המדריך לשפת C

Authorized translation from the English language edition of :
Learning C
By Peter Aitken
Published by Sams Publishing, USA
Copyright © by Sams Publishing

מהדורה ראשונה בעברית - 1996
הדפסה: 3 2 1

הוצאת פוקוס-מחשבים בע"מ
ת"ד 863 ר"ג, 52108
טל' 03-677-38-98

כל הזכויות למהדורה העברית שמורות להוצאת פוקוס-מחשבים בע"מ
© 1996

אין להעתיק, לשכפל, לצלם ולהקליט ספר זה, ו/או את הדיסקט המצורף לו, או קטעים מהם בשום צורה ובשום אמצעי אלקטרוני, אופטי, או מכני לכל מטרה שהיא, ללא אישור בכתב מהוצאת פוקוס-מחשבים בע"מ.

תרגום: שאול שילו
עימוד במחשב: ענת קדם בן-צבי
עטיפה: בלום עיצוב, הפקה וכתיבה
לוחות והדפסה: טופSפרינט בע"מ
כריכה: כריכיית אהרון בע"מ

הודעת זכויות

הספרים מוגנים בזכויות יוצרים. הוצאת פוקוס-מחשבים מרשה את הורדתם כשירות לציבור, לשימוש אישי של המוריד בלבד. אין להפיץ או להעביר לאחרים, בין בתמורה ובין ללא תמורה, חלק כלשהו מן החומר המורד. אין להפיץ תדפיס או העתק של החומר בכל צורה שהיא ללא רשות מפורשת, בכתב ומראש מהוצאת פוקוס-מחשבים.

תוכן העניינים

הקדמה 21

פרק 1: צעדים ראשונים 23

היסטוריה קצרה של שפת C 23
מדוע להשתמש ב-C 24
מחזור הפיתוח של תוכנית 24
התקנת Zortech C 26
התקנה בדיסק הקשיח 26
התוכנית הראשונה שלך ב-C 28
הכנסה והידור של HELLO.C 28
סיכום הפרק 32
בוחן 32

פרק 2: השימוש ב-Zortech C 33

המרכיבים של Zortech C 33
Zortech Workbench 34
הפעלת Zortech Workbench 34
המסך של Zortech Workbench 35
שימוש בעכבר 36
עריכת קוד המקור שלך 37
סמן העריכה 37
הכנסת טקסט 38
מחיקת טקסט 38
תווים בלתי נראים 39

שמירת העבודה שלך 39
יציאה מ-Zortech Workbench..... 40
מערכת התפריטים של Zortech Workbench..... 41
בחירה בפקודות מתוך התפריט הראשי..... 42
מבנה תפריט המשנה..... 43
בחירה בפקודה מתוך תפריט משנה..... 44
שימוש בתיבות דו-שיח 44
עבודה עם קבצים..... 46
עבודה עם בלוקים..... 46
סימון בלוק 47
שימוש במצב בלוק..... 48
הלוח..... 49
פעולות בבלוקים 49
חיפוש טקסט..... 50
מציאת טקסט..... 50
חיפוש והחלפה 51
הידור וקישור של התוכנית שלך 53
הידור וקישור מתוך שורת הפקודה 53
שימוש בשירות הבקרה של המהדר של Zortech 54
שגיאות הידור..... 54
הודעות שגיאה של המקשר..... 57
הידור וקישור מתוך Zortech Workbench..... 57
הרצת התוכנית שלך מתוך Zortech Workbench..... 60
קביעה של תצוגה מונוכרומטית..... 61
סיכום הפרק..... 62
בוחן..... 62

פרק 3: המרכיבים של תוכנית בשפת C 63

תוכנית קצרה ב-C..... 63
מרכיבי התוכנית..... 64
הפונקציה main( ) (שורות 5 - 16)..... 65
ההוראה #include (שורה 2) 65
הגדרת משתנים (שורה 3)..... 65
אבטיפוס של פונקציה (שורה 4)..... 65
משפטי תוכנית (שורות 8, 9, 11, 12, 14, 15, 20)..... 65
printf( )..... 66
scanf( ) 66
c=product(a,b);..... 66
return(x*y);..... 66

הגדרת פונקציה (שורות 17 - 21) 66
הערות בתוכנית (שורות 1, 7, 10, 13, 18) 67
סוגריים מסולסלים (שורות 6, 16, 19, 21) 67
הרצת התוכנית 67
הערה בנוגע לדיוק 68
סיכום הפרק 68
בוחן 68

פרק 4: משתנים מספריים וקבועים 69

זכרון המחשב 69
משתנים 70
שמות משתנים 71
טיפוסי משתנים 71
הצהרות על משתנים 73
מילת המפתח typedef 74
אתחול של משתנים מספריים 74
קבועים 75
קבועים מפורשים 75
קבועים סימבוליים 76
סיכום הפרק 80
בוחן 80

פרק 5: משפטים, ביטויים ואופרטורים 81

משפטים 81
משפטים ורווח לבן 82
משפטים מורכבים 83
ביטויים 83
ביטויים פשוטים 83
ביטויים מורכבים 84
אופרטורים 85
אופרטור ההצבה 85
אופרטורים מתמטיים 85
האופרטורים המתמטיים האונריים 85
האופרטורים המתמטיים הבינריים 88
קדימות של אופרטור וסוגריים 89
סדר התחשיב של תת-ביטוי 91
אופרטורים של יחס 91
משפט if 92

תחשיב של ביטויי יחס 95
קדימות של אופרטורים של יחס 97
אופרטורים לוגיים 98
עוד על ערכי אמת/שקר 99
קדימות של אופרטורים לוגיים 100
אופרטורי הצבה מורכבים 102
אופרטור התנאי 103
האופרטור פסיק 104
סיכום הפרק 104
בוחן 105

פרק 6: פונקציות: היסודות 107

מהי פונקציה? 107
הגדרת פונקציה 107
הדגמה של פונקציה 108
כיצד פונקציה פועלת 109
פונקציות ותכנות מובנה 110
היתרונות של תכנות מובנה 110
התכנון של תוכנית מובנית 111
גישת מעלה-מטה 112
כתיבת פונקציה 112
כותרת הפונקציה 113
טיפוס הערך המוחזר על ידי הפונקציה 113
שם הפונקציה 113
רשימת הפרמטרים 113
גוף הפונקציה 115
משתנים מקומיים 115
משפטי פונקציה 117
החזרה של ערך 118
האבטיפוס של פונקציה 119
העברת ארגומנטים לפונקציה 120
קריאה לפונקציות 121
רקורסיה 121
היכן למקם את הפונקציות? 122
סיכום הפרק 123
בוחן 124

פרק 7: יסודות בבקרת תוכנית ובקלט/פלט 125

פיקוח על ביצוע התוכנית 125
מערכים: היסודות..... 126
משפט for 126
משפט while 130
לולאת do...while..... 132
לולאות מקוננות..... 133
הצגת מידע על המסך 134
הפונקציה printf( )..... 134
הצגת הודעות באמצעות puts( )..... 137
קליטת נתונים מספריים באמצעות scanf( ) 138
סיכום הפרק..... 139
בוחן..... 140

פרק 8: מערכים מספריים 141

מהו מערך? 141
מערכים חד-מימדיים..... 142
מערכים רב-מימדיים 144
מתן שמות למערכים והצהרה עליהם..... 145
אתחול מערכים..... 146
הגודל המקסימלי של מערך 149
סיכום הפרק..... 151
בוחן..... 151

פרק 9: מצביעים 153

מהו מצביע?..... 153
זכרון המחשב שלך 153
יצירת מצביע..... 154
מצביעים ומשתנים פשוטים..... 155
הצהרה על מצביעים..... 155
אתחול של מצביעים..... 156
שימוש במצביעים..... 156
מצביעים וטיפוסי משתנים 158

מצביעים ומערכים 159
שם המערך כמצביע 160
אחסון איברים של מערך 160
חשבון מצביעים 163
הוספה לערכם של מצביעים 163
מניפולציות אחרות במצביעים 165
אזהרה בנוגע למצביעים 166
כתיב אינדקסים של מערך ומצביעים 167
העברת מערכים לפונקציות 167
סיכום הפרק 170
בוחן 171

