = 100

return 0;

}

### משתני רפרנס קבועים

בגלל שרפרנס אמור להצביע לעצם שאפשר לשנות אותו הוא לא יכול להצביע למספר או לביטוי זמני, לכן אי אפשר לאתחל רפרנס עם דבר שהוא לא lvalue. אולם אם מגדירים את הרפרנס כקבוע const אפשר להציב rvalue בתוך הרפרנס, שהרי אז הדבר שמפנים אליו מוגדר כ"קבוע" והקומפיילר ממילא לא ייתן לנו לשנות אותו. במקרה זה, הקומפיילר מאריך את אורך החיים של המשתנה הזמני, כך שיישאר בזיכרון (בד"כ על המחסנית) למשך כל אורך החיים של הרפרנס.

const int& r = 5; // legal

const int& r = x+y; // legal

השימוש העיקרי של const reference הוא כשרוצים להעביר ארגומנטים לפונקציות. זה שימושי במיוחד כשהארגומנט הוא אובייקט גדול. יש כמה דרכים להעביר אותו לפונקציה:

* אם מעבירים אותו כערך - הוא מועתק למשתנה זמני שמועבר לפונקציה. זה יכול להיות מאד בזבזני כשהמשתנה גדול.
* אם מעבירים אותו כרפרנס - הוא לא מועתק, רק הכתובת שלו מועתקת. אבל, הפונקציה יכולה לשנות את ערכו, וזה עלול לגרום שגיאות לוגיות.
* אם מעבירים אותו כרפרנס קבוע (const &) - רק הכתובת שלו מועתקת, ובנוסף, הקומפיילר מוודא שאי אפשר לשנות את ערכו. **יתרון נוסף** הוא, שאפשר להעביר לפונקציה rvalue - למשל ערך זמני המגיע מקריאה לפונקציה קודמת.

## משתנה רפרנס כערך מוחזר מפונקציה

כאשר מחזירים רפרנס מתוך פונקציה ניתן לבצע השמה על הערך המוחזר מהפונקציה. לדוגמא, עבור מחלקה המייצגת רשימה של מספרים שמוגדר בה השיטה get, אם משיטה זו נחזיר רפרנס (int& במקום int) נוכל להשתמש באותה השיטה לקריאה ולכתיבה. בנוסף, כשכותבים שיטה שמשנה מחלקה, מקובל להחזיר רפרנס למחלקה עצמה, כדי שיהיה אפשר לקרוא לכמה שיטות עדכון בשרשרת.

class Point {

int x, y;

**Point&** add (const Point& b) {

x += b.x;

y += b.y;

**return \*this;**

}

};

int main() {

Point p1(2,3), p2(4,5), p3(1,1);

**p1.add(p2).add(p3);** // now p1 is (7,9)

{

אם לא מעוניינים לבצע שרשור כזה, אזי אין הבדל בין להחזיר משתנה מקומי לרפרנס מבחינת היעילות. רק בשליחת פרמטרים לפונקציה יש הבדל כזה.

בעת החזרת רפרנס צריך להיזהר שהאובייקט אותו מחזירים מוגדר גם ב-scope שממנו קראנו לפונקציה, ואינו משתנה מקומי בתוך הפונקציה בלבד, אחרת נקבל שגיאה. אמנם תמיד אפשר להחזיר במצב כזה משתנה סטטי שמוגדר בכל התוכנית.

## רפרנס למצביע

ניתן ליצור רפרנס למצביע שהוא תווית של המצביע. ואזי בכל פעם שנגדיר שהרפרנס הזה מצביע למקום אחר גם המצביע המקורי יצביע למקום זה.

int i = 5, k = 10;

int\* p = &i;

**int\*& rp = p;**

rp = &k; // p\* = 10

**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);