מרקו קנטו

יסודות הפסקל

פיאצ'נזה, איטליה
מהדורה רביעית, אפריל 2008

מחבר ומוציא לאור: מרקו קנטו
עורך: פיטר וו' א' וווד
עריכה טכנית (של מהדורה זו): פטריציו מושקוביץ', קלד ר' הנסן
עיצוב כריכה: פבריציו שיאבי

Copyright © 1995-2008 Marco Cantù, Piacenza, Italy. World rights reserved.

תרגום לעברית: עידו גנדל (http://www.idogendel.com)

© כל הזכויות על התרגום לעברית שמורות לעידו גנדל, 2010

בפרסום זה יצר המחבר קוד לדוגמה, למטרה מפורשת של שימוש חופשי על ידי קוראיו. קוד המקור עבור ספר זה הינו תוכנה חופשית (Freeware) המוגנת בזכויות יוצרים ומופצת דרך אתר האינטרנט http://www.marcocantu.com. זכויות היוצרים מונעות מכם לפרסם מחדש את הקוד במדיה מודפסת ללא אישור המחבר. ניתנת רשות מוגבלת לקוראים להשתמש בקוד זה ביישומים שלהם, כל עוד הקוד עצמו אינו מופץ, נמכר או מנוצל לצורך מסחרי כמוצר בפני עצמו. ניתן להשתמש בחלקים מקטעי הקוד ביישומים שלכם, ללא תלות ברשיון השימוש של יישומים אלה.

פרט להרשאות הספציפיות שניתנו עבור קוד מקור, אין לאחסן במערכת אחזור, לשדר או לשכפל בכל דרך שהיא כל חלק שהוא מפרסום זה, במקורו או מתורגם, כולל (אך לא מוגבל ל) מכונת צילום, מצלמה, אחסון מגנטי או כל אמצעי תיעוד אחר, ללא הסכמתו מראש ואישורו בכתב של המוציא לאור.

ISBN: לא הוקצה.

Delphi הינו סימן מסחרי של CodeGear, חברה בת של Borland. Windows ו- Windows Vista הינם סימנים מסחריים של Microsoft. סימנים מסחריים אחרים שייכים לבעליהם כמצוין בגוף הטקסט.

המחבר (והמוציא לאור) עשה כמיטב יכולתו בהכנת הספר, והתוכן מבוסס ככל שניתן על גרסאות התוכנה העדכניות ביותר. המחבר, המוציא לאור והמתרגם אינם מתחייבים או אחראים בשום צורה ואופן לשלמות או לדיוק של תוכן ספר זה ו/או תרגומו, ולא יישאו באחריות מכל סוג שהוא, כולל (אך לא מוגבל ל) ביצועים, התאמה ליישומים מסחריים, התאמה לכל מטרה מסוימת, או אובדן או נזק כלשהם מכל סוג שהוא שנגרמו או נגרמו לכאורה, ישירות או בעקיפין, בשל ספר זה.

המהדורה המודפסת הראשונה הופיעה ב-Lulu.com. מהדורה מתוקנת ראשונה. 21 באפריל 2008.

ניתן להזמין עותקים מודפסים של המהדורה האנגלית בכתובת
http://www.lulu.com/content/2398448.

מבוא

ספר זה מוקדש למשפחתי,
ללה, בנדטה וג'קופו

המהדורות הראשונות של *Mastering Delphi*, סדרת ספרי דלפי הנמכרת ביותר שלי שנכתבה בין השנים 1999 ו-2005, סיפקו מבוא לשפת התכנות פסקל בדלפי. בשל מגבלות מקום, ומכיוון שמתכנתי דלפי רבים חיפשו מידע מתקדם יותר, חומר זו הושמט לחלוטין מהמהדורות המאוחרות של סדרת *Mastering Delphi*. כדי להתגבר על היעדר המידע התחלתי לכתוב ספר אלקטרוני שכותרתו *Essential Pascal*. לאחר מספר מהדורות שיועדו לאינטרנט בלבד, הפכתי את הספר לזמין בשלושה פורמטים: ספר מבוסס HTML בחינם (עם פרסומות), גרסה מבוססת PDF בתשלום, וספר מודפס. הקפדתי שמחירי ה-PDF והגרסה המודפסת יהיו נמוכים מאד על מנת להתאימם לקהל הסטודנטים והחובבים, אם כי הספר יכול לשרת גם מפתחים מקצועיים.

פסקל במובן דלפי

זהו ספר מפורט על שפת התכנות פסקל המשמשת בדלפי, ובכמה מניבי דלפי הזמינים (כלומר FreePascal ו-Chrome)[1]. הספר מתמקד בכוונה תחילה במבני השפה ה*מסורתיים* של פסקל, ואינו מתעמק בהרחבות הקשורות לתכנות מונחה עצמים. מפעם לפעם יוצעו עדכונים שיכסו הרחבות של ליבת השפה, אשר יופיעו במהדורות עדכניות של דלפי (שוב, לא כולל הרחבות הקשורות לתכנות מונחה עצמים של Object Pascal).

המהדורה השלמה הראשונה של ספר זה, שהושלמה ביולי 1999, צורפה לתקליטור שליווה את דלפי 5. המהדורות הבאות עודכנו, הן בתוכנן והן בעיצובן, ונוספו להן הערות בנוגע ל-Kylix (דלפי ללינוקס) ולדלפי עבור דוט נט[2]. מהדורה חדשה זו, הראשונה הזמינה בדפוס, מרחיבה את הכיסוי לניבים אחרים של פסקל/דלפי.

ממהדורה למהדורה, הדוגמאות המשמשות בספר התרחקו מספריית הרכיבים החזותיים (VCL) של ממשק המשתמש הגרפי (GUI) של דלפי, כדי להפוך את הספר למתאים יותר לפלטפורמות ולמהדרים שונים. המעבר מדוגמאות חזותיות לדוגמאות מבוססות מסוף בקרה (Console) הביא עמו את היתרון, שכעת הקוראים יכולים להתרכז אף יותר בשפה ולהתעלם ממנהלי אירועים (Event Handlers), מתודות, מרכיבים (Components) ונושאים מתקדמים אחרים. כמו כן, התוכניות יכולות לרוץ בפלטפורמות שאינן Windows.

[1] קיים גם מהדר פסקל עבור GNU התומך בתקני ISO 7185 ו-ISO 10206, אך קהילת הפסקל ממעטת להשתמש בו, עובדה המקושרת לרוב להרחבות הלא-תקניות ש-Borland הוסיפה לשפה.

[2] המוצר Delphi for .NET אינו מופיע בגרסאות 2009 ומעלה של דלפי – הוא הוחלף ב-Delphi Prism. המוצר Kylix אינו קיים עוד. [המתרגם]

זכויות יוצרים על הספר וזמינות

זכויות היוצרים על הטקסט ועל קוד המקור שבספר זה שייכות למרקו קנטו. כמובן, אתם רשאים להשתמש בתוכניות ולהתאים אותן לצורכיכם, אך אינכם רשאים להשתמש בהן ללא רשות בספרים, בחומרי הדרכה ובפורמטים אחרים המוגנים בזכויות יוצרים (אלא אם, כמובן, אתם משתמשים בכמות מוגבלת וסבירה של הטקסט או של קוד המקור ומציינים במפורש את מקורו). זכויות היוצרים על התרגום לעברית שמורות לעידו גנדל, באותם התנאים.

אין להפיץ את המהדורה האנגלית של ספר זה בפורמט אלקטרוני. גרסת ה-HTML זמינה (ותישאר זמינה) בחינם בכתובת:

http://www.marcocantu.com/epascal

בראשית הדרך, הספר היה קיים בפורמט HTML בלבד. הגרסאות הבאות היו זמינות כקובצי PDF שניתנו בחינם, אם כי ביקשתי תשלום של רצון טוב (או תרומה). כעת אני מפרסם את הספר בשלושה פורמטים (התוכן בכולם זהה לחלוטין):

- גרסת HTML בחינם עם פרסומות
- גרסת PDF אותה ניתן לרכוש באתר Lulu.com (אם השגתם אותה במקום אחר, כנראה שמדובר בעותק לא חוקי).
- גרסה מודפסת, אותה ניתן לרכוש באתר Lulu.com.

הערה: כמחווה לקהל המתכנתים הישראלי, המחבר הסכים להפיץ תרגום זה של המהדורה העדכנית ביותר לעברית כקובץ PDF בחינם. אם הספר הביא לכם תועלת, הנכם מוזמנים להביע את תודתכם ולתמוך במחבר ובהמשך העבודה על הספר באמצעות תרומה באתר המחבר: http://www.marcocantu.com/epascal/donate.htm

קוד מקור

קוד המקור של כל הדוגמאות המוזכרות בספר זמין בחינם. על הקוד חלות אותן זכויות יוצרים כמו על הספר עצמו: השתמשו בו כרצונכם, אך אל תפרסמו אותו במסמכים או באתרי אינטרנט אחרים. פרטי הורדת הקוד ורשימה של דוגמאות מופיעים בנספח של ספר זה.

משוב

נא הודיעו לי על כל שגיאה שתמצאו, אך גם על נושאים שאינם ברורים דיים עבור מתחילים. ספרו לי גם אילו נושאים אחרים הייתם רוצים שאכסה בספרים עתידיים.

הדרך המועדפת לשליחת משוב היא בקבוצת הדיון (Newsgroup) הפומבית שלי (ממשק רשת לקבוצות שלי נמצא באתר האינטרנט http://delphi.newswhat.com), באזור המוקדש לספרים. אם אתם מתקשים בשימוש בקבוצות הדיון, שלחו במקום זאת דואר אלקטרוני לכתובת marco.cantu@gmail.com.

תודות

הסיבה העיקרית לפרסום ספר זה בחינם ברשת היא התנסותו של ברוס אקל (Eckel) עם ספרו *Thinking in Java*. ברוס הוא ידידי, ואני סבור שהוא עשה עבודה מצוינת בספר הנ"ל ובאחרים, לא רק באיכות הגבוהה של תוכנם אלא גם בניסויים שערך בשיטות הפצה.

כאשר הזכרתי את הפרויקט בפני אנשים בחברת Borland (כפי שנקראה באותה עת), קיבלתי המון משוב חיובי. כמובן, עליי להודות לחברה על יצירת סדרת המהדרים "טורבו פסקל", ומאוחר יותר גם דלפי – סדרת סביבות הפיתוח החזותיות המשולבות.

בעקבות המהדורות הראשונות של הספר קיבלתי משוב חשוב מאד. הקוראים הראשונים שסייעו לשפר את החומרים הללו היו צ'רלס וד ו-וויאט וונג. מרק גרינהאו ופרדריק גאוטייר-בוטין סייעו בעריכה של חלק מהטקסט. רפאל ברנקו-דרוגה הציע תיקונים טכניים רבים ועריכה לשונית. תודה.

בזמן העבודה על המהדורה החדשה הזו קיבלתי סיוע בעריכה מפיטר וו' א' וד והגהות טכניות מאת פטריציו מושקוביץ' וקלד ר' הנסן. כריכת הספר המודפס עוצבה על ידי פבריציו שיאבי. היא מייצגת משולש פסקל[3], ותואמת את הכריכה של ספרי *Delphi 2007 Handbook*.

על המחבר

אני חי בפיאצ'נזה שבאיטליה. בשנת 1995, לאחר שלימדתי את שפת C++ וכתבתי ספרים ומאמרים על C++ ועל Object Windows Library, התחלתי לעסוק בתכנות בדלפי. אני המחבר של סדרת הספרים *Mastering Delphi* שראתה אור בהוצאת Sybex, הספר למתקדמים *Delphi Developers Handbook* (שקשה להשיגו בימים אלה, אך ייתכן שאוציא אותו לאור שוב בעצמי), וגם הספר *Delphi 2007 Handbook* שראה אור לאחרונה. כתבתי מאמרים לכתבי עת רבים, ביניהם *The Delphi Magazine*, הרציתי בכנסים של דלפי ושל Borland ברחבי העולם, ולימדתי דלפי ברמות מתחילים ומתקדמים.

לאחרונה נעשיתי מעורב יותר ויותר בטכניקות לפיתוח Web 2.0 ובטכנולוגיות הקשורות ל-XML, אם כי בעיקר מנקודת המבט של דלפי. תוכלו למצוא פרטים נוספים עליי ועל עבודתי:

באתר האינטרנט שלי: http://www.marcocantu.com
ובבלוג שלי: http://blog.marcocantu.com

[3] למידע נוסף, ראו בכתובת: http://he.wikipedia.org/wiki/משולש_פסקל

תוכן העניינים

מבוא 4

פסקל במובן דלפי 4
זכויות יוצרים על הספר וזמינות 5
קוד מקור 5
משוב 5
תודות 6
על המחבר 6

פרק 1: היסטוריה קצרה של שפת התכנות פסקל 10

הפסקל של וירת' 10
טורבו פסקל 10
הפסקל של דלפי 11

פרק 2: תכנות בפסקל 13

תחביר וסגנון 13
הערות 14
השימוש באותיות רישיות 15
סימנים לבנים 15
עיצוב נאה 16
הבלטת תחביר 17
זיהוי שגיאות ומסייעי קוד 18
הצהרות שפה 18
מילות מפתח 18
ערכים ליטרליים 19
ביטויים ואופרטורים 20
הצגת תוצאה של ביטוי (תוכנית ראשונה) 20
אופרטורים וקדימות 21
אופרטורים של קבוצות 22
סיכום 23

פרק 3: טיפוסים, משתנים וקבועים 24

משתנים 24
קבועים 25
קבועי מחרוזות משאב (RESOURCE STRING) 26
טיפוסי נתונים 27
טיפוסים סודרים 27
טיפוסי מספרים שלמים 28
בוליאני 28
תווים 29
הצגת טווחים סודרים 29
רוטינות של טיפוסים סודרים 31
טיפוסים ממשיים 32
תאריך ושעה 33
טיפוסים ספציפיים ל-WINDOWS 36
הטלת טיפוסים והמרות טיפוסים 36

סיכום 38

פרק 4: טיפוסי נתונים מוגדרים על ידי המשתמש 39

טיפוסים משוימים ולא-משוימים 39
טיפוסי תת-טווח 40
טיפוסים מונים 41
טיפוסי קבוצה 42
טיפוסי מערכים 43
טיפוסי רשומות 45
מצביעים 46
טיפוסי קבצים 48
סיכום 49

פרק 5: הצהרות 50

הצהרות פשוטות ומרוכבות 50
הצהרות השמה 51
הצהרות תנאי 51
הצהרות if *52*
הצהרות case *53*
לולאות בפסקל 53
הלולאה for *54*
הצהרות while ו-repeat *55*
דוגמאות ללולאות *55*
ההצהרה WITH 57
סיכום 59

פרק 6: פרוצדורות ופונקציות 60

פרוצדורות ופונקציות בפסקל 60
פרמטרי הפניה 61
פרמטרים קבועים 62
פרמטרי מערכים פתוחים 63
פרמטרים של מערך פתוח בעלי טיפוס משתנה *64*
מוסכמות קריאה של דלפי 66
מהי מתודה? 66
הכרזות מקדימות 67
טיפוסים פרוצדורליים 68
העמסת פונקציות 70
פרמטרים של ברירת מחדל 71
סיכום 73

פרק 7: טיפול במחרוזות 74

טיפוסי מחרוזות 74
מחרוזות פסקל מסורתיות 75
שימוש במחרוזות ארוכות 75
המחרוזות בזכרון 76
מחרוזות של דלפי ו-PCHAR של WINDOWS 77
עיצוב מחרוזות 79
סיכום 81

פרק 8: זכרון 82

זכרון גלובלי 82
זכרון המחסנית 82
זכרון הערמה 83

מערכים דינמיים 83
סיכום 85

פרק 9: תכנות ל-WINDOWS 86

ידיות של WINDOWS 86
הכרזות חיצוניות 87
פונקציית CALLBACK של WINDOWS 88
תוכנית WINDOWS מינימלית 89
סיכום 91

פרק 10: וריאנטים 92

לווריאנטים אין טיפוס 92
הסתכלות מעמיקה על וריאנטים 93
וריאנטים הם איטיים! 94
סיכום 95

פרק 11: תוכניות ויחידות 96

יחידות 96
יחידות והיקף 98
יחידות כמרחבי שמות 99
יחידות ותוכניות 100
סיכום 100

פרק 12: קבצים בשפת פסקל 101

רוטינות לעבודה עם קבצים 101
טיפול בקובצי טקסט 103
ממיר קובץ טקסט 104
סיכום 106

אחרית דבר 107

נספח: דוגמאות 108

אינדקס 109

פרק 1: היסטוריה קצרה של שפת התכנות פסקל

שפת התכנות Object Pascal, בה אנו משתמשים בדלפי, לא הומצאה בשנת 1995 ביחד עם סביבת הפיתוח החזותית של Borland. היתה זו פשוט הרחבה של שפת Object Pascal שכבר נעשה בה שימוש במוצרי פסקל של Borland. למעשה, Borland לא המציאה את שפת פסקל – רק סייעה להפוך אותה לפופולרית ביותר והרחיבה אותה מעט.

הפסקל של וירת'

שפת פסקל תוכננה במקור בשנת 1971 על ידי ניקלאוס וירת' (Niklaus Wirth)[4], פרופסור בפוליטכניון של ציריך בשוויץ. היא עוצבה כגרסה מפושטת, למטרות חינוך, של שפת אלגול שהופיעה בשנת 1960.

בימים בהם תוכננה פסקל היו שפות תכנות רבות, אך רק מעטות היו בשימוש נרחב: פורטרן, אסמבלר, קובול. הרעיון המרכזי של השפה החדשה היה סדר, שמושג באמצעות תפיסה קפדנית של טיפוסי נתונים, הכרזות משתנים ובקרות תוכנית מובנות. השפה תוכננה גם במטרה להיות כלי לימוד.

טורבו פסקל

מהדר הפסקל המפורסם של Borland, טורבו פסקל, יצא לשוק בשנת 1983 ויישם את הנחיות המסמך "Pascal User Manual and Report" מאת ינסן ו-וירת'. סדרת מהדרי טורבו פסקל היתה מהנמכרות ביותר בכל הזמנים, והפכה את השפה לפופולרית במיוחד עבור פלטפורמת ה-PC הודות לאיזון שסיפקה בין פשטות ועוצמה.

טורבו פסקל הציגה סביבת פיתוח משולבת (IDE), בה יכולתם לערוך את הקוד (בעורך תואם WordStar), להריץ את המהדר, לראות את השגיאות ולדלג בחזרה אל השורות שהכילו את השגיאות הללו. כיום הדבר נשמע מובן מאליו, אך לפני כן היה עליכם לצאת מהעורך, לחזור ל-DOS, להריץ את המהדר משורת הפקודה, לכתוב בצד את מספרי שורות השגיאה, לפתוח שוב את העורך ולעבור אליהן.

בנוסף, Borland מכרה את טורבו פסקל במחיר של 49 דולרים, בעוד שמהדר הפסקל של מיקרוסופט נמכר בכמה מאות. שנות ההצלחה הרבות של טורבו פסקל תרמו לכך שמיקרוסופט זנחה בסופו של דבר את מוצר מהדר הפסקל שלה.

[4] את הביוגרפיה הרשמית של וירת' אפשר לקרוא בכתובת: http://www.cs.int.ethz.ch/~wirth/

למעשה, אתם יכולים להוריד עותק של הגרסה המקורית של טורבו פסקל של Borland מאזור המוזיאון באתר רשת המפתחים של CodeGear: http://dn.codegear.com/museum.

הפסקל של דלפי

בשנת 1995, כעבור תשע גרסאות של מהדרי טורבו פסקל ו-Borland Pascal שהרחיבו לאיטן את השפה אל עולם התכנות מונחה העצמים (OOP), הוציאה Borland לשוק את דלפי והפכה את פסקל לשפת תכנות חזותית. דלפי הרחיבה את שפת פסקל במספר אופנים, ביניהם הרחבות רבות של תכנות מונחה עצמים שהיו שונות מאלה שבסוגי Object Pascal אחרים, כולל אלה ששימשו במהדר *Borland Pascal with Objects* (הגלגול האחרון של טורבו פסקל).

בדלפי 2 קידמה Borland את מהדר הפסקל לעולם 32 הביטים, ולמעשה תכננה אותו מחדש במטרה ליצור מחולל קוד מכונה תואם לזה של מהדר ה-C++. הדבר איפשר אופטימיזציות רבות לשפת פסקל, שהיו קיימות לפני כן רק במהדרי C/C++.

בדלפי 3 הוסיפה Borland לשפה את תפיסת הממשק (Interface), שהיווה זינוק קדימה מבחינת אפשרויות הביטוי של מחלקות (Classes) ושל הקשרים ביניהן. חברת Borland ביצעה צעד נוסף לפנים עם Kylix, שפתחה בפני מתכנתי פסקל/דלפי את מערכת ההפעלה לינוקס (ולו רק בגרסאות שהתבססו על מעבדים של אינטל). ניתן להריץ את רוב הדוגמאות בספר זה, ללא שינוי כמעט, גם בלינוקס.

עם יציאתה לשוק של דלפי גרסה 7 (ו-Kylix גרסה 3), החלה Borland לכנות את שפת פסקל (או Object Pascal) באופן רשמי בשם שפת דלפי. כך, דלפי 7 משתמשת בשפת דלפי, Kylix 3 תומכת הן בדלפי והן ב-C++, ויש גם מהדר מבית Borland של שפת דלפי עבור ארכיטקטורת דוט נט של מיקרוסופט. זהו שינוי קוסמטי ושיווקי בעיקרו, קרוב לוודאי בשל העובדה ששפת פסקל מעולם לא היתה פופולרית בארצות הברית כפי שהיתה (והינה עד היום) באירופה ובאזורים אחרים בעולם.

דלפי 8 עבור דוט נט הציגה סוג חדש של סביבת הפיתוח המשולבת של דלפי והוסיפה תמיכה נרחבת בשפה ובספריות שלה עבור ארכיטקטורת דוט נט של מיקרוסופט. דלפי 8, שהופיעה בסוף שנת 2003, הביאה עמה את אוסף השינויים הגדול והדרמטי ביותר בשפת Object Pascal/דלפי מאז דלפי 1 בשנת 1995. שינויים אלה אומצו גם במהדר דלפי עבור Win32.

נכון לכתיבת שורות אלה, הגרסה העדכנית ביותר של דלפי היא RAD Studio 2007 מבית CodeGear, חברה בת של Borland[5]. בין שאר השינויים, CodeGear החזירה לשפה את שמה המקורי והיא נקראת שוב, רשמית, Object Pascal. דלפי 2007 היציב והחזק ראוי לציון אחרי דלפי 2005 שלא היה "מי יודע מה" ודלפי 2006 שלקה בחוסר יציבות מסוים.

[5] נכון לשעת התרגום, הגרסה העדכנית ביותר הינה RAD Studio 2010

מבין הניבים האחרים של דלפי, שני הנפוצים ביותר הינם FPC (Free Pascal Compiler) ו-Chrome (שפה מבוססת דוט נט מבית RemObjects). אני אזכיר את שני הניבים הללו לעתים קרובות בספר. האתרים שלהם הם, בהתאמה:

http://www.freepascal.org
http://www.remobjects.com/chrome

פרק 2: תכנות בפסקל

פרק זה מתאר את רכיביה של תוכנית בפסקל, כגון מילות מפתח, סימנים לבנים וביטויים. כאן תמצאו את אבני הבניין הבסיסיות של שפת פסקל.

בתור התחלה, אראה לכם את הקוד של יישום "שלום, עולם!" פשוט שמציג כמה מהרכיבים המבניים של תוכנית בפסקל. בינתיים לא אסביר מה המשמעות של רכיבים אלה, מכיוון שבזה עוסקים הפרקים הראשונים בספר. הנה הקוד:

```
program EssHello;
{$APPTYPE CONSOLE}

var
  strMessage: string;

begin
  strMessage := 'Hello, Small World';
  writeln (strMessage);
  // המתן עד שהמשתמש ילחץ על מקש האנטר
  readln;
end.
```

בקוד זה תוכלו לראות את שם התוכנית בשורה הראשונה, הנחיית מהדר (Compiler directive), הכרזת משתנה ושלוש שורות קוד (בתוספת הערה) בתוך בלוק begin-end הראשי. שוב, נלמד על כל הרכיבים הללו בקרוב; הם מופיעים כאן רק כדי לתת לכם מושג כיצד נראית תוכנית פסקל קטנה אך שלמה.

תחביר וסגנון

לפני שנעבור לנושא של כתיבת הצהרות בשפת פסקל, חשוב להדגיש מספר רכיבים של סגנון הכתיבה בפסקל. השאלה שאני מתייחס אליה כאן היא זו: מלבד כללי התחביר, כיצד יש לכתוב קוד? אין תשובה אחת לשאלה הזו, מכיוון שטעם אישי יכול להכתיב סגנונות שונים. עם זאת, קיימים מספר עקרונות שעליכם להכיר בנוגע להערות, אותיות רישיות, רווחים ומה שמכונה בשם "עיצוב נאה" (Pretty-printing) – נאה עבורנו בני האדם, לא עבור המחשב.

ככלל, המטרה של כל סגנון כתיבת קוד היא בהירות. החלטות הסגנון והעיצוב שתקבלו יוצרות מעין קיצורים שמעידים על מטרתו של כל קטע קוד נתון. אחד הכלים החיוניים להשגת בהירות הוא העקביות: יהיה הסגנון שתבחרו אשר יהיה, הקפידו לעבוד לפיו בתוך כל פרויקט ובכל הפרויקטים.

הערות

בשפת פסקל המסורתית, הערות נכתבו בתוך סוגריים מסולסלים או בין סוגריים שאליהם צמודה כוכבית. הגרסאות המודרניות מכירות גם בהערות בסגנון C++ שמתחילות בלוכסן כפול ומשתרעות עד לסוף השורה, בלי שום סמל שיציין את סופן:

```
{זו הערה}
(*  זו הערה נוספת  *)
// וזו הערה שמגיעה עד לקצה השורה מימין
```

הצורה הראשונה היא קצרה ונפוצה יותר. הצורה השניה היתה מועדפת באירופה, מכיוון שמקלדות אירופאיות רבות לא כללו את סמלי הסוגריים המסולסלים. צורת ההערה השלישית הושאלה משפת C++ והתווספה החל מדלפי 2. הערות שמשתרעות עד לקצה השורה הן שימושיות מאד עבור הערות קצרות ולנטרול של שורות קוד בודדות[6].

שימו לב שבקוד שמופיע בספר אנסה לעצב את ההערות בגופן נטוי ואת מילות המפתח בגופן מודגש. זאת כדי לשמור על עקביות עם הבלטת התחביר שדלפי (ורוב העורכים האחרים) מבצעת כברירת מחדל.

שלוש צורות שונות של הערות יכולות להיות שימושית לצורך קינון של הערות. אם ברצונכם לנטרל מספר שורות קוד באמצעות הערה, ושורות אלה עצמן מכילות הערות אמיתיות, לא תוכלו להשתמש באותו מזהה הערה:

```
{   ... קוד
{הערה שיוצרת בעיות}
... קוד  }
```

לעומת זאת, בעזרת מזהה הערה שני, תוכלו לכתוב את הקוד התקין הבא:

```
{   ... קוד
// הערה זו תקינה
... קוד  }
```

שימו לב שאם סוגריים מסולסלים שמאליים, או צירוף סוגריים שמאליים וכוכבית, מלווים בסימן דולר ($), הם הופכים להנחיית מהדר, לדוגמה[7]:

```
{$X+}
```

[6] החל מדלפי גרסת 2006 (כולל גרסת Turbo Delphi) ניתן לנטרל או לבטל נטרול של שורה (או קבוצת שורות) בעזרת סימוני הערות באמצעות לחיצה ישירה על Ctrl+/ במקלדת אמריקנית, או על שילוב אחר שכולל את המקש "/" במקלדות אחרות.

[7] למעשה, הנחיות מהדר הן הערות גם כן. לדוגמה, הקוד {$X+ This is a comment} הוא חוקי. הוא מהווה הנחיית מהדר תקינה וגם הערה, אם כי מתכנתים *שפויים* יעדיפו ככל הנראה להפריד בין הנחיות המהדר לבין ההערות.

הנחיות המהדר החוקיות מפורטות בתיעוד ו/או בעזרה של המהדר. ככלל, הן משפיעות על האופן בו המהדר יוצר קוד מכונה והן ספציפיות למהדר. בכל מקרה הן אינן מהוות חלק משפת התכנות עצמה, והן מתקדמות מכדי לכסותן בספר זה.

השימוש באותיות רישיות

שלא כמו שפות תכנות אחרות, ביניהן כל אלה שמקורן בשפת C כגון C++, ג'אווה ו-C#, מהדר הפסקל מתעלם מבחירה באותיות רישיות או רגילות. מבחינתו, המזהים Myname, MyName, myname, myName ו-MYNAME הם כולם בדיוק אותו הדבר. לדעתי, חוסר רגישות שכזה הוא תכונה חיובית בהחלט, מכיוון ששימוש שגוי באותיות רישיות ולא-רישיות בשפות רגישות לנושא עלול לגרום לשגיאות תחביר ולטעויות אחרות קשות לאיתור[8].