פרק 10: תווים ומחרוזות 173

טיפוס הנתונים char 173
שימוש במשתנים תוויים 174
שימוש במחרוזות 176
מערכים של תווים 176
אתחול מערכים של תווים 177
מחרוזות ומצביעים 178
מחרוזות ללא מערכים 178
הקצאת שטח למחרוזות בזמן ההידור 178
הפונקציה ( )malloc 179
הצגת מחרוזות ותווים 181
הפונקציה ( )puts 182
הפונקציה ( )printf 182
קריאת מחרוזות מן המקלדת 183
קליטת מחרוזות באמצעות הפונקציה ( )gets 183
קליטת מחרוזות באמצעות הפונקציה ( )scanf 186
סיכום הפרק 189
בוחן 189

פרק 11: מבנים 191

מבנים פשוטים 191
הגדרת מבנים והצהרה עליהם 192
גישה לשדות במבנה 193
מבנים מורכבים יותר 193
מבנים המכילים מבנים 193
מבנים המכילים מערכים 196
מערכים של מבנים 198

אתחול של מבנים 201
מבנים ומצביעים 203
מצביעים כשדות במבנה 204
מצביעים למבנים 206
מצביעים ומערכים של מבנים 208
העברת מבנים כארגומנטים לפונקציות 211
רשימות מקושרות 213
הארגון של רשימה מקושרת 213
הפונקציה malloc( ) 215
מימוש רשימה מקושרת 216
typedef ומבנים 216
סיכום הפרק 217
בוחן 217

פרק 12: תחום הכרה של משתנה 219

מהו תחום הכרה 219
הדגמה של תחום ההכרה 220
מדוע תחום ההכרה חשוב? 221
משתנים חיצוניים 222
תחום ההכרה של משתנה חיצוני 222
מתי להשתמש במשתנים חיצוניים 222
מילת המפתח extern 223
משתנים מקומיים 224
משתנים סטטיים לעומת משתנים אוטומטיים 224
תחום ההכרה של הפרמטרים של הפונקציה 227
משתנים חיצוניים סטטיים 227
משתני אוגר 227
משתנים מקומיים והפונקציה main( ) 228
באיזה סוג של אחסון עליך להשתמש? 228
משתנים מקומיים ובלוקים 229
סיכום הפרק 231
בוחן 231

פרק 13: מצביעים: נושאים מתקדמים 233

מצביעים למצביעים 233
מצביעים ומערכים רב-מימדיים 235
מערכים של מצביעים 242
מחרוזות ומצביעים – חזרה 243
מערך של מצביעים ל-char 243
דוגמה 246

מצביעים לפונקציות 251
הצהרה על מצביע לפונקציה 251
אתחול ושימוש במצביע לפונקציה 252
סיכום הפרק 260
בוחן 260

פרק 14: בקרת תוכנית: נושאים מתקדמים 263

משפט goto 263
סיום מוקדם של לולאות 265
משפט break 266
משפט continue 267
לולאות אינסופיות 268
משפט switch 271
יציאה מן התוכנית 278
הפונקציה ( )exit 278
הפונקציה ( )atexit 278
ביצוע של פקודות DOS מתוך התוכנית 281
סיכום הפרק 282
בוחן 283

פרק 15: קלט ופלט של תוכנית: מסך, מדפסת, ומקלדת 285

streams ו-C 285
קלט/פלט של תוכנית 286
מהו stream? 286
streams של טקסט לעומת streams בינריים 287
streams שהוגדרו מראש 287
פונקציות של stream ב-C 288
דוגמה 289
קבלת קלט מן המקלדת 289
קלט של תווים 289
getchar( ) 290
getch( ) 291
getche( ) 293
getc( ) ו-fgetc( ) 293
קלט של תווים: מקשים מיוחדים 293
ביטול קליטה של תו באמצעות ungetc( ) 300
קלט של שורות 301
gets( ) 301
fgets( ) 301

קלט לפי תבנית 302
הארגומנטים של scanf( ) 303
טיפול בתווים עודפים 305
דוגמאות ל-scanf( ) 307
פלט למסך 309
פלט של תווים בודדים: putchar( ), putc( ), ו-fputc( ) 309
putchar( ) 309
putc( ) ו-fputc( ) 310
פלט של מחרוזת: puts( ) ו-fputs( ) 311
פלט לפי תבנית: printf( ) ו-fprintf( ) 312
ניתוב מחדש של קלט ופלט 317
מתי להשתמש ב-fprintf( )? 319
שימוש ב-stderr 319
פלט למדפסת 320
סיכום הפרק 320
בוחן 321
תרגילים 321

פרק 16: שימוש בקובצי דיסק 323

streams וקובצי דיסק 323
טיפוסים של קובצי דיסק 324
שמות קבצים 324
פתיחת קובץ לשימוש 325
כתיבה וקריאה של נתוני קובץ 328
קלט ופלט לפי תבנית בקבצים 328
פלט לפי תבנית לקובץ 328
קלט לפי תבנית מקובץ 330
קלט ופלט של תווים 332
קלט של תווים 332
הפונקציות getc( ) ו-fgetc( ) 332
הפונקציה fgets( ) 332
פלט של תווים 333
הפונקציה putc( ) 333
הפונקציה fputs( ) 333
קלט ופלט ישירים בקבצים 334
fwrite( ) 334
fread( ) 335
מאגרים בעבודה עם קבצים: סגירה וריקון של קבצים 338

שימוש בסמן המקום בקובץ 339
ftell( ) ו-rewind( ) 340
fseek( ) 342
איתור EOF (סוף קובץ) 344
פונקציות לניהול קבצים 346
מחיקת קובץ 346
שינוי שם של קובץ 347
העתקת קובץ 348
שימוש בקבצים זמניים 350
סיכום הפרק 351
בוחן 352
תרגילים 352

פרק 17: מניפולציה במחרוזות 353

אורך מחרוזת ואחסון 353
העתקת מחרוזות 354
strcpy( ) 355
strncpy( ) 356
strdup( ) 357
שרשור מחרוזות 358
strcat( ) 358
strncat( ) 359
השוואת מחרוזות 360
strcmp( ) 360
strcmpl( ) 362
strncmp( ) 362
חיפוש מחרוזות 363
strchr( ) 363
strrchr( ) 364
strcspn( ) 365
strspn( ) 366
strpbrk( ) 367
strstr( ) 367
המרות של מחרוזת 368
פונקציות מחרוזת שונות 369
strrev( ) 369
strset( ) ו-strnset( ) 370

המרות של מחרוזת למספר 370
atoi( ) 370
atol( ) 371
atof( ) 371
פונקציות לבדיקת תווים 372
סיכום הפרק 375
בוחן 376
תרגילים 376

פרק 18: פונקציות: נושאים מתקדמים 377

העברת מצביעים לפונקציות 377
מצביעים מטיפוס void 381
פונקציות עם מספר משתנה של ארגומנטים 384
פונקציות המחזירות מצביע 387
סיכום הפרק 389
בוחן 389
תרגילים 389

פרק 19: בחינה של ספריית הפונקציות 391

פונקציות מתמטיות 391
פונקציות טריגונומטריות 392
פונקציות מעריכיות ולוגריתמיות 392
פונקציות היפרבוליות 392
פונקציות מתמטיות אחרות 393
טיפול בזמן 393
ייצוג של זמן 393
פונקציות הזמן 394
קבלת הזמן הנוכחי 394
המרה בין ייצוגי זמן 395
הצגת זמנים 395
חישוב הפרשי זמן 397
שימוש בפונקציות הזמן 397
פונקציות לטיפול בשגיאות 399
assert( ) 399
errno.h 401
perror( ) 402

חיפוש ומיון 403
חיפוש באמצעות bsearch( ) 403
מיון באמצעות qsort( ) 405
חיפוש ומיון – שתי הדגמות 405
סיכום הפרק 408
בוחן 409
תרגיל 409

פרק 20: "שאר ירקות" 411

המרות של טיפוסים 411
המרות טיפוס אוטומטיות 411
קידום של טיפוס בביטויים 412
המרה באמצעות הצבה 412
המרות מפורשות תוך שימוש בהסבות טיפוס 413
הסבה של ביטויים אריתמטיים 413
הסבה של מצביעים 414
הקצאת שטח אחסון בזיכרון 414
malloc( ) 415
calloc( ) 415
realloc( ) 416
free( ) 418
שימוש בארגומנטים של שורת הפקודה 419
פעולה בסיביות 421
אופרטורים של הזזה 421
אופרטורים לוגיים בפעולה בסיביות 422
אופרטור המשלים 424
שדות של סיביות במבנים 424
סיכום הפרק 426
בוחן 426
תרגילים 426

פרק 21: שימוש מתקדם ב-Zortech C++ 427

תכנות עם קובצי קוד מרובים 427
יתרונות של תכנות מודולרי 427
טכניקות של תכנות מודולרי 428
מרכיבי המודול 429
משתנים חיצוניים ותכנות מודולרי 430

שימוש ב-Zortech Worbench בעבודה עם מודולים מרובים 431
עריכת קבצים מרובים 432
הידור של מודולים מרובים מתוך Zortech Workbench 433
שימוש בקובצי OBJ 434
הקדם-מעבד של C 434
#define 435
מאקרו(ים) פשוטים של החלפה באמצעות #define 435
מאקרו(ים) כפונקציות באמצעות #define 436
מאקרו(ים) לעומת פונקציות 440
התבוננות בהתפתחות מאקרו 440
#include 440
#if, #elif, #else, ו-#endif 441
#undef 444
סיכום הפרק 444
בוחן 445
תרגילים 445

נספח א' מילים שמורות 447

נספח ב' אבטיפוסים של פונקציות וקובצי כותרת 449

נספח ג' לוח תווי ASCII 453

נספח ד' כתיב בינרי והקסדצימלי 457

נספח ה' קדימות של אופרטורים ב-C 461

נספח ו' קודי חילוף 463

נספח ז' תשובות לשאלות בבחנים 465

תשובות לשאלות בבוחן שבפרק 1 465
תשובות לשאלות בבוחן שבפרק 2 465
תשובות לשאלות בבוחן שבפרק 3 466
תשובות לשאלות בבוחן שבפרק 4 466
תשובות לשאלות בבוחן שבפרק 5 467

תשובות לשאלות בבוחן שבפרק 6 467
תשובות לשאלות בבוחן שבפרק 7 468
תשובות לשאלות בבוחן שבפרק 8 470
תשובות לשאלות בבוחן שבפרק 9 471
תשובות לשאלות בבוחן שבפרק 10 472
תשובות לשאלות בבוחן שבפרק 11 472
תשובות לשאלות בבוחן שבפרק 12 473
תשובות לשאלות בבוחן שבפרק 13 473
תשובות לשאלות בבוחן שבפרק 14 474
תשובות לשאלות בבוחן שבפרק 15 474
תשובות לשאלות בבוחן שבפרק 16 475
תשובות לשאלות בבוחן שבפרק 17 475
תשובות לשאלות בבוחן שבפרק 18 475
תשובות לשאלות בבוחן שבפרק 19 476
תשובות לשאלות בבוחן שבפרק 20 476
תשובות לשאלות בבוחן שבפרק 21 476

נספח ח' מדריך מקשים/פקודות 477

מפתח העניינים 479

הקדמה

אם אתה קורא את השורות האלה, קרוב לוודאי שאתה מעוניין ללמוד את שפת התכנות C. בחירתך מצוינת! קיימת הסכמה כללית ש-**C** היא רבת העוצמה והגמישה ביותר מבין שפות התכנות השונות הזמינות; והיא בהחלט השפה שתוכניתנים מקצועיים משתמשים בה בשכיחות הרבה ביותר. בשל סיבות שאני מפרט בפרק 1, אינך יכול לטעות בבחירה ב-**C** כשפת התכנות שלך.

אומנם יהיה זה מעט בלתי-צנוע מצידי לומר זאת, אך אני חושב שבחירתך בספר זה ככלי ללימוד **C** היא החלטה נבונה. סיבה אחת לכך היא שבמחיר סביר ביותר אתה מקבל לא רק מדריך למידה אלא גם חבילת פיתוח תפקודית מלאה ב-**C** של **Zortech**, אחד השמות המכובדים ביותר בתחום שפה זו. התקליטון המגיע עם ספר זה מכיל את כל הכלים שאתה זקוק להם לשם כתיבה, הידור, והרצה של תוכניות קטנות עד בינוניות ב-**C**. התקליטון מכיל גם את כל תוכניות ההדגמה המוצגות בספר. הוראות ההתקנה של התקליטון ניתנות בפרק 1.

נכון הוא שחבילה זו אינה כוללת את כל המאפיינים המצויים בחבילות פיתוח מסחריות ב-**C**, של **Zortech** או של יצרנים אחרים – אולם חבילה זו גם אינה עולה מאות דולרים! חשוב יותר, אינך זקוק לכל המאפיינים הנוספים כדי ללמוד את שפת **C**. לאמיתו של דבר, קרוב לוודאי שהם פשוט יעמדו בדרכך. מאוחר יותר, עם שהמיומנויות שלך בתכנות והצרכים שלך יגדלו, תוכל לעבור להשתמש בחבילה מקיפה יותר.

ספר זה מציג את **C** בסדר אשר לפי דעתי הוא ההגיוני והנוח ביותר ללמידה. אני ממליץ שתתחיל בתחילת הספר ותעבור על הפרקים לפי הסדר, מכיוון שפרקים מאוחרים יותר מסתמכים על חומר שהוצג בפרקים קודמים. לא מונח שיש לך ניסיון קודם בתכנות, אם כי ניסיון מסוים בשפה אחרת (כמו **BASIC**), עשוי להחיש את תהליך הלמידה.

בסוף כל פרק תמצא בוחן קצר. שאלות הבוחן מוצעות כדרך שבה תוכל לבדוק את ידיעתך את החומר שהוצג בפרק. שאלות הבוחן אינן ממצות, אלא דוגמות כמה נושאים חשובים מתוך הפרק. אני מציע שתבדוק את עצמך באמצעות שאלות אלו; תשובות ניתנות לקראת סוף הספר, בנספח ז'. אם אתה לא חש ביטחון בתשובותיך לכמה מן השאלות, טוב יהיה אם תחזור על הפרק שאליו הן מתייחסות.

פרקים מאוחרים כוללים גם תרגילים. כל תרגיל מציג בעיה תכנותית קטנה שאתה צריך לפתור על ידי כתיבה של פונקציה או תוכנית ב-**C**. לא מוצעות תשובות לתרגילים, מאחר שקיימות בדרך כלל מספר דרכים לפתירה של בעיה תכנותית כלשהי (וחוץ מזה, אתה עשוי להציץ!). אני ממליץ ביותר ש"תעבוד על" כל תרגיל, מאחר שהניסיון בתכנות שתרכוש כתוצאה מכך יהיה מכשיר לימודי מצוין.

פרק 6

פונקציות: היסודות

פונקציות הן עניין מרכזי בתכנות ב-C ובפילוסופיה של עיצוב תוכנית ב-C. כבר הוצגו בפניך כמה פונקציות ספריה של **C**, שהן פונקציות שלמות ב-**C** המגיעות ביחד עם המהדר שלך. פרק זה דן בפונקציות **המוגדרות על ידי המשתמש (user-defined)**, אשר, כפי שמשתמע משמן, הן פונקציות שאתה, התוכניתן, יוצר. כאן תלמד

- מהי פונקציה, ומה הם חלקיה.
- מהם היתרונות של תכנות מובנה עם פונקציות.
- כיצד ליצור פונקציה.
- כיצד להצהיר על משתנים מקומיים בתוך פונקציה.
- כיצד להחזיר ערך מפונקציה אל התוכנית.
- כיצד להעביר ארגומנטים לפונקציה.

מהי פונקציה?

פרק זה מתמודד עם השאלה "מהי פונקציה?" בשתי דרכים. ראשית הוא **מספר** לך מהי פונקציה, ולאחר מכן, הוא **מדגים** בפניך מהי.

הגדרת פונקציה

כדי להתחיל, הנה הגדרה של המונח **פונקציה (function)**: קטע עצמאי של קוד ב-**C** שיש לו שם, המבצע משימה ספציפית ואפשר (קיימת האפשרות) שהוא מחזיר ערך לתוכנית הקוראת. הבנת? יפה. התבונן כעת בחלקיה של הגדרה זו.

- לפונקציה **ניתן שם**. לכל פונקציה יש שם ייחודי. על ידי שימוש בשם הפונקציה בחלקים אחרים של התוכנית (דבר הידוע **כקריאה – calling** – לפונקציה) אתה יכול להוציא אל הפועל את המשפטים הכלולים בפונקציה. פונקציה יכולה להיקרא מתוך פונקציה אחרת.
- הפונקציה היא **בלתי-תלויה (independent)**. פירוש הדבר שהפונקציה יכולה לבצע את המשימה שלה בלי הפרעה לביצוע זה מצד חלקים אחרים בתוכנית, או הפרעה מצד הפונקציה לחלקים אלה.
- פונקציה מבצעת **משימה ספציפית**. ובכן, זהו חלק קל בהגדרה (אתה כבר יודע בוודאי מהי משימה!) זו מטלה מובחנת שהתוכנית שלך חייבת לבצע כחלק מפעולתה הכוללת, כמו שליחה של שורת טקסט למדפסת שלך, מיון מערך לפי סדר מספרי, או חישוב השורש השלישי של מספר.
- פונקציה יכולה **להחזיר ערך (return a value)** אל התוכנית הקוראת. כאשר התוכנית שלך קוראת לפונקציה, המשפטים הנכללים בפונקציה מבוצעים. משפטים אלה יכולים, אם זה רצונך, להעביר מידע חזרה אל התוכנית הקוראת.

החלק שבו "סיפרנו" לך מהי פונקציה מסתיים כאן. זכור את ההגדרה הקודמת בשעה שתעבור לקטע הבא.

הדגמה של פונקציה

התוכנית בתדפיס 6.1 משתמשת בפונקציה. זו תוכנית פשוטה וזו פונקציה פשוטה, אך הן משרתות את המטרה שהיא להקנות את המושג. מספרי השורות אינם חלק של התוכנית, כמובן. במספרים אלה נעשה שימוש לשם זיהוי של שורות בתוכנית בהסבר הניתן לאחר התדפיס.

תדפיס 6.1. פונקציה פשוטה.

```
 1:  /* Demonstrates a simple function */
 2:  #include <stdio.h>
 3:  long cube(long x);
 4:  long input, answer;
 5:  main()
     {
 6:     printf("Enter an integer value: ");
 7:     scanf("%d", &input);
 8:     answer = cube(input);
 9:     /* Note: %ld is the conversion specifier for */
10:     /* a long integer */
11:     printf("\n\nThe cube of %ld is %ld.", input, answer);
     }
12:  long cube(long x)
```

```
13:  {
14:      long x_cubed;
15:      x_cubed = x * x * x;
16:      return x_cubed;
17:  }
```

התבונן כעת בחלקים של תוכנית זו אשר יש להם קשר לפונקציה. כל שאתה צריך בקטע זה הוא להכיר מרכיבים של פונקציה, כך שאל תצפה שהדברים יוסברו באופן מלא.

קיימות שתי שורות במסגרת הגוף המרכזי של התוכנית שיש קשר בינן לבין הפונקציה:

- שורה 3 היא **האבטיפוס של הפונקציה (function prototype)** היכול להשתרע על פני יותר משורה אחת, והוא מספק למהדר מידע מסוים אודות הפונקציה: שמה (שהוא **cube**), טיפוס הנתונים שהיא מחזירה, והארגומנטים שלה.
- שורה 8 קוראת לפונקציה **cube**, ומעבירה אליה את המשתנה **input** כארגומנט. הערך המוחזר מן הפונקציה מוצב לאחר מכן במשתנה **answer**.

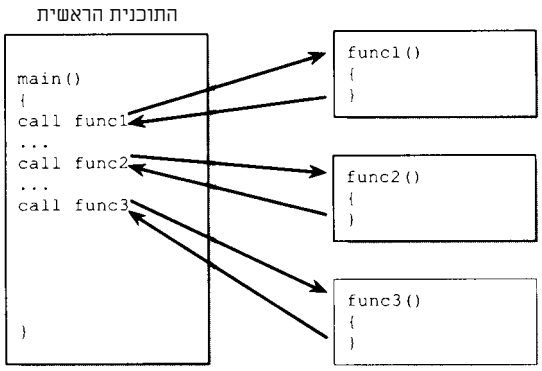
שורות אחרות בתוכנית מרכיבות את הפונקציה **cube**. הפונקציה **cube** עצמה – **הגדרת הפונקציה (function definition)** – משתרעת על פני השורות 12 - 17. להגדרת הפונקציה יש מספר חלקים אשר יתוארו להלן.

- שורה 12 היא **כותרת הפונקציה (function header)**, שהיא בתחילת הפונקציה ונותנת שם לפונקציה (במקרה זה השם הוא **cube**), ונותנת לפונקציה גם את טיפוס הנתונים שהיא מחזירה ומתארת את הארגומנטים שלה. שים לב לכך שכותרת הפונקציה זהה לאבטיפוס של הפונקציה (אך בלי הנקודה-פסיק).
- גוף הפונקציה, שורות 13 - 17, מוקף בסוגריים מסולסלים. הגוף מכיל משפטים, כמו אלה המוצגים בשורה 15. אלה המשפטים המבוצעים בכל פעם כשיש קריאה לפונקציה.
- שורה 14 היא הצהרה על משתנה מקומי. **מקומי (local)** פירושו שמוצהר על המשתנה במסגרת הפונקציה.
- שורה 16 היא משפט **החזרה**, המאותת על סוף הפונקציה. משפט **החזרה** גם מעביר ערך חזרה לתוכנית הקוראת. במקרה זה מוחזר הערך של המשתנה **x_cubed**.

כיצד פונקציה פועלת

תוכנית ב-**C** משתמשת בפונקציות באופן הבא. המשפטים בפונקציה אינם מוצאים אל הפועל עד שהפונקציה נקראת על ידי חלק אחר של התוכנית. כאשר פונקציה נקראת, התוכנית יכולה לשלוח אליה מידע בצורה של ארגומנט אחד או יותר. **ארגומנט (argument)** אינו יותר מאשר נתוני תוכנית הנחוצים לפונקציה כדי שתוכל לבצע את משימתה. לאחר מכן מתבצעים המשפטים במסגרת הפונקציה, כשכל אחד מהם מבצע את מה שהוא תוכנן לבצע. לאחר שהמשפטים בפונקציה גמרו להתבצע, הביצוע חוזר אל אותה נקודה בתוכנית שקראה לפונקציה. פונקציות יכולות לשלוח מידע חזרה אל התוכנית בצורה של **ערך מוחזר (return value)**.

דבר זה מודגם באיור 6.1, המציג תוכנית שיש בה שלוש פונקציות, כשכל אחת מהן נקראת פעם אחת. בכל פעם שפונקציה נקראת, הביצוע עובר לאותה פונקציה. כאשר הפונקציה מסתיימת, הביצוע עובר חזרה אל המקום שממנו הפונקציה נקראה. כמובן, פונקציה נתונה יכולה להיקרא מספר כלשהו של פעמים כפי שנחוץ, ואין צורך לקרוא לפונקציות בסדר מסוים כלשהו.



איור 6.1. כאשר תוכנית קוראת לפונקציה, הביצוע עובר לפונקציה ולאחר מכן חזרה לתוכנית הקוראת.

בנקודה זו אמור להיות לך מושג ברור באשר למהותה של פונקציה. בהמשך תוסבר החשיבות של פונקציות, ולאחר מכן תלמד כיצד ליצור פונקציות ולהשתמש בפונקציות שלך.

פונקציות ותכנות מובנה

כאשר אתה משתמש בפונקציות בתוכניות שלך ב-C, אתה ממש **תכנות מובנה (structured programming)**, שהוא העיקרון בעיצוב תוכנית אשר במסגרתו משימות שונות בתוכנית מבוצעות על ידי קטעים עצמאיים של קוד תוכנית. החלק האחרון המדבר אודות "קטעים עצמאיים של קוד תוכנית" מצלצל ממש כמו חלק מתוך הגדרת הפונקציות אשר ניתנה קודם לכן, האין זאת? זו הסיבה לכך שפונקציות ותכנות מובנה קשורים באופן כה הדוק.

היתרונות של תכנות מובנה

מדוע תכנות מובנה הוא כה מוצלח?

- קל יותר לכתוב תוכנית מובנית מכיוון שהיא מפרקת בעיות תכנותיות מורכבות למספר משימות קטנות יותר ופשוטות יותר. כל אחת מן המשימות הללו מבוצעת על ידי פונקציה שהקוד שלה והמשתנים שלה מבודדים מיתר התוכנית. אתה יכול להתקדם מהר יותר כשאתה מטפל במשימה פשוטה יחסית אחת בכל פעם.
- קל יותר לנפות שגיאות מתוכנית מובנית. אם בתוכנית שלך יש **תקלה – bug** – (כלומר, משהו הגורם לתוכנית לפעול שלא כהלכה), עיצוב מובנה הופך לקל יותר את בידוד הבעיה ושיוכה לקטע ספציפי של קוד (כלומר, לפונקציה ספציפית).

יתרון של תכנות מובנה המתקשר לכך הוא שתכנות מובנה יכול לחסוך המון זמן אם אתה כותב תוכניות רבות. אם אתה כותב פונקציה כדי לבצע משימה מסוימת בתוכנית אחת, אתה יכול בקלות ובמהירות להשתמש בה בתוכנית אחרת הצריכה שפונקציה זו תתבצע. גם אם התוכנית החדשה מצריכה את הביצוע של משימה שונה במקצת, פעמים רבות תמצא שקל לך יותר לשנות פונקציה שיצרת קודם לכן מאשר לכתוב פונקציה חדשה מן ההתחלה.

התכנון של תוכנית מובנית

אם אתה מתכוון לכתוב תוכנית המשתמשת בגישה המובנית, עליך לעסוק תחילה בתכנון. על התכנון להיעשות עוד לפני שאתה כותב שורת קוד אחת, ובדרך כלל יספיקו לכך נייר ועיפרון. אתה צריך לקבוע את המשימות הספציפיות שהתוכנית שלך מבצעת. אתה מתחיל, כמובן, ברעיון כללי באשר למטרה של התוכנית. נניח, למשל, שאתה מתכנן תוכנית לניהול רשימת השמות והכתובות שלך. ובכן, מה אתה רוצה שהתוכנית תבצע? הנה כמה דברים מובנים מאליהם:

- להכניס שמות וכתובות חדשים.
- לשנות ערכים קיימים.
- למיין ערכים לפי שם המשפחה.
- להדפיס תוויות דיוור.

באמצעות רשימה זו, חילקת את התוכנית לארבע משימות ראשיות אשר כל אחת מהן יכולה להתבצע על ידי פונקציה. כעת עליך להתקדם מעט יותר, בכך שתחלק את המשימות הללו לתת-משימות. לדוגמה, המשימה "הכנס שמות וכתובות חדשים" יכולה להיות מחולקת בחלוקת משנה לתת-משימות הבאות:

- קרא את רשימת הכתובות הקיימות מתוך הדיסק.
- בקש מהמשתמש להכניס ערך חדש אחד או יותר.
- הוסף את הנתונים החדשים לרשימה.
- שמור את הרשימה המעודכנת בדיסק.

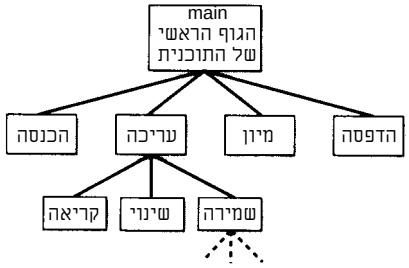
באופן דומה, המשימה "שנה ערכים קיימים" יכולה להיות מחולקת בחלוקת משנה כדלקמן:

- קרא את רשימת הכתובות הקיימות מתוך הדיסק.
- שנה ערך קיים אחד או יותר.
- שמור את הרשימה המעודכנת בדיסק.

ייתכן ששמת לב לכך שלשתי הרשימות הללו יש שתי תת-משימות במשותף – אלו המטפלות בקריאה מן הדיסק ובשמירה בדיסק. אתה יכול לכתוב פונקציה אחת "קרא את רשימת הכתובות הקיימות מן הדיסק"; ופונקציה זו יכולה להיקרא הן על ידי הפונקציה "הכנס שמות וכתובות חדשים" והן על ידי הפונקציה "שנה ערכים קיימים". דברים אלה נכונים גם בנוגע לפונקציה "שמור את הרשימה המעודכנת בדיסק".

אתה צריך לעמוד כבר על יתרון אחד לפחות של תכנות מובנה. על ידי חלוקה זהירה של התוכנית למשימות, אתה מזהה שתי משימות שביצוען נדרש על ידי חלקים שונים של התוכנית. אתה יכול לכתוב פונקציות לשם גישה לדיסק המבצעות "משימה כפולה" – חוסכות לך זמן, והופכות את התוכנית שלך לקטנה יותר ויעילה יותר.

התוצאה של שיטת תכנות זו היא מבנה תוכנית היררכי או שכבתי. דבר זה מודגם באיור 6.2 עבור תוכנית רשימת הכתובות.



איור 6.2. תוכנית מובנית מאורגנת באופן היררכי.

גישת מעלה-מטה

תכנות מובנה מעודד תוכניתנים ב-**C** לנקוט גישה הנקראת לעיתים גישת **מעלה-מטה (top-down)**. ראית זאת באיור 6.2 שבו מבנה התוכנית נראה כמו עץ הפוך. פעמים רבות, רוב העבודה הממשית של התוכנית מתבצעת על ידי הפונקציות ב"קצות הענפים". הפונקציות ברמות "גבוהות" יותר משמשות בראש ובראשונה לניתוב הביצוע של התוכנית בין הפונקציות הללו.

כתוצאה מכך, בתוכניות רבות ב-**C** יש מעט מאוד קוד בגוף הראשי של התוכנית, כלומר, ב-**main()** . הרוב המכריע של קוד התוכנית מצוי בפונקציות. ב-**main( )** אתה עשוי למצוא מספר מועט של שורות קוד המנתבות את ביצוע התוכנית בין הפונקציות. לעיתים קרובות ביותר, סוג כלשהו של תפריט מוצג לאדם המשתמש בתוכנית, כשביצוע התוכנית מסתעף בהתאם לבחירות של משתמש זה.

זו בעצם גישה טובה מאוד לעיצוב של תוכנית. פרק 15 מראה לך כיצד אתה יכול להשתמש במשפט **switch** כדי ליצור מערכת תפריטים רב-תכליתית.

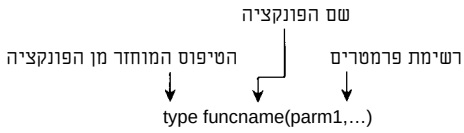
כעת אתה יודע מהן פונקציות ומדוע הן כה חשובות. ועתה הגיע הזמן שבו תלמד כיצד לכתוב פונקציה בעצמך!

כתיבת פונקציה

השלב הראשון בכתיבת פונקציה הוא, כמובן, לדעת מה אתה רוצה שהפונקציה תעשה! משעה שאתה יודע זאת, תהליכי כתיבת הפונקציה עצמם אינם קשים במיוחד.

כותרת הפונקציה

השורה הראשונה בכל פונקציה היא **כותרת הפונקציה (function header)**, שיש בה שלושה חלקים, כשלכל חלק תפקיד מסוים. חלקים אלה מתוארים בדיאגרמה שבאיור 6.3 ומוסברים כדלקמן.



איור 6.3. שלושת המרכיבים של כותרת פונקציה.

טיפוס הערך המוחזר על ידי הפונקציה

הטיפוס המוחזר (return type) על ידי הפונקציה מגדיר את טיפוס הנתונים שהפונקציה מחזירה לתוכנית הקוראת. הטיפוס המוחזר יכול להיות כל אחד מטיפוסי הנתונים ב-**C**: **char**, **int**, **long**, **float**, או **double**. אתה יכול גם להגדיר פונקציה שאינה מחזירה ערך. במקרה כזה, הגדר את הערך המוחזר כ-**void**. הנה כמה דוגמאות:

```
int func1(...)              /* returns a type int. */
float func2(...)            /* returns a type float. */
void func3(...)             /* returns nothing. */
```

שם הפונקציה

אתה יכול לקרוא לפונקציה בשם כלשהו, כל עוד אתה מציית לכללי מתן השמות למשתנה ב-**C** (הניתנים בפרק 4). שם של פונקציה חייב להיות ייחודי (אין לתת אותו לפונקציה או למשתנה אחרים כלשהם). כמובן, רעיון טוב הוא לקבוע שם המשקף את מה שהפונקציה עושה!

רשימת הפרמטרים

פונקציות רבות משתמשות **בארגומנטים (arguments)**, שהם ערכים המועברים לפונקציה כאשר היא נקראת. הפונקציה צריכה לדעת לאילו סוגים של ארגומנטים עליה לצפות – כלומר, את טיפוס הנתונים של כל ארגומנט. אתה יכול להעביר לפונקציה טיפוס נתונים כלשהו ב-**C**. מידע באשר לטיפוס הארגומנט ניתן בכותרת הפונקציה על ידי **רשימת הפרמטרים (parameter list)**.

עבור כל ארגומנט המועבר לפונקציה, רשימת הפרמטרים חייבת להכיל כניסה אחת. כניסה זו מגדירה את טיפוס הנתונים ואת שם הפרמטר. לדוגמה, הנה כותרת פונקציה מתוך הפונקציה בתדפיס 6.1:

```
long cube(long x)
```

ברשימת הפרמטרים כתוב **long x**, ועובדה זו מציינת שפונקציה זו מקבלת ארגומנט אחד מטיפוס **long**, המיוצג על ידי הפרמטר **x**. אם קיים יותר מפרמטר אחד, כל פרמטר חייב להיות מופרד על ידי פסיקים. כותרת הפונקציה

```
void func1(int x, float y, char z)
```

מגדירה פונקציה שיש לה שלושה ארגומנטים: מטיפוס **int** הנקרא **x**, מטיפוס **float** הנקרא **y**, ומטיפוס **char** הנקרא **z**. יש פונקציות שאינן מקבלות שום ארגומנט, ובמקרה כזה ברשימת הפרמטרים צריך להיות כתוב **void**:

```
void func2(void)
```

שים לב שאינך שם נקודה-פסיק בסוף כותרת של פונקציה. אם תעשה זאת בטעות, המהדר יוציא הודעה על שגיאה.

לעיתים נוצר בלבול בנוגע לאבחנה בין פרמטר לארגומנט. **פרמטר (parameter)** הוא כניסה בכותרת של פונקציה; הוא משמש כ"שומר מקום" עבור ארגומנט. הפרמטרים של פונקציה קבועים; הם אינם משתנים במהלך הביצוע של התוכנית.

ארגומנט (argument) הוא ערך בפועל המועבר לפונקציה על ידי התוכנית הקוראת. בכל פעם שפונקציה נקראת, ארגומנטים שונים יכולים להיות מועברים אליה (כמובן, חובה תמיד להעביר לפונקציה מספר מסוים של ארגומנטים, כשהארגומנטים מטיפוסים מסוימים, אך ערכי הארגומנטים יכולים להיות שונים). בפונקציה, הגישה לארגומנט מתבצעת תוך שימוש בשם הפרמטר המתאים.

דוגמה עשויה לעזור להבהיר זאת. תדפיס 6.2 מציג תוכנית פשוטה מאוד שבה פונקציה אחת הנקראת **twice**.

תדפיס 6.2. ההבדל בין ארגומנטים לפרמטרים.

```
/* Illustrates the difference between arguments and */
/* parameters. */

#include <stdio.h>

float x = 3.5, y = 65.11, z;

float half_of(float k);
```

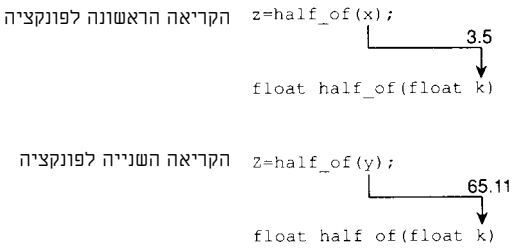
```
main()
{
   /* In this call, x is the argument to half_of(). */
   z = half_of(x);
   printf("The value of z = %f\n", z);

   /* In this call, y is the argument to half_of(). */
   z = half_of(y);
   printf("The value of z = %f\n", z);
}

float half_of(float k)
{
   /* k is the parameter. Each time half_of() is called, */
   /* k has the value that was passed as an argument. */

   return (k/2);
}
```

איור 6.4 מציג את הקשר שבין ארגומנטים לפרמטרים באופן סכימטי.



איור 6.4. בכל פעם שפונקציה נקראת, הארגומנטים מועברים לפרמטרים של הפונקציה.

גוף הפונקציה

גוף הפונקציה מוקף על ידי סוגריים מסולסלים ובא מייד לאחר כותרת הפונקציה. כאן, בגוף הפונקציה, העבודה הממשית מתבצעת. כאשר פונקציה נקראת, הביצוע מתחיל בתחילת גוף הפונקציה, **ומסתיים (terminates)** – חוזר אל התוכנית הקוראת – במשפט **החזרה** או בסוגר מסולסל ימני.

משתנים מקומיים

אתה יכול להצהיר על משתנים בתוך גוף הפונקציה. משתנים אשר מוצהר עליהם בתוך פונקציה נקראים **משתנים מקומיים (local variables)**. פירוש המונח **מקומי (local)** הוא שהמשתנים משתייכים לפונקציה באופן פרטי, והם נבדלים ממשתנים אחרים הנושאים שמות זהים שהוצהר

עליהם במקום אחר בתוכנית. דבר זה יוסבר עוד מעט, אולם עתה עליך ללמוד כיצד להצהיר על משתנים מקומיים.

על משתנה מקומי מצהירים כמו על כל משתנה אחר, תוך שימוש באותם טיפוסי משתנים ובאותם כללים למתן שמות אשר למדת אודותם בפרק 4. ניתן גם לאתחל משתנים מקומיים כאשר מצהירים עליהם. אתה יכול להצהיר על טיפוס כלשהו של משתנה ב-**C** בתוך פונקציה. הנה כמה דוגמאות:

```
int func1(int y)
{
    int a, b = 10;
    float rate;
    double cost = 12.55;
...
}
```

ההצהרות אשר לעיל יוצרות את המשתנים המקומיים **a**, **b**, **rate** ו-**cost** אשר הקוד בתוך הפונקציה יכול להשתמש בהם. שים לב שהפרמטרים של הפונקציה נחשבים להצהרות על משתנים; כך שהמשתנים, אם יש כאלה, ברשימת הפרמטרים של הפונקציה זמינים גם כן.

כעת (אנו חוזרים אל המובן של "מקומי"), כאשר אתה מצהיר ומשתמש במשתנה בתוך פונקציה, הוא נפרד ונבדל לחלוטין ממשתנים אחרים כלשהם אשר מוצהר עליהם במקום אחר בתוכנית. דבר זה נכון גם אם יש למשתנים שם זהה. התוכנית בתדפיס 6.3 מדגימה אי-תלות זו.

תדפיס 6.3. הדגמה של משתנים מקומיים.

```
/* Demonstrates local variables. */

#include <stdio.h>

int x = 1, y = 2;

void demo(void);

main()
{
   printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
   demo();
   printf("\nAfter calling demo(), x = %d and y = %d.", x, y);
}

void demo(void)
```

```
{
   /* Define and initialize two local variables. */

   int x = 88, y = 99;

   /* Display their values. */

   printf("\nWithin demo(), x = %d and y = %d.", x, y);
}
```

הפלט של תוכנית זו מוצג כאן:

```
Before calling demo( ), x = 1 and y = 2.
Within demo( ), x = 88 and y = 99.
After calling demo( ), x = 1 and y = 2.
```

דבר זה מראה בבירור שהמשתנים המקומיים **x** ו-**y** – בתוך הפונקציה – בלתי-תלויים לחלוטין במשתנים **x** ו-**y** – אשר מוצהר עליהם מחוץ לפונקציה. מכך אתה יכול ללמוד שקיימים שלושה כללים המפקחים על השימוש במשתנים בפונקציות:

1. כדי להשתמש במשתנה בתוך פונקציה, אתה חייב להצהיר עליו בכותרת הפונקציה או בגוף הפונקציה (למעט משתנים **גלובליים – global**, הנידונים בפרק 12).
2. כדי שפונקציה תקבל ערך מן התוכנית הקוראת, ערך זה חייב להיות מועבר כארגומנט.
3. כדי שהתוכנית הקוראת תקבל ערך מן הפונקציה, ערך זה חייב להיות מוחזר באופן מפורש מן הפונקציה.

אם להיות ישר לגמרי, הכללים אינם מדויקים לחלוטין, מכיוון שתלמד לעקוף אותם מאוחר יותר בספר זה. אולם בינתיים פעל כאילו הם נכונים וכך תימנע מבעיות.

משתנים מקומיים הם היבט חשוב של כוחן של פונקציות. החזקה של משתני הפונקציה בנפרד ממשתנים אחרים בתוכנית היא דרך אחת שבה הפונקציות הן עצמאיות. פונקציה יכולה לבצע כל סוג של מניפולציה בנתונים כרצונך, תוך שימוש בקבוצת המשתנים המקומיים שלה. אין מקום לחשוש מפני האפשרות שלמניפולציות אלו תהיה השפעה שלא התכוונת אליה על חלק אחר של התוכנית.

משפטי פונקציה

באופן מהותי, לא קיימות הגבלות על משפטים שניתן לכלול בתוך פונקציה. הדבר היחיד **שאינך** יכול לעשות בתוך פונקציה הוא להגדיר פונקציה אחרת. אולם, אתה יכול להשתמש בכל המשפטים ב-**C**, כולל לולאות, משפטי **if**, ומשפטי הצבה. אתה יכול לקרוא לפונקציות ספריה ולפונקציות אחרות המוגדרות על ידי המשתמש. השמים הם הגבול!

מה בדבר אורך של פונקציה? **C** אינה מציבה מגבלה של אורך על פונקציות, אך משיקולים מעשיים עליך לדאוג שהפונקציות שלך יהיו קצרות באופן יחסי. באופן זה תנהג בהתאם לעקרונות של תכנות מובנה, אשר לפיהם אמורה כל פונקציה לבצע משימה פשוטה יחסית. אם תבחין שפונקציה מסוימת הופכת לארוכה באמת, ייתכן שהסיבה לכך היא שאתה מנסה לבצע משימה שהיא מורכבת מדי עבור פונקציה אחת. קרוב לוודאי שניתן לפצל פונקציה זו לשתי פונקציות או יותר שהן קטנות יותר.

מתי ארוך הוא ארוך מדי? אין תשובה מוחלטת לשאלה זו, אך הניסיון המעשי מלמד שלעיתים נדירות תיתקל בפונקציה המשתרעת על פני יותר מ-25 או 30 שורות קוד. אתה חייב להשתמש בשיקול הדעת שלך. כמה משימות תכנותיות דורשות פונקציות ארוכות יותר, בעוד שאורכן של פונקציות רבות הוא שורות מעטות בלבד. עם שתרכוש ניסיון בתכנות, אתה תהפוך מיומן יותר לקבוע מה ניתן לפצל לפונקציות קטנות יותר ומה לא.

החזרה של ערך

כדי להחזיר ערך מפונקציה, אתה משתמש במילת המפתח **return** בפונקציה. מילת המפתח **return** נעקבת על ידי ביטוי ב-**C**. כאשר הביצוע נתקל במשפט **return**, הביטוי מחושב והביצוע חוזר אל התוכנית הקוראת. הערך המוחזר על ידי הפונקציה הוא הערך של הביטוי. התבונן בפונקציה:

```
int func1(int var)
{
    int x;
    ...
    ...
    return x;
}
```

כאשר פונקציה זו נקראת, המשפטים בגוף הפונקציה מתבצעים עד משפט **return**. **return** מסיים את הפונקציה ומחזיר את הערך של **x** לתוכנית הקוראת. הביטוי העוקב את מילת המפתח **return** יכול להיות ביטוי תקף כלשהו ב-**C**.

פונקציה יכולה להכיל משפטי **return** מרובים. משפט **return** הראשון שמבוצע הוא היחיד שיש לו השפעה כלשהי. משפטי **return** מרובים הם דרך יעילה להחזרת ערכים שונים מפונקציה. התבונן בדוגמה שבתדפיס 6.4.

תדפיס 6.4. הדגמה של שימוש במשפטי return מרובים בפונקציה.

```
/* Demonstrates using multiple return statements */
/* in a function. */

#include <stdio.h>
```

```
int x, y, z;

int larger_of( int a, int b);

main()
{
   puts("Enter two different integer values: ");
   scanf("%d%d", &x, &y);

   z = larger_of(x,y);

   printf("\nThe larger value is %d.", z);
}

int larger_of( int a, int b)
{
   if (a > b)
        return a;
   else
        return b;
}
```

כשהדבר תלוי בארגומנטים המועברים לפונקציה **larger_of( )**, משפט **return** הראשון או השני מבוצעים, כשהם מעבירים את הערך המתאים חזרה אל התוכנית הקוראת.

זכור מדיון קודם שהערך המוחזר מפונקציה הוא מטיפוס מסוים המוגדר בכותרת הפונקציה ובאבטיפוס של הפונקציה. הערך המוחזר מן הפונקציה חייב להיות מאותו טיפוס, או שיהיו שגיאות מהדר.

האבטיפוס של פונקציה

תוכנית חייבת לכלול אבטיפוס עבור כל פונקציה שהיא משתמשת בה. ראית דוגמה לאבטיפוס של פונקציה בשורה 3 שבתדפיס 6.1, והיו אבטיפוסים של פונקציה בתדפיסים אחרים גם כן. מהו אבטיפוס של פונקציה, ולשם מה הוא נועד?

אתה יכול להבחין מתוך הדוגמאות הקודמות שהאבטיפוס של פונקציה זהה לכותרת הפונקציה, למעט הנקודה-פסיק המושם בסופו. כמו כותרת הפונקציה, האבטיפוס של הפונקציה כולל מידע אודות טיפוס הערך המוחזר על ידי הפונקציה, שמה, והפרמטרים שלה. כשהוא מצויד במידע זה, המהדר יכול לבדוק בכל פעם שקוד המקור שלך קורא לפונקציה, ולאמת שאתה מעביר לפונקציה מספר נכון של ארגומנטים ומן הטיפוסים הנכונים, ומשתמש נכון בערך המוחזר על ידי הפונקציה. אם קיימת אי-התאמה, המהדר מאתר אותה ומוציא הודעה על שגיאה.

לאמיתו של דבר, אבטיפוס של פונקציה לא חייב להתאים בדיוק לכותרת הפונקציה. שמות הפרמטרים יכולים להיות שונים, כל עוד הם מאותו טיפוס, מספרם זהה, והם נתונים באותו סדר.

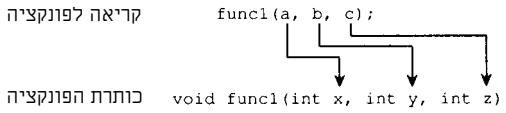
אולם לא קיימת סיבה שבגללה לא תהיה התאמה בין כותרת הפונקציה לאבטיפוס שלה, והזהות ביניהם הופכת את קוד המקור לקל יותר להבנה. היא גם הופכת את כתיבת התוכנית לקלה יותר. כאשר אתה משלים את כתיבת הגדרת הפונקציה, פשוט השתמש בתכונה גזור-והדבק של העורך שלך כדי להעתיק את כותרת הפונקציה וליצור באופן זה את האבטיפוס. הקפד להוסיף נקודה-פסיק בסוף.

היכן יש להכניס אבטיפוסים של פונקציות בקוד המקור? חובה להכניס אותם לפני תחילת **( )main**, ולשם הקריאות, הטוב ביותר הוא לקבץ את כל האבטיפוסים ביחד במקום אחד.

העברת ארגומנטים לפונקציה

כדי להעביר ארגומנטים לפונקציה, אתה רושם אותם בסוגריים הבאים לאחר שם הפונקציה. מספר הארגומנטים והטיפוס של כל ארגומנט חייבים להתאים לפרמטרים שבכותרת הפונקציה ובאבטיפוס. לדוגמה, אם פונקציה מוגדרת כך שעליה לקבל שני ארגומנטים מטיפוס **int**, אתה חייב להעביר אליה בדיוק שני ארגומנטים מטיפוס **int** – לא פחות ולא יותר, ולא שום טיפוס אחר. אם אתה מנסה להעביר לפונקציה מספר ו/או טיפוס לא נכונים של ארגומנטים, המהדר מאתר זאת, בהתבסס על המידע שבאבטיפוס של הפונקציה.

אם הפונקציה מקבלת ארגומנטים מרובים, אזי הארגומנטים הרשומים בקריאה לפונקציה מוקצבים לפרמטרים של הפונקציה בסדר: הארגומנט הראשון לפרמטר הראשון, וכן הלאה. דבר זה מודגם באיור 6.5.



איור 6.5. ארגומנטים מרובים מוקצבים בסדר לפרמטרים של הפונקציה.

כל ארגומנט יכול להיות ביטוי תקף כלשהו ב-C: קבוע, משתנה, ביטוי מתמטי או לוגי, או אפילו פונקציה אחרת (פונקציה המחזירה ערך). לדוגמה, אם **( )half**, **( )square**, ו-**( )third** הן כולן פונקציות המחזירות ערך, יכולת לכתוב

```
x = half(third(square(half(y))));
```

התוכנית קוראת תחילה ל-**( )half**, ומעבירה אליה את **y** כארגומנט. כאשר הביצוע חוזר מ-**( )half**, התוכנית קוראת ל-**( )square**, ומעבירה אליה את הערך שהוחזר מן הפונקציה **( )half** כארגומנט. לאחר מכן, הפונקציה **( )third** נקראת כשהארגומנט המועבר הוא הערך המוחזר מ-**( )square**. בשלב זה, **( )half** נקראת שוב, כשהפעם הערך המוחזר מ-**( )third** הוא הארגומנט. לבסוף, הערך המוחזר מ-**( )half** מוצב במשתנה **x**.

קריאה לפונקציות

קיימות שתי דרכים לקריאה לפונקציה. פונקציה כלשהי יכולה להיקרא פשוט על ידי שימוש במשפט שיכיל רק את שמה ואת רשימת הארגומנטים בתוכנית שלך. אם לפונקציה יש ערך מוחזר, מתעלמים ממנו. לדוגמה,

```
wait(12);
```

בשיטה השנייה ניתן להשתמש רק בהתייחס לפונקציות שיש להן ערך מוחזר. מאחר שפונקציות אלו מתחשבות לערך (כלומר, לערך המוחזר שלהן), הן ביטויים תקפים ב-**C** וניתן להשתמש בהן בכל מקום שבו ניתן להשתמש בביטוי ב-**C**. ראית כבר כיצד ניתן להשתמש בביטוי שיש לו ערך מוחזר בצד הימני של משפט הצבה. הנה כמה דוגמאות נוספות:

```
printf("Half of %d is %d.", x, half_of(x));
y = half_of(x) + half_of(z);
if ( half_of(x) > 10 )
    ...;
```

אם תנסה להשתמש בפונקציה עם ערך מוחזר מטיפוס **void** כביטוי, המהדר יוציא הודעה על שגיאה.

רקורסיה

המונח **רקורסיה (recursion)** מתייחס למצב שבו פונקציה קוראת לעצמה באופן ישיר או באופן עקיף. המובן של רקורסיה עקיפה הוא שפונקציה אחת קוראת לפונקציה אחרת אשר לאחר מכן קוראת לפונקציה הראשונה. **C** מאפשרת פונקציות רקורסיביות, והן יכולות, לאמיתו של דבר, להיות שימושיות במצבים מסוימים.

דוגמה טובה אחת היא חישוב העצרת של מספר. אתה עשוי להיזכר, משיעורי המתמטיקה, שהעצרת של מספר **x** נכתבת **x!**, והיא מחושבת כ-

```
x! = x * (x-1) * (x-2) * (x-3) .... * (2) * 1
```

האפשרות להשתמש בפונקציה רקורסיבית עולה מתוך העובדה, אשר אפשר שכבר שמת לב אליה, שאתה יכול לחשב את **x!** גם באופן הבא:

```
x! = x * (x-1)!
```

אם נתקדם צעד אחד נוסף, אתה יכול לחשב את **(x-1)!** תוך שימוש בנוהל זהה:

```
(x-1)! = (x-1) * (x-2)!
```

אתה יכול להמשיך לפעול כך עד שתגיע לערך של 1, ואז לסיים. התוכנית בתדפיס 6.5 משתמשת בפונקציה רקורסיבית כדי לחשב עצרות. מכיוון שהתוכנית משתמשת במספרים שלמים (integers) בלי סימן, היא מוגבלת לערך קלט של 8; העצרת של 9 ושל ערכים גדולים יותר מציבים אותה מחוץ לטווח המותר עבור מספרים שלמים.

תדפיס 6.5. שימוש בפונקציה רקורסיבית לחישוב עצרות.

```
/* Demonstrates function recursion. Calculates the */
/* factorial of a number. */

#include <stdio.h>

unsigned int f, x;
unsigned int factorial(unsigned int a);

main()
{
   puts("Enter an integer value between 1 and 8: "); scanf("%d", &x);

   f = factorial(x);

   printf("%u factorial equals %u", x, f);
}

unsigned int factorial(unsigned int a)
{
   if (a == 1)
        return 1;
   else
        {
        a *= factorial(a-1);
        return a;
        }
}
```

היכן למקם את הפונקציות?

אתה עשוי לתהות היכן יש למקם את הגדרות הפונקציה בתוך קוד המקור שלך. בינתיים, יש לכלול אותן באותו קובץ קוד מקור שבו נמצאת **main( )**, לאחר הסוף של **main( )**. המבנה הבסיסי של תוכנית המשתמשת בפונקציות מוצג באיור 6.6.

```
/* start of source code */
   ...
   prototypes here
   ...
   main()
   {
   ...
   ...
   }
   func1()
   {
   ...
   }
   func2()
   {
   ...
   }
/* end of source code */
```

איור 6.6. מקם את האבטיפוסים של הפונקציות לפני ( )main ואת ההגדרות של הפונקציות לאחר ( )main.

אתה יכול לשמור את הפונקציות המוגדרות על ידי המשתמש (הפונקציות שלך) בקובץ קוד מקור אחר, בנפרד מ-**main( )**. טכניקה זו שימושית במקרה של תוכניות גדולות, וכאשר אתה רוצה להשתמש באותה קבוצה של פונקציות ביותר מאשר בתוכנית אחת. האופן שבו יש לעשות זאת מוצג בפרק 21.

סיכום הפרק

פרק זה הציג בפניך חלק חשוב מאוד בתכנות ב-C: פונקציות, שהן קטע עצמאי של קוד המבצע משימות ספציפיות. בכל פעם שהתוכנית זקוקה לביצוע של המשימה, היא קוראת לפונקציה המתאימה. השימוש בפונקציות חיוני עבור תכנות מובנה – שיטה לעיצוב תוכנית המדגישה גישה מודולרית של מעלה-מטה. תכנות מובנה יוצר תוכניות יעילות יותר, והוא גם קל הרבה יותר לשימוש עבורך, התוכניתן.

למדת גם שפונקציה מורכבת מכותרת ומגוף. הכותרת כוללת מידע אודות הערך המוחזר מן הפונקציה, שמה, והפרמטרים שלה. הגוף כולל הצהרות על משתנים מקומיים ואת המשפטים ב-C המבוצעים כאשר הפונקציה נקראת. לבסוף, ראית שמשתנים מקומיים – אלה אשר מוצהר עליהם בתוך הפונקציה – הם בלתי תלויים לחלוטין במשתני תוכנית אחרים כלשהם אשר מוצהר עליהם במקום אחר.

בוחן

1 האם אתה מתכוון להשתמש בתכנות מובנה כאשר תכתוב את התוכניות שלך ב-C?

2 כיצד תכנות מובנה פועל?

3 כיצד פונקציות ב-**C** משתלבות בתוך תכנות מובנה?

4 מה צריך להיות המשפט הראשון בהגדרה של פונקציה, ואיזה מידע הוא מכיל?

5 כתוב כותרת של פונקציה עבור פונקציה הנקראת **do_it( )** המקבלת שלושה ארגומנטים מטיפוס **char** ומחזירה ערך מטיפוס **float** אל התוכנית הקוראת.

6 מה שגוי בהגדרת הפונקציה הבאה:

```
int twice(int y);
{
return (2 * y);
}
```

7 מהו ההבדל בין הגדרת פונקציה לאבטיפוס של פונקציה?

8 מהו משתנה מקומי?

9 מה הייחוד של משתנים מקומיים?

10 כתוב תוכנית המשתמשת בפונקציה כדי לחשב את הממוצע של חמישה ערכים מטיפוס **float** המוכנסים על ידי המשתמש.