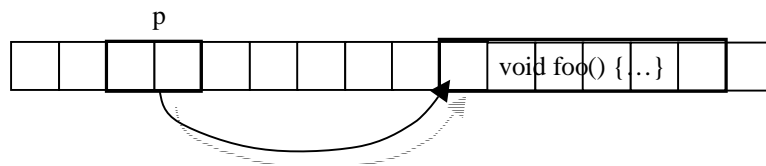


## מצביעים לפונקציות

### משתנה המכיל כתובת של קוד

מצביע הוא משתנה המכיל כתובת זיכרון. האינפורמציה שנמצאת באותו מקום בזיכרון יכולה להיות מסוגים שונים. כבר ראינו מצביעים למשתנים כמו `int` או `double`. ראינו מצביעים למבנים וראינו מצביעים למצביעים מסוגים שונים. כעת נראה מצביעים המצביעים לאינפורמציה מסוג אחר – קוד. כידוע, כאשר תוכנית רצה במחשב, היא נמצאת (לפחות בחלקה) בזיכרון המחשב. מצביע יכול להחזיק את הכתובת של תחילת `main` או כל פונקציה אחרת.



C מאפשרת להפעיל את הפונקציה המוצבעת דרך התייחסות למצביע. לדוגמה:

```
#include <stdio.h>

double foo(int a, float b) {
    return a+b;
}

int main() {
    double (*p)(int,float);

    p = foo;
    //p = &foo; // causes the same effect

    printf("%g\n",p(1,2));
    printf("%g\n",(*p)(1,2)); // same thing
    return 0;
}
```

בתחילת הפונקציה `main`, מוגדר משתנה לוקאלי בשם `p`. טיפוס משתנה זה הוא:

```
double (*)(int, float)
```

כלומר, מצביע לפונקציה המקבלת פרמטר ראשון מסוג `int`, פרמטר שני מסוג `float` ומחזירה ערך מסוג `double`. כזכור, טיפוס המצביע הוא זה שמגדיר את האינטרפרטציה של האינפורמציה באזור המוצבע. במקרה זה, כאשר `p` יכיל כתובת, התוכנית תתייחס לכתובת זו כתחילה של פונקציה המקבלת `int` ו-`float` ומחזירה `double`. התחביר של שורת ההצהרה עלול להיראות קצת מוזר, היינו מצפים אולי להצהרה כזו:

```
double (*)(int,float) p;
```

אך לא אלו חוקי השפה.

השורה השנייה של `main` משימה ל-`p` את ערך הביטוי `foo`. ב-`C` ערך שם פונקציה הוא הכתובת של הפונקציה בזיכרון. המצביע `p`, אם כן, מצביע כרגע על תחילת הקוד של הפונקציה `foo`. כרגיל, השמה דורשת התאמה של טיפוסים. במקרה זה ל-`p` ול-`foo` יש באמת אותו הטיפוס.

ניתן להתייחס גם לכתובת של שם הפונקציה ולקבל שוב את כתובת ההתחלה שלה. הביטויים foo ו-foo & שווים מבחינת ערך ומבחינת טיפוס (וחבל). הפעלה של הקוד עליו מצביע p נעשית כאילו p הוא שם הפונקציה: p(1,2). גם להפעלה יש תחביר נוסף בעל משמעות זהה: (\*p)(1,2). הפלט של הקוד יהיה אם כן:

3  
3

### תכנות גנרי

היכולת להתייחס לקוד כמשתנה מאפשרת כתיבת קוד ברמת אבסטרקציה גבוהה יותר. לדוגמה, הפונקציה הבאה מדפיסה "לוח כפלי" לפעולה בינארית כלשהי:

```
void printBoard(double (*f)(double,double)) {  
    int i,j;  
    for(i=1; i<=10; ++i) {  
        for(j=1; j<=10; ++j)  
            printf("%.2f\t",f(i,j));  
        printf("\n");  
    }  
}
```

כעת ניתן להוסיף את הקוד הבא:

```
double mul(double a, double b) { return a*b; }
```

```
double div(double a, double b) { return a/b; }
```

```
int main() {  
    printBoard(mul);  
    printf("\n\n");  
    printBoard(div);  
    return 0;  
}
```

שידפיס את לוח הכפל ולוח החילוק.

ללא שימוש במצביע לפונקציה היינו כותבים קוד כזה:

```
void printBoard(int funType) {  
    int i,j;  
    for(i=1; i<=10; ++i) {  
        for(j=1; j<=10; ++j) {  
            if(funType == MUL)  
                printf("%.2f\t",mul(i,j));  
            else if (funType == DIV)  
                printf("%.2f\t",div(i,j));  
            ...  
        }  
        printf("\n");  
    }  
}
```

(זאת בהנחה שאנו לא רוצים לשכפל את printBoard עבור כל פעולה בינארית) קל לראות שגרסה זו פחות טובה מהגרסה הקודמת. חיסרון בולט אחד הוא היעילות: פונקציה זו צריכה לברר לעצמה עבור כל אבר שהיא מדפיסה, איזה פעולה עליה לבצע. כל ברור כזה דורש מעבר על כל הפעולות האפשריות. חיסרון נוסף הוא הארכיטקטורה. אם רוצים להוסיף פעולה בינארית חדשה, את הגרסה הראשונה כלל לא צריך לשנות. את הגרסה השנייה צריך.

הדוגמה הבאה מראה שימוש במצביע לפונקציה על מנת לכתוב פונקציה כללית לחישוב קירוב של אינטגרל מסוים ע"י סכומי רימן:

```
#include <stdio.h>
#include <math.h>

double integral(double (*f)(double), double x1, double x2) {
    static const double eps = 1e-7;
    double x, sum = 0;
    for(x=x1; x<=x2; x+=eps)
        sum += eps*f(x);
    return sum;
}

double linear(double x) {
    return 2*x+1; // integral = x^2+x
}

double quadratic(double x) {
    return pow(x,2); // integral: x^3 / 3
}

double exponential(double x) {
    return pow(M_E,x); // integral: e^x
}

int main() {
    printf("%g\n",integral(linear,0,2)); // 2^2+2 - 0 = 6
    printf("%g\n",integral(quadratic,0,2)); // 2^3/3 - 0 = 8/3 = 2.6666
    printf("%g\n",integral(cos,0,M_PI)); // sin(pi)-sin(0) = 0
    printf("%g\n",integral(exponential,0,1)); // e-1
    return 0;
}
```

### שם חדש לטיפוס מצביע לפונקציה

כמו לכל טיפוס אחר, גם לטיפוס מצביע לפונקציה ניתן לתת שם חדש בעזרת typedef. התחביר שבו יש להשתמש עלול להיראות לא אינטואיטיבי. לדוגמה:

```
#include <stdio.h>
#include <math.h>

typedef double(*func_t)(double); // that is the strange correct syntax.
// typedef double(*)(double) func_t; // this somewhat more reasonable syntax won't compile

func_t greaterOnZero( func_t f, func_t g) {
    if(f(0) > g(0))
        return f;
    return g;
}

int main() {
    printf("%f\n",greaterOnZero(sin,cos)(M_PI)); // output: -1
}
```

```
return 0;
}
```

בדוגמה זו, השימוש ב- typedef היה הכרחי מכיוון שהקומפילר לא יודע להתמודד עם כתיבה מפורשת של מצביע לפונקציה כערך מוחזר. ניסיון לכתוב את הפונקציה באופן הבא לא יעבור קומפילציה:

```
double(*) (double) greaterOnZero( ...) {...}
```

### דוגמאות נוספות לשימוש במצביעים לפונקציות

הקוד הבא נעזר בספרייה curses על מנת לאפשר למשתמש להזיז את הסמן על המסך. בכל שלב, המשתמש יכל לבחור להזיז את הסמן לארבעה כיוונים: שמאלה, ימינה למעלה ולמטה. אם היינו כותבים את הקוד ללא שימוש במצביע לפונקציה, בכל שלב היה צריך לברר ע"י סידרה של שאלות, על איזה מקש לחץ המשתמש. במקרה הגרוע מדובר בארבע שאלות לחיצה. מובן שאם היו יותר אפשרויות, סידרת השאלות הייתה מתארכת וחוסר היעילות היה גדל. בדוגמה זו, במקום לברר את סוג המקש, מפעילים קוד המשודך עליו. השידוך נעשה ע"י שימוש במערך של מצביעים לפונקציות:

```
#include <curses.h>
```

```
typedef void (*fun_t)(int*,int*);
```

```
void doNothing(int *x, int *y) {
    addstr("this key is not bound to a function");
}
```

```
void left(int *x, int *y) {(*x)--;}
```

```
void right(int *x, int *y) {(*x)++;}
```

```
void up(int *x, int *y) {(*y)--;}
```

```
void down(int *x, int *y) {(*y)++;}
```

```
int main() {
    int x = 10, y = 10;
    unsigned char c;
    initscr();
    fun_t arr[256];
    for(i=0; i<256; ++i)
        arr[i] = doNothing;
    arr['b'] = right;
    arr['v'] = left;
    arr['n'] = up;
    arr['m'] = down;
    while(1) {
        move(y,x);
        refresh();
        c = getch();
        if(c == (unsigned char)27)
            break;
        arr[c](&x,&y);
    }
    endwin();
}
```

```
    return 0;
}
```

בקומפילציה יש כמובן ללנקגי עם `curses`:

```
gcc -Wall check.c -lcurses -o check
```

הדוגמה הבאה מראה איך אפשר להשתמש במצביע לפונקצית השוואה על מנת לכתוב קוד מיון כללי. בנוסף, ניתן לראות דוגמה לפונקציה המקבלת מצביע לפונקציה המקבלת מצביע לפונקציה (הקוד המודגש):

```
#include <stdio.h>
const int SIZE = 10;

int less(double v1, double v2) { return v1 < v2; }
int more(double v1, double v2) { return v1 > v2; }
void swap(int *p1, int *p2) {
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}

void print(int *v) {
    int i;
    for(i=0; i<SIZE; i++)
        printf("%d ",v[i]);
    printf("\n");
}

void mysort(int * v, int (*comp)( double, double)) {
    int i,j;
    for( i=0; i<SIZE-1; i++)
        for(j = i+1; j<SIZE; j++) {
            if(comp(v[j],v[i]))
                swap(v+j,v+i);
        }
}

void printMost(int* v, int (*comp)( double, double)) {
    int i,most = v[0];
    for(i=1; i<SIZE; i++)
        if(comp(v[i],most))
            most = v[i];
    printf("The most: %d\n",most);
}

void test(int *v, void (*f)(int*, int*)( double, double)) {
    f(v,less);
    print(v);
    f(v,more);
    print(v);
}

int main() {
```

```

int v[] = {7,2,8,5,1,9,6,3,10,4};
test(v,mysort);
test(v,printMost);
return 0;
}

```

פלט:

```

1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
The most: 1
10 9 8 7 6 5 4 3 2 1
The most: 10
10 9 8 7 6 5 4 3 2 1

```

### מימוש polymorphism בעזרת מצביעים לפונקציות

polymorphism הוא מנגנון המאפשר לבקש מאובייקט לעשות משהוא דרך התייחסות אבסטרקטית. האובייקט אמור למלא את הבקשה בצורה האופיינית לו. המנגנון מאפשר כתיבת קוד מופשט המקבל משמעויות שונות בהתאם לאובייקטים אליהם הוא פועל. אחת הדוגמאות הקנוניות לשימוש בפולימורפיזם הוא קוד הפועל על צורות מופשטות, מבקש מהן לחשב את שיטחן, לזוז וכד' והן מבצעות את הבקשות כל אחת בדרכה. אפשר לממש פולימורפיזם גם ב - C.

נתבונן בקוד הבא:

```

int main() {
    const unsigned N = 3;
    unsigned i;
    Shape** shapes = (Shape**)malloc(sizeof(Shape*)*N);
    shapes[0] = createCircle(2,2,5); // a circle at (2,2) with radius = 5
    shapes[1] = createRectangle(10,8,7,3.4); // a rectangle at (10,8) width = 7 height = 3.4
    shapes[2] = createRectangle(1,2,8,10);

    for(i = 0; i<N; ++i) {
        shapes[i]->print(shapes[i]);
        shapes[i]->move(shapes[i],1,2);
        printf("area: %g\n",shapes[i]->area(shapes[i]));
    }
    ... free allocated memory...
}

```

כפי שניתן לראות, הקוד מגדיר מערך של מצביעים לצורה כללית ושם בו עיגול ושני מלבנים. לאחר מכן הוא עובר על המערך סדרתית ומבקש מכל צורה להדפיס את עצמה, לזוז ולחשב את שיטחה. הקוד הוא מופשט במובן זה שהוא איננו מתעניין בסוג הצורה שבמערך, הוא מתייחס אליה כצורה כלשהי ומבקש ממנה לעשות דברים שכל צורה יודעת. כל צורה ספציפית כמובן אמורה לבצע את המטלות בצורה האופיינית לה.

נתבונן על בפקודת ההדפסה הכללית (השורה המודגשת). ניתן לראות כי print צריך להיות שדה של Shape מטיפוס מצביע לפונקציה המקבלת צורה. כיצד print יראה את המאפיינים של הצורה הספציפית ? איך בכלל יראה היחס בין צורה ספציפית לצורה כללית כאשר ב - C אין תמיכה בירושה ? איך תיקרא דווקא פונקציה ההדפסה המתאימה לצורה הספציפית ?

הקוד שניתן בהמשך עונה על השאלות בדרך הבאה: למבנה Shape יהיו שדות print, area ו - move שיהיו מצביעים לפונקציות ובנוסף מצביע מסוג void\* למבנה המכיל פרמטרים ספציפיים

לכול צורה. כאשר יבנה עיגול למשל, יבנה מבנה מסוג Shape, המצביע מסוג void\* שלו יצביע על מבנה ספציפי לעיגול המכיל רדיוס, והשדות print, ו- area יצביעו על פונקציות ספציפיות לעיגול. כאשר תופעל הפונקציה עליה מצביע print, היא תדע שמדובר בעיגול ותוכל להמיר את המבנה המוצבע ע"י ה- void\* למבנה פרמטרים של עיגול. נתבונן בקוד:

```
typedef struct shape Shape;
typedef void (*FunV)(Shape*);
typedef double (*FunD)(Shape*);
typedef void (*FunSB) (Shape*, char*);
```

מבנה צורה כללית מכיל מיקום, הצבעה למבנה פרמטרים ספציפי לצורה (m\_param) ומצביעים לפונקציות:

```
struct shape {
    double m_x,m_y;
    void *m_param;
    FunD area;
    FunV print;
    void (*move)(Shape*, double , double);
};
```

הצהרה על מבנה פרמטרים של עיגול המכיל רק רדיוס:

```
typedef struct {
    double m_rad;
} CircleParam;
```

ומבנה פרמטרים של מלבן המכיל ממדים:

```
typedef struct {
    double m_width, m_height;
} RectangleParam;
```

הפונקציה הבאה בונה Shape חדש לפי ספציפיקציות מוכתבות. הפונקציה שבונה עיגול והפונקציה שבונה מלבן יקראו לפונקציה זו על מנת לבנות צורה ספציפית:

```
Shape* createShape(double x, double y, void *param, FunD area, FunV print) {
    Shape *p = (Shape*)malloc(sizeof(Shape));
    p->m_x = x;
    p->m_y = y;
    p->m_param = param;
    p->area = area;
    p->print = print;
    p->move = moveShape;
    return p;
}
```

שימו לב שמכיוון שבדוגמה זו, הזזת כל הצורות מתבצעת באותו אופן, אפשר לכתוב את קוד ההזזה ברמת הצורה הכללית.

```
void moveShape(Shape *shape, double dx, double dy) {
    shape->m_x += dx;
    shape->m_y += dy;
}
```

גם קוד השחרור יכל במקרה זה להתבצע ברמת הצורה הכללית:

```
void freeShape(Shape* shape) {
    free(shape->m_param);
    free(shape);
}
```

הפונקציה הבאה יוצרת עיגול. ראשית נוצר מבנה הפרמטרים ואז מופעלת הפונקציה שכתבנו קודם עם הפרמטרים הספציפיים לעיגול:

```
Shape * createCircle(double x, double y, double rad) {
    CircleParam *cp = (CircleParam*)malloc(sizeof(CircleParam));
    cp->m_rad = rad;
    return createShape(x,y,cp,circleArea,circlePrint);
}
```

פונקציות השטח ופונקציית ההדפסה של העיגול. פונקציות אלה יושמו בשדות area ו print של מבנה ה Shape שיבנה עבור העיגול. שימו לב להמרה שנעשית כאן. כאשר הפונקציה תופעל מבנה הפרמטרים שעליו יצביע m\_param יהיה אכן מבנה פרמטרים של עיגול:

```
double circleArea(Shape *shape) {
    double r = ((CircleParam*)(shape->m_param))->m_rad;
    return M_PI*r*r;
}
```

```
void circlePrint(Shape *shape) {
    double r = ((CircleParam*)(shape->m_param))->m_rad;
    printf("<Circle: (%g,%g) R = %g>\n",shape->m_x,shape->m_y,r);
}
```

קוד מקביל עבור מלבן:

```
Shape *createRectangle(double x, double y, double width, double height) {
    RectangleParam *rp = (RectangleParam*)malloc(sizeof(RectangleParam));
    rp->m_width = width;
    rp->m_height = height;
    return createShape(x,y,rp,rectangleArea,rectanglePrint);
}
```

```
double rectangleArea(Shape *shape) {
    RectangleParam *rp = (RectangleParam*)shape->m_param;
    return rp->m_width * rp->m_height;
}
```

```
void rectanglePrint(Shape *shape) {
    RectangleParam *rp = (RectangleParam*)shape->m_param;
    printf("<Rectangle: (%g,%g) Width = %g, Height = %g>\n",
        shape->m_x,shape->m_y,rp->m_width,rp->m_height);
}
```

**הערה:** שימו לב שלדרך מימוש זו עלות מסוימת בזיכרון. עבור כל צורה שניצור אנו דורשים שלושה מצביעים לפונקציות. קיימת אפשרות לשנות קצת את המימוש ולהקטין את צריכת הזיכרון. במקום לשמור מצביע עבור כל פונקציה של הצורה אפשר לשמור מצביע יחיד למערך של מצביעים לפונקציות. למימוש זה יש עלות בזמן ריצה.