עם זאת, יש לדבר מספר חסרונות. ראשית, עליכם להיות מודעים לכך שהמזהים הללו הם באמת אותו הדבר, כך שעליכם להימנע משימוש בהם כרכיבים שונים. שנית, עליכם לנסות ולהיות עקביים באופן השימוש באותיות רישיות, וזאת במטרה לשפר את הקריאות של הקוד.

המהדר אינו מחייב שימוש עקבי באותיות רישיות, אך זהו הרגל שכדאי לאמץ. אחת הגישות הנפוצות היא להשתמש באות רישית רק עבור האות הראשונה של כל מזהה. אם מזהה מורכב ממספר מילים עוקבות (לא ניתן להשתמש ברווחים בתוך שמות מזהים), יש להשתמש באות רישית עבור האות הראשונה של כל מילה:

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

שיטה זו מכונה לרוב בשם "Pascal-casing", כדי להבדילה מה-"Camel-casing" של ג'אווה ושל שפות אחרות שמקורן ב-C, בה משתמשים באות רישית עבור המילים הפנימיות אך מתחילים את שם המזהה באות רגילה, כגון:

```
myLongIdentifier
```

סימנים לבנים

רכיבים אחרים שהמהדר מתעלם מהם לחלוטין הם רווחים, תווי מעבר שורה וטאבים שאתם מוסיפים לקוד המקור. כל הרכיבים הללו ידועים בשם הכללי "סימנים לבנים" (White spaces). סימנים לבנים נועדו אך ורק לשיפור הקריאות של הקוד; הם אינם משפיעים על ההידור בשום צורה.

[8] בדלפי קיים מקרה יוצא דופן יחיד להתעלמות מרישיות של שפת פסקל: הפרוצדורה Register שנמצאת בחבילה (Package) של רכיב (Component) חייבת להתחיל ב-R רישית, בשל דרישות תאימות עם מהדר C++ Builder. כמובן, כאשר תבצעו הפניות למזהים שיוצאו משפות אחרות (כגון פונקציות של Win32 או מחלקות של דוט נט), ייתכן שיהיה עליכם להשתמש באותיות רישיות על פי הכללים המתאימים.

שלא כמו שפת BASIC המסורתית, פסקל מאפשרת לכתוב הצהרה על פני מספר שורות ו"לשבור" הוראה מורכבת לשתי שורות או יותר. החסרון של כתיבת הצהרות על פני יותר משורה אחת (לפחות בעיני מתכנתי BASIC רבים) הוא שעליכם לזכור להוסיף תו נקודה-פסיק לציון סוף של הצהרה, או ליתר דיוק, כדי להפריד אותה מזו שבאה אחריה. המגבלה היחידה על פיצול הצהרות תכנות למספר שורות היא שמחרוזת שמהווה יחידת תחביר אינה יכולה להשתרע על פני מספר שורות.

שוב, אין כללים קבועים לשימוש ברווחים ובהצהרות מרובות-שורות, רק מספר כללי אצבע:

- העורך של דלפי, ועורכים רבים אחרים, כוללים קו אנכי שניתן להציב במרחק ששים או שבעים תווים. אם אתם משתמשים בקו זה ומשתדלים לא לחצות אותו, קוד המקור שלכם ייראה טוב יותר כאשר תדפיסו אותו על נייר. אחרת, שורות ארוכות עלולות להישבר בכל נקודה שהיא בעת ההדפסה.
- כאשר פונקציה או פרוצדורה כוללות מספר פרמטרים, נהוג להציב את הפרמטרים בשורות נפרדות.
- ניתן להשאיר שורה ריקה לחלוטין לפני הערה, או לחלק קטע קוד ארוך לחלקים קטנים יותר. אפילו רעיון פשוט זה יכול לשפר את קריאות הקוד, הן על גבי המסך והן בעת ההדפסה.
- השתמשו ברווחים כדי להפריד בין הפרמטרים בקריאה לפונקציה, ואולי אף ברווח לפני הסוגריים הפותחים. כמו כן, הפרידו בין האופרנדים בביטוי. אני מכיר מספר מתכנתים שלא יסכימו עם הרעיונות הללו, אך אני מתעקש: רווחים הם בחינם; אינכם משלמים עבורם.

עיצוב נאה

ההצעה האחרונה בנוגע לשימוש בסימנים לבנים קשורה לעיצוב טיפוסי של קוד בשפת פסקל, המכונה בשם עיצוב נאה. הכלל הוא פשוט: בכל פעם שאתם כותבים פקודה מורכבת, הזיזו אותה שני רווחים פנימה (ולא באמצעות טאב, כפי שנוהגים לעשות מתכנתי C!), לימין ההצהרה הנוכחית. הצהרה מורכבת בתוך הצהרה מורכבת אחרת תוזח למרחק ארבעה רווחים, וכן הלאה:

```
if ... then
  הצהרה;

if ... then
begin
  הצהרה1;
  הצהרה2;
end;

if ... then
begin
  if ... then
    הצהרה1;
  הצהרה2;
```

```
end;
```

העיצוב לעיל מבוסס על עיצוב נאה, אך מתכנתים שונים מפרשים את ההנחיה הכללית בדרכים שונות. חלקם מיישרים את הצהרות ה-begin וה-end עם הקוד הפנימי, חלקם מזיחים את ה-begin וה-end פנימה ואת הקוד הפנימי עוד יותר, אחרים מציבים את ה-begin באותה השורה כמו תנאי ה-if (בסגנון דמוי C). זהו בעיקר עניין של טעם.

קיימים תוספי תוכנה לדלפי בהם תוכלו להשתמש כדי להמיר עיצוב של קוד מקור קיים לעיצוב הזחה לפי בחירתכם. עיצוב הזחה דומה משמש לעתים קרובות עבור רשימות של משתנים או של טיפוסי נתונים:

```
type
  Letters = set of Char;

var
  Name: string;
```

ההזחה משמשת גם עבור הצהרות שממשיכות משורה קודמת:

```
MessageDlg ('This is a message',
  mtInformation, [mbOk], 0);
```

כמובן, כל מוסכמה שכזו היא רק הצעה לשיפור הקריאות של הקוד עבור מתכנתים אחרים, ואילו המהדר מתעלם ממנה לגמרי. ניסיתי להשתמש בכלל זה בצורה עקבית בכל הדוגמאות וקטעי הקוד שבספר. קוד המקור, המדריכים למשתמש והדוגמאות בעזרה של דלפי משתמשים כולם בעיצוב דומה.

הבלטת תחביר

כדי להקל על קריאת וכתיבת קוד פסקל, העורך של דלפי ועורכים רבים אחרים כוללים מאפיין שנקרא הבלטת תחביר צבעונית. בהתאם למשמעות בשפת פסקל של המילים שאתם מקלידים בעורך, הוא יציג אותן בצבעים שונים. כברירת מחדל, מילות מפתח הן מודגשות, מחרוזות והערות הן צבעוניות (ולרוב גם נטויות), וכן הלאה.

מילים שמורות, הערות ומחרוזות הן ככל הנראה שלושת הרכיבים שנהנים ביותר ממאפיין זה. כך תוכלו לזהות במבט אחד מילת מפתח שהוקלדה באופן שגוי, מחרוזת שלא נסגרה היטב, ואת מספר השורות שנכללות בהערה.

בדלפי תוכלו להתאים בקלות את הגדרות הבלטת התחביר באמצעות הדף "צבעי עורך" (Editor Colors) שבתיבת הדו-שיח "אפשרויות סביבה" (Environment Options). אם אתם היחידים שמשתמשים במחשב למטרות תכנות בפסקל, בחרו את הצבעים כרצונכם. לעומת זאת, אם אתם עובדים בשיתוף עם מתכנתים אחרים, כדאי לכם להסכים במשותף על סכמת צביעה אחידה. לי עצמי קשה מאד לעבוד במחשב עם צביעת תחביר שונה מזו שאני רגיל לה.

זיהוי שגיאות ומסייעי קוד

הגרסאות העדכניות של עורך דלפי כוללות מאפיינים רבים נוספים שיסייעו לכם בכתיבת קוד תקין. המאפיין הבולט ביותר הוא זיהוי השגיאות (Error Insight), שמציב זיגזג אדום מתחת לרכיבי קוד מקור שהוא אינו מבין, בדומה לאופן בו מעבד תמלילים מסמן שגיאות כתיב.

מאפיינים אחרים, כגון השלמת קוד (Code Completion), מסייעים בכתיבת הקוד בכך שהם מספקים רשימה של סמלים חוקיים עבור המיקום בו אתם כותבים. עם זאת, מדובר במאפיינים ספציפיים לעורך ואינני מעוניין להיכנס לפרטיהם, מכיוון שאני רוצה להתמקד בשפת התכנות ולא בעורך של דלפי (אף על פי שהעורך של דלפי הוא אחד הכלים הנפוצים ביותר לכתיבת קוד בפסקל).

הצהרות שפה

לאחר שהגדרתם מספר מזהים, תוכלו להשתמש בהם בהצהרות וגם בביטויים שמהווים חלק מהצהרה. שפת פסקל כוללת מספר סוגי הצהרות וביטויים. ראשית, בואו ונעיף מבט על מילות מפתח, ביטויים ואופרטורים.

מילות מפתח

מילות מפתח הן כל המזהים ששמורים על ידי שפת פסקל (או Object Pascal). מדובר על סמלים שיש להם משמעות ותפקיד מוגדרים מראש. העזרה של דלפי מבדילה בין מילים שמורות לבין הנחיות: מילים שמורות אינן יכולות לשמש בתור מזהים, ואילו בהנחיות אין להשתמש כמזהים, אף על פי שהמהדר יקבל שימוש כזה. בפועל, עליכם להימנע משימוש בכל מילת מפתח שהיא בתור מזהה.

להלן רשימה מלאה של המזהים השמורים, כולל מילות מפתח ומילים שמורות אחרות. לכמה מהם יש יותר ממשמעות אחת, חלקם נפוצים מאד ואחרים נדירים למדי. אפילו אם אתם מתכנתים מנוסים בדלפי, ייתכן שתגלו אחד או שניים שמעולם לא שמעתם עליהם. חפשו אותם בקובצי העזרה!

```
absolute        abstract        and
array           as              asm
assembler       at              automated
begin           case            cdecl
class           const           constructor
contains        default         destructor
dispid          dispinterface   div
do              downto          dynamic
else            end             except
export          exports         external
far             file            finalization
finally         for             forward
function        goto            if
implementation  implements      in
```

```
index           inherited       initialization
inline          interface       is
label           library         message
mod             name            near
nil             nodefault       not
object          of              on
or              out             overload
override        package         packed
pascal          private         procedure
program         property        protected
public          published       raise
read            readonly        record
register        reintroduce     repeat
requires        resident        resourcestring
safecall        set             shl
shr             stdcall         stored
string          then            threadvar
to              try             type
unit            until           uses
var             virtual         while
with            write           writeonly
xor
```

הניב FreePascal כולל מספר מילים שמורות נוספות[9]:

```
dispose         exit        false
new             true
```

ערכים ליטרליים

ערך ליטרלי (Literal) הינו ערך שאתם מקלידים ישירות לתוך קוד המקור של התוכנית. לדוגמה, אם אתם זקוקים למספר שערכו שתיים, אתם פשוט כותבים:

```
2
```

זהו הערך הליטרלי עבור מספר שלם. אם אתם מעוניינים באותו ערך כמספר עשרוני, הוסיפו ספרה עשרונית ריקה אחריו:

```
2.0
```

ערכים ליטרליים אינם מוגבלים למספרים. הם יכולים להיות גם תווים ומחרוזות. אלה מסומנים בתו גרש בודד מכל אחד מצדדיהם:

```
// תו ליטרלי
'K'

// מחרוזת ליטרלית
'Marco'
```

באפשרותכם לציין תווים גם באמצעות קוד ה-ASCII שלהם, ולפניו התו #. את זאת אציג ביתר פירוט בסעיף שמדבר על טיפוס הנתונים Char בפרק הבא.

[9] מקור: http://www.freepascal.org/docs-html/ref/refsu3.html

אם אתם זקוקים לתו הגרש בתוך מחרוזת, יהיה עליכם להכפיל אותו. כך, כשאני רוצה לכתוב את שמי באנגלית (בה הוא מאוית עם גרש בסוף), אני כותב:

```
'Marco Cantu'''
```

שני תווי הגרש מייצגים גרש בודד בתוך המחרוזת, ואילו הגרש השלישי שאחריהם מסמן את סוף המחרוזת. שימו לב שיש לכתוב את המחרוזת הליטרלית כולה בשורה אחת.

ביטויים ואופרטורים

אין כלל גורף לבניית ביטויים, מכיוון שהם תלויים בעיקר באופרטורים שבהם נעשה שימוש, ושפת פסקל כוללת מספר אופרטורים. אלו הם אופרטורים לוגיים, אריתמטיים, בוליאניים ויחסיים, וכן אופרטורים של קבוצות ועוד כמה:

```
// ביטויים לדוגמה
20 * 5
30 + n
a < b
c = 10
```

רוב שפות התכנות כוללות ביטויים. ביטוי הוא כל צירוף חוקי של קבועים, משתנים, ערכים ליטרליים, אופרטורים ותוצאות של פונקציות. ביטויים יכולים לשמש לצורך קביעה של הערך שמשתנה יקבל, לצורך חישוב של פרמטר עבור פונקציה או פרוצדורה, או לצורך בדיקה של תנאי. בכל פעם שאתם מבצעים פעולה על ערך של מזהה במקום להשתמש במזהה עצמו, אתם משתמשים למעשה בביטוי.

הצגת תוצאה של ביטוי (תוכנית ראשונה)

אם ברצונכם להתנסות מעט בביטויים, הדרך הטובה ביותר היא לכתוב תוכנית פשוטה. כמו ברוב הדוגמאות בספר זה, ניצור יישום של מסוף בקרה (Console Application) ונשתמש בהצהרות עם הפרוצדורה Writeln כדי להציג דברים על מסך הפלט של המסוף[10]. בסיום התוכנית מקובל להוסיף קריאה לפרוצדורה Readln, כדי שהתוכנית תמתין עד שתלחצו על המקש אנטר (Enter) ולא תיסגר מייד – אחרת לא תספיקו לראות את הפלט.

להלן הקוד השלם עבור התוכנית לדוגמה, ששמה EPExpressions:

```
program EPExpressions;

{$APPTYPE CONSOLE}

begin
  writeln (20 * 5);
```

[10] בשפת פסקל, פרמטרים שמועברים לפונקציות או לפרוצדורות מוקפים בסוגריים. שפות אחרות, ביניהן Rebol ו-Ruby, מאפשרות לכם להעביר פרמטרים פשוט באמצעות כתיבתם אחרי שם הפונקציה או הפרוצדורה.

```
  writeln (30 + 222);
  writeln (3 < 30);
  writeln (12 = 10);

  readln;
end.
```

מעתה והלאה אראה רק את הקוד הרלוונטי ואדלג על חלק מהפרטים כגון הקריאה ל-readln והנחיית המהדר APPTYPE. עם זאת, דוגמאות קוד המקור הזמינות להורדה יהיו תוכניות שלמות ומוכנות להרצה. הנה הפלט שמתקבל מהרצת התוכנית:

```
100
252
TRUE
FALSE
```

אופרטורים וקדימות

אם כתבתם אי-פעם תוכנית, אתם כבר יודעים מהו ביטוי, מכיוון שביטויים הם אבני הבניין הבסיסיות של כל שפת תכנות. כאן אדגיש את הרכיבים הספציפיים של האופרטורים של שפת פסקל. להלן רשימה של האופרטורים הללו, כשהם מקובצים על פי קדימות:

אופרטורים אונריים (הקדימות הגבוהה ביותר)

@	כתובת של משתנה או של פונקציה (מחזיר מצביע)
not	פעולת not בוליאנית, או שמבוצעת על ביטים

אופרטורים של הכפלה ואופרטורים של פעולות על ביטים

*	כפל אריתמטי או חיתוך קבוצות
/	חילוק של מספרים עשרוניים
div	חילוק של מספרים שלמים
mod	שארית (מחילוק מספרים שלמים)
as	המרת טיפוסים בטוחה בזמן ריצה
and	פעולת and בוליאנית, או שמבוצעת על ביטים
shl	הזזה שמאלה של ביטים
shr	הזזה ימינה של ביטים

אופרטורים של הוספה

+	חיבור אריתמטי, איחוד קבוצות, שרשור מחרוזות, תוספת להסטה של מצביע
-	חיסור אריתמטי, הבדל בין קבוצות, החסרה מהסטה של מצביע
or	פעולת or בוליאנית או שמבוצעת על ביטים
xor	פעולת Exclusive or בוליאנית או שמבוצעת על ביטים

אופרטורים של יחס והשוואה (הקדימות הנמוכה ביותר)

=	בדיקת שוויון
<>	בדיקת אי-שוויון
>	בדיקת "קטן מ-"
<	בדיקת "גדול מ-"
<=	בדיקת "קטן או שווה ל-", או בדיקה האם תת-קבוצה מוכלת בקבוצה
>=	בדיקת "גדול או שווה ל-", או בדיקה האם קבוצה מכילה תת-קבוצה
in	בדיקת הימצאות של פריט בקבוצה

is בדיקת תאימות של אובייקט להגדרת הטיפוס שצוינה (בשימוש רק בתכנות מונחה עצמים)

בניגוד לרוב שפות התכנות האחרות, לאופרטורים and ו-or יש קדימות גבוהה יותר מאשר לאופרטורים של השוואה. לכן, אם תכתבו:

```
a < b and c < d
```

המהדר יבצע ראשית כל את פעולת ה-and, והביטוי כולו יגרום לשגיאת מהדר. לכן יש לסגור את כל אחד מביטויי ה-"<" בסוגריים:

```
(a < b) and (c < d)
```

לכמה מהאופרטורים הנפוצים יש משמעויות שונות כאשר משתמשים בהם עם טיפוסי נתונים שונים. לדוגמה, האופרטור "+" יכול לשמש לחיבור של שני מספרים, לשרשור של שתי מחרוזות, לאיחוד שתי קבוצות ואפילו להוספת הסטה למצביע מטיפוס PChar. עם זאת, לא ניתן לחבר שני תווים כפי שאפשר לעשות בשפת C.

אופרטור יוצא דופן נוסף הוא div. בפסקל אפשר לחלק כל זוג מספרים (עשרוניים או שלמים) בעזרת האופרטור "/", והתוצאה תהיה תמיד מטיפוס מספר עשרוני. אם אתם צריכים לחלק שני מספרים שלמים ומעוניינים בתוצאה שגם היא מספר שלם, השתמשו במקום זאת באופרטור div. הנה שתי השמות ערכים לדוגמה (קוד זה יהיה ברור יותר כאשר נכסה את נושא טיפוסי הנתונים בפרק הבא):

```
realVal := 123 / 12;
integerVal := 123 div 12;
```

אופרטורים של קבוצות

אופרטורי הקבוצות כוללים איחוד (+), הבדל (-), חיתוך (*), בדיקת הימצאות (in) ומספר אופרטורי יחס. כדי להוסיף פריטים לקבוצה, אתם יכולים לבצע איחוד של הקבוצה עם קבוצה אחרת שמכילה רק את הפריטים להם אתם זקוקים. הנה דוגמה בדלפי שעוסקת בסגנונות גופן:

```
Style := Style + [fsBold];
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

לחלופין, תוכלו להשתמש בפרוצדורות Include ו-Exclude הסטנדרטיות, שהן יעילות הרבה יותר, אם כי לא ניתן להשתמש בהן על תכונות (Properties) מטיפוס קבוצה ששייכות לרכיבים (Components):

```
Include (Style, fsBold);
```

סיכום

כעת, כשאנו מכירים את המבנה הבסיסי של תוכנית בשפת פסקל, אנו מוכנים להתחיל לחקור את משמעותה. נפתח בהסתכלות על טיפוסי נתונים מוגדרים מראש וכאלה שמוגדרים על ידי המשתמש, ולאחר מכן נתחיל להשתמש במילות מפתח כדי לבנות הצהרות תכנותיות.

פרק 3: טיפוסים, משתנים וקבועים

שפת פסקל המקורית הביאה לעולם מספר תפיסות חדשות, שהפכו מאז נפוצות למדי בשפות התכנות. התפיסה המהפכנית-דאז הראשונה היתה של *טיפוס נתונים* (Data Type). הטיפוס קובע את הערכים שהמשתנה יכול להחזיק, ואת הפעולות שניתן לבצע עליו.

רעיון הטיפוס חזק יותר בפסקל מאשר ב-C, שם טיפוסי הנתונים האריתמטיים הם כמעט ברי החלפה, והרבה יותר חזק מאשר ב-BASIC בה אין שום תפיסה דומה. זוהי הסיבה לכך שמתכנתים מתייחסים לפסקל בתור שפה בעלת "טיפוסיות חזקה" (Strongly typed).

משתנים

שפת פסקל דורשת שכל המשתנים יוכרזו לפני שנעשה בהם שימוש. בכל פעם שאתם מכריזים (Declare) על משתנה, עליכם לציין עבורו טיפוס נתונים. הנה מספר הכרזות משתנים לדוגמה:

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

מילת המפתח var יכולה להופיע במספר מקומות בתוכנית, כגון בתחילת פונקציה או פרוצדורה, לצורך הכרזה על משתנים מקומיים של רוטינה, או בתוך יחידה (Unit) לצורך הכרזה על משתנים גלובליים. לאחר מילת המפתח var מופיע רשימה של שמות משתנים, ואחריהם נקודתיים ושם טיפוס הנתונים. ניתן לכתוב יותר משם משתנה יחיד בשורה אחת, כפי שנעשה עם A ו-B בהצהרה האחרונה בקטע בקוד לעיל.

לאחר שהגדרתם משתנה מטיפוס כלשהו, תוכלו לבצע עליו רק את הפעולות שנתמכות על ידי אותו טיפוס נתונים. לדוגמה, תוכלו להשתמש בערך בוליאני בבדיקת תנאי, ובערך של מספר שלם בביטוי מספרי. אינכם יכולים לערבב בוליאנים ומספרים שלמים (כפי שניתן לעשות בשפת C).

בעזרת השמות פשוטות אנו יכולים לכתוב את הקוד הבא (חלק מהתוכנית לדוגמה Variables[11]):

[11] התוכנית לדוגמה Variables כוללת את הכרזות המשתנים ואת ההשמות שמפורטות בסעיף זה, וכן מספר הצהרות writeln להצגת דבר-מה על גבי המסך.

```
Value := 10;
IsCorrect := True;
```

בהינתן הכרזות המשתנים הקודמות, שתי השמות אלה הן חוקיות. ההצהרה הבאה, לעומת זאת, אינה חוקית מכיוון שלשני המשתנים יש טיפוסי נתונים שונים:

```
Value := IsCorrect; // שגיאה
```

אם תנסו להדר את הקוד הזה, המהדר יפיק שגיאה עם תאור כמו זה:

```
[DCC Error]: Incompatible types: 'Integer' and 'Boolean'
```

ברוב המקרים, שגיאות שכאלה הן שגיאות תכנות, מכיוון שאין הגיון במתן ערך True או False למשתנה מטיפוס הנתונים Integer (מספר שלם). אל תאשימו את המהדר בשגיאות כאלה. הוא רק מתריע בפניכם שיש בעיה בקוד שכתבתם.

כמובן, לעתים קרובות ניתן להמיר את ערכו של משתנה מטיפוס אחד לאחר. במקרים מסוימים ההמרה היא אוטומטית, אך בדרך כלל יהיה עליכם לקרוא לפונקצית מערכת ספציפית שתשנה את הייצוג הפנימי של הנתונים.

בשפת פסקל ניתן להקצות ערך התחלתי למשתנה גלובלי בעת ההכרזה. לדוגמה, אתם יכולים לכתוב:

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

טכניקת אתחול זו עובדת רק עבור משתנים גלובליים, לא עבור משתנים שמוכרזים בתוך פרוצדורה או פונקציה.

קבועים

פסקל מאפשרת גם הכרזה של קבועים, שמאפשרים לכם להעניק שמות בעלי משמעות לערכים שאינכם רוצים שישתנו במהלך הרצת התוכנית. כדי להכריז על קבוע אין צורך לציין טיפוס נתונים, רק להקצות ערך התחלתי. המהדר יתבונן בערך ויבחר באופן אוטומטי בטיפוס הנתונים המתאים. הנה מספר הכרזות לדוגמה (מהתוכנית לדוגמה EPConstants):

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantu';
```

שפת פסקל קובעת את טיפוס הנתונים של הקבוע בהתבסס על ערכו. בדוגמה שלמעלה, הקבוע Thousand מפורש כשייך לטיפוס SmallInt, שהוא טיפוס המספר השלם הקטן ביותר

שיכול להכיל את הערך 1000. אם ברצונכם לומר לפסקל להשתמש בטיפוס ספציפי, אתם יכולים פשוט להוסיף את שם הטיפוס להכרזה, כמו כאן:

```
const
  Thousand: Integer = 1000;
```

כאשר אתם מכריזים על קבוע, המהדר בוחר אם להקצות מקום בזכרון המחשב ולשמור שם את הערך, או לשכפל את הערך עצמו בכל מקום בתוכנית בו נעשה שימוש בקבוע. הגישה השניה הגיונית במיוחד כאשר מדובר בקבועים פשוטים.

בדומה לטורבו פסקל, גרסת 16 ביט של דלפי אפשרה לשנות בזמן הריצה את הערך של קבוע שהוגדר עבורו טיפוס, כאילו היה משתנה. גרסאות 32 ביט מתירות התנהגות זו לצורך שמירה על תאימות לאחור אם מפעילים את הנחיית המהדר J$, או בוחרים בתיבת הסימון Assignable typed constants בדף Compiler שבתיבת הדו-שיח Project Options. הגדרה זו הופעלה כברירת מחדל עד לדלפי 6, אך בכל מקרה מומלץ מאד לא להשתמש בתעלול זה כטכניקת תכנות. השמת ערך חדש לקבוע מונעת כל אפשרות לאופטימיזציה של קבועים על ידי המהדר. במקרים כאלה, פשוט הכריזו על משתנה במקום על קבוע.

קבועי מחרוזות משאב (Resource String)

כאשר אתם מגדירים קבוע שהוא מחרוזת, במקום לכתוב הכרזת קבוע רגילה, אתם יכולים להשתמש בהנחיה ספציפית – resourcestring, שמורה למהדר ולמקשר (linker) להתייחס למחרוזת כאילו היתה משאב של מערכת ההפעלה Windows:

```
const
  sAuthorName = 'Marco';

resourcestring
  strAuthorName = 'Marco';
```

בשני המקרים אתם מגדירים קבוע, כלומר ערך שאינו משתנה במהלך הרצת התוכנית. ההבדל הוא רק ביישום: קבוע מחרוזת שהוגדר בעזרת ההנחיה resourcestring יאוחסן בין משאבי התוכנית, בטבלת מחרוזות.

כדי לראות את היכולת הזו בפעולה, הסתכלו על הדוגמה EPConstants שכוללת את הקוד הבא:

```
resourcestring
  strAuthorName = 'Marco Cantù';
  strBookName = 'Essential Pascal';

begin
  writeln (strBookName + ' ' + strAuthorName);
```

פלט שתי המחרוזות יופיע עם רווח ביניהן. הדבר המעניין בתוכנית זו הוא שאם תבחנו אותה בעזרת סייר משאבים (resource explorer), כגון זה שמסופק בין שאר הדוגמאות שמגיעות

עם דלפי, תוכלו לראות את המחרוזות החדשות בין המשאבים. פירוש הדבר שהמחרוזות אינן חלק מהקוד המהודר, אלא מאוחסנות במיקום נפרד של קובץ ההרצה (ה-EXE)[12].

במילים פשוטות, היתרונות של השימוש במשאבים הם ניהול יעיל יותר של הזכרון על ידי Windows, וסיוע לתהליך הלוקליזציה של התוכנה (תרגום המחרוזות לשפה אחרת) בכך שאין צורך לשנות את קוד המקור שלה. ככלל אצבע, עליכם להשתמש ב-resourcestring עבור כל טקסט שמוצג בפני המשתמשים ושעשוי לעבור תרגום, ולהשתמש בקבועים פנימיים עבור כל מחרוזת פנימית אחרת של התוכנה, כגון שם של קובץ הגדרת תצורה.

טיפוסי נתונים

בשפת פקסל קיימים מספר טיפוסי נתונים מוגדרים מראש, אותם ניתן לחלק לשלוש קבוצות: *טיפוסים סודרים* (Ordinal), *טיפוסים ממשיים* (Real) ו*מחרוזות*. בסעיפים הבאים נדון בטיפוסים הסודרים והממשיים, ועל המחרוזות נדבר מאוחר יותר[13].

דלפי כוללת גם טיפוס נתונים *נטול טיפוס* (Non-typed) בשם Variant (וריאנט), עליו נדבר בפרק 10. זהו טיפוס משונה שלא מתבצעת עבורו בדיקת טיפוסים הולמת. הוא הופיע לראשונה בדלפי 2 כדי לנהל אוטומציית OLE של Windows, אך מצא את דרכו גם לאזורים אחרים בספריות של דלפי.

טיפוסים סודרים

טיפוסים סודרים מבוססים על התפיסה של סדר, או רצף. אתם יכולים לא רק להשוות שני ערכים כדי לבדוק מי מהם גבוה יותר, אלא גם לקבל את הערך הבא או הקודם עבור כל ערך נתון, ולחשב את הערכים הנמוכים והגבוהים ביותר האפשריים.

שלושת הטיפוסים הסודרים המוגדרים-מראש החשובים ביותר הם Integer (מספר שלם), Boolean (בוליאני) ו-Char (תו, מהמילה Character). קיימים טיפוסים אחרים הקשורים אליהם ובעלי אותה משמעות, אך יש להם ייצוג פנימי אחר והם תומכים בטווחי ערכים שונים. הטבלה הבאה מפרטת את טיפוסי הנתונים הסודרים המשמשים לייצוג מספרים:

גודל	עם סימן (Signed)	ללא סימן (Unsigned)
8 ביטים	ShortInt: מ-128- ועד 127	Byte: מ-0 ועד 255
16 ביטים	SmallInt: מ-32,768- ועד 32,767	Word: מ-0 ועד 65,535
16/32 ביטים	Integer	Cardinal
32 ביטים	LongInt: מ-2,147,483,648- ועד 2,147,483,647	LongWord: מ-0 ועד 4,294,967,295

[12] זה יישמע מוזר, אך הגדרת resourcestring זמינה גם ב-Kylix.

[13] אני אציג גם מספר טיפוסים שמוגדרים על ידי ספריות דלפי (ולא מוגדרים מראש על ידי המהדר), אשר ניתן להתייחס אליהם כאל טיפוסים מוגדרים מראש לכל מטרה מעשית.

גודל	עם סימן (Signed)	ללא סימן (Unsigned)
64 ביטים	Int64: מ-9,223,372,036,854,775,808 ועד 9,223,372,036,854,775,807	-

כפי שאפשר לראות, טיפוסים אלה תואמים לייצוגים שונים של מספרים התלויים במספר הביטים שמשמשים כדי לבטא את הערך, וכן בהימצאותו או בהיעדרו של ביט סימן. ערכים בעלי סימן יכולים להיות חיוביים או שליליים, אך טווח הערכים שלהם קטן יותר מכיוון שחסר לערך המספרי אותו ביט שמשמש לסימן. אתם יכולים להסתכל בדוגמה EPRange, המוזכרת בסעיף הבא, כדי לראות את טווח הערכים בפועל של כל טיפוס.

הקבוצה המסומנת כגודל 16/32 ביט מציינת ערכים שיש להם ייצוגים שונים בגרסאות 16 ביט ו-32 ביט של דלפי. ב-Integer וב-Cardinal נעשה שימוש לעתים קרובות מכיוון שהם תואמים לייצוג הטבעי של מספרים במעבד הראשי של המחשב. מאוחר יותר אפרט את היתרונות של השימוש ב-Integer וב-Cardinal.

טיפוסי מספרים שלמים

בדלפי 2 ו-3, המספרים חסרי הסימן בגודל 32 ביט שנקראו Cardinal היו למעשה בעלי 31 ביטים בלבד, עם ערך קצה שיכול לציין, למשל, כתובות בזכרון עד 2 גיגהבייט. לדלפי 4 התווסף טיפוס מספרי חדש ללא סימן, LongWord, שהיה 32 ביט הלכה למעשה והגיע עד 4 גיגהבייט. כיום, הטיפוס Cardinal הוא כינוי אחר לטיפוס LongWord. הטיפוס LongWord מאפשר, כאמור, הפניה לכתובות של שני גיגהבייט נוספים של נתונים באמצעות מספר ללא סימן. יתרה מזאת, הוא תואם לייצוג הטבעי של המספרים במעבד הראשי.

טיפוס חדש נוסף שהתווסף לדלפי 4 הוא הטיפוס Int64, שמייצג מספרים שלמים בעלי עד 18 ספרות. לטיפוס חדש זה יש תמיכה מלאה בחלק מהרוטינות המוקדשות לטיפוסים סודרים (כגון High ו-Low), ברוטינות הנומריות (כגון Inc ו-Dec) וברוטינות המרת המחרוזות (כגון IntToStr).[14]

בוליאני

רק לעתים רחוקות ביותר משתמשים בערכים בוליאניים שאינם מטיפוס Boolean. לעתים, פונקציות API (וספריות COM) של Windows מצריכות ערכים בוליאניים בייצוגים ספציפיים. טיפוסים אלה נקראים ByteBool, WordBool ו-LongBool.

בדלפי 3, במטרה ליצור תאימות ל-Visual Basic ולאוטומציית OLE, טיפוסי הנתונים ByteBool, WordBool ו-LongBool שונו כך שייצגו את הערך True בעזרת המספר 1-, בעוד שהערך False נותר 0. טיפוס הנתונים Boolean לא השתנה (True הוא 1, False הוא 0), אם כי בכל מקרה המספרים בפועל לא אמורים להיות רלוונטיים ואין לנצל אותם לרעה (כפי שנעשה בשפת C).

[14] קיימות גם פונקציות שמסוגלות להמיר מחרוזות למספרים שלמים בני 64 ביטים. מאוחר יותר בספר אתאר העברה של נתונים ממחרוזות למספרים שלמים ולהפך.

תווים

לסיום, ישנם שני ייצוגים שונים של תווים: ANSIChar ו-WideChar. הטיפוס הראשון מייצג תווים ב-8 ביטים, התואמים לקבוצת התווים ANSI שמשמשת באופן מסורתי ב-Windows. הטיפוס השני מייצג תווים ב-16 ביטים, התואמים לתווי Unicode החדשים יחסית שנתמכים, בנוסף על התווים המסורתיים, בגרסאות עדכניות של Windows.

ברוב המקרים אתם תשתמשו בפשטות בטיפוס Char, שתואם לטיפוס ANSIChar מאז דלפי 3 ועד דלפי 2007. בכל מקרה, זכרו ש-256 התווים הראשונים ב-Unicode תואמים בדיוק לתווים של ANSI.

אפשר לייצג תווים קבועים באמצעות הייצוג הסמלי שלהם, לדוגמה 'k', או בייצוג מספרי כגון #78. את הייצוג המספרי אפשר לבטא גם בעזרת הפונקציה Chr כך: Chr(78). את ההמרה בכיוון ההפוך אפשר לבצע בעזרת הפונקציה Ord. ככלל, מוטב להשתמש בייצוג הסמלי כאשר מציינים אותיות, ספרות או סמלים.

לעומת זאת, כאשר מתייחסים לתווים מיוחדים, כגון אלה שקטנים מ-#32, משתמשים לרוב בייצוג המספרי. הרשימה הבאה כוללת כמה מהתווים המיוחדים הנפוצים ביותר:

#8	Backspace (חזרה אות אחת לאחור)
#9	Tabulator ("טאב")
#10	Newline (מעבר שורה)
#13	Carriage Return (חזרה לתחילת השורה)
#27	Escape (כמו בלחצן Esc)

הצגת טווחים סודרים

כדי לתת לכם מושג על הטווחים השונים של כמה מהטיפוסים הסודרים כתבתי תוכנית דלפי פשוטה בשם EPRange. התוכנית מציגה את השם, הגודל והטווח של מספר טיפוסי נתונים, ומפרידה בין הערכים ששייכים לאותו טיפוס נתונים בעזרת טאבים:

```
program EPRange;

{$APPTYPE CONSOLE}

begin
  write ('Integer');
  write (#9);
  write (SizeOf (Integer));
  write (#9);
  write (Low (Integer));
  write (#9);
  write (High (Integer));
```

```
  writeln;

  write ('SmallInt');
  write (#9);
  write (SizeOf (SmallInt));
  write (#9);
  write (Low (SmallInt));
  write (#9);
  write (High (SmallInt));
  writeln;

  write ('Int64');
  write (#9);
  write (SizeOf (Int64));
  write (#9);
  write (Low (Int64));
  write (#9);
  write (High (Int64));
  writeln;

  write ('Char');
  write (#9);
  write (SizeOf (Char));
  write (#9);
  write (Ord(Low (Char)));
  write (#9);
  write (Ord(High (Char)));
  writeln;

  readln;
end.
```

הקוד חוזר על עצמו במידת מה, וניתן היה לכתוב אותו בצורה יפה הרבה יותר[15], אך לא רציתי להציג יותר מדי מושגים בעת ובעונה אחת.

והנה הפלט (מעוצב מעט מחדש לצורך הבהירות):

```
Integer     4     -2147483648 2147483647
SmallInt    2     -32768      32767
Int64       8     -9223372036854775808
                              9223372036854775807
Char        1     0           255
```

התוכנית משתמשת בשלוש פונקציות: SizeOf, High ו-Low. התוצאה של פונקציית SizeOf היא מספר שלם שמציין את מספר הבתים (Bytes) הדרושים לייצוג ערכים בטיפוס הנתון.

התוצאות של שתי הפונקציות האחרות הן סודרים, מאותו הטיפוס כמו של הערך שהועבר אליהן, שמציינים את טווח הערכים החוקי שהטיפוס עצמו מסוגל לייצג. כדי להציג את הטווח של הטיפוס Char, התוכנית הנ"ל ממירה את התווים לייצוגים המספריים שלהם באמצעות Ord, מכיוון שהתו 0# אינו ניתן להדפסה ואילו התו 255# הוא סימן לבן.

הגודל של הטיפוס Integer משתנה בהתאם למעבד הראשי ולמערכת ההפעלה בהם אתם משתמשים. ב-Windows 16 ביטים (לדוגמה, בעת שימוש בדלפי 1), גודלו של משתנה

[15] בדלפי קיימת תמיכה נרחבת במידע טיפוסים של זמן ריצה (RTTI), בו תוכלו להשתמש כדי לבצע פעולות ישירות על טיפוסי נתונים בזמן ריצה. אך נושא זה, כפי שאתם יכולים לנחש, מתקדם למדי ונמצא הרחק מעבר להיקף החומר של ספר זה.

Integer הוא שני בתים. ב-Windows 32 ביטים (כל הגרסאות של דלפי מאז עד היום), גודלו הוא ארבעה בתים.

הייצוגים השונים של הטיפוס Integer אינם מהווים בעיה כל עוד התוכנית שלכם אינה מניחה הנחות כלשהן בנוגע לגודלם של מספרים שלמים. עם זאת, אם במקרה שמרתם Integer בקובץ באמצעות גרסה אחת וטענתם אותו בחזרה באמצעות גרסה אחרת, הדבר ייצור בעיות. במקרה כזה, עליכם לבחור בטיפוס משתנים שאינו תלוי בפלטפורמה (כגון LongInt, SmallInt או Int64).

לצורך חישובים מתמטיים או קוד גנרי, מוטב להיצמד לייצוג ה-Integer הרגיל של הפלטפורמה הספציפית – כלומר, להשתמש בטיפוס Integer – מכיוון שזהו הטיפוס המועדף על המעבד הראשי.

בעת טיפול במספרים שלמים, הטיפוס Integer צריך להיות הבחירה הראשונה שלכם. השתמשו בייצוגים אחרים רק אשר יש לכם סיבה טובה לעשות זאת.

רוטינות של טיפוסים סודרים

קיימות מספר רוטינות מערכת (רוטינות שמוגדרות ביחידה System בדלפי ובשפת פסקל) שפועלות על טיפוסים סודרים. הטבלה הבאה מציגה אותן:

Dec	מקטינה את ערך המשתנה שהועבר כפרמטר, באחד או בערך של הפרמטר השני האופציונלי.
Inc	מגדילה את ערך המשתנה שהועבר כפרמטר, באחד או בערך שצוין[16].
Odd	מחזירה True אם הארגומנט הוא אי-זוגי
Pred	מחזירה את הערך שלפני הארגומנט, על פי הסדר שקובע טיפוס הנתונים.
Succ	מחזירה את הערך שאחרי הארגומנט, "הבא בתור".
Ord	מחזירה מספר שמציין את המיקום של הארגומנט בקבוצת הערכים של טיפוס הנתונים.
Low	מחזירה את הערך הנמוך ביותר בטווח של טיפוס הסודר שהועבר כפרמטר.
High	מחזירה את הערך הגבוה ביותר בטווח של טיפוס הסודר שהועבר כפרמטר.

שימו לב שכמה מהרוטינות הללו, כאשר מחילים אותן על קבועים, מוערכות באופן אוטומטי על ידי המהדר ומוחלפות בערכן. לדוגמה, אם תכתבו את הקריאה High(x) כאשר x מוגדר כ-Integer, המהדר יחליף את הביטוי בערך הגבוה ביותר האפשרי עבור טיפוס הנתונים Integer.

[16] על מתכנתי C++ לשים לב ששתי הגרסאות של הפרוצדורה Inc, עם פרמטר יחיד או שניים, מקבילות לאופרטורים ++ ו-+= (אותו הדבר נכון לגבי הפרוצדורה Dec).

טיפוסים ממשיים

טיפוסים ממשיים מייצגים מספרים בעלי נקודה עשרונית במגוון פורמטים. הנה רשימה של טיפוסי נתונים בעלי נקודה עשרונית:

Single	גודל האחסון הקטן ביותר הוא במספרים מטיפוס Single, שמייצגים ערכים בני 4 בתים.
Double	אלו הם מספרים בעלי נקודה עשרונית שמיושמים ב-8 בתים.
Extended	אלו הם מספרים שמיושמים ב-10 בתים.

שלושת אלה הם טיפוסי נתונים בעלי נקודה עשרונית, ברמות דיוק שונות, שתואמים לייצוגי הספרות העשרוניות התקניים של IEEE ונתמכים ישירות על ידי המעבד הראשי להשגת מהירות מירבית.

בדלפי 2 ו-3, לטיפוס Real היתה אותה הגדרה כמו בגרסת ה-16 ביטים: היה זה טיפוס בן 48 ביטים. אלא ש-Borland הפסיקה לתמוך בשימוש בו והציעה להשתמש, במקומו, בטיפוסים Single, Double ו-Extended. הסיבה להצעה זו היתה שהפורמט הישן בן 6 הבתים לא נתמך על ידי המעבדים של אינטל ולא נכלל בין הטיפוסים הממשיים הרשמיים של IEEE. כדי להתגבר על הבעיה לגמרי, דלפי 4 שינתה את הגדרת הטיפוס Real כך שייצג מספר תקני בעל נקודה עשרונית בן 8 בתים (64 ביטים)[17].

בנוסף על היתרון של השימוש בהגדרה תקנית, שינוי זה אפשר לרכיבים (Components) לפרסם תכונות שמבוססות על הטיפוס Real, דבר שלא היה אפשרי בדלפי 3. בין החסרונות ניתן למנות בעיות תאימות. במקרה הצורך, אפשר להתגבר על האפשרות של אי-תאימות באמצעות היצמדות להגדרת הטיפוס של דלפי 2 ו-3. את זאת מבצעים בעזרת אפשרות המהדר הבאה:

```
{$REALCOMPATIBILITY ON}
```

בנוסף, קיימים שני סוגי נתונים לא סודרים משונים[18]:

Comp	מתאר מספרים שלמים גדולים מאד באמצעות 8 בתים (שיכולים לאחסן מספרים בעלי 18 ספרות)
Currency	(לא זמין בדלפי 16 ביטים) מציין ערך בעל ארבע ספרות בדיוק אחרי הנקודה, בייצוג 64 ביטים זהה לזה של הטיפוס Comp. כפי ששמו מלמד, טיפוס הנתונים Currency הוסף כדי לטפל בערכים כספיים מדויקים מאד, עד ארבע ספרות אחרי הנקודה.

[17] למטרות תאימות לאחור עם הטיפוס Real של מהדרי פסקל ישנים יותר של Borland, הוצג טיפוס חדש בשם Real48.

[18] הן FreePascal והן GNU Pascal כוללים את טיפוס הנתונים Comp. ב-GNU Pascal הוא נחשב לטיפוס של מספר שלם (Integer) ובצדק. FreePascal כולל את טיפוס הנתונים Currency.

איננו יכולים לכתוב תוכנית דומה ל-EPRange עם טיפוסי נתונים ממשיים מכיוון שאי אפשר להפעיל על משתנים מטיפוסים ממשיים את הפונקציות High, Low או Ord. טיפוסים ממשיים מייצגים (בתאוריה) קבוצה בלתי מוגבלת של מספרים; ערכים סודרים מייצגים קבוצה קבועה של ערכים.

אסביר זאת באופן אחר. כאשר נתון לכם המספר השלם 23, אתם יכולים לקבוע מהו הערך הבא בתור. טיפוסי מספרים שלמים הם סופיים (יש להם טווח קבוע מראש וסדר). מספרים בעלי נקודה עשרונית הם אינסופיים אפילו בתוך טווח קטן, ואין להם סדר. כמה ערכים קיימים בין 23 ל-24? ואיזה מספר בא אחרי 23.46? האם זה 23.47, או 23.461, או אולי 23.4601? אי אפשר לדעת!

לכן, בעוד שהגיוני לדבר על המיקום הסודר של התו 'w' בתוך הטווח של טיפוס הנתונים Char, אין הגיון בשאילת אותה השאלה לגבי 7143.1562 בטווח של טיפוס נתונים בעל נקודה עשרונית. ניתן אמנם לדעת אם ערך מסוים גבוה יותר מאחר, אך אי אפשר לשאול כמה מספרים ממשיים קיימים לפני מספר נתון (וזו המשמעות של הפונקציה Ord).

לטיפוסים ממשיים יש תפקיד מוגבל בקטעי הקוד שעוסקים בממשק המשתמש (חלק ה-Windows של התוכנית), אך דלפי תומכת בהם בצורה מלאה, כולל בתחום מסדי הנתונים. התמיכה בטיפוסים בעלי נקודה עשרונית תקניים של IEEE הופכת את שפת Object Pascal למתאימה לגמרי למגוון רחב של תוכניות שמצריכות חישובים נומריים. אם אתם מעוניינים בפן זה של עולם התכנות, תוכלו לעיין בפונקציות האריתמטיות שמסופקות על ידי היחידה System של המהדר (לפרטים נוספים, ראו בתיעוד המהדר או השתמשו במערכת העזרה).

דלפי ו-FreePascal כוללות גם יחידה בשם Math שמגדירה רוטינות מתמטיות מתקדמות, ביניהן פונקציות טריגונומטריות (כגון הפונקציה ArcCosh), פיננסיות (כגון הפונקציה InterestPayment) וסטטיסטיות (כגון הפרוצדורה MedianAndStdDev). קיימות רוטינות רבות כאלה, וחלקן נשמעות לי משונות מאד, למשל הפונקציה MomentSkewKurtosis (אשאיר לכם לגלות מה זה).

תאריך ושעה

בדרך כלל, יישומי פסקל משתמשים בטיפוסי נתונים בעלי נקודה עשרונית כדי לטפל במידע שקשור לתאריכים ולשעות. ליתר דיוק, הן דלפי והן FreePascal מגדירים טיפוס נתונים ספציפי בשם TDateTime[19].

טיפוסי נתונים בעלי נקודה עשרונית דרושים מכיוון שהטיפוס חייב להיות רחב דיו להכיל שנים, חודשים, ימים, שעות, דקות ושניות עד לרמת דיוק של אלפיות השניה במשתנה יחיד. בחלק השלם של ערך TDateTime, התאריכים מאוחסנים כמספר הימים מאז ה-30 בדצמבר

[19] GNU Pascal מאמץ גישה שונה לאחסון תאריכים ושעות. הוא מגדיר רשומת TimeStamp (על רשומות נדבר מאוחר יותר), שמאחסנת כל מרכיב של התאריך ושל השעה בנפרד. הוא כולל מספר מצומצם יחסית של רוטינות לטיפול בתאריכים ובשעות מכיוון שניתן לגשת ישירות לרכיבים הנפרדים כגון "דקה". גישה זו אמנם פשוטה ומעט מהירה יותר, אך היא מצריכה שימוש בהרבה יותר זכרון.

1899[20] (כאשר ערכים שליליים מצביעים על תאריכים שלפני 1899). בחלק שאחרי הנקודה העשרונית, השעות מאוחסנות כחלק היחסי מהיממה.

הטיפוס TDateTime אינו טיפוס מוגדר מראש שהמהדר מכיר. הוא מוגדר ביחידה system בתור:

```
type
  TDateTime = type Double;
```

השימוש בטיפוס TDateTime פשוט למדי, מכיוון שדלפי כוללת מספר פונקציות שפועלות על טיפוס זה. הנה רשימה של כמה מהפונקציות הללו:

Now	מחזירה את התאריך והשעה הנוכחיים כערך תאריך/שעה
Date	מחזירה רק את התאריך הנוכחי
Time	מחזירה רק את השעה הנוכחית
DateTimeToStr	ממירה ערך תאריך ושעה למחרוזת בעיצוב ברירת המחדל. לשליטה רבה יותר בהמרה, השתמשו במקום זאת בפונקציה FormatDateTime.
DateTimeToString	מעתיקה את ערכי התאריך והשעה לתוך מחרוזת חוצץ (Buffer), עם עיצוב ברירת מחדל.
DateToStr	ממירה את חלק התאריך שבערך תאריך/שעה למחרוזת
TimeToStr	ממירה את חלק השעה שבערך תאריך/שעה למחרוזת.
FormatDateTime	ממירה את התאריך ואת השעה תוך שימוש בעיצוב שצוין. אתם יכולים לציין אילו ערכים אתם רוצים לראות ובאיזה עיצוב להשתמש בעזרת מחרוזת עיצוב מורכבת.
StrToDateTime	ממירה מחרוזת עם מידע תאריך ושעה לערך תאריך/שעה, ומעלה חריגה (Exception) במקרה של שגיאה בעיצוב המחרוזת. הפונקציה האחות, StrToDateTimeDef, אינה מעלה חריגה במקרה של שגיאה אלא מחזירה את ערך ברירת המחדל.
StrToDate	ממירה מחרוזת שמייצגת תאריך לערך תאריך/שעה.
StrToTime	ממירה מחרוזת שמייצגת שעה לערך תאריך/שעה.
DayOfWeek	מחזירה מספר שתואם ליום בשבוע של ערך התאריך/שעה שהועבר כפרמטר.
DecodeDate	מחלצת את השנה, החודש והיום מערך תאריך/שעה.
DecodeTime	מחלצת את השעות, הדקות, השניות ואלפיות השניה מערך תאריך/שעה.
EncodeDate	הופכת ערכי שנה, חודש ויום לערך תאריך/שעה.

[20] דלפי 1 השתמש בנקודת אפס שונה עבור ערכי TDateTime.

EncodeTime הופכת ערכי שעה, דקה, שניה ואלפיות השניה לערך תאריך/שעה.

כדי להדגים את אופן השימוש בטיפוס נתונים זה ובכמה מהרוטינות הקשורות אליו, בניתי דוגמה פשוטה בשם TimeNow. כאשר התוכנית מתחילה לפעול, היא מחשבת ומציגה אוטומטית את התאריך ואת השעה הנוכחיים.

```
StartTime := Now;
writeln (TimeToStr (StartTime));
writeln (DateToStr (StartTime));
```

ההצהרה הראשונה היא קריאה לפונקציה Now, שמחזירה את התאריך ואת השעה הנוכחיים. ערך זה מאוחסן במשתנה StartTime, שמוכרז כמשתנה גלובלי באופן הבא:

```
var
  StartTime: TDateTime;
```

שתי ההצהרות הבאות מציגות את חלק השעה של ערך ה-TDateTime כשהוא מומר למחרוזת, ואת חלק התאריך של אותו ערך. להלן פלט של התוכנית:

```
6:33:14 PM
10/7/2007
```

על מנת להדר את התוכנית, יהיה עליכם לפנות לפונקציות שמהוות חלק מיחידה (קובץ של קוד מקור) נפוצה של פסקל בשם SysUtils. הדבר נעשה באמצעות הקוד הבא[21]:

```
uses
  SysUtils;
```

מלבד הקריאה ל-TimeToStr ול-DateToStr, אתם יכולים להשתמש בפונקציה החזקה יותר FormatDateTime (לפרטים על פרמטרי העיצוב, עיינו בקובץ העזרה או בתיעוד).

שימו לב שערכי השעה והתאריך מומרים למחרוזות בהתאם להגדרות הבינלאומיות של המערכת. קובץ ההרצה קורא את הערכים הללו מהמערכת ומעתיק אותם למספר משתנים גלובליים שמוכרזים ביחידה SysUtils. הנה כמה מהם:

```
DateSeparator: Char;
ShortDateFormat: string;
LongDateFormat: string;
TimeSeparator: Char;
TimeAMString: string;
TimePMString: string;
ShortTimeFormat: string;
LongTimeFormat: string;
ShortMonthNames: array [1..12] of string;
LongMonthNames: array [1..12] of string;
ShortDayNames: array [1..7] of string;
LongDayNames: array [1..7] of string;
```

[21] יחידות והצהרות uses מוסברות בפירוט בפרק 11.

ישנם משתנים גלובליים נוספים, שקשורים לעיצוב של מטבע ושל מספרים בעלי נקודה עשרונית. תוכלו למצוא את הרשימה המלאה בקובץ העזרה או בתיעוד.

טיפוסים ספציפיים ל-Windows

טיפוסי הנתונים המוגדרים מראש שראינו עד כה הם חלק משפת פסקל. דלפי ו-FreePascal כוללים גם טיפוסי נתונים אחרים שמוגדרים על ידי Windows. טיפוסים אלה אינם חלק אינטגרלי מהשפה, אלא חלק מספריות Windows. טיפוסי Windows כוללים טיפוסים משותפים חדשים (כגון DWORD או UINT), רשומות (או מבנים) רבות, מספר טיפוסי מצביעים וכן הלאה.

מבין טיפוסי הנתונים של Windows, הטיפוס החשוב ביותר מיוצג על ידי ידיות (Handles), עליהן נדבר בפרק 9.

הטלת טיפוסים והמרות טיפוסים

כפי שראינו, לא ניתן להשים משתנה למשתנה מטיפוס אחר. כאשר יש צורך לעשות זאת, קיימות שתי אפשרויות.

הראשונה היא *הטלת טיפוסים* (typecasting), שמתבצעת בעזרת סימון פונקציונלי פשוט הכולל את השם של טיפוס הנתונים המיועד:

```
var
  N: Integer;
  C: Char;
  B: Boolean;

begin
  N := Integer ('X');
  C := Char (N);
  B := Boolean (0);
```

ניתן להטיל טיפוסים בין סוגי נתונים בעלי גודל זהה. לרוב, אפשר להטיל בבטחה טיפוסים בין טיפוסים סודרים, או בין טיפוסים ממשיים, אך ניתן גם להטיל טיפוסים בין טיפוסי מצביעים (ואובייקטים) כל עוד אתם יודעים מה אתם עושים.

אף על פי כן, הטלה נחשבת לנוהג תכנות מסוכן מכיוון שהיא מאפשרת לכם לגשת לערך כאילו ייצג משהו אחר. ככלל, הייצוגים הפנימיים של טיפוסי הנתונים אינם תואמים זה לזה, ועל כן אתם מסתכנים ביצירת שגיאות קשות לאיתור. מסיבה זו רצוי בדרך כלל להימנע מהטלת טיפוסים.

האפשרות השניה היא להשתמש ברוטינה להמרת טיפוסים. הרשימה הבאה מסכמת את הרוטינות להמרת הטיפוסים השונים:

Chr	ממירה מספר סודר לתו ANSI.
Ord	ממירה ערך מטיפוס סודר למספר שמציין את סדרו.
Round	ממירה ערך של טיפוס ממשי לערך מטיפוס של מספר שלם, תוך שהיא מעגלת אותו.
Trunc	ממירה ערך של טיפוס ממשי לערך מטיפוס של מספר שלם, תוך שהיא מקצצת אותו.
Int	מחזירה את החלק השלם של ארגומנט שכולל נקודה עשרונית.
IntToStr	ממירה מספר שלם למחרוזת.
IntToHex	ממירה מספר שלם למחרוזת בייצוג הקסדצימלי.
StrToInt	ממירה מחרוזת למספר, ומעלה חריגה אם המחרוזת אינה מייצגת מספר שלם חוקי.
StrToIntDef	ממירה מחרוזת למספר, ומשתמשת בערך ברירת מחדל אם המחרוזת שגויה.
Val	ממירה מחרוזת למספר (רוטינה מסורתית של טורבו פסקל, זמינה לצורך תאימות לאחור).
Str	ממירה מספר למחרוזת ומשתמשת בפרמטרי עיצוב (רוטינה מסורתית של טורבו פסקל, זמינה לצורך תאימות לאחור).
StrPas	ממירה מחרוזת בסיום null (null-terminated) למחרוזת בנוסח פסקל. המרה זו מתבצעת אוטומטית עבור משתנים מטיפוס AnsiString בדלפי 32 ביטים (ראו בפרק 7 על מחרוזות).
StrPCopy	מעתיקה מחרוזת בנוסח פסקל למחרוזת בסיום null. המרה זו מתבצעת בעת הטלה פשוטה של הטיפוס PChar בדלפי 32 ביט (ראו בפרק 7 על מחרוזות).
StrPLCopy	מעתיקה חלק ממחרוזת בנוסח פסקל למחרוזת בסיום null.
FloatToDecimal	ממירה ערך בעל נקודה עשרונית לרשומה שכוללת את הייצוג העשרוני שלו (חזקה, ספרות וסימן).
FloatToStr	ממירה ערך בעל נקודה עשרונית לייצוג במחרוזת, תוך שימוש בעיצוב ברירת המחדל.
FloatToStrF	ממירה ערך בעל נקודה עשרונית לייצוג במחרוזת, תוך שימוש בעיצוב שצוין.
FloatToText	מעתיקה ערך בעל נקודה עשרונית למחרוזת חוצץ, תוך שימוש בעיצוב שצוין.
FloatToTextFmt	בדומה לקודמת, מעתיקה את הערך בעל הנקודה העשרונית למחרוזת חוצץ תוך שימוש בעיצוב שצוין.
StrToFloat	ממירה מחרוזת נוסח פסקל לערך בעל נקודה עשרונית.
TextToFloat	ממירה מחרוזת בסיום null לערך בעל נקודה עשרונית.

חלק מהרוטינות הללו פועלות על טיפוסי נתונים עליהם נדבר בהמשך. שימו לב שהטבלה אינה כוללת רוטינות עבור טיפוסים מיוחדים (כגון TDateTime או וריאנטים) או רוטינות שמיועדות ספציפית לעיצוב, כמו הרוטינות רבות העוצמה Format ו-FormatFloat.

בגרסאות העדכניות של מהדר הפסקל של דלפי[22], הפונקציה Round מתבססת על היישום שהמעבד הראשי עצמו מציע. המעבד מאמץ את שיטת "עיגול הבנקאים", שמעגלת ערכי אמצע (כגון 5.5 או 6.5) למעלה או למטה על סמך אי-הזוגיות או הזוגיות, בהתאמה, של החלק השלם.

סיכום

בפרק זה סקרנו את המושג הבסיסי של "טיפוס" בפסקל, אך לשפה יש מאפיין חשוב מאד נוסף: היא מאפשרת למתכנתים להגדיר טיפוסי נתונים חדשים ומותאמים, שמכונים "טיפוסי נתונים מוגדרים על ידי המשתמש" (user-defined). על אלה נדבר בפרק הבא.

[22] ושל FreePascal

פרק 4: טיפוסי נתונים מוגדרים על ידי המשתמש

אחד מהרעיונות החדשים והגדולים שהופיעו בשפת פסקל, ביחד עם מושג הטיפוס, היה היכולת להגדיר טיפוסי נתונים חדשים בתוכנית. מתכנתים יכולים להגדיר טיפוסי נתונים משלהם באמצעות *הגדרות טיפוסים* (Type definitions) כגון טיפוסי תת-טווח, טיפוסי מערכים, טיפוסי רשומות, טיפוסים מונים, טיפוסי מצביעים וטיפוסי קבוצות. טיפוס הנתונים החשוב ביותר שמוגדר על ידי המשתמש הוא המחלקה (class), שמהווה חלק מהרחבות התכנות מונחה העצמים של Object Pascal שאינן מתוארות בספר זה.

אם אתם חושבים שבניית טיפוסים משותפת לשפות תכנות רבות, אתם צודקים – אך פסקל היתה השפה הראשונה שהציגה את הרעיון בצורה פורמלית ומדויקת מאד.

טיפוסים משוימים ולא-משוימים

טיפוסי נתונים שמוגדרים על ידי המשתמש יכולים לקבל שם לשימוש מאוחר יותר, או לחול ישירות על משתנים. המוסכמה בדלפי היא להשתמש בתחילית T כדי לציין כל טיפוס נתונים שהוא, כולל מחלקות אך לא רק אותן. אני ממליץ בחום לנהוג לפי כלל זה, אפילו אם הדבר לא נראה לכם טבעי בתחילה.

כאשר אתם מעניקים שם לטיפוס, עליכם לספק קטע קוד ספציפי כגון:

```
type
  // הגדרת תת-טווח
  TUppercase = 'A'..'Z';

  // הגדרת מערך
  TDayTemperatures = array [1..24] of Integer;

  // הגדרת רשומה
  TMyDate = record
    Month: Byte;
    Day: Byte;
    Year: Integer;
  end;

  // הגדרת טיפוס מונה
  TColors = (Red, Yellow, Green, Cyan, Blue, Violet);
```

```
// הגדרת קבוצה
TLetters = set of Char;
```

ניתן להשתמש בהגדרות טיפוסים דומות כדי להגדיר משתנים ישירות, ללא שם טיפוס מפורש, כפי שנעשה בקוד הבא:

```
var
  DecemberTemperature: array [1..31] of Byte;
  ColorCode: array [Red..Violet] of Word;
  Palette: set of Colors;
```

ככלל, כדאי להימנע משימוש בטיפוסים לא-משוימים כמו בקוד הנ"ל, מכיוון שלא ניתן להעביר אותם כפרמטרים לרוטינות או להכריז משתנים אחרים מאותו הטיפוס.

למעשה, כללי תאימות הטיפוסים בפסקל מבוססים על שמות הטיפוסים, לא על ההגדרות שלהם בפועל. שני משתנים בעלי טיפוס זהה עדיין לא ייחשבו זהים, אלא אם לטיפוסים שלהם יש שם זהה – והמהדר מקצה שמות פנימיים לטיפוסים לא-משוימים. התרגלו להגדיר טיפוס נתונים בכל פעם שאתם זקוקים למשתנה בעל מבנה מורכב, ולא תתחרטו על הזמן שהשקעתם בכך.

אך מה המשמעות של הגדרות טיפוסים אלה? אספק מספר תיאורים לאלו מכם שאינם מכירים את מבני הטיפוסים של פסקל. אנסה גם להדגיש את ההבדלים בינם לבין מבנים מקבילים בשפות תכנות אחרות, כך שייתכן שהסעיפים הבאים יעניינו אתכם אפילו אם אתם מכירים את הגדרות הטיפוסים מהסוג שתואר לעיל. לסיום, אראה מספר דוגמאות ואציג כלים שיאפשרו לכם לגשת למידע על טיפוסים באופן דינמי.

טיפוסי תת-טווח

טיפוס תת-טווח (Subrange type) מגדיר טווח ערכים בתוך הטווח של טיפוס אחר (מכאן השם *תת-טווח*). לדוגמה, אתם יכולים להגדיר תת-טווח של הטיפוס Integer מ-1 ועד 10 או מ-100 ועד 1000, או להגדיר תת-טווח של הטיפוס Char שיכיל רק את האותיות הרישיות, כך:

```
type
  TTen = 1..10;
  TOverHundred = 100..1000;
  TUppercase = 'A'..'Z';
```

בהגדרת תת-הטווח אין צורך לציין את השם של טיפוס הבסיס; צריך רק לספק שני קבועים מאותו הטיפוס. טיפוס המקור חייב להיות סודר, והטיפוס החדש שיתקבל יהיה סודר גם כן. לאחר שהגדרתם משתנה בתור תת-טווח, תוכלו לתת לו כל ערך בתוך הטווח שהוגדר עבורו. הקוד הבא חוקי:

```
var
  UppLetter: TUpperCase;

begin
```

```
  UppLetter := 'F';
```

אך זה לא:

```
var
  UppLetter: TUpperCase;

begin
  UppLetter := 'e'; // שגיאה בזמן ההידור
```

כתיבת הקוד השני תגרום לשגיאה של זמן הידור, שנוסחה "*Constant expression violates subrange bounds.*" ("ביטוי קבוע מפר את גבולות תת-הטווח"). לעומת זאת, אם תכתבו את הקוד הבא:

```
var
  UppLetter: TUppercase;
  Letter: Char;

begin
  Letter :='e';
  UppLetter := Letter;
```

הוא יהודר בלי בעיה על ידי דלפי. בזמן הריצה, אם הפעלתם את אפשרות המהדר לבדיקת טווחים (Range Checking) בדף Compiler שבתיבת הדו-שיח Project Options, תקבלו הודעת "*Range check error*" ("שגיאה של בדיקת טווח").

כאשר אתם מפתחים תוכנה, אני מציע להפעיל את אפשרות המהדר הזו על מנת שהתוכנה תהיה יציבה יותר ואיתור השגיאות בה יהיה קל יותר, מכיוון שבמקרה של שגיאה אתם תקבלו הודעה מפורשת במקום התנהגות לא צפויה. בסופו של דבר תוכלו להשבית את האפשרות הזו עבור הגרסה הסופית של התוכנית, כדי שהיא תעבוד קצת יותר מהר. עם זאת, השיפור במהירות יהיה כה קטן שאני מציע להשאיר את כל בדיקות זמן הריצה הללו מופעלות אפילו במוצר הסופי. אותו הדבר נכון לגבי אפשרויות בדיקה אחרות של זמן ריצה, כגון בדיקות גלישה (Overflow) ובדיקות מחסנית (Stack).

טיפוסים מונים

טיפוסים מונים (Enumerated types), המכונים לרוב "enums", הם טיפוס סודר שמוגדר על ידי המשתמש. במקום לציין טווח מתוך טיפוס קיים, בטיפוס מונה אנו מפרטים את כל הערכים האפשריים של הטיפוס. במילים אחרות, מניה הינה רשימת ערכים. הנה שתי דוגמאות:

```
type
  TColors = (Red, Yellow, Green, Cyan, Blue, Violet);
  TSuit = (Club, Diamond, Heart, Spade);
```

לכל ערך ברשימה יש "מספר סודר" משלו, ומספרים אלה מתחילים באפס. כאשר אתם מפעילים את הפונקציה Ord על ערך בטיפוס מונה, אתם מקבלים את הערך "מבוסס-האפס" הזה. לדוגמה, Ord(Diamond) תחזיר 1.

לטיפוסים מונים יכולים להיות ייצוגים פנימיים שונים. כברירת מחדל, דלפי משתמשת בייצוג בן 8 ביטים, אלא אם יש יותר מ-256 ערכים שונים – ואז היא משתמשת בייצוג בן 16 ביטים. קיים גם ייצוג בן 32 ביטים, שעשוי להיות שימושי לצורך תאימות עם ספריות של שפת C או C++.[23]

ספרית הרכיבים החזותיים (VCL) של דלפי משתמשת בטיפוסים מונים במקומות רבים. לדוגמה, סגנון העיצוב של שולי טופס מוגדר כך[24]:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsSizeToolWin, bsToolWindow);
```

טיפוסי קבוצה

טיפוסי קבוצה מציינים קבוצות של ערכים, בהן רשימת הערכים הזמינים תלויה בטיפוס הסודר שעליו מבוססת כל קבוצה. טיפוסים סודרים אלה מוגבלים בדרך כלל, ומיוצגים לעתים קרובות על ידי טיפוס מונה או תת-טווח. אם ניקח את תת-הטווח 1..3, הערכים האפשריים בקבוצה שמבוססת עליו כוללים את 1 בלבד, 2 בלבד, 3 בלבד, 1 ו-2, 1 ו-3, 2 ו-3, כל השלושה או אף אחד מהם.

בעוד שמשתנה רגיל מחזיק ערך יחיד מתוך הערכים האפשריים שבטווח הטיפוס שלו, משתנה מטיפוס קבוצה יכול להכיל אפס ערכים, אחד, שניים, שלושה או יותר. הוא יכול אף לכלול את כל הערכים. הנה דוגמה לקבוצה:

```
type
  TLetters = set of TUppercase;
```

כעת אני יכול להגדיר משתנה מטיפוס זה ולתת לו מספר ערכים מתוך הטיפוס המקורי. כדי לציין ערכים בקבוצה כותבים רשימה מופרדת בפסיקים ומוקפת בסוגריים מרובעים. הקוד הבא מציג השמה של מספר ערכים, של ערך יחיד ושל ערך ריק למשתנה:

```
var
  Letters1, Letters2, Letters3: TLetters;

begin
  Letters1 := ['A', 'B', 'C'];
  Letters2 := ['K'];
  Letters3 := [];
```

[23] תוכלו לשנות את ייצוג ברירת המחדל של טיפוסים מונים ולבקש ייצוג גדול יותר בעזרת הנחית המהדר $Z.

[24] בסביבת הפיתוח המשולבת, כאשר הערך של תכונת אובייקט הוא טיפוס מונה, תוכלו לבחור את הערך מתוך הרשימה שמוצגת ב-Object Inspector

את הפעולות שניתן לבצע על קבוצה (ההצהרות Include ו-Exclude) תוכלו לראות בסעיף "אופרטורים של קבוצות" בפרק 2.

בשפת פסקל, קבוצה משמשת לרוב לציון "דגלים" שאינם בלעדיים. לדוגמה, שורות הקוד הבאות (שהן חלק מהספריה של דלפי) מכריזות טיפוס מונה של סמלים אפשריים בשולי חלון, ואת טיפוס הקבוצה המתאים[25]:

```
type
  TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
  TBorderIcons = set of TBorderIcon;
```

למעשה, חלון נתון יכול שלא לכלול אף אחד מהסמלים הללו, או לכלול אחד מהם או יותר מאחד. תכונה נוספת שמבוססת על טיפוס קבוצה היא עיצוב של גופן. הערכים אפשריים כוללים גופן מודגש, נטוי, בעל קו תחתי או בעל קו חוצה. כמובן שאותו הגופן יכול להיות גם מודגש וגם נטוי, או ללא מאפיינים כלשהם, או עם כולם. על כן התכונה מוגדרת כקבוצה. ניתן להקצות ערכים לקבוצה זו בקוד התוכנית באופן הבא:

```
Font.Style := []; // ללא סגנון
Font.Style := [fsBold]; // סגנון מודגש בלבד
Font.Style := [fsBold, fsItalic]; // שני סגנונות
```

ניתן גם לבצע פעולות רבות ושונות על קבוצות, ביניהן חיבור שני משתנים מאותו טיפוס קבוצה (או, ליתר דיוק, איחוד של שני משתני קבוצה):

```
Font.Style := OldStyle + [fsUnderline]; // מיזוג שתי קבוצות
```

טיפוסי מערכים

טיפוסי מערכים מגדירים רשימות בעלות מספר קבוע של רכיבים מטיפוס מסוים. לרוב משתמשים ב*אינדקס* בתוך סוגריים מרובעים כדי לגשת לאחד מהרכיבים שבמערך. סוגריים מרובעים משמשים גם לציון הערכים האפשריים של האינדקס בעת הכרזת המערך. לדוגמה, ניתן להגדיר אוסף של 24 מספרים שלמים בעזרת הקוד הבא:

```
type
  TDayTemperatures = array [1..24] of Integer;
```

בהגדרת המערך עליכם לציין טיפוס תת-טווח בתוך הסוגריים המרובעים, או להגדיר תת-טווח ספציפי חדש באמצעות שני קבועים מטיפוס סודר. תת-טווח זה מציין את האינדקסים החוקיים של המערך. מכיוון שאתם מציינים הן את האינדקס הגבוה והן את הנמוך, האינדקסים אינם חייבים להיות מבוססי-אפס כפי שהם חייבים להיות ב-C, C++, ג'אווה ושפות אחרות.

[25] בעת עבודה עם ה- Object Inspector של דלפי, תוכלו לקבוע ערכים בקבוצה באמצעות הרחבת הבחירה (לחיצה כפולה על שם התכונה או לחיצה על סימן הפלוס שלצדה), והפעלה או כיבוי של כל ערך.

מכיוון שאינדקסים של מערכים מבוססים על תת-טווחים, שפת פסקל יכולה לבדוק את הטווח שלהם כפי שכבר ראינו. תת-טווח קבוע לא חוקי יגרום לשגיאת זמן הידור, ואינדקס מחוץ לטווח בזמן ריצה יגרום לשגיאת זמן ריצה אם אפשרות המהדר המתאימה מופעלת.

באמצעות הגדרת המערך לעיל ניתן לקבוע את הערך של המשתנה DayTemp1 מהטיפוס TDayTemperatures באופן הבא:

```
type
  TDayTemperatures = array [1..24] of Integer;

var
  DayTemp1: TDayTemperatures;

procedure AssignTemp;
begin
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;
  DayTemp1 [25] := 67; // שגיאת זמן הידור
```

למערך יכול להיות יותר מממד אחד, כפי שמראות הדוגמאות הבאות:

```
type
  TMonthTemps = array [1..24, 1..31] of Integer;
  TYearTemps = array [1..24, 1..31, Jan..Dec] of Integer;
```

שני מערכים אלה מבוססים על אותם טיפוסי ליבה, ולכן תוכלו להכריז עליהם באמצעות טיפוסי הנתונים שמופיעים לפניהם, כפי שנעשה כאן:

```
type
  TMonthTemps = array [1..31] of TDayTemperatures;
  TYearTemps = array [Jan..Dec] of TMonthTemps;
```

הכרזה זו הופכת את סדר האינדקסים שהוצג לעיל, אך מאפשרת השמה של בלוקים שלמים בין משתנים. לדוגמה, ההצהרה הבאה מעתיקה את הטמפרטורות של ינואר לפברואר:

```
var
  ThisYear: TYearTemps;

begin
  ThisYear[Feb] := ThisYear[Jan];
```

ניתן להגדיר גם מערכים מבוססי-אפס, כלומר טיפוס מערך שהגבול התחתון שלו הוא אפס. ככלל, השימוש בגבולות לוגיים מהווה יתרון, כי אז אין צורך להשתמש באינדקס 2 כדי לגשת לפריט השלישי וכן הלאה. עם זאת, Windows, Linux ו-Mac OS X משתמשות כולן במערכים מבוססי-אפס (מכיוון שהן מבוססות על שפת C), ומסיבה זו דלפי וספריות רכיבים רבות אחרות נוטות להתנהג באותו האופן.

כאשר אתם צריכים לעבוד עם מערך, תוכלו למצוא תמיד את גבולותיו באמצעות הפונקציות הסטנדרטיות High ו-Low, שמחזירות את הגבול העליון והתחתון בהתאמה. מומלץ מאד להשתמש ב-Low וב-High בעת עבודה עם מערך, בעיקר בתוך לולאות, מכיוון ששימוש כזה

הופך את הקוד לבלתי תלוי בטווח המערך. מאוחר יותר תוכלו לשנות את הטווח המוכרז של האינדקסים במערך, והקוד שמשתמש ב-Low וב-High עדיין יעבוד. לעומת זאת, אם תכתבו את הטווח של המערך בצורה מפורשת בלולאה, יהיה עליכם לעדכן את קוד הלולאה בכל פעם שגודל המערך ישתנה. הפונקציות Low ו-High הופכות את הקוד שלכם לקל יותר לתחזוקה ולאמין יותר[26].

דלפי 4 כללה לראשונה ב-Object Pascal את המערכים הדינמיים, מערכים שניתן לשנות את גודלם בזמן הריצה תוך הקצאת כמות הזכרון המתאימה. השימוש במערכים דינמיים הוא פשוט, אך לדעתי מוטב לדבר עליהם מאוחר יותר, בדיון על האופן בו שפת פסקל מטפלת בזכרון, בפרק 8.

טיפוסי רשומות

טיפוס רשומה (Record) מגדיר אוספים של פריטים מטיפוסים שונים. לכל רכיב, או *שדה*, יש טיפוס משלו. הגדרה של טיפוס רשומה מפרטת את כל השדות הללו ומעניקה לכל אחד מהם שם, בו תשתמשו מאוחר יותר כדי לגשת אליו.

הנה קוד קצר שכולל הגדרה של טיפוס רשומה, הכרזה של משתנה מטיפוס זה ומספר הצהרות שמשתמשות במשתנה הנ"ל:

```
type
  TMyDate = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: TMyDate;

begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
end;
```

אפשר לראות במחלקות ובאובייקטים הרחבה של טיפוס הרשומה[27]. הספריות של דלפי נוטות להשתמש בטיפוסי מחלקות במקום בטיפוסי רשומות, אך ה-API של Windows מגדיר טיפוסי רשומות רבים.

טיפוסי רשומות יכולים לכלול גם חלק וריאנטי; כלומר, ניתן למפות שדות רבים על אותו האזור בזכרון, אפילו אם טיפוסי הנתונים שלהם שונים (הדבר מקביל ל-union בשפת C).

[26] חשוב לציין שאין מחיר בזמן ריצה לשימוש ב-Low וב-High של מערכים. הן מפוענחות בזמן ההידור לביטויים קבועים במקום לקריאות של ממש לפונקציות. פענוח זה בזמן הידור של ביטויים ושל קריאות לפונקציות מתרחש גם עבור פונקציות מערכת פשוטות אחרות ורבות.

[27] בגרסאות העדכניות של דלפי, רשומות יכולות לכלול גם מתודות (פונקציות שמשויכות להן) והן תומכות בהעמסת אופרטורים של השפה (Operator Overloading). אני דן בנושאים אלה בספרי "Delphi 2007 Handbook" (לפרטים, ראו באתר האינטרנט שלי).

לחלופין, תוכלו להשתמש בשדות או בקבוצות שדות אלה כדי לגשת לאותו מיקום זכרון בתוך רשומה, אך להתייחס לערכים מנקודות מבט שונות. במקור, השימוש העיקרי של שדות כאלה היה לאחסון נתונים שונים אך דומים ולהשיג אפקט דומה לזה של הטלת טיפוסים (בימיה הראשונים של שפת פסקל, כאשר הטלת טיפוסים ישירה היתה אסורה). כיום, תכנות מונחה עצמים וטכניקות מודרניות אחרות החליפו ברוב המקרים את טיפוסי הרשומות הווריאנטיים, אם כי דלפי עדיין עושה בהם שימוש פנימי במקרים מיוחדים מסוימים.

השימוש בטיפוסי רשומה וריאנטיים אינו בטיחותי מבחינת טיפוסים ואינו מומלץ כטכניקת תכנות, במיוחד עבור מתחילים. מתכנתים מומחים יכולים להשתמש בהם, וכאמור נעשה בהם שימוש בליבה של ספריות דלפי. בכל מקרה, לא יהיה עליכם להתעסק איתם עד שתהיו מומחי דלפי של ממש.

לפני שתשתמשו בטיפוסי רשומות וריאנטיים, קחו בחשבון גם שגרסת דוט נט של דלפי אינה תומכת בהם באופן מלא, וניתן להשתמש בהם רק בחלקים ה"לא בטוחים" ("unsafe") של הקוד. למעשה, סביבת ההרצה של דוט נט רואה במאפיינים אלה סכנה לבטיחות, ויש לכך סיבה טובה.

מצביעים

טיפוס מצביע (Pointer) מגדיר משתנה, שמאחסן את כתובתו בזכרון של משתנה אחר מטיפוס נתון (או מטיפוס לא מוגדר). כלומר, משתנה מטיפוס מצביע מפנה בצורה עקיפה לערך. ההגדרה של טיפוס המצביע אינה מבוססת על מילת מפתח מסוימת אלא על שימוש בתו מיוחד שנקרא קארה (^ - Caret):

```
type
  TPointerToInt = ^Integer;
```

לאחר שהגדרתם משתנה מטיפוס מצביע, תוכלו לשים בתוכו את הכתובת של משתנה אחר בעל הטיפוס המוצבע, וזאת באמצעות האופרטור @:

```
var
  P: ^Integer;
  X: Integer;

begin
  P := @X;
  // שינוי הערך בשתי דרכים שונות
  X := 10;
  P^ := 20;
```

כאשר יש לכם מצביע בשם P, הביטוי P מפנה לכתובת בזכרון אליה מצביע P, ואילו הביטוי P^ מפנה לתוכן בפועל של אותו מיקום בזכרון. על כן, בקטע הקוד שלעיל, הביטוי P^ מקביל ל-X.

מצביע אינו חייב להפנות למיקום תפוס בזכרון – הוא יכול להפנות לבלוק זכרון חדש שהוקצה (Allocated) בצורה דינמית בערמה (Heap)[28] באמצעות הפרוצדורה New[29]. במקרה כזה, כאשר לא תזדקקו עוד לערך שהמצביע מפנה אליו, יהיה עליכם להיפטר גם מהזכרון שהקציתם בצורה דינמית – וזאת באמצעות קריאה לפרוצדורה Dispose.

אם לא תיפטרו מהזכרון לאחר השימוש בו, התוכנית שלכם עלולה לנצל בסופו של דבר את כל הזכרון הזמין ולהתרסק. תופעה זו מכונה בשם "זליגת זכרון" (Memory Leak).

הנה קטע קוד[30]:

```
var
  P: ^Integer;

begin
  // אתחול
  New (P);
  // פעולות
  P^ := 20;
  writeln (P^);
  // סיום
  Dispose (P);
end;
```

אם אין למצביע שום ערך, ניתן לתת לו את הערך nil. כך אפשר יהיה לבדוק, לאחר מכן, אם המצביע הוא nil כדי לגלות אם הוא מפנה לערך כלשהו.

שיטה זו משמשת לעתים קרובות, מכיוון שמימוש הפניה (Dereferencing) – כלומר גישה לערך שבכתובת שנמצאת במצביע – של מצביע לא תקף גורם להפרת גישה (Access Violation), הידועה גם כשגיאת הגנה כללית (General Protection Fault):

```
var
  P: ^Integer;

begin
  P := nil;
  writeln (P^);
```

תוכלו לראות דוגמה להשפעה של הקוד הזה באמצעות הרצה של הדוגמה Pointers אחרי הסרה של סימון ההערות משורות הקוד האחרונות. השגיאה שתראו תהיה דומה לזו:

```
Exception EAccessViolation in module Pointers.exe at 000083AC.
Access violation at address 004083AC in module 'Pointers.exe'. Read
of address 00000000.
```

[28] ניהול זכרון ככלל, ואופן הפעולה של הערמה בפרט, מוצגים בפרק 8. בקצרה, הערמה היא אזור (גדול) בזכרון שבו ניתן להקצות ולשחרר בלוקים של זכרון בלי סדר מיוחד.

[29] כתחליף ל-New ול-Dispose ניתן להשתמש ב-GetMem וב-FreeMem. עם זאת, לדברי העזרה של דלפי, "*שימוש בפרוצדורות New ו-Dispose נחשב עדיף*".

[30] בפועל, קוד זה צריך להיות סגור בתוך בלוק try-finally, נושא שהחלטתי לא לכלול בספר זה מכיוון שהוא אמנם רלוונטי, אך אינו חלק משפת פסקל המסורתית או מההרחבות של טורבו פסקל.

דרך אחת לשפר את הבטיחות של גישה לנתונים דרך מצביעים היא להוסיף בדיקת "המצביע אינו מאופס", כגון:

```
if P <> nil then
  writeln (P^);
```

דרך אחרת, שמועדפת לרוב מכיוון שהיא קריאה יותר, היא להשתמש בפונקציה המדומה Assigned[31]:

```
if Assigned(P) then
  writeln (P^);
```

דלפי מגדירה גם טיפוס נתונים בשם Pointer, שמציין מצביע נטול טיפוס (בדומה ל-*void בשפת C). אם אתם משתמשים במצביע נטול טיפוס, עליכם להשתמש ב-GetMem במקום ב-New. הפרוצדורה GetMem נדרשת בכל פעם שגודל הזכרון שיש להקצות אינו מוגדר מראש.

העובדה שמצביעים אינם נדרשים לעתים קרובות בעת תכנות בפסקל היא יתרון מעניין של שפה זו. אף על פי כן, הבנה של מצביעים היא חשובה לצורך תכנות מתקדם ולהבנה מלאה של מודל האובייקטים של דלפי, שמשתמש במצביעים "מאחורי הקלעים".

טיפוסי קבצים

סוג טיפוסים נוסף ייחודי לפסקל הינו הטיפוס *file* (קובץ). טיפוסי קבצים מייצגים קבצים פיזיים בדיסק, וזהו מאפיין מוזר שקיים רק בשפת פסקל. באפשרותכם להגדיר טיפוס נתונים חדש מסוג קובץ באופן הבא[32]:

```
type
  IntFile = file of Integer;
```

לאחר מכן תוכלו לפתוח קובץ פיזי שמשויך למבנה זה ולכתוב לתוכו ערכי מספרים שלמים, או לקרוא את הערכים שקיימים בו.

השימוש בקבצים בפסקל הוא פשוט למדי, אך בדלפי קיימים מספר רכיבים נוספים שמסוגלים לאחסן או לטעון את תוכנם אל או מתוך קובץ. קיימת תמיכה מסוימת בסריאליזציה (Serialization, הפיכה של אובייקט לסדרת ביטים לצורך אחסון -המתרגם) באמצעות שטפים (Streams), ויש גם תמיכה במסדי נתונים.

[31] Assigned אינה פונקציה אמיתית, מכיוון שהיא מפוענחת על ידי המהדר שמפיק את הקוד הסופי.
[32] דוגמאות לשימוש בקבצים מוצגות בפרק 12. עם זאת, שימו לב שטיפוס הקובץ אינו זמין בדלפי לדוט נט.

סיכום

פרק זה, שדן בטיפוסי נתונים מוגדרים על ידי המשתמש, משלים את הכיסוי של מערכת הטיפוסים בשפת פסקל. כעת אנו מוכנים להתבונן בהצהרות שהשפה מספקת במטרה לפעול על המשתנים שהגדרנו.

פרק 5: הצהרות

אם טיפוסי הנתונים הם אחד מהיסודות של התכנות בפסקל, הרי שהשני הוא ההצהרות. ניקלאוס וירת' הבהיר את הרעיון, בזמנו, בספרו המצוין " Algorithms + Data Structures = Programs", שפורסם בהוצאת Prentice Hall בפברואר 1976 (ספר קלאסי, שעדיין מודפס וניתן להשיגו).

הצהרות בשפת תכנות מבוססות על מילות מפתח ועל רכיבים אחרים, שמאפשרים לכם לציין בפני המהדר רצף של פעולות לביצוע. ההצהרות נמצאות לרוב בתוך פרוצדורות או פונקציות, כפי שנראה בפרק הבא. בינתיים נתמקד רק בסוגי הפקודות הבסיסיים בהם תוכלו להשתמש כדי ליצור תוכנית.

הצהרות פשוטות ומרוכבות

הצהרה בשפת פסקל נקראת פשוטה אם היא אינה מכילה כל הצהרה אחרת. דוגמאות להצהרות פשוטות הן השמות ערכים וקריאות לפרוצדורות. הצהרות פשוטות מופרדות בנקודה-פסיק:

```
X := Y + Z;   // השמה
Randomize;    // קריאה לפרוצדורה
```

בדרך כלל, הצהרות פשוטות מופיעות כחלקים של הצהרה מרוכבת (Compound), שמוקפת במילים begin ו-end. הצהרה מרוכבת יכולה להופיע בכל מקום בו יכולה להופיע הצהרה פשוטה. הנה דוגמה:

```
begin
  A := B;
  C := A * 2;
end;
```

הנקודה-פסיק שאחרי ההצהרה האחרונה בתוך ההצהרה המרוכבת (כלומר, זו שלפני ה-end) אינה נדרשת, כפי שניתן לראות כאן:

```
begin
  A := B;
  C := A * 2
end;
```

שתי הגרסאות תקינות. הראשונה כוללת נקודה-פסיק מיותרת אך לא מזיקה. למעשה, זוהי הצהרה ריקה (null), כלומר הצהרה ללא קוד[33].

[33] שימו לב שלעתים ניתן להשתמש בהצהרות ריקות בתוך לולאות, או במקרים ספציפיים אחרים:

```
while condition_with_side_effect do
  ; // הצהרה ריקה
```

נקודה-פסיק מסוג זה אינה משרתת כל מטרה, אך אני נוטה להשתמש בה ומציע שאתם תעשו את אותו הדבר. לפעמים, אחרי שכתבתם מספר שורות קוד, תרצו להוסיף עוד הצהרה. אם הנקודה-פסיק האחרונה חסרה, יהיה עליכם לזכור להוסיף אותה, כך שעדיף להוסיפה מלכתחילה.

הצהרות השמה

השמות (Assignments) בשפת פסקל משתמשות באופרטור נקודתיים-שווה (:=), צורת כתיבה משונה עבור מתכנתים שרגילים לשפות אחרות[34]. האופרטור =, שמשמש להשמה בהרבה שפות אחרות, שמור בפסקל לבדיקת שוויון.

באמצעות שימוש בסמלים שונים עבור השמה ועבור בדיקת שוויון, מהדר הפסקל (בדומה למהדר C) מסוגל לתרגם קוד מקור מהר יותר, וזאת מכיוון שאינו צריך לבחון את ההקשר שבו האופרטור נמצא כדי לקבוע את משמעותו. השימוש באופרטורים השונים מקל גם על בני אדם שקוראים את הקוד. בפסקל נבחרו אופרטורים שונים מאלה שב-C (ובנגזרותיה, כגון ג'אווה, C# או JavaScript), שם = משמש להשמה ואילו == (שני סימני "=" צמודים) לבדיקת שוויון.

שני הרכיבים של הצהרת השמה מכונים לרוב rvalue ו-lvalue, קיצור של Right Value (המשתנה או המיקום בזכרון שמהם מגיע הערך) ו-Left Value, הביטויים שמקבלים את הערך.

התוצאה של הביטוי מועתקת בדרך כלל לתוך משתנה. כאשר אתם מעתיקים רשומה או מערך, לדוגמה, כל מבנה הנתונים מועתק למיקום חדש בזכרון, וזו פעולה שעשויה להצריך זמן. כפי שנראה בפרק 7, מחרוזות מנוהלות באופן אחר.

הצהרות תנאי

הצהרת תנאי משמשת לביצוע אחת מתוך ההצהרות שמוכלות בתוכה, או אף אחת מהן, בהתאם לבדיקה שצוינה. קיימים שני סוגים בסיסיים של הצהרות תנאי: הצהרות if והצהרות case.

---

[34] למעט מתכנתים, בדומה לעורך ספר זה, שלמדו לתכנת לראשונה בשפת אלגול וששפת התכנות הראשונה המקצועית שלהם היתה PL/1...

הצהרות if

ניתן להשתמש בהצהרת ה-if כדי לבצע הצהרה רק אם תנאי כלשהו מתקיים (אם-אז, if-then), או כדי לבחור בין שתי אפשרויות שונות (אם-אז-אחרת, if-then-else). התנאי מתואר באמצעות ביטוי בוליאני.

דוגמה פשוטה בפסקל תראה כיצד כותבים הצהרות תנאי. בתוכנית זו נבקש קלט מהמשתמש, באמצעות הפרוצדורה read עם תו יחיד כפרמטר:

```
var
  aChar: Char;

begin
  write ('Enter a character: ');
  readln (aChar);

  if aChar = 'a' then
    writeln ('You pressed [a]');
```

אם תקלידו את התו a (A קטנה) התוכנית תציג הודעה פשוטה, אחרת לא יקרה דבר. במקרה כזה עדיף להיות ברורים יותר, כפי שנעשה בקטע הקוד הבא שמשתמש בהצהרת if-then-else:

```
  // if-then-else הצהרת
  if aChar = 'b' then
    writeln ('You pressed [b]')
  else
    writeln ('You pressed something else than [b]');
```

שימו לב שלא מקלידים נקודה-פסיק אחרי ההצהרה הראשונה ולפני מילת המפתח else, אחרת המהדר יפיק שגיאת תחביר. למעשה, ההצהרה if-then-else היא הצהרה יחידה ולא ניתן להציב נקודה-פסיק באמצע הצהרה.

הצהרת if יכולה להיות מורכבת למדי. אפשר להפוך את התנאי לסדרת תנאים (באמצעות האופרטורים הבוליאניים and, or ו-not), ואפשר לקנן הצהרת if אחת בתוך הצהרת if אחרת. החלק האחרון שבדוגמה IfTest מדגים את המקרים הללו:

```
  // הצהרה עם תנאי כפול
  // בודקת אם האות אינה ריישית
  if (aChar >= 'a') and (aChar <= 'z') then
    writeln ('lowercase');

  // הצהרת תנאי מרוכבת
  if aChar >= Char(32) then
  begin
    if (aChar >= '0') and (aChar <= '9') then
      writeln ('a number')
    else
      writeln ('not a number');
  end
  else
    writeln ('non-printable char');
```

הסתכלו על הקוד בתשומת לב והריצו את התוכנית כדי לוודא שאתם מבינים הכל (אפשר להשתמש במקש Tab כדי להזין תו שאינו ניתן להדפסה). אם אינכם בטוחים בנוגע למבנה תכנותי כלשהו, תוכלו להסתייע מאד בכתיבת תוכנית בדיקה פשוטה בסגנון זה. ייתכן שתרצו להוסיף אפשרויות וביטויים בוליאניים, להגדיל את המורכבות של הדוגמה הקטנה הזו ועל ידי כך ליצור כל בדיקה שתבחרו.

הצהרות case

אם הצהרות ה-if שלכם הופכות למורכבות מאד, תוכלו לפעמים להחליף אותן בהצהרות case. הצהרת case מורכבת מביטוי שמשמש לבחירת ערך (הערך ה"בוחר"), ומרשימה של ערכים אפשריים (או טווחי ערכים). ערכים אלה הם קבועים, ועליהם להיות בלעדיים ומטיפוס סודר. בסיום יכולה להופיע גם הצהרת else שמבוצעת אם אף אחד מהערכים לא תואם לערך הבוחר. להלן שתי דוגמאות פשוטות (חלק מפרויקט CaseTest):

```
case Number of
  1: Text := 'One';
  2: Text := 'Two';
  3: Text := 'Three';
end;

case aChar of
  '+' : Text := 'Plus sign';
  '-' : Text := 'Minus sign';
  '*', '/': Text := 'Multiplication or division';
  '0'..'9': Text := 'Number';
  'a'..'z': Text := 'Lowercase character';
  'A'..'Z': Text := 'Uppercase character';
else
  Text := 'Unknown character: ' + aChar;
end;
```

הוספה של סעיף else ללכידה של תנאי לא מוגדר או לא צפוי נחשבת לפרקטיקת תכנות טובה.

לולאות בפסקל

שפת פסקל כוללת את ההצהרות המתבצעות שוב ושוב שיש ברוב שפות התכנות, כולל הצהרות for, while ו-repeat. אם אתם מכירים שפות תכנות אחרות, רוב מה שהלולאות הללו עושות יהיה לכם מוכר, כך שאדבר עליהן בקצרה.

הלולאה for

הלולאה for בפסקל מבוססת באופן בלעדי על מונה, אותו ניתן להגדיל או להקטין בכל פעם שהלולאה מבוצעת. הנה דוגמה פשוטה ללולאת for שמשמשת לחיבור עשרת המספרים הראשונים.

```
var
  total, I: Integer;

begin
  total := 0;
  for I := 1 to 10 do
    total := total + I;
```

את אותה הצהרת for אפשר היה לכתוב באמצעות מונה הפוך:

```
var
  total, I: Integer;

begin
  total := 0;
  for I := 10 downto 1 do
    total := total + I;
```

לולאת for בפסקל היא פחות גמישה מאשר בשפות אחרות (לא ניתן להגדיר צעד שונה מ-1), אך קל ופשוט להבין אותה. אם תרצו לבדוק תנאי מורכב יותר, או לספק מונה מותאם, יהיה עליכם להשתמש בהצהרת while או repeat במקום בלולאת for.

המונה בלולאת for אינו חייב להיות מספר. הוא יכול להיות כל ערך שהוא של כל טיפוס סודר, כגון תו או טיפוס מונה. הנה דוגמה עם תווים:

```
var
  aChar: Char;

begin
  for aChar := 'a' to 'z' do
  begin
    write (aChar);
    write (' ');
  end;
```

והנה דוגמה נוספת, עם טיפוס מונה:

```
type
  TSuit = (Club, Diamond, Heart, Spade);

var
  aSuit: TSuit;

begin
  for aSuit := Club to Spade do
    ...
```

כל קטעי הקוד הללו נמצאים בדוגמה ForTest. את הלולאה האחרונה אפשר לכתוב גם כך שתפעל מפורשות על כל פריט בטיפוס המונה:

```
for aSuit := Low (TSuit) to High (TSuit) do
```

הגרסאות העדכניות של דלפי כוללות צורה חדשה של לולאה, בשם for-in, שדומה ללולאה for-each המסורתית ב-Visual Basic. בלולאת for כזו, הקוד מופעל כל רכיב במערך, ברשימה או במכולה (container) מסוג אחר[35].

הצהרות while ו-repeat

ההבדל בין הלולאה while-do ללולאה repeat-until הוא שהקוד של הצהרת repeat מבוצע בכל מקרה לפחות פעם אחת. אפשר להבין זאת בקלות אם מסתכלים על הדוגמה הפשוטה הבאה:

```
while (I <= 100) and (J <= 100) do
begin
  // משתמשים בשני המשתנים כדי לחשב משהו...
  I := I + 1;
  J := J + 1;
end;

repeat
  // משתמשים בשני המשתנים כדי לחשב משהו...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);
```

אם הערך הראשוני של I או של J גדול מ-100, ההצהרות שבתוך הלולאה repeat-until מבוצעות בכל זאת פעם אחת.

ההבדל הגדול הנוסף בין שתי הלולאות הללו הוא שללולאה repeat-until יש תנאי הפוך: לולאה זו מבוצעת כל עוד התנאי *אינו* מתקיים. כאשר התנאי מתקיים, הלולאה מפסיקה. המצב הפוך בלולאה while-do, שמבוצעת כל עוד התנאי מתקיים. זו הסיבה לכך שהיה עליי להפוך את התנאי בקוד שלמעלה כדי להשיג תוצאה דומה[36].

דוגמאות ללולאות

במטרה לחקור את פרטי הלולאות, הבה נתבונן בדוגמה קטנה בפסקל. התוכנית LoopsTest מדגישה את ההבדל בין לולאה עם מונה קבוע ללולאה עם מונה אקראי כמעט. הלולאה הראשונה מציגה מספר מחרוזות:

[35] פרטי הלולאה for-in מוצגים בספרי "Delphi 2007 Handbook"
[36] מאפיין זה ידוע בשמו הפורמלי "חוקי דמורגן" (עליהם ניתן לקרוא בכתובת (http://en.wikipedia.org/wiki/De_Morgan%27s_laws)

```
var
  I: Integer;

begin
  for I := 1 to 20 do
    writeln ('String ' + IntToStr (I));
end;
```

הקטע השני מסובך קצת יותר: במקרה זה, יש לולאת while שמבוססת על מונה שמוגדל באופן אקראי. כדי לבצע זאת קראתי לפרוצדורה Randomize, שמאפסת את מחולל המספרים האקראיים, ולפונקציה Random עם ערך טווח של 100. התוצאה של פונקציה זו היא מספר בין 0 ל-99 (כולל), שנבחר באקראי. סדרת המספרים האקראיים קובעת כמה פעמים הלולאה תרוץ.

```
var
  I: Integer;

begin
  Randomize;
  I := 0;
  while I < 1000 do
  begin
    I := I + Random (100);
    writeln ('Random Number: ' + IntToStr (I));
  end;
end;
```

בכל פעם שתריצו את התוכנית המספרים יהיו שונים, מכיוון שהם תלויים במחולל המספרים האקראיים. הנה פלטים של שתי הרצות שונות, מוצגים במקביל:

```
Random Number: 25              Random Number: 82
Random Number: 68              Random Number: 130
Random Number: 131             Random Number: 140
Random Number: 192             Random Number: 186
Random Number: 263             Random Number: 195
Random Number: 347             Random Number: 196
Random Number: 379             Random Number: 214
Random Number: 437             Random Number: 311
Random Number: 531             Random Number: 403
Random Number: 583             Random Number: 429
Random Number: 660             Random Number: 468
Random Number: 683             Random Number: 515
Random Number: 689             Random Number: 608
Random Number: 751             Random Number: 628
Random Number: 775             Random Number: 722
Random Number: 798             Random Number: 776
Random Number: 888             Random Number: 824
Random Number: 910             Random Number: 889
Random Number: 948             Random Number: 967
Random Number: 970             Random Number: 1062
Random Number: 1019
```

שימו לב שלא רק המספרים שהופקו שונים בכל פעם, אלא שגם מספר הפריטים עצמו שונה. לולאת while זו מופעלת מספר אקראי של פעמים. אם תריצו אותה מספר פעמים ברצף, תראו שיש בפלט מספר שונה של שורות.

אפשר לשנות את הזרימה הרגילה של ביצוע לולאה באמצעות פרוצדורות המערכת Break ו-Continue. הראשונה מפסיקה את הלולאה, ואילו השניה משמשת לקפיצה הישר אל תנאי הלולאה או אל הגדלת המונה, וביצוע המעבר הבא בתור (אלא אם התנאי אינו מתקיים או שהמונה הגיע לערכו המקסימלי)[37]. שתי פרוצדורות מערכת נוספות, Exit ו-Halt, מאפשרות לכם לחזור באופן מיידי מהפונקציה או הפרוצדורה, או לעזוב את התוכנית.

ההצהרה with

הסוג האחרון של הצהרות פסקל בו אתמקד הוא ההצהרה with, שהיתה בעבר ייחודית לשפה זו (אך הופיעה מאז ב-JavaScript וב-Visual Basic), ויכולה להיות שימושית מאד בתכנות בפסקל.

ההצהרה with אינה אלא קיצור דרך. כאשר עליכם לפנות למשתנה מטיפוס record (או לאובייקט), במקום לחזור על שמו כל פעם, תוכלו להשתמש בהצהרת with. לדוגמה, כאשר הצגתי את טיפוס הרשומה, כתבתי את הקוד הבא:

```
type
  TMyDate = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: TMyDate;

begin
  BirthDay.Year := 2007;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

בעזרת הצהרת with אני מסוגל לשפר את החלק האחרון של הקוד, באופן הבא:

```
  with BirthDay do
  begin
    Year := 2008;
    Month := 2;
    Day := 14;
  end;
```

גישה זו יכולה לשמש בתוכניות פסקל גם כדי להפנות לרכיבים ולטיפוסי מחלקות אחרים. כאשר אתם עובדים עם רכיבים או עם מחלקות באופן כללי, ההצהרה with מאפשרת לכם לדלג על כתיבה של קצת קוד, במיוחד בשדות שמכילים האחד את השני.

לדוגמה, בקוד של ה-VCL של דלפי, נניח שאתם צריכים לשנות את ה-width (רוחב) ואת ה-color (צבע) של כלי הציור "עט" (pen) של טופס. אפשר לכתוב את הקוד הבא:

[37] לדעתי, עדיף להשתמש בהצהרות תנאי (if) במקום ב-Break וב-Continue מכיוון שכך הקוד יהיה קריא יותר.

```
Form1.Canvas.Pen.Width := 2;
Form1.Canvas.Pen.Color := clRed;
```

אך ללא ספק, יהיה קל יותר לכתוב את הקוד הזה:

```
with Form1.Canvas.Pen do
begin
  Width := 2;
  Color := clRed;
end;
```

בעת כתיבת קוד מורכב, ההצהרה with יכולה להיות אפקטיבית ולחסוך הכרזה של משתנים זמניים, אך יש לה חסרון. היא עלולה להפוך את הקוד לפחות קריא, בעיקר בעת עבודה עם טיפוסים מונים, רשומות ואובייקטים שונים שיש להם תכונות דומות או מקבילות. חסרון נוסף הוא שהשימוש בהצהרה with עלול לגרום לשגיאות לוגיות נסתרות בקוד, שהמהדר אינו יכול לזהות. הדוגמה הבאה מראה כיצד המהדר יחמיץ שגיאה לוגית (ובצדק) כאשר משתמשים ב-with ביחד עם אובייקטים נגזרים:

```
with Button1 do
begin
  Width := 200;
  Caption := 'New Caption';
  Color := clRed;
end;
```

הקוד הזה ישנה את ה-Caption (תווית) ואת ה-Width (רוחב) של הכפתור (Button), אך ישפיע על תכונת ה-Color (צבע) של הטופס עצמו ולא של הכפתור! הסיבה לכך היא שלרכיבים מטיפוס TButton אין תכונה בשם Color, ומכיוון שהקוד הזה מבוצע עבור אובייקט טופס (מדובר כאן בקוד שנמצא בתוך מתודה של הטופס), מתבצעת גישה לאובייקט הטופס כברירת מחדל. במקום זאת, אילו היינו כותבים:

```
Button1.Width := 200;
Button1.Caption := 'New Caption';
Button1.Color := clRed; // שגיאה!
```

המהדר היה מתריע על שגיאה. ככלל, ניתן לומר שמכיוון שההצהרה with מוסיפה מזהים חדשים להיקף הנוכחי, אנו עלולים להסתיר מזהים קיימים או לגשת בטעות למזהים אחרים באותו ההיקף (כפי שנעשה בגרסה הראשונה של הקוד לעיל). גם תוך התחשבות במגרעה זו, אני מציע להתרגל לשימוש בהצהרות with מכיוון שהן יכולות להיות מועילות מאד ולעתים אף להפוך את הקוד לקריא יותר. עם זאת, עליכם להימנע מהצהרות with מרובות כגון:

```
with ListBox1, Button1 do...
```

הקוד שיבוא אחרי הצהרה כזו יהיה, ככל הנראה, בלתי קריא, מכיוון שעבור כל תכונה שמוגדרת בתוך בלוק הקוד תצטרכו לחשוב לאיזה רכיב היא שייכת, בהתאם לתכונות השונות ולסדר הרכיבים בהצהרת ה-with.

אם כבר מדברים על קלות הקריאה, בשפת פסקל אין הצהרות endif או endcase. אם יש להצהרת if בלוק begin-end, אזי סוף הבלוק מציין את סוף ההצהרה. לעומת זאת, ההצהרה

case מסתיימת תמיד ב-end. כל הצהרות ה-end הללו, שמופיעות לעתים קרובות בזו אחר זו, עלולות להפוך את הקוד לקשה לקריאה. רק באמצעות מעקב אחר ההסטה ימינה ניתן לזהות לאיזו הצהרה מתייחסת הצהרת end מסוימת. דרך נפוצה לפתור את הבעיה ולהפוך את הקוד לקריא יותר היא להוסיף הערה לאחר הצהרת ה-end שמציינת את תפקידה, כמו כאן:

```
end; // if
```

סיכום

פרק זה הראה כיצד לכתוב הצהרות תנאי ולולאות. במקום לכתוב רשימות ארוכות של הצהרות שכאלה, התוכניות מחולקות לרוב לרוטינות, פרוצדורות או פונקציות. זהו הנושא של הפרק הבא, שמציג גם מספר רכיבים מתקדמים.

פרק 6: פרוצדורות ופונקציות

רעיון חשוב נוסף ששפת פסקל מדגישה הוא התפיסה של רוטינה, שהיא בבסיסה סדרה של הצהרות שיש לה שם ייחודי, ושניתן להפעיל אותה פעמים רבות באמצעות שימוש בשם זה. כך אנו נמנעים מהצורך לכתוב את אותן ההצהרות שוב ושוב, ומקבלים גרסה יחידה של הקוד שמשמש במקומות רבים בתוכנית. מנקודת מבט כזו אפשר לחשוב על רוטינות כעל מכניזם בסיסי לכימוס (encapsulation) של קוד. אחזור לנושא זה עם דוגמה לאחר שאציג את תחביר הרוטינות של פסקל.

פרוצדורות ופונקציות בפסקל

רוטינות בפסקל יכולות להופיע בשתי צורות: פרוצדורה ופונקציה. בתאוריה, פרוצדורה היא פעולה שאתם מבקשים מהמחשב לבצע, ואילו פונקציה היא חישוב שמחזיר ערך. ההבדל מובלט בעובדה שלפונקציה יש תוצאה, ערך חוזר, ואילו לפרוצדורה לא. שני סוגי הרוטינות יכולים לקבל פרמטרים מרובים מטיפוסי נתונים שונים.

בפועל, לעומת זאת, ההבדל בין פונקציות לפרוצדורות מוגבל מאד: ניתן לקרוא לפונקציה כדי שתבצע פעולה מסוימת ולדלג על התוצאה (שעשויה להיות, למשל, קוד שגיאה אופציונלי) או לקרוא לפרוצדורה שמעבירה בחזרה תוצאה דרך הפרמטרים שלה (ועוד על פרמטרי הפניה בהמשך הפרק).

הנה הגדרה של פרוצדורה, והגדרות של שתי גרסאות של אותה הפונקציה בתחביר שונה מעט:

```
procedure Hello;
begin
  writeln ('Hello world!');
end;

function Double (Value: Integer) : Integer;
begin
  Double := Value * 2;
end;

// או, כחלופה,
function Double2 (Value: Integer) : Integer;
begin
  Result := Value * 2;
end;
```

השימוש ב-Result במקום בשם הפונקציה לצורך השמת ערך ההחזרה הולך ונהיה פופולרי, והוא הופך, בדרך כלל, את הקוד לקריא יותר. לאחר שרוטינות אלה הוגדרו, תוכלו לקרוא להן פעם אחת או יותר. קוראים לפרוצדורה כדי לגרום לה לבצע את הפעולה שלה, ולפונקציה – כדי לחשב את הערך:

```
// קריאה לפרוצדורה
Hello;

// קריאה לפונקציה
X := Double (100);
Y := Double (X);
writeln (IntToStr (Y));
```

זהו מימוש בפועל של רעיון כימוס הקוד שהצגתי קודם לכן. כאשר אתם קוראים לפונקציה *Double*, אינכם צריכים לדעת איזה אלגוריתם שימש לצורך יישומה. אם תגלו בהמשך דרך טובה יותר להכפיל מספרים פי שניים, תוכלו לשנות בקלות את הקוד של הפונקציה, ואילו הקוד שקורא לה יישאר זהה (אם כי ההרצה תהיה מהירה יותר!). אותו העקרון חל על הפרוצדורה *Hello*: אנו יכולים לשנות את פלט התוכנית באמצעות שינוי הקוד של הפרוצדורה, וקוד התוכנית המרכזי ישנה את התוצאה באופן אוטומטי. הנה דוגמה לאופן בו אנו יכולים לשנות את הקוד:

```
procedure Hello;
begin
  writeln ('Hello world, again!');
end;
```

כאשר אתם קוראים לפונקציה או לפרוצדורה קיימת בפסקל, עליכם לזכור את מספר וטיפוסי הפרמטרים שלה[38].

פרמטרי הפניה

רוטינות של פסקל מאפשרות העברת פרמטרים לפי ערך (value) ולפי הפניה (reference). העברת פרמטרים לפי ערך היא ברירת המחדל: הערך מועתק למחסנית (stack), והרוטינה משתמשת ומשנה את העותק, לא את הערך המקורי.

העברת פרמטר לפי הפניה פירושה שהערך אינו מועתק למחסנית כפרמטר רשמי של הרוטינה (ההימנעות מהעתקה מובילה בדרך כלל לכך שהתוכנית רצה מהר יותר). במקום זאת, התוכנית מפנה אל הערך המקורי מתוך הקוד של הרוטינה. הדבר מאפשר לפרוצדורה או לפונקציה לשנות את הערך של הפרמטר. העברת פרמטרים לפי הפניה מבוטאת באמצעות מילת המפתח *var*.

[38] העורך של דלפי מסייע בכך ומציג את רשימת הפרמטרים של פונקציה או של פרוצדורה בעזרת "חלון רמז" שצץ ברגע שאתם מקלידים את שמה ואת הסוגריים הפותחים. מאפיין זה נקרא Code Parameters והוא מהווה חלק מטכנולוגיית Code Insight.

טכניקה זו זמינה ברוב שפות התכנות. היא אינה קיימת ב-C, אך הוספה ב-C++, שם ניתן להשתמש בסמל & (העברה לפי הפניה). בשפת Visual Basic, כל פרמטר שלא צוין במפורש כ-ByVal מועבר לפי הפניה. הנה דוגמה להעברה של פרמטר לפי הפניה באמצעות מילת המפתח var:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

במקרה זה, הפרמטר משמש הן להעברת ערך לפרוצדורה והן להחזרת ערך חדש לקוד הקורא. כאשר אתם כותבים:

```
var
  X: Integer;

begin
  X := 10;
  DoubleTheValue (X);
```

הערך של המשתנה x הופך להיות 20, מכיוון שהפרוצדורה משתמשת בהפניה למיקום המקורי בזכרון של x ומשפיעה על ערכו ההתחלתי.

העברת פרמטרים לפי הפניה הגיונית כשמדובר בטיפוסים סודרים, במחרוזות מהסוג הישן וברשומות גדולות[39]. למחרוזות דלפי יש התנהגות שונה מעט: הן מתנהגות כמו הפניות למיקום מסוים בזכרון, אך אם תשנו אחד ממשתני המחרוזת שמפנים לאותה המחרוזת שבזכרון, היא תועתק אליו בטרם העדכון. מחרוזת ארוכה שמועברת כפרמטר ערך מתנהגת כהפניה רק במובן של שימוש בזכרון ושל מהירות; אם תשנו את הערך של המחרוזת שהועברה, הערך המקורי לא יושפע. לעומת זאת, אם תעבירו את המחרוזת הארוכה לפי הפניה, תוכלו לשנות את ערכה המקורי.

בדלפי 3 הוצג סוג חדש של פרמטר, out. לפרמטר out אין ערך התחלתי והוא משמש אך ורק להחזרת ערך. יש להשתמש בפרמטרים אלה רק במסגרת פרוצדורות ופונקציות של COM (Component Object Model); ככלל, מומלץ לדבוק בפרמטרי var היעילים יותר. פרט לכך שאין להם ערך התחלתי, פרמטרי out מתנהגים כמו פרמטרי var.

פרמטרים קבועים

כחלופה לפרמטרי הפניה, ניתן להשתמש בפרמטרי const. מכיוון שלא ניתן לתת ערך חדש לפרמטר קבוע בתוך הרוטינה, המהדר יכול למטב את העברת הפרמטרים. הוא יכול לבחור גישה דומה לזו שמשמשת עבור פרמטרי הפניה (או "const reference" במונחי C++), אך ההתנהגות תישאר דומה לזו של פרמטרי ערך מכיוון שהערך המקורי לא יושפע מהרוטינה.

[39] למעשה, אובייקטים בפסקל מועברים תמיד לפי ערך, מכיוון שהם עצמם הפניות. לכן, העברה של אובייקט לפי הפניה היא חסרת תועלת (למעט במקרים מאד מיוחדים) – היא בעצם העברה של "הפניה להפניה".

למעשה, אם תנסו להדר את הקוד (הטפשי) הבא, דלפי תפיק הודעת שגיאה:

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2;       // שגיאת מהדר
  Result := Value;
end;
```

פרמטרי מערכים פתוחים

שלא כמו ב-C, לפונקציה או פרוצדורה בפסקל יש תמיד מספר פרמטרים קבוע. עם זאת, קיימת דרך להעביר מספר משתנה של פרמטרים לרוטינה, וזאת באמצעות מערך פתוח (Open array). ההגדרה הבסיסית של פרמטר מסוג מערך פתוח היא כזו של מערך פתוח מטיפוס מסוים. פירוש הדבר שאתם מציינים את טיפוס הפרמטר, אך אינכם יודעים מראש כמה רכיבים מאותו טיפוס יהיו במערך. הנה דוגמה להגדרה שכזו:

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;

begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

באמצעות High(A) אנו מסוגלים לגלות את הגבול העליון של המערך. שימו לב גם לשימוש שנעשה בערך ההחזרה של הפונקציה, Result, לאחסון ערכים זמניים. לפונקציה זו אפשר לקרוא באמצעות העברה של ביטוי מטיפוס *Array of Integer*:

```
X := Sum ([10, Y, 27*I]);
```

בהנתן Array of Integer בכל גודל שהוא, אפשר להעביר אותו ישירות לרוטינה שדורשת פרמטר מערך פתוח או, לחלופין, לקרוא לפונקציה Slice כדי להעביר רק חלק מהמערך (כפי שמציין הפרמטר השני שלה). הנה דוגמה שבה המערך השלם מועבר כפרמטר:

```
var
  List: array [1..10] of Integer;
  X, I: Integer;

begin
  // אתחול המערך
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // קריאה
  X := Sum (List);
```

אם ברצונכם להעביר רק חלק מהמערך לפונקציה Sum, פשוט קראו לה באופן הבא, בעזרת הפונקציה המובנית Slice:

```
X := Sum (Slice (List, 5));
```

שמתם לב כיצד התוצאה של פונקציה אחת יכולה להיות הפרמטר של אחרת? את כל קטעי הקוד שהוצגו בסעיף זה תוכלו למצוא בדוגמה OpenArr.

קיימת תאימות מלאה בין מערכים פתוחים בעלי טיפוס לבין מערכים דינמיים (שהופיעו לראשונה בדלפי 4 ומתוארים בפרק 8). מערכים דינמיים משתמשים באותו התחביר כמו מערכים פתוחים, בהבדל אחד – ניתן להשתמש בסימון כמו Array of Integer כדי להכריז על משתנה ולא רק כדי להעביר פרמטר.

פרמטרים של מערך פתוח בעלי טיפוס משתנה

פרט למערכים פתוחים בעלי טיפוס, דלפי מאפשרת לכם להגדיר גם מערכים פתוחים בעלי טיפוס משתנה, או נטולי טיפוס. לסוג מערך מיוחד זה יש מספר לא מוגדר של ערכים, שיכולים להיות שימושיים להעברת פרמטרים.

מבחינה טכנית, המבנה Array of const מאפשר להעביר לרוטינה בבת אחת מערך עם מספר לא מוגדר של רכיבים מסוגים שונים. לדוגמה, הנה ההגדרה של הפונקציה Format (אנו נראה כיצד להשתמש בפונקציה זו בפרק 7 שעוסק במחרוזות):

```
function Format (const Format: string;
  const Args: array of const): string;
```

הפרמטר השני הוא מערך פתוח, שמקבל מספר לא מוגדר של משתנים. למעשה, ניתן לקרוא לפונקציה הזו בכל הדרכים הבאות:

```
N := 20;
S := 'Total:';
writeln (Format ('Total: %d', [N]));
writeln (Format ('Int: %d, Float: %f', [N, 12.4]));
writeln (Format ('%s %d', [S, N * 2]));
```

שימו לב שניתן להעביר פרמטר כערך קבוע, כערך של משתנה או כביטוי. הכרזה של פונקציה שכזו היא פשוטה, אבל איך כותבים את הקוד שלה? כיצד יודעים מהם הטיפוסים של הפרמטרים? הערכים של מערך פתוח בעל טיפוס משתנה תואמים את רכיבי הטיפוס TVarRec.[40]

להלן המבנה של טיפוס הרשומה TVarRec:

```
type
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
```

[40] אל תבלבלו בין טיפוס הרשומה TVarRec לבין טיפוס הרשומה TVarData, שנעשה בו שימוש על ידי הטיפוס הנתונים וריאנט (Variant). לשני מבנים אלה מטרות שונות והם אינם תואמים. אפילו רשימת הטיפוסים האפשריים שונה, מכיוון ש-TVarRec מסוגל לאחסן טיפוסי נתונים של דלפי ואילו TVarData מאחסן טיפוסי נתונים של Windows OLE.

```
      vtChar:        (VChar: Char);
      vtExtended:    (VExtended: PExtended);
      vtString:      (VString: PShortString);
      vtPointer:     (VPointer: Pointer);
      vtPChar:       (VPChar: PChar);
      vtObject:      (VObject: TObject);
      vtClass:       (VClass: TClass);
      vtWideChar:    (VWideChar: WideChar);
      vtPWideChar:   (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:    (VCurrency: PCurrency);
      vtVariant:     (VVariant: PVariant);
      vtInterface:   (VInterface: Pointer);
  end;
```

לכל רשומה אפשרית יש שדה בשם VType, אם כי קשה לראות אותו במבט ראשון מכיוון שהוא מוגדר רק פעם אחת, ביחד עם הנתון בגודל Integer בהתחלה (לרוב, הוא מציין הפניה או מצביע). בעזרת המידע הזה אנו יכולים לכתוב פונקציה שמסוגלת לפעול על טיפוסי נתונים שונים. בדוגמה של הפונקציה SumAll להלן, ברצוני לסכם ערכים של טיפוסים שונים, להמיר מחרוזות למספרים, תווים לטיפוס הסודר המתאים ואת הערך True של משתנה בוליאני ל-1.

הקוד מבוסס על הצהרת case והוא פשוט למדי, אם כי עלינו לממש לעתים קרובות הפניות של מצביעים:

```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;

begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger: Result :=
        Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtChar:
        Result := Result + Ord (Args [I].VChar);
      vtExtended:
        Result := Result + Args [I].VExtended^;
      vtString, vtAnsiString:
        Result := Result + StrToIntDef ((
          Args [I].VString^), 0);
      vtWideChar:
        Result := Result + Ord (Args [I].VWideChar);
      vtCurrency:
        Result := Result + Args [I].VCurrency^;
    end; // case
end;
```

הוספתי את הקוד הזה לדוגמה OpenArr, שקוראת לפונקציה SumAll בקטע הבא:

```
var
  X: Extended;
  Y: Integer;
```

```
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34, '99999']);
  writeln (Format (
    'SumAll ([Y*Y, ''k'', True, 10.34, ''99999'']) => %n',
   [X]));
end;
```

את הפלט של קריאה זו אפשר לראות כאן:

```
SumAll ([Y*Y, 'k', True, 10.34, '99999']) => 10,217.34
```

מוסכמות קריאה של דלפי

עם גרסת 32 הביטים של דלפי הגיעה גישה חדשה להעברת פרמטרים, שמכונה בשם fastcall: כאשר הדבר אפשרי, ניתן להעביר עד שלושה פרמטרים ברגיסטרים (registers) של המעבד הראשי ולהאיץ על ידי כך מאד את הקריאה לפונקציה. מוסכמת הקריאה המהירה הזו (שמהווה את ברירת המחדל מגרסה 3 של דלפי והלאה) מצוינת על ידי מילת המפתח register.

הבעיה היא שגישת ברירת מחדל זו, ופונקציות שמשתמשות בה, אינן תואמות ל-Windows. את הפונקציות של Win32 API יש להכריז בעזרת מוסכמת הקריאה stdcall, שהיא שילוב של מוסכמת הקריאה המקורית של שפת פסקל ל-Win16 API ושל מוסכמת הקריאה cdecl של שפת C.

בדרך כלל אין שום סיבה להימנע ממוסכמת הקריאה המהירה החדשה, אלא אם אתם מבצעים קריאות חיצוניות לפונקציות של Windows או מגדירים פונקציות callback עבור Windows. לפני סוף הפרק נראה דוגמה שמשתמשת במוסכמה stdcall. סיכום של מוסכמות הקריאה של דלפי אפשר למצוא תחת הנושא "Calling conventions" שבעזרה של דלפי.

מהי מתודה?

אם כבר עבדתם עם דלפי או קראתם את המדריכים למשתמש, בוודאי שמעתם את המונח "מתודה" (Method). מתודה היא סוג מיוחד של פונקציה או פרוצדורה, שמקושרת לטיפוס נתונים מסוג מחלקה (Class). בדלפי, בכל פעם שאנו מטפלים באירוע מערכת (Event) עלינו להגדיר מתודה – לרוב פרוצדורה. עם זאת, המונח "מתודה" ככלל משמש לציון הן פונקציות והן פרוצדורות שקשורות למחלקה.

הנה מתודה ריקה שדלפי מוסיפה באופן אוטומטי לקוד המקור של טופס:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { כאן נכתב הקוד שלכם }
end;
```

הכרזות מקדימות

כאשר אתם רוצים להשתמש במזהה (מכל סוג שהוא), המהדר חייב לקבל מראש הכרזה כלשהי כדי לדעת למה המזהה הזה מתייחס. זו הסיבה לכך שאנו מספקים, לרוב, הכרזה מלאה לפני שאנו משתמשים ברוטינה כלשהי. עם זאת, ישנם מקרים בהם הדבר אינו אפשרי. אם פרוצדורה A קוראת לפרוצדורה B, ופרוצדורה B קוראת לפרוצדורה A, יהיה עליכם לכתוב בשלב כלשהו קוד שיקרא לרוטינה שהמהדר טרם ראה את ההכרזה שלה.

אם אתם רוצים להכריז על קיום של פרוצדורה או פונקציה בעלת שם ופרמטרים נתונים, מבלי לספק את הקוד שלה בפועל, תוכלו לכתוב אותה עם מילת הקוד forward אחריה:

```
procedure Hello; forward;
```

הקוד שלכם יצטרך לספק הגדרה מלאה של הפרוצדורה מאוחר יותר, אך ניתן יהיה לקרוא לה גם לפני שהוגדרה במלואה. הנה דוגמה טפשית, רק בשביל להמחיש את הרעיון:

```
procedure DoubleHello; forward;

procedure Hello;
begin
  if MessageDlg ('Do you want a double message?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    DoubleHello
  else
    ShowMessage ('Hello');
end;

procedure DoubleHello;
begin
  Hello;
  Hello;
end;
```

גישה זו מאפשרת לכם לכתוב רקורסיה הדדית: DoubleHello קוראת ל-Hello, אך גם Hello עשויה לקרוא ל-DoubleHello. כמובן שחייב להיות תנאי שיסיים את הרקורסיה, כדי למנוע גלישת מחסנית (Stack overflow).

הכרזות מקדימות לפרוצדורות אמנם אינן נפוצות בפסקל, אך קיים מקרה דומה נפוץ הרבה יותר. כאשר אתם מכריזים על פרוצדורה או על פונקציה בחלק הממשק (interface) של יחידה (עוד על יחידות בפרק הבא), ההכרזה נחשבת מקדימה אפילו אם המילה forward לא מופיעה. למעשה, אינכם יכולים לכתוב את "גוף" הרוטינה בחלק הממשק של היחידה. במקביל, עליכם לספק את היישום בפועל של כל רוטינה שהכרזתם בתוך אותה יחידה[41].

[41] אותו הדבר נכון לגבי הכרזות של מתודות בתוך טיפוסי מחלקה שהופקו באופן אוטומטי על ידי סביבת הפיתוח המשולבת של דלפי (עם הוספת אירוע לטופס או לרכיב בו). מנהלי האירועים שהוכרזו בתוך המחלקה TForm הינם הכרזות מקדימות: הקוד יסופק בחלק היישום (implementation) של היחידה.

טיפוסים פרוצדורליים

מאפיין ייחודי נוסף של Object Pascal הוא קיומם של טיפוסים פרוצדורליים (Procedural types). למעשה, מדובר בנושא ברמה גבוהה שרק מתכנתי דלפי מעטים משתמשים בו ביומיום. עם זאת, מכיוון שבפרקים הבאים נעסוק בנושאים קשורים (מצביעים למתודות, טכניקה שדלפי עושה בה שימוש נרחב), כדאי להעיף בו מבט. אם אתם מתכנתים מתחילים, תוכלו לדלג בינתיים על הסעיף הזה ולחזור אליו כשתרגישו מוכנים.

בשפת פסקל קיים המושג של טיפוס פרוצדורלי (דומה לרעיון של מצביע לפונקציה בשפת C). הכרזה של סוג פרוצדורלי מציינת את רשימת הפרמטרים, ואם מדובר בפונקציה, גם את טיפוס הערך החוזר. לדוגמה, בעזרת הקוד הבא אתם יכולים להכריז על טיפוס פרוצדורלי חדש עם פרמטר מטיפוס Integer שמועבר בהפניה:

```
type
  TIntProc = procedure (var Num: Integer);
```

טיפוס פרוצדורלי זה תואם לכל רוטינה שיש לה את אותם הפרמטרים בדיוק (או את אותה "חתימת הפונקציה", אם נשתמש בעגת C). הנה דוגמה לרוטינה תואמת[42]:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

ניתן להשתמש בטיפוסים פרוצדורליים לשתי מטרות שונות: אפשר להכריז על משתנים מטיפוס פרוצדורלי, או להעביר טיפוס פרוצדורלי (כלומר מצביע לפונקציה) כפרמטר לרוטינה אחרת. בהינתן הטיפוס הפרוצדורלי והפרוצדורה שהוגדרו קודם לכן, אפשר לכתוב את הקוד הבא:

```
var
  IP: TIntProc;
  X: Integer;

begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

קוד זה יביא לאותה תוצאה כמו הגרסה הקצרה יותר שלהלן:

```
var
  X: Integer;

begin
  X := 5;
  DoubleTheValue (X);
```

[42] בגרסת 16 הביטים של דלפי, אם מתכוונים להשתמש ברוטינות כערכים בפועל של טיפוס פרוצדורלי, יש להכריז עליהן בתוספת ההנחיה far .

```
end;
```

הגרסה הראשונה היא בבירור מסובכת יותר, אז לשם מה להשתמש בה? במקרים מסוימים, היכולת להחליט לאיזו פונקציה לקרוא ולממש את הקריאה מאוחר יותר יכולה להיות שימושית. אפשר ליצור דוגמה מורכבת שתציג את הגישה הזו, אך אני מעדיף לתת לכם לבחון דוגמה פשוטה יחסית בשם ProcType. דוגמה זו מורכבת יותר ממה שראינו עד כה, וזאת כדי להפוך את הסיטואציה למוחשית קצת יותר.

דוגמה זו מבוססת על שתי פרוצדורות. אחת מהן משמשת להכפלת הערך של הפרמטר, והיא דומה לגרסה שהוצגה בסעיף זה. הפרוצדורה השניה משמשת לשילוש הערך של הפרמטר, ועל כן היא נקראת TripleTheValue:

```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
  writeln ('Value tripled: ' + IntToStr (Value));
end;
```

שתי הפרוצדורות מראות מה מתרחש על מנת שנוכל לדעת שבוצעה קריאה להן. זהו מאפיין איתור שגיאות ("דיבוג") פשוט שאפשר להשתמש בו כדי לבדוק אם קטע קוד מסוים בוצע, במקום להשתמש בנקודת עצירה (Breakpoint).

כדי להדגים את השימוש בטיפוסים פרוצדורליים, במקום לקרוא ישירות לפרוצדורות השתמשתי בגישה ארוכה אך מעניינת יותר. בכל פעם שהמשתמש מזין 2 או 3 באמצעות המקלדת, אחת מהפרוצדורות מאוחסנת במשתנה:

```
var
  ch: Char;
  IP: TIntProc;
  X: Integer;

begin
  // ערכי ברירת מחדל התחלתיים
  X := 10;
  IP := DoubleTheValue;

  while True do
  begin
    // בצע קריאה מהמקלדת וקרא לפרוצדורה
    read (ch);
    case ch of
      '2': IP := DoubleTheValue;
      '3': IP := TripleTheValue;
      'x': Break;
      else
        IP (X);
  end;
end;
```

כאשר המשתמש לוחץ על מקש אחר כלשהו (ענף ה-else של הצהרת ה-case לעיל), הפרוצדורה שאוחסנה מבוצעת. תוכלו לראות דוגמה מעשית יותר של השימוש בטיפוסים פרוצדורליים בפרק 9, בסעיף "פונקצית callback של Windows".

העמסת פונקציות

רעיון ההעמסה (Overloading) הוא פשוט: המהדר מאפשר לכם להגדיר שתי פונקציות או פרוצדורות בעלות אותו שם, כל עוד הפרמטרים שלהן שונים. למעשה, על ידי בדיקת הפרמטרים מסוגל המהדר לקבוע לאיזו גרסה של הרוטינה אתם מבקשים לקרוא. הנה למשל סדרה של פונקציות שנלקחו מתוך היחידה Math של ה-VCL:

```
function Min (A,B: Integer): Integer; overload;
function Min (A,B: Int64): Int64; overload;
function Min (A,B: Single): Single; overload;
function Min (A,B: Double): Double; overload;
function Min (A,B: Extended): Extended; overload;
```

כאשר אתם קוראים ל-Min(10, 20), המהדר קובע בקלות שאתם מתכוונים לפונקציה הראשונה ברשימה, ולכן הערך החוזר יהיה מטיפוס Integer.

ישנם שני כללים בסיסיים:

- כל גרסה של הרוטינה חייבת להיות מלווה, בסיום ההכרזה, במילת המפתח overload.
- ההבדלים חייבים להיות במספר או בטיפוס הפרמטרים, או בשניהם. הערך החוזר אינו יכול לשמש להבחנה בין שתי רוטינות.

להלן שלוש גרסאות מועמסות של של הפרוצדורה ShowMsg שהוספתי לדוגמה OverDef (יישום שמדגים העמסה ופרמטרים של ברירת מחדל):

```
procedure ShowMsg (str: string); overload;
begin
  writeln ('Message: ' + str);
end;

procedure ShowMsg (FormatStr: string;
  Params: array of const); overload;
begin
  writeln ('Message: ' + Format (FormatStr, Params));
end;

procedure ShowMsg (I: Integer; Str: string); overload;
begin
  ShowMsg (IntToStr (I) + ' ' + Str);
end;
```

שלוש הפרוצדורות מציגות תיבת הודעה עם מחרוזת, לאחר עיצוב אופציונלי של המחרוזת בדרכים שונות. הנה שלוש הקריאות[43] שבתוכנית:

[43] טכנולוגיית Code Parameters של סביבת הפיתוח המשולבת של דלפי עובדת היטב עם פרוצדורות ופונקציות מועמסות. כאשר אתם מקלידים את הסוגריים הפותחים לאחר שם הרוטינה, כל האפשרויות הזמינות מוצגות. עם הזנת הפרמטרים, דלפי משתמשת בטיפוסים שלהם כדי לקבוע אילו מהחלופות עדיין רלוונטיות.

```
ShowMsg ('Hello');
ShowMsg ('Total = %d.', [100]);
ShowMsg (10, 'MBytes');
```

ואלה תוצאותיהן:

```
Message: Hello
Message: Total = 100.
Message: 10 MBytes
```

העובדה שצריך לציין בצורה מפורשת כל רוטינה מועמסת מלמדת שלא ניתן להעמיס על רוטינה קיימת באותה יחידה, שאינה מסומנת בעצמה במילת המפתח overload. הודעת השגיאה שמתקבלת אם מנסים לעשות זאת היא:

```
Previous declaration of '<name>' was not marked with the 'overload'
directive.
```

עם זאת, ניתן להעמיס רוטינה שהוכרזה במקור ביחידה אחרת[44]. אפשרות זו נועדה לצורך תאימות עם גרסאות קודמות של דלפי, שאפשרו ליחידות שונות להשתמש באותם שמות רוטינות. בכל אופן, מקרה מיוחד זה אינו באמת מאפיין נוסף של העמסה, מכיוון שהשפעתו קטנה מאד.

לדוגמה, ניתן להוסיף לתוכנית (במקרה זה, הדוגמה OverDef) את הקוד הבא:

```
procedure MessageBox (str: string); overload;
begin
  Windows.MessageBox (0, PChar(str), 'Title', MB_OK);
end;
```

קוד זה אינו מועמס כלל על הרוטינה המקורית MessageBox של ה-API של Windows. למעשה, אם תנסו לקרוא לגרסה המקורית בעזרת הקוד:

```
MessageBox (0, 'Message', 'Title', MB_OK);
```

תקבלו הודאת שגיאה נאה שמציינת שכמה מהפרמטרים חסרים. הדרך היחידה לקרוא לגרסה הרשמית במקום לזו המקומית הוא להפנות במפורש ליחידה, פעולה שסותרת את עצם הרעיון של ההעמסה:

```
Windows.MessageBox (0, 'Message', 'Title', MB_OK);
```

פרמטרים של ברירת מחדל

מאפיין נוסף שקשור להעמסה הוא שניתן להעניק ערך ברירת מחדל לפרמטר או לפרמטרים של רוטינה, כך שתוכלו לקרוא לאותה רוטינה עם או ללא הפרמטר. אם הפרמטר יהיה חסר בקריאה, הוא יקבל את ערך ברירת המחדל.

[44] למידע נוסף על יחידות עיינו בפרק 11.

הרשו לי להדגים. אנו יכולים להגדיר את הכימוס הבא של הקריאה write, תוך אספקת שני פרמטרים של ברירת מחדל:

```
procedure NewMessage (Msg: string;
  Caption: string = 'Message';
  Separator: string = sLineBreak);
begin
  write (Caption);
  write (': ');
  write (Msg);
  write (Separator);
end;
```

בעזרת הגדרה זו ניתן לקרוא לפרוצדורה בכל אחת מהדרכים הבאות:

```
NewMessage ('Something wrong here!');
NewMessage ('Something wrong here!', 'Attention');
NewMessage ('Hello', 'Message', '--');
```

הפלט נראה כך:

```
Message: Something wrong here!
Attention: Something wrong here!
Message: Hello--
```

שימו לב שדלפי אינה יוצרת קוד מיוחד כלשהו לתמיכה בפרמטרים של ברירת מחדל; היא גם אינה מייצרת עותקים מרובים של הרוטינות. הפרמטרים החסרים פשוט מוספים על ידי המהדר לקוד הקורא. קיימת מגבלה אחת חשובה שמשפיעה על השימוש בפרמטרים של ברירת מחדל: לא ניתן "לדלג" על פרמטרים. לדוגמה, לא תוכלו להעביר את הפרמטר השלישי של פונקציה לאחר שהשמטתם את השני.

הנה הכלל הראשי עבור פרמטרים של ברירת מחדל: בעת קריאה, ניתן להשמיט פרמטרים רק החל מהאחרון. במילים אחרות, אם אתם משמיטים פרמטר, עליכם להשמיט גם את הבאים אחריו.

לפרמטרים של ברירת המחדל יש מספר כללים נוספים:

- פרמטרים בעלי ערכים של ברירת מחדל חייבים להיות בסוף רשימת הפרמטרים.

- ערכי ברירת מחדל חייבים להיות קבועים. הדבר מגביל, כמובן, את הטיפוסים בהם ניתן להשתמש עבור פרמטרים של ברירת מחדל. לדוגמה, מערך דינמי או טיפוס ממשק (interface) אינם יכולים לקבל פרמטר של ברירת מחדל למעט nil, וכלל לא ניתן להשתמש ברשומות.

- פרמטרים של ברירת מחדל חייבים להיות מועברים לפי ערך או כ-const. פרמטר הפניה (var) אינו יכול לקבל ערך ברירת מחדל.

שימוש בפרמטרים של ברירת מחדל ובהעמסה בו זמנית עלול לגרום ללא מעט בעיות, ושני המאפיינים הללו עלולים להתנגש. לדוגמה, אם אני מוסיף לדוגמה הקודמת את הגרסה החדשה הבאה לפרוצדורה NewMessage:

```
procedure NewMessage (Str: string; I: Integer = 0);
  overload;
begin
  writeln (Str + '; ' + IntToStr (I))
end;
```

המהדר לא יתלונן מכיוון שזו הגדרה חוקית. עם זאת, הקריאה:

```
NewMessage ('Hello');
```

תסומן על ידו כ:

```
NewMessageTest.dpr(24):
E2251 Ambiguous overloaded call to 'NewMessage'
```

שימו לב ששגיאה זו מוצגת עבור שורה שהודרה היטב לפני ההגדרה המעמיסה החדשה. בפועל, אין לנו שום דרך לקרוא לפרוצדורה NewMessage עם פרמטר מחרוזת יחיד, מכיוון שהמהדר אינו יודע האם ברצוננו לקרוא לגרסה עם פרמטר המחרוזת בלבד או לגרסה עם המחרוזת ופרמטר ה-Integer תוך שימוש בערך ברירת המחדל. כאשר הוא ניצב בפני התלבטות כזו, המהדר עוצר ומבקש מהמתכנת לציין את כוונותיו ביתר בהירות.

סיכום

כתיבה של פרוצדורות ושל פונקציות היא מרכיב מפתח בתכנות, ורוב המאפיינים שלהן משותפים גם למתודות (המקבילות להן בתכנות מונחה עצמים). עם זאת, במקום לעבור למאפיינים של תכנות מונחה עצמים, הפרקים הבאים יספקו לכם פרטים על מרכיבים אחרים של התכנות בפסקל, והראשונים מביניהם – מחרוזות.

פרק 7: טיפול במחרוזות

הטיפול במחרוזות בפסקל נראה פשוט למדי, אך מאחורי הקלעים מדובר בעסק מסובך. לפסקל יש דרך מסורתית לטפל במחרוזות, למערכת ההפעלה Windows יש דרך משלה, שנשאלה משפת התכנות C, והגרסאות המודרניות של פסקל כוללות טיפוס נתונים רב-עוצמה של מחרוזת ארוכה, שמהווה כעת את טיפוס המחרוזות של ברירת המחדל בדלפי.

טיפוסי מחרוזות

בימים הראשונים של טורבו פסקל ושל דלפי 16 ביט, טיפוס המחרוזת הרגיל היה רצף של תווים בעל בית (Byte) אורך בהתחלה, שציין את האורך הנוכחי של המחרוזת. מכיוון שהאורך צוין באמצעות בית יחיד, הוא הוגבל ל-255 תווים, ערך נמוך מאד שיצר בעיות רבות בעיבוד מחרוזות. כל מחרוזת הוגדרה עם אורך קבוע (שהיה, כברירת מחדל, הערך המקסימלי של בית – 255), אם כי ניתן היה להכריז על מחרוזות קצרות יותר כדי לחסוך מקום בזכרון.

טיפוס מחרוזת דומה לטיפוס מערך. למעשה, מחרוזת היא כמעט מערך של תווים. המחשה לכך היא העובדה שניתן לגשת לתו ספציפי במחרוזת באמצעות הסימון [], בדומה לשפת C.

כדי להתגבר על המגבלות של מחרוזות פסקל מסורתיות, הגרסאות המודרניות של פסקל תומכות במחרוזות ארוכות. למעשה, ישנם שלושה טיפוסי מחרוזות:

- הטיפוס ShortString תואם למחרוזות פסקל המקוריות שתוארו לעיל. למחרוזות אלה יש מגבלה של 255 תווים והן מקבילות למחרוזות בגרסת 16 הביט של דלפי. כל מרכיב במחרוזת קצרה שכזו הוא מטיפוס ANSIChar (טיפוס התו הרגיל).

- הטיפוס ANSIString תואם למחרוזות החדשות והארוכות בעלות האורך המשתנה. מחרוזות אלה מוקצות דינמית, ההפניות שלהן נספרות (כלומר שאינכם צריכים לדאוג לשחרור הזכרון בו הן משתמשות), והן עושות שימוש בטכניקת העתקה-בעת-כתיבה (Copy-on-write). האורך של מחרוזות אלה כמעט ואינו מוגבל (הן יכולות לאחסן עד שני מיליארדי תווים!). גם הן מבוססות על הטיפוס ANSIChar.

- הטיפוס WideString דומה לטיפוס ANSIString, אך הוא מבוסס על הטיפוס WideChar (שמאחסן תווי Unicode). הוא אינו חזק ויעיל כמו טיפוסי המחרוזות הסטנדרטיים, מכיוון שהתמיכה שלו בספירת הפניות אינה מקיפה באותה מידה.

מחרוזות פסקל מסורתיות

מחרוזות פסקל מסורתיות הן מבנה נתונים פשוט ואפקטיבי מאד. באפשרותכם להגדיר מחרוזת שמכילה לכל היותר עשרים תווים באמצעות ההכרזה:

```
var
  strShortName: ShortString [20];[45]

begin
  strShortName := 'marco';
```

הנתונים יוקצו מקומית (במחסנית, אותו אזור בזכרון שמשמש פונקציות ופרוצדורות עבור המשתנים המקומיים שלהן) ולא דינמית[46]. במקרה זה אתם מנצלים 21 בתים, 20 עבור התווים ואחד עבור האורך. נהוג לכנות בית זה בשם "בית אורך" (Length byte), והוא מאוחסן בתחילת נתוני המחרוזת. שימו לב שלמחרוזת יש גודל מרבי של 20 תווים, אך במהלך תקופת חייה האורך שלה עשוי להשתנות. בעקבות ההקצאה בדוגמה שלמעלה, האורך בפועל יהיה 5 והמחרוזת תתפוס 6 בתים.

הכרזת אורך המחרוזת מראש מגבילה אתכם לגודל המרבי שצוין ולא תוכלו לעבור אותו בשום מקרה. יותר מזה, הגודל המרבי שניתן להכריז הוא 255 פשוט מכיוון שבית האורך הוא בית, וככזה יכול לייצג מספרים רק בין 0 ל-255.

מצד שני, מחרוזות קצרות כאלה הן מהירות מאד (מכיוון שלא מעורבים בהן שום הקצאה דינמית, ניקוי או ספירת הפניות). אורכן הקבוע מאפשר אחסון קל ברשומות ובמבני נתונים אחרים. גודל מחרוזת קבוע הוא מגבלה, אך כזו שמפתח מסדי נתונים, למשל, יכול לחיות איתה בלי בעיה.

כיום, השימוש במחרוזות קצרות בפסקל אינו נפוץ, אף על פי שבפסקל המסורתי הן בהחלט היוו מבנה נתונים עיקרי. לכן העדפתי להתמקד בפרק זה במחרוזות ארוכות.

שימוש במחרוזות ארוכות

אם אתם משתמשים בטיפוס הנתונים string, תקבלו מחרוזות קצרות או מחרוזות ANSI בהתאם לערך של הנחיית המהדר $H. ברירת המחדל $H+ מציינת מחרוזות ארוכות (מטיפוס ANSIString), אותו הטיפוס שמשמש ברכיבים של ספריית דלפי.

המחרוזות הארוכות של פסקל מבוססות על מנגנון ספירת הפניות, שעוקב אחר מספר משתני המחרוזת שמצביעים לאותה מחרוזת שבזכרון. ספירת הפניות זו משמשת גם לשחרור הזכרון כאשר לא נעשה במחרוזת כל שימוש נוסף – כלומר, כאשר מספר ההפניות שלה מגיע לאפס.

[45] אפשר לכתוב זאת גם בעזרת הסימון הקלאסי - StrShortName: string[20];

[46] מיקום המחרוזת מאחסן את הנתונים עצמם, לא מצביע לנתונים הממשיים כפי שנעשה במחרוזות ארוכות.

אם אתם מעוניינים להגדיל את המחרוזת בזכרון אך המיקומים הצמודים תפוסים על ידי דבר-מה אחר, המחרוזת אינה יכולה לגדול במקומה הנוכחי ויש לבצע העתקה מלאה של המחרוזת למיקום אחר. במצב כזה, התמיכה של זמן הריצה מקצה מחדש את המחרוזת עבורכם באופן שקוף לגמרי. אתם פשוט קובעים את הגודל המרבי של המחרוזת באמצעות הפרוצדורה SetLength, שמקצה למעשה את כמות הזכרון הדרושה:

```
SetLength (String1, 200);
```

הפרוצדורה SetLength מבצעת בקשה להקצאת זכרון, לא את ההקצאה עצמה. היא שומרת את המקום הדרוש בזכרון לשימוש עתידי ולא משתמשת בו בפועל. טכניקה זו מבוססת על מאפיין של מערכת ההפעלה Windows ומשמשת עבור כל הקצאות הזכרון הדינמיות של דלפי. לדוגמה, כאשר אתם מבקשים מערך גדול מאד, הזכרון נשמר אך אינו מוקצה.

רק לעתים רחוקות יש צורך לקבוע את אורך המחרוזת. המקרה היחיד בו חייבים להקצות זכרון עבור מחרוזת ארוכה בעזרת SetLength הוא כאשר צריך להעביר את המחרוזת כפרמטר לפונקציית API של Windows (לאחר הטלת טיפוס מתאימה), כפי שאראה בקרוב.

המחרוזות בזכרון

כדי לסייע לכם להבין טוב יותר את הפרטים של ניהול הזכרון עבור מחרוזות כתבתי את הדוגמה הפשוטה StrRef. בתוכנית זו הגדרתי שתי מחרוזות גלובליות: Str1 ו-Str2. התוכנית מקצה מחרוזת קבועה לראשון מבין שני המשתנים, ולאחר מכן מקצה את המשתנה הראשון לשני:

```
Str1 := 'Hello';
Str2 := Str1;
```

פרט לעבודה עם המחרוזות, התוכנית מציגה את המצב הפנימי שלהן בתיבת רשימה, וזאת באמצעות הפונקציה StringStatus הבאה:

```
function StringStatus (const Str: string): string;
begin
  Result := 'Address: ' +
    IntToStr (Integer (Str)) +
    ', Length: ' +
    IntToStr (Length (Str)) +
    ', References: ' +
    IntToStr (PInteger (Integer (Str) - 8)^) +
    ', Value: ' + Str;
end;
```

ההעברה של פרמטר המחרוזת כפרמטר const לפונקציה StringStatus היא חיונית. העברת הפרמטר באמצעות העתקה תגרום לתופעת לוואי של הפניה אחת נוספת למחרוזת בעת הרצת הפונקציה. לעומת זאת, העברה של הפרמטר דרך הפניה (var) או כקבוע (const) אינה יוצרת הפניה נוספת למחרוזת. במקרה זה השתמשתי בפרמטר const מכיוון שהפונקציה אינה מיועדת לשנות את המחרוזת.

כדי להשיג את הכתובת של המחרוזת בזכרון (לצורך קביעת זהותה האמיתית ובדיקה מתי שתי מחרוזות שונות מפנות לאותו מקום בזכרון), ביצעתי הטלת טיפוס נוקשה מטיפוס מחרוזת לטיפוס Integer. מחרוזות ארוכות הן הפניות – למעשה, הן מצביעים: הערך שלהן מחזיק את המיקום של המחרוזת בפועל בזכרון, לא את המחרוזת עצמה.

כדי לחלץ את ספירת ההפניות ביססתי את הקוד על עובדה שלא רבים מכירים – האורך וספירת ההפניות מאוחסנים למעשה במחרוזת, לפני הטקסט עצמו ולפני המיקום אליו מצביע משתנה המחרוזת. ההזחה (השלילית) היא 4- עבור אורך המחרוזת (ערך שניתן לחלץ ביתר קלות באמצעות הפונקציה Length) ו-8- עבור ספירת ההפניות[47].

הרצת הדוגמה תיתן לכם שתי מחרוזות בעלות אותו תוכן, באותו מיקום בזכרון ובעלות ספירת הפניות שערכה 2:

```
Str1 - Address: 13419088, Length: 5,
  References: 2, Value: Hello
Str2 - Address: 13419088, Length: 5,
  References: 2, Value: Hello
```

כעת, אם תשנו את הערך של אחת משתי המחרוזות (לא משנה מי), המיקום בזכרון של המחרוזת שהשתנתה ישתנה גם הוא. זוהי תוצאה של טכניקת ההעתקה-בעת-כתיבה. למעשה, אנו יכולים ליצור את האפקט הזה באמצעות כתיבת הקוד הבא:

```
  Str1 [2] := 'a';
  writeln ('Str1 [2] := ''a''');
  writeln ('Str1 - ' + StringStatus (Str1));
  writeln ('Str2 - ' + StringStatus (Str2));
```

הפלט שמתקבל הוא:

```
Str1 - Address: 13419112, Length: 5,
  References: 1, Value: Hallo
Str2 - Address: 13419088, Length: 5,
  References: 1, Value: Hello
```

אתם יכולים להרחיב את הדוגמה הזו כרצונכם ולהשתמש בפונקציה StringStatus כדי לחקור את ההתנהגות של מחרוזות ארוכות במגוון נסיבות שונות.

מחרוזות של דלפי ו-PChar של Windows

נקודה חשובה נוספת לטובת המחרוזות הארוכות היא שהן בעלות סיום null (null-terminated). פירוש הדבר שהן תואמות לגמרי למחרוזות בסיום null של שפת C, בהן נעשה שימוש ב-Windows. מחרוזת בסיום null היא רצף של תווים ואחריהם בית שערכו אפס (בשם אחר, null). הדבר יכול לבוא לידי ביטוי בדלפי באמצעות מערך מבוסס-אפס של תווים,

[47] זכרו שמידע פנימי זה על ההזחות עשוי להשתנות בכל גרסה עתידית של דלפי; אין כל אחריות שמאפיינים לא מתועדים כאלה ואחרים יישמרו בעתיד.

אותו טיפוס נתונים שמשמש לרוב ליישום מחרוזות בשפת C. זו הסיבה לכך שמערכי תווים בסיום null נפוצים כל כך בפונקציות ה-API של Windows (שמבוססות על שפת C). מכיוון שהמחרוזות הארוכות של פסקל תואמות באופן מלא למחרוזות בסיום null של C, תוכלו פשוט להשתמש בהן ולהטיל אותן לטיפוס PChar כאשר תצטרכו להעביר מחרוזת לפונקציית API של Windows[48].

לדוגמה, כדי להעתיק את השם של משתמש Windows הנוכחי למחרוזת PChar (באמצעות פונקציית ה-API שנקראת GetUserName) ולהציג אותו:

```
var
  Str1: string;
  nSize : Cardinal = 20;

begin
  SetLength (Str1, nSize);
  GetUserName(PChar (Str1), nSize);
  writeln (str1);
end;
```

את הקוד הזה תוכלו למצוא בדוגמה StringAndPChar. שימו לב שאם תכתבו את הקוד הזה מבלי להקצות קודם לכן זכרון עבור המחרוזת באמצעות SetLength, סביר להניח שהתוכנית תקרוס. אם אתם משתמשים ב-PChar כדי להעביר ערך, ולא כדי לקבל ערך כפי שנעשה בקוד לעיל, הקוד יהיה פשוט יותר כי לא יהיה צורך להגדיר מחרוזת זמנית ולאתחל אותה.

לאחר שהצגתי את התמונה היפה, ברצוני להתמקד במכשולים. קיימות מספר בעיות שעלולות לצוץ כאשר אתם ממירים מחרוזת ארוכה ל-PChar. בעיקרון, הבעיה הבסיסית היא שלאחר ההמרה אתם הופכים לאחראים למחרוזת ולתוכנה, ושפת פסקל לא תסייע לכם עוד. הביטו בשינוי הקטן בדוגמה StringAndPChar, כאשר מוסיפים עוד קצת טקסט למחרוזת הפלט:

```
  GetUserName(PChar (Str1), nSize);
  writeln (str1 + '*');
```

תוכנית זו עוברת הדרה, אך כשתריצו אותה צפויה לכם הפתעה; מחרוזת הפלט תהיה:

```
Marco                *
```

(זהו השם שלי, אחריו 15 רווחים ואחריהם כוכבית). הבעיה היא שכאשר Windows כותבת למחרוזת (בתוך קריאת GetUserName של ה-API), מערכת ההפעלה אינה קובעת בצורה נכונה את האורך של מחרוזת פסקל הארוכה. שפת פסקל עדיין יכולה להשתמש במחרוזת זו לצורך פלט ולהבין מתי היא נגמרת באמצעות חיפוש סיום ה-null התואם לשפת C (ושמתווסף על ידי Windows), אך היא תוסיף כל טקסט נוסף אחרי הסוף ו(בדוגמאות אחרות) תדלג על טקסט שיתווסף אחרי תו ה-null.

[48] כאשר אתם צריכים להטיל WideString לטיפוס תואם Windows, עליכם להשתמש לצורך ההמרה ב-PWideChar במקום ב-PChar. מחרוזות מטיפוס Wide משמשות לעתים קרובות בתוכניות COM.

כיצד לתקן את הבעיה? הפתרון הוא לומר למערכת להמיר את המחרוזת שהוחזרה מקריאת GetUserName של ה-API בחזרה לפסקל. עם זאת, אם תכתבו את הקוד הבא:

```
Str1 := String (Str1);
```

המערכת תתעלם ממנו, מכיוון שהמרה של טיפוס נתונים לעצמו היא פעולה חסרת משמעות. כדי לקבל את המחרוזת הארוכה הרצויה של פסקל, עליכם להטיל מחדש את המחרוזת ל-PChar ולתת לפסקל להמיר אותה בחזרה ל-string כהלכה:

```
Str1 := String (PChar (Str1));
```

למעשה, באפשרותכם לדלג על המרת המחרוזת מפני שהמרות PChar-ל-string הן אוטומטיות בפסקל, כך שאפשר לכתוב:

```
GetUserName(PChar (Str1), nSize);
Str1 := PChar (Str1);
writeln (Str1 + '*');
```

ולקבל את התוצאה הצפויה:

```
Marco*
```

דרך חלופית היא לאפס את האורך של מחרוזת הפסקל בהתאם לאורך של מחרוזת ה-PChar, וזאת באמצעות הקוד:

```
SetLength (Str1, StrLen (PChar (Str1)));
```

את רוב קטעי הקוד הללו תוכלו למצוא בדוגמה StringAndPChar.

עיצוב מחרוזות

ניתן לבנות מחרוזות מורכבות מתוך ערכים קיימים באמצעות האופרטור פלוס (+) וחלק מפונקציות ההמרה (כגון IntToStr). עם זאת, קיימת גישה אחרת לעיצוב מספרים, ערכי מטבע ומחרוזות אחרות לכדי מחרוזת סופית. אפשר להשתמש בפונקציה Format או באחת מהפונקציות הנלוות לה.

הפונקציה Format מקבלת כפרמטרים מחרוזת, שמכילה את הטקסט הבסיסי וכן מספר שומרי מקום (Placeholders, שמסומנים לרוב בסימן %), וכן מערך של ערכים – אחד עבור כל שומר מקום. לדוגמה, כדי לעצב מחרוזת משני מספרים אפשר לכתוב:

```
Format ('First %d, Second %d', [n1, n2]);
```

כאשר n1 ו-n2 הם שני ערכי מספרים שלמים. שומר המקום הראשון מוחלף בערך הראשון, השני בערך השני וכן הלאה. אם טיפוס הפלט של שומר המקום (שמצוין על ידי האות שלאחר

הסימן %) אינו תואם לטיפוס של הפרמטר המקביל מתרחשת שגיאת זמן ריצה. למעשה, היעדר בדיקת טיפוסים בזמן ההידור היא החסרון הגדול ביותר של השימוש בפונקציה Format. באותו האופן, שגיאת זמן ריצה תתרחש אם לא יועברו די פרמטרים.

הפונקציה Format משתמשת בפרמטר של מערך פתוח (לפרמטר כזה יכול להיות מספר ערכים שרירותי, כפי שהודגם בפרק 6). פרט לשימוש ב-%d תוכלו להשתמש בשומרי מקום רבים אחרים שמוגדרים על ידי פונקציה זו ומפורטים בקצרה בטבלה הבאה. שומרי מקום אלה מספקים פלט ברירת מחדל עבור כל טיפוס נתונים, אך באפשרותכם להשתמש במצייני עיצוב נוספים כדי לשנות את פלט ברירת המחדל. מציין רוחב, לדוגמה, מגדיר מספר קבוע של תווים בפלט, ואילו מציין דיוק קובע את מספר הספרות לאחר הנקודה. לדוגמה,

```
Format ('%8d', [n1]);
```

ממיר את המספר n1 למחרוזת באורך שמונה תווים, מיושרת לימין (השתמשו בסימן המינוס (-) כדי לציין יישור לשמאל) ומרופדת בסימנים לבנים. הנה הרשימה של שומרי המקום לעיצוב:

d (עשרוני)	ערך המספר השלם התואם מומר למחרוזת של ספרות עשרוניות.
x (הקסדצימלי)	ערך המספר השלם התואם מומר למחרוזת של ספרות הקסדצימליות.
p (מצביע)	ערך המצביע התואם מומר למחרוזת שמבוטאת בספרות הקסדצימליות.
s (מחרוזת)	המחרוזת, התו או ערך ה-PChar התואמים מועתקים למחרוזת הפלט
e (אקספוננציאלי)	הערך העשרוני התואם מומר למחרוזת בסימון מדעי.
f (נקודה עשרונית)	הערך העשרוני התואם מומר למחרוזת בסימון של נקודה עשרונית.
g (כללי)	הערך העשרוני התואם מומר למחרוזת העשרונית הקצרה ביותר האפשרית, בסימון נקודה עשרונית או בסימון מדעי.
n (מספר)	הערך העשרוני התואם מומר למחרוזת עשרונית, שעושה שימוש גם במפרידי אלפים.
m (כסף)	הערך העשרוני התואם מומר למחרוזת שמייצגת סכום כספי. ההמרה מבוססת על הגדרות אזוריות – ראו בקובץ העזרה של דלפי תחת Currency ומשתנים לעיצוב תאריך/שעה.

הדרך הטובה ביותר לראות דוגמאות להמרות אלה היא להתנסות לבד בעיצוב מחרוזות. כדי להקל עליכם במשימה זו כתבתי את התוכנית FmtTest, שמאפשרת למשתמש לספק מחרוזות עיצוב למספרים שלמים ועשרוניים. תוכנית זו אינטראקטיבית הרבה יותר מרוב התוכניות שבספר זה. ראשית היא שואלת אם לבצע עיצוב של מספר שלם או של מספר

עשרוני, ואז קוראת לאחת משתי רוטינות שונות בהתאם לקלט. להלן החלק העיקרי של התוכנית:

```
begin
  Done := False;
  while not Done do
  begin
    writeln ('Work with [I]nteger or [F]loating' +
      ' point numbers? [I or F or X to exit]');
    readln (chInput);

    case Upcase (chInput) of
      'I': TestFormatInteger;
      'F': TestFormatFloat;
      'X': Done := True;
    else
      writeln ('Wrong selection');
    end;
  end;
  writeln ('Bye');
  readln;
end.
```

כל אחת מהרוטינות מבקשת קלט מספרי ומחרוזת עיצוב, ומספקת מספר הצעות:

```
procedure TestFormatInteger;
var
  n: Integer;
  strFmt: string;

begin
  writeln ('Enter value');
  readln (n);

  writeln ('Enter a format string: (examples below)');
  // כאן דילגתי על קוד ההצעות
  readln (strFmt);

  writeln (Format ('%d %s => %s',
    [n, strFmt, Format (strFmt, [n])]));
end;
```

בעיקרון, הקוד מבצע את פעולת העיצוב תוך שימוש בקלט השני כמחרוזת עיצוב ובערך של הקלט הראשון כערך לעיצוב. עיצוב המספר העשרוני נעשה באופן דומה.

סיכום

מחרוזות הן טיפוס נתונים נפוץ מאד. ברוב המקרים תוכלו אמנם להשתמש בהן בבטחה גם בלי להבין איך הן עובדות, אך פרק זה נועד להבהיר את אופן התנהגותן המדויק כדי שתוכלו לנצל את מלוא כוחו של טיפוס הנתונים הזה.

המחרוזות מטופלות בזכרון בדרך דינמית מיוחדת, כפי שנעשה עבור מערכים דינמיים. זהו הנושא של הפרק הבא.

פרק 8: זכרון

פרק זה עוסק בטיפול בזכרון ובאזורים השונים בזכרון אליהם תוכלו לגשת ביישום, וכן מציג את המערכים הדינמיים.

זכרון גלובלי

הזכרון הגלובלי הוא האזור בזכרון בו שמורים משתנים גלובליים. כאשר אתם מגדירים משתנה גלובלי מסוג Integer, לדוגמה, המהדר ישמור עבורו ארבעה בתים של זכרון גלובלי. בלי קשר לנראות של נתון גלובלי זה (קראו בפרק 11 על יחידות ועל הגבלות הנראות הקשורות אליהן), הוא מקבל הקצאה של זכרון גלובלי.

משתנים גלובליים מוקצים בזכרון נתונים גלובלי ספציפי עם התחלת התוכנית, והגודל והפריסה של זכרון זה נקבעים על ידי המהדר והמקשר (Linker). המשתנים הגלובליים נשארים באזור זה של הזכרון הגלובלי עד לסיום התוכנית, אפילו אם נעשה בהם שימוש רק חלק קטן מאד מהזמן. זו הסיבה לכך שהשימוש בזכרון הגלובלי מוגבל מאד ביישומי שפת פסקל ודלפי, לטובת שני אזורי הזכרון הדינמיים שיישומים יכולים להשתמש בהם – הערמה והמחסנית.

זכרון המחסנית

המונח מחסנית (Stack) מציין נתח זכרון שזמין לתוכנית והוא דינמי, אך מוקצה (והקצאתו מבוטלת) על פי סדר נוקשה. ההקצאה במחסנית היא מסוג "האחרון שנכנס הוא הראשון שיוצא" (LIFO – Last In First Out). פירוש הדבר שהאובייקט האחרון שהוקצה לו מקום במחסנית יהיה הראשון שיימחק. זכרון המחסנית משמש לרוב רוטינות (קריאות לפרוצדורות, לפונקציות ולמתודות) לצורך פרמטרים ומשתנים מקומיים.

כאשר אתם קוראים לרוטינה, הפרמטרים וטיפוס ההחזרה שלה מוצבים במחסנית (אלא אם אתם ממטבים את הקריאה, כפי שדלפי עושה כברירת מחדל). כמו כן, המשתנים שאתם מכריזים ברוטינה (באמצעות הבלוק var שלפני ההצהרה begin) מאוחסנים במחסנית, כך שכאשר הרוטינה מסתיימת הם מוסרים אוטומטית (לפני שהשליטה חוזרת לרוטינה הקוראת, לפי סדר LIFO).

שפת פסקל משתמשת במחסנית עבור פרמטרים של רוטינות וערכי החזרה (אלא אם אתם משתמשים במוסכמת הקריאה של ברירת המחדל – register), עבור משתנים מקומיים של הרוטינות, עבור קריאות לפונקציות ה-API של Windows וכן הלאה.

יישומים יכולים להקצות כמות גדולה של זכרון עבור המחסנית. בדלפי אפשר לקבוע את הגודל המרבי בדף המקשר של אפשרויות הפרויקט, אם כי ברירת המחדל מספיקה ברוב

המקרים. אם תקבלו אי פעם הודעת שגיאה בנוגע למחסנית מלאה, הסיבה היא ככל הנראה פונקציה שקראה לעצמה באופן רקורסיבי ללא מעצור, ולא נפח קטן מדי של המחסנית.

זכרון הערמה

המונח ערמה (Heap) מציין נתח זכרון שזמין לתוכנית, ושמכונה גם בשם 'אזור זכרון דינמי'. הערמה היא האזור בו הקצאה וביטול הקצאה של זכרון מתרחשים בסדר שרירותי. פירוש הדבר שאם אתם מקצים שלושה בלוקים של זכרון ברצף, ההשמדה שלהם מאוחר יותר יכולה להתבצע בכל סדר שהוא. מנהל הערמה (או מנהל הזכרון) מטפל בכל הפרטים עבורכם, כך שאתם פשוט מבקשים מקום בזכרון באמצעות הפקודה GetMem או באמצעות קריאה ל-Constructor של אובייקט (שמביאה ליצירת האובייקט), וקובץ ההרצה יחזיר לכם בלוק חדש בזכרון – לפעמים תוך שימוש חוזר בבלוקים בזכרון שפונו קודם לכן.

שפת פסקל משתמשת בערמה לצורך הקצאת זכרון עבור כל אובייקט ואובייקט, עבור טקסט במחרוזות, עבור תוכן של מערכים דינמיים ועבור בקשות מותאמות ספציפיות לזכרון דינמי. מערכת ההפעלה Windows, לדוגמה, מאפשרת ליישום לקבל עד שני ג'יגהבייטים של מרחב כתובות, והערמה יכולה להשתמש ברובו.

מערכים דינמיים

שפת פסקל הכילה מאז ומעולם מערכים בעלי גודל קבוע. כאשר אתם מכריזים על טיפוס נתונים באמצעות מבנה מערך, עליכם לציין את מספר המרכיבים במערך. מתכנתים מומחים יודעים בוודאי שהיו מספר טכניקות בהן ניתן היה להשתמש כדי ליצור מערכים דינמיים, בדרך כלל תוך שימוש במצביעים והקצאה ושחרור ידניים של הזכרון הדרוש.

בדלפי 4 הוצג לראשונה יישום פשוט מאד של מערכים דינמיים, שנבנו על פי אותו עקרון כמו טיפוס המחרוזת הארוכה שתיארתי קודם. המערכים הדינמיים, כמו מחרוזות ארוכות, מוקצים באופן דינמי ומתבצעת עבורם ספירת הפניות, אך הם אינם משתמשים בטכניקת העתקה-בעת-כתיבה. ביטול הקצאה של מערך דינמי מתבצע באמצעות מתן הערך nil למשתנה שלו, או שינוי אורכו לאפס.

אפשר פשוט להגדיר מערך בלי לציין את מספר המרכיבים שבו, ואז להקצות לו גודל מסוים בעזרת הפרוצדורה SetLength. אותה פרוצדורה יכולה לשמש גם לשינוי גודל של מערך בלי לאבד את תוכנו. קיימות פרוצדורות נוספות בהשראת מחרוזות, כגון Copy, שניתן להפעיל על מערכים. הנה קטע קוד קטן שממחיש את העובדה שחייבים גם להכריז על המערך וגם להקצות לו זכרון לפני שניתן להתחיל להשתמש בו:

```
var
  Array1: array of Integer;

begin
  Array1 [1] := 100; // שגיאה
  SetLength (Array1, 100);
```

```
  Array1 [99] := 100; // תקין
end;
```

מכיוון שאתם מציינים רק מספר המרכיבים, האינדקס מתחיל תמיד ב-0. מערכים גנריים בפסקל מאפשרים גבול תחתון שאינו אפס ואינדקסים שאינם מספרים שלמים; מערכים דינמיים אינם תומכים באפשרויות אלה. כדי לגלות את הסטטוס של מערך דינמי ניתן להשתמש בפונקציות Length, High ו-Low, כמו בכל מערך רגיל. עם זאת, עבור מערכים דינמיים Low מחזירה תמיד 0 ואילו High מחזירה תמיד את האורך פחות 1. מכאן נובע שעבור מערך דינמי ריק, High תחזיר 1- (ערך משונה, כשחושבים עליו, מכיוון שהוא נמוך יותר מזה שמחזירה Low).

לאחר מבוא קצר זה אני יכול להראות לכם דוגמה פשוטה בשם DynArr. היא פשוטה מאד מכיוון שאין שום דבר מורכב במיוחד במערכים דינמיים. אני אשתמש בה גם כדי להראות מספר שגיאות אפשריות שמתכנתים עלולים לבצע. התוכנית מגדירה שני מערכים גלובליים ומאתחלת את הראשון מביניהם עם הפעלתה:

```
var
  Array1, Array2: array of Integer;

begin
  // בצע הקצאה
  SetLength (Array1, 100);
```

פקודה זו מאפסת את כל הערכים. קוד האתחול מאפשר לנו להתחיל לקרוא ולכתוב ערכים במערך מיד, בלי חשש משגיאות זכרון (כמובן, בהנחה שאיננו מנסים לגשת לפריטים מעבר לגבול העליון של המערך). כעת, הנה קוד נוסף שכותב לכל תא במערך ומבצע אתחול טוב עוד יותר:

```
var
  I: Integer;

begin
  for I := Low (Array1) to High (Array1) do
    Array1 [I] := I;
```

באמצעות קריאה חוזרת ל-SetLength אפשר לשנות את גודל המערך מבלי לאבד את תוכנו. תוכלו לבדוק זאת באמצעות קריאת ערך:

```
  // הגדלה תוך שמירה על הערכים הקיימים
  SetLength (Array1, 200);

  // חילוץ
  writeln(IntToStr (Array1 [99]));
```

הקוד המורכב (מעט) היחיד הוא בסוף התוכנית, ומטרתו להעתיק מערך אחד לאחר באמצעות האופרטור :=, ועל ידי כך ליצור "כינוי" (Alias), משתנה חדש שמפנה לאותו המערך בזכרון. בנקודה זו, אם תשנו אחד מהמערכים, גם השני ישתנה מכיוון ששניהם מפנים לאותו אזור בזכרון:

```
  // כינוי
```

```
Array2 := Array1;

// שינוי של אחד (שניהם משתנים)
Array2 [99] := 1000;

// הצגת האחר
writeln (IntToStr (Array1 [99]));
```

בנקודה זו, התוכנית לדוגמה מבצעת שתי פעולות נוספות. הראשונה היא בדיקת שוויון של המערכים. בדיקה זו אינה משווה את המרכיבים בפועל אלא את המיקומים בזכרון שהמערכים מפנים אליהם, כלומר האם שני המשתנים הם כינויים של אותו המערך בזכרון:

```
if Array1 = Array2 then
  Beep;

// קיצון המערך הראשון
Array1 := Copy (Array2, 0, 10);
```

הפעולה השניה היא קריאה לפונקציה Copy, שלא רק מעתיקה נתונים ממערך אחד לאחר אלא גם מחליפה את המערך הראשון במערך חדש שהפונקציה יצרה. התוצאה היא שהמשתנה Array1 מפנה כעת למערך בן 11 מרכיבים, ולכן פניה חוזרת לערך ה-99 תגרום לשגיאת זכרון ולהעלאת חריגה (אלא אם כיביתם את בדיקת הטווחים, ואז השגיאה תישאר אך החריגה לא תוצג).

סיכום

פרק זה הציג את תפקידיהם של האזורים השונים בזכרון וכיסה את נושא המערכים הדינמיים, מרכיב חשוב בניהול זכרון. הפרק הבא יתמקד ביישומי Windows בסיסיים, וזה שאחריו ידבר על הרחבה לליבה של שפת פסקל – טיפוס הנתונים Variant.

פרק 9: תכנות ל-Windows

דלפי מספקת כימוס מלא ל-API הבסיסי של Windows דרך Object Pascal וספריית הרכיבים החזותיים (VCL)[49]. רק לעתים רחוקות עולה צורך לבנות יישומי Windows באמצעות שפת פסקל פשוטה וקריאה ישירה לפונקציות ה-API של Windows. למרות זאת, האופציה עדיין קיימת עבור מתכנתים שמעוניינים להשתמש בטכניקות מיוחדות שה-VCL אינה תומכת בהן, או שאינם רוצים להשתמש בסביבה ויזואלית.

מכיוון שספר זה מתמקד בפסקל, אראה לכם כיצד אתם יכולים לבנות יישום Windows בפסקל – תירוץ טוב ללמוד כמה ממאפייני הליבה של מערכת ההפעלה, ידע שיכול להיות שימושי אפילו בזמן עבודה במסגרת מתקדמת יותר. אתם יכולים לבנות יישומי Linux ו- Mac OS X באופן דומה באמצעות גישה ל-API אחר, כגון GTK 2 תחת Linux או Carbon ב- Mac OS X. העקרונות זהים, אך בפרק זה אתמקד ב-Windows.

באופן דומה, העיקרון של גישה ל-API של Windows מתוך FreePascal או GNU Pascal זהה לזה של גישה מתוך דלפי, אם כי חלק מהפרטים עשויים להשתנות. אתם יכולים להסתכל עליהם בתיעוד של FreePascal ושל GNU Pascal. כך שלמעשה, בפרק זה אתמקד ב-Windows ובדלפי.

ידיות של Windows

מבין טיפוסי הנתונים שמערכת ההפעלה Windows הוסיפה לדלפי, הקבוצה החשובה ביותר היא הידיות (Handles). שם טיפוס נתונים זה הוא THandle, והטיפוס עצמו מוגדר ביחידה Windows כך:

```
type
  THandle = LongWord;
```

היישום של טיפוסי נתונים מסוג ידית הוא מספרי, אך השימוש בהם בפועל אינו כזה. ב-Windows, ידית היא הפניה למבנה נתונים פנימי של המערכת. לדוגמה, כאשר אתם עובדים עם חלון (או "טופס" במונחי דלפי), המערכת נותנת לכם ידית לאותו חלון. המערכת מודיעה לכם שהחלון עמו אתם עובדים הוא חלון מספר 142, למשל. מאותו רגע והלאה, היישום שלכם יכול לבקש מהמערכת לבצע פעולות על חלון מספר 142 – להזיז אותו, לשנות את גודלו, להקטינו לסמל וכן הלאה. למעשה, הידית היא הפרמטר הראשון בפונקציות רבות ב-API של Windows. הדבר נכון לא רק לפונקציות שפועלות על חלונות: גם פונקציות אחרות

[49] מהדר FreePascal מבצע דבר דומה עם ספריית רכיבי לזרוס (LCL), שמהווה במובנים רבים מעין עותק של ה-VCL.

של ה-API של Windows מקבלות כפרמטר ראשון ידית GDI, ידית תפריט, ידית מופע, ידית של מפת ביטים או כל אחד ממגוון טיפוסי ידיות אחרים.

במילים אחרות, ידית היא קוד פנימי (מעין מספר זהות) שבו אתם יכולים להשתמש כדי לפנות לרכיב ספציפי שמנוהל על ידי המערכת, כולל חלון, מפת ביטים, סמל, בלוק בזכרון, סמן, גופן, תפריט וכן הלאה. בדלפי כמעט לעולם אין צורך להשתמש בידיות ישירות, מכיוון שהן מסתתרות בתוך טפסים, מפות ביטים ואובייקטים אחרים של דלפי. הן הופכות לשימושיות כאשר אתם רוצים לקרוא לפונקציית API של Windows שאינה נתמכת ישירות על ידי דלפי.

הכרזות חיצוניות

מרכיב חשוב נוסף בתכנות ל-Windows מיוצג על ידי הכרזות חיצוניות. הכרזות אלה, ששימשו במקור לקישור בין קוד פסקל לפונקציות חיצוניות שנכתבו בשפת אסמבלי, משמשות בתכנות ל-Windows לקריאה לפונקציות שנמצאות בקובצי DLL (Dynamic Link Library). בדלפי קיימות מספר הכרזות שכאלה ביחידה Windows:

```
// הכרזה מקדימה
function GetUserName(lpBuffer: PChar;
  var nSize: DWORD): BOOL; stdcall;

// הכרזה חיצונית (במקום קוד בפועל)
function GetUserName; external advapi32
  name 'GetUserNameA';
```

רק לעתים רחוקות תצטרכו לכתוב הכרזות כגון אלה, מכיוון שהן כבר קיימות ביחידה Windows וביחידות מערכת אחרות של דלפי. הסיבה היחידה שבגללה תצטרכו להקליד את קוד ההכרזה החיצונית היא קריאה לפונקציות שנמצאות ב-DLL משלכם, או קריאה לפונקציות Windows שאינן מתועדות.

ההכרזה שלמעלה פירושה שהקוד עבור הפונקציה GetUserName נמצא בספריה הדינמית advapi32 (advapi32 הוא קבוע שמשויך לשם המלא של ה-DLL, 'advapi32.dll') ונקרא שם GetUserNameA מכיוון שלפונקציה זו יש גם גרסת ASCII וגם גרסת WideString. למעשה, אנו יכולים לציין בתוך ההכרזה החיצונית שהפונקציה שלנו מתייחסת לפונקציה ב-DLL שיש לה שם אחר[50].

[50] בגרסת 16 ביט של דלפי, ההכרזות החיצוניות השתמשו בשם הספריה ללא הסיומת, כאשר אחריו באה הנחית השם (כמו בקוד שבדוגמה) או בהנחית אינדקס חלופית שאחריה המספר הסודר של הפונקציה בתוך ה-DLL. השינוי משקף שינוי מערכתי באופן בו מתבצעת גישה לספריות: אמנם Win32 מאפשרת גישה לפונקציות DLL לפי מספר, אך מיקרוסופט ציינה שאפשרות זו לא תיתמך בעתיד. שימו לב גם שהיחידה Windows החליפה את היחידות WinProcs ו-WinTypes שהיו קיימות בדלפי בגרסת 16 ביט.

פונקציית Callback של Windows

בפרק 6 ראינו ששפת פסקל תומכת בטיפוסים פרוצדורליים. שימוש נפוץ לטיפוסים פרוצדורליים הוא באספקת פונקציות callback לפונקציות API של Windows.

ראשית, מהי פונקציית callback? הרעיון הוא שפונקציות API מסוימות מבצעות פעולה מסוימת על מספר רכיבים פנימיים של המערכת, לדוגמה כל החלונות מסוג מסוים. פונקציה שכזו, שמכונה גם בשם פונקציית מניה (enumeration), מצריכה הגדרה של הפעולה, שמתבצעת על כל אחד מהמרכיבים, כפרמטר שמועבר כפונקציה או כפרוצדורה שתואמים לטיפוס פרוצדורלי נתון. מערכת ההפעלה Windows משתמשת בפונקציות callback גם בנסיבות אחרות, אך אנו נגביל את הדיון למקרה הפשוט הזה.

בואו ונביט בפונקציית API בשם EnumWindows, בעלת האבטיפוס הבא (כפי שהוא מוגדר בשפת פסקל):

```
function EnumWindows (
  lpEnumFunc: TFNWndEnumProc;
  lParam: LPARAM): BOOL; stdcall;
```

על פי קובץ העזרה, הפונקציה שמועברת כפרמטר צריכה להיות מהסוג הבא (שוב, בגרסה הפסקלית):

```
type
  EnumWindowsProc = function (Hwnd: THandle;
    Param: Pointer): Boolean; stdcall;
```

הפרמטר הראשון הוא הידית של כל חלון ראשי בתורו, ואילו השני הוא הערך שהעברנו בעת הקריאה לפונקציה EnumWindows. למעשה, הטיפוס TFNWndEnumProc אינו מוגדר היטב בפסקל: זהו בסך הכל מצביע. פירוש הדבר שעלינו לספק פונקציה עם הפרמטרים המתאימים ולהשתמש בה כמצביע, באמצעות לקיחת הכתובת של הפונקציה במקום ביצוע קריאה לה. לרוע המזל, פירוש הדבר גם שהמהדר לא יסייע לנו כלל במקרה של שגיאה בטיפוס של אחד מהפרמטרים.

מערכת ההפעלה Windows דורשת מאיתנו המתכנתים להשתמש במוסכמת הקריאה stdcall בכל פעם שאנו קוראים לפונקציית API של Windows או מעבירים פונקציית callback למערכת. דלפי משתמשת, כברירת מחדל, במוסכמת קריאה שונה ויעילה יותר שמסומנת במילת המפתח register.

הנה הגדרה הולמת של פונקציה תואמת, אשר קוראת את הכותרת של החלון לתוך מחרוזת ומציגה אותה על גבי המסך:

```
function GetTitle (Hwnd: THandle;
  Param: Pointer): Boolean; stdcall;
var
  Text: string;
```

```
begin
  SetLength (Text, 100);
  GetWindowText (Hwnd, PChar (Text), 100);
  Text := PChar (Text);
  // דלג על חלונות בעלי כותרות ריקות
  if Text <> '' then
    writeln (IntToStr (Hwnd) + ': ' + Text);
  Result := True;
end;
```

התוכנית קוראת לפונקציית ה-EnumWindows API ומעבירה לה כפרמטר את הפונקציה GetTitle:

```
var
  EWProc: TFNWndEnumProc;

begin
  EWProc := @GetTitle;
  EnumWindows (EWProc, 0);
end;
```

יכולתי לקרוא לפונקציה בלי לאחסן קודם לכן את הערך בטיפוס פרוצדורלי זמני, אך רציתי להבהיר מה קורה בדוגמה הזו. התוצאה המתקבלת דומה לפירוט הבא (השמטתי ממנו כתריסר שורות):

```
66762: ClamWin
66602: Interwise Push Client
66700: SkypeÖ - marco.cantu
66696: TrayIconManager
66676: HP ProtectToolsSystemTrayWindowInstance
66390: Windows Sidebar
66382: Apache Service Monitor
262732: TaskEng - Task Scheduler Engine Process
66022: HiddenFaxWindow
66018: BluetoothNotificationAreaIconWindowClass
131128: MMDEVAPI Device Window
196902: Battery Meter
131346: TSVNCacheWindow
133024: EnumTitles - CodeGear RAD Studio for Microsoft
  Windows - EnumTitles.dproj
132950: EssentialPascalv3.odt - OpenOffice.org Writer
262650: Wintech Italia Srl - Inbox (4) - Mozilla Firefox
133642: XanaNews 1.18.1.6
65780: Program Manager
```

זהו חלק מרשימה של כל החלונות הראשיים הקיימים שרצים אצלי במערכת. רובם הם חלונות חבויים שלרוב איננו רואים כלל (וישנם גם רבים נטולי כותרת, שהתוכנה עצמה משמיטה).

תוכנית Windows מינימלית

על מנת להשלים את נושא התכנות ל-Windows בשפת פסקל, ברצוני להראות לכם יישום פשוט מאד אך שלם שנבנה ללא ה-VCL. התוכנית פשוט לוקחת את הפרמטר של שורת הפקודה (שמאוחסן על ידי המערכת במשתנה הגלובלי CmdLine) ומחלצת ממנו מידע

בעזרת פונקציות פסקל ParamCount ו-ParamStr. הראשונה מחזירה את מספר הפרמטרים, והשניה מחזירה את הפרמטר שבמיקום נתון.

בסביבה של ממשק משתמש גרפי, המשתמשים מציינים פרמטרים של שורת פקודה רק לעתים רחוקות מאד, אך פרמטרים אלה חשובים עבור המערכת. לדוגמה, מרגע שהגדרתם קשר בין סיומת קובץ לבין יישום, אתם יכולים להריץ תוכנית פשוט באמצעות בחירה בקובץ המשויך. בפועל, כשאתם לוחצים לחיצה כפולה על סמל הקובץ, מערכת ההפעלה Windows מפעילה את התוכנית המשויכת ומעבירה אליה דרך פרמטר של שורת פקודה את שם הקובץ שנבחר.

הנה קוד המקור המלא של הפרויקט:

```
program Strparam;

uses
  Windows;

begin
  // הצגת המחרוזת במלואה
  MessageBox (0, cmdLine,
    'StrParam Command Line', MB_OK);

  // הצגת הפרמטר הראשון
  if ParamCount > 0 then
    MessageBox (0, PChar (ParamStr (1)),
      '1st StrParam Parameter', MB_OK)
  else
    MessageBox (0, PChar ('No parameters'),
      '1st StrParam Parameter', MB_OK);
end.
```

קוד הפלט משתמש בפונקציית ה-MessageBox API, פשוט כדי להימנע מהוספה של כל ה-VCL לפרויקט. למעשה, לתוכנית Windows "נקיה" כמו זו יש יתרון של חתימת זכרון קטנה מאד: גודלו של קובץ ההרצה של התוכנית כ-18 קילובייט.

כדי לספק לתוכנית זו פרמטר של שורת פקודה, תוכלו להשתמש בפקודת התפריט של דלפי Run > Parameters. טכניקה אחרת היא לפתוח את סייר החלונות (Windows Explorer), לאתר את התיקיה שמכילה את קובץ ההרצה של התוכנית ולגרור את סמל הקובץ שברצונכם להריץ אל סמל קובץ ההרצה. הסייר יפעיל את התוכנית עם השם של הקובץ שנגרר כפרמטר של שורת פקודה.

סיכום

בפרק זה ראינו מבוא לתכנות windows ברמה הבסיסית, דיברנו על ידיות והצגנו תוכנית Windows פשוטה מאד. למשימות תכנות נורמליות ב-Windows תשתמשו לרוב בתמיכה בפיתוח חזותי מבוססת ה-VCL שדלפי מספקת, אלא שתמיכה זו היא מחוץ להיקף הספר, שעוסק בשפת פסקל עצמה.

הפרק הבא עוסק בווריאנטים, תוספת משונה מאד למערכת הטיפוסים של פסקל שהופיעה במקור במטרה לספק תמיכה מלאה ל-OLE של Windows[51].

[51] נושאי OLE (ו-COM) מורכבים הרבה יותר מדי מכדי לכסותם בטקסט זה. הם היוו טכנולוגיות ליבה של Windows במשך מספר שנים ועדיין נעשה בהם שימוש נרחב, אם כי מיקרוסופט נוטשת אותם בהדרגה לטובת ארכיטקטורת דוט נט.

פרק 10: וריאנטים

במטרה לספק תמיכה מלאה ב-OLE של Windows, גרסת 32 ביט של דלפי כוללת את טיפוס הנתונים Variant[52]. אני מתכוון לדון בטיפוס נתונים זה כאן מנקודת מבט כללית. למעשה, לטיפוס Variant יש השפעה נרחבת על השפה כולה, וספריית הרכיבים של דלפי משתמשת בו גם בדרכים שאינן קשורות לתכנות OLE.

לווריאנטים אין טיפוס

ככלל, ניתן להשתמש בווריאנטים כדי לאחסן נתונים מכל טיפוס ולבצע מגוון פעולות והמרות טיפוסים. שימו לב שהדבר עומד בניגוד לגישה הכללית של שפת פסקל, והוא מהווה יישום של טיפוסים דינמיים כגון אלה שקיימים בשפות התכנות Smalltlalk, Objective-C ושפות תסריט נפוצות רבות כגון JavaScript, PHP, פייתון ורובי[53]. הווריאנטים מחושבים ועוברים בדיקת טיפוסים (Type checking) בזמן ריצה. המהדר לא יזהיר אתכם מפני שגיאות אפשריות בקוד, אותן תוכלו לגלות רק באמצעות בדיקות מקיפות. באופן כללי, אפשר לחשוב על קטעי קוד שמשתמשים בווריאנטים כעל קוד מפוענח (Interpreted) מכיוון שכמו בקוד מפוענח, רוב הפעולות אינן ניתנות לפענוח עד זמן הריצה. הדבר משפיע במיוחד על מהירות הקוד.

לאחר שהזהרתי אתכם מפני השימוש בטיפוס Variant, הגיע הזמן לראות מה הוא מסוגל לעשות. בעיקרון, לאחר שהכרזתם על משתנה וריאנטי כמו כאן:

```
var
  V: Variant;
```

אתם יכולים לתת לו ערכים ממספר טיפוסים שונים:

```
V := 10;
V := 'Hello, World';
V := 45.55;
```

לאחר שהוגדר ערך לווריאנט, תוכלו להעתיק אותו לכל טיפוס נתונים תואם או לא תואם. אם תתנו את הערך לטיפוס נתונים לא תואם, דלפי תבצע המרה אם תוכל – אחרת היא תפיק שגיאת זמן ריצה. למעשה, וריאנט מאחסן מידע על הטיפוס ביחד עם הנתונים ומאפשר מגוון פעולות בזמן ריצה. פעולות אלה יכולות להיות שימושיות, אך הן איטיות וגם לא בטוחות.

הביטו בדוגמה הבאה (בשם VariTest), שמהווה הרחבה של הקוד לעיל:

```
var
```

[52] FreePascal מציע טיפוס Variant זהה.

[53] את גישת "הרחבת שפה דינמית" הזו הדגמתי בהרצאה שהעברתי לראשונה לכנס הווירטואלי CodeRage II בנובמבר 2007, אשר עסקה בנושא "שפות תחומיות (Domain Specific) בדלפי".

```
  V: Variant;
  s: string;

begin
  V := 10;
  s := v;
  writeln (s);
  V := 'Hello, World';
  s := v;
  writeln (s);
  V := 45.55;
  s := v;
  writeln (s);
```

מצחיק, לא? פרט להשמה של וריאנט שמחזיק מחרוזת אל המשתנה s, אפשר להשים אל s גם וריאנט שמחזיק מספר שלם או מספר עשרוני. *גרוע* מכך, אתם יכולים להשתמש בווריאנטים כדי לחשב ערכים, כפי שמדגים הקוד הבא:

```
var
  V: Variant;
  N: Integer;
  s: string;

begin
  V := s;
  N := Integer(V) * 2;
  V := N;
  s := V;
```

כתיבה של קוד כזה נושאת בחובה סיכונים, בלשון המעטה. אם המחרוזת מכילה מספר, הכל עובד. אם לא, תועלה חריגה. אפשר לכתוב קוד כזה, אך אם אין סיבה טובה לעשות זאת, לא מומלץ להשתמש בטיפוס Variant. היצמדו לטיפוסי הנתונים המסורתיים ולגישת בדיקת הטיפוסים של פסקל. בדלפי וב-VCL (ספריית הרכיבים החזותיים) הווריאנטים משמשים בעיקר לתמיכה ב-OLE ולגישה לשדות של מסדי נתונים.

הסתכלות מעמיקה על וריאנטים

בדלפי וב-FreePascal קיים טיפוס רשומה וריאנטי בשם TVarData, שמיוצג בזכרון המחשב כמו הטיפוס Variant. אתם יכולים להשתמש בו כדי לגשת לטיפוס בפועל של הווריאנט. רשומה מסוג TVarData כוללת את טיפוס הווריאנט (כפי שמציין המשתנה VType), מספר שדות שמורים ואת הערך בפועל.

הערכים האפשריים של השדה VType תואמים לטיפוסי הנתונים בהם ניתן להשתמש באוטומציית OLE, אשר מכונים לרוב בשם טיפוסי OLE או טיפוסים וריאנטיים. הנה רשימה אלפביתית מלאה של הטיפוסים הווריאנטיים הזמינים:

```
varArray        varBoolean      varByRef
varCurrency     varDate         varDispatch
varDouble       varEmpty        varError
varInteger      varNull         varOleStr
varSingle       varSmallint     varString
varTypeMask     varUnknown      varVariant
```

אפשר למצוא תיאורים של טיפוסים אלה תחת הנושא Values in variants שבמערכת העזרה של דלפי. קיימות גם פונקציות רבות לביצוע פעולות על וריאנטים, בהן ניתן להשתמש כדי לבצע המרות טיפוסים ספציפיות או לקבל מידע על טיפוס הוריאנט (ראו, לדוגמה, את הפונקציה VarType). למעשה, רוב פונקצית ההשמה והמרות הטיפוסים הללו מופעלות אוטומטית כאשר אתם כותבים ביטויים שמשתמשים בוריאנטים. רוטינות אחרות לתמיכה בוריאנטים (חפשו את הנושא Variant support routines בקובץ העזרה) פועלות למעשה על מערכי וריאנטים.

וריאנטים הם איטיים!

קוד שמשתמש בטיפוס Variant הוא איטי, לא רק בעת המרת טיפוסי נתונים אלא גם כאשר מחברים שני ערכי וריאנטים שמאחסנים מספרים שלמים. הוא איטי כמעט כמו קוד מפוענח של שפת Visual Basic! להשוואה בין המהירויות של אלגורתים מבוסס-וריאנטים לקוד זהה שמבוסס על טיפוס Integer, הסתכלו בדוגמה VSpeed.

תוכנית זו מריצה לולאה, מודדת את מהירותה ומציגה את התוצאה על גבי המסך. הנה הראשונה מתוך שתי הלולאות הדומות שמבוססות על וריאנטים ועל מספרים מטיפוס Integer:

```
var
  time1, time2: TDateTime;
  n1, n2: Variant;

begin
  time1 := Now;
  n1 := 0;
  n2 := 0;

  while n1 < 5000000 do
  begin
    n2 := n2 + n1;
    Inc (n1);
  end;

  // אנו חייבים להשתמש בתוצאה כדי שהמהדר לא יבחר להתעלם מהחישוב
  writeln (n2);
  time2 := Now;
  writeln ('Variants: ' + FormatDateTime (
     'ss.hhh', Time2-Time1) + ' seconds');
```

כדאי לעיין בקוד התזמון, מכיוון שזו טכניקה שתוכלו להתאים בקלות לכל סוג של מבחן ביצועים. כפי שאתם יכולים לראות, התוכנית משתמשת בפונקציה Now כדי לקבל את השעה הנוכחית ובפונקציה FormatDateTime כדי להציג את הפרש הזמנים – באמצעות מחרוזת העיצוב – רק ברמת השניות ("ss") ואלפיות השניה ("hhh"). לחלופין, אפשר להשתמש בפונקציה GetTickCount מה-API של Windows, אשר מחזירה בדיוק רב את מספר אלפיות השניה שעברו מאז אתחול מערכת ההפעלה.

בדוגמה זו הפרש המהירויות הוא כה גדול, שתוכלו להבחין בו אף ללא תזמון מדויק:

```
Variants: 00.922 seconds
Integers: 00.005 seconds
```

הערכים בפועל ישתנו בהתאם למחשב עליו תריצו את התוכנית, אך יחס ההבדלים לא ישתנה במידה רבה.

סיכום

וריאנטים הם כה שונים מטיפוסי נתונים מסורתיים בפסקל, שהחלטתי לדבר עליהם בפרק קצר ונפרד זה. התפקיד העיקרי שלהם הוא אמנם בתכנות OLE, אך אפשר להשתמש בהם לכתיבת תוכניות מאולתרות, מבלי שיהיה צורך לחשוב על טיפוסי נתונים. כפי שראינו, יש לכך מחיר כבד בביצועים.

כעת, לאחר שכיסינו את רוב המאפיינים של השפה, הרשו לי לדבר על המבנה הכללי של התוכנית ועל המודולריזציה שאפשר לבצע בעזרת יחידות.

פרק 11: תוכניות ויחידות

יישומי פסקל עושים שימוש נרחב ביחידות (Units), שהן מודולים (Modules) של תוכניות. למעשה, היחידות היוו את הבסיס למודולריות בשפת פסקל לפני שנוספו לה המחלקות. ביישומי דלפי, לכל טופס יש יחידה תואמת. כאשר אתם מוסיפים טופס חדש לפרויקט (בעזרת הלחצן המתאים בסרגל הכלים או בפקודת התפריט File -> New Form), דלפי מוסיפה למעשה יחידה חדשה שמגדירה את המחלקה עבור הטופס החדש.

יחידות

כל טופס מוגדר אמנם ביחידה, אך ההיפך אינו נכון. יחידות אינן חייבות להגדיר טפסים: הן יכולות פשוט להגדיר אוסף של רוטינות ולהפכן לזמינות לתוכניות שונות. כדי להוסיף יחידה ריקה לפרויקט הנוכחי, בחרו בפקודת התפריט File -> New ואז בסמל Unit בדף New של ה-Object Repository. יחידה ריקה זו מכילה את הקוד הבא, שמציין את החלקים מהם מורכבת היחידה:

```
unit Unit1;

interface

implementation

end.
```

הרעיון של יחידה הוא פשוט: ליחידה יש שם ייחודי שתואם את שם הקובץ שמכיל אותה, סעיף ממשק (Interface) שמכריז מה יהיה גלוי לעיני יחידות אחרות, וסעיף מימוש (Implementation) שבו נמצאים הקוד בפועל וכן הכרזות סמויות אחרות. לסיום, היחידה יכולה לכלול סעיף אתחול (Initialization) אופציונלי עם קוד אתחול כלשהו, שיורץ כאשר התוכנית נטענת לזכרון, וגם סעיף סגירה (Finalization) אופציונלי שיורץ בעת סיום הרצת התוכנית.

המבנה הכללי של יחידה עם כל הסעיפים האפשריים נראה כך:

```
unit unitName;

interface

// יחידות אחרות אליהן אנו מתייחסים בסעיף הממשק
uses
  A, B, C;

// הגדרות טיפוסים מיוצאות
```

```
type
  newType = TypeDefinition;

// קבועים מיוצאים
const
  Zero = 0;

// משתנים גלובליים
var
  Total: Integer;

// רשימת פונקציות ופרוצדורות מיוצאות
procedure MyProc;

implementation

// יחידות אחרות אליהן אנו מתייחסים בסעיף המימוש
uses
  D, E;

// משתנה גלובלי סמוי
var
  PartialTotal: Integer;

// יש לכתוב את הקוד של כל הפונקציות המיוצאות
procedure MyProc;
begin
  // ... הקוד של הפרוצדורה
end;

initialization
  // קוד אתחול אופציונלי

finalization
  // קוד סגירה/"ניקוי" אופציונלי

end.
```

הקטע uses בתחילת סעיף הממשק מציין את היחידות האחרות להן אנו זקוקים בחלק הממשק של היחידה. אלה כוללות יחידות שמגדירות את טיפוסי הנתונים בהם אנו משתמשים לצורך הגדרת טיפוסי נתונים אחרים – לדוגמה, הרכיבים שנעשה בהם שימוש במסגרת טופס שאנו יוצרים.

קטע ה-uses השני, בתחילת סעיף היישום, מציין יחידות נוספות שאנו זקוקים להן רק בקוד היישום. כאשר אתם צריכים הפניה ליחידות אחרות עבור הקוד של הרוטינות והמתודות, עליכם להוסיף אלמנטים לקטע uses זה ולא לקודם. כל היחידות בהן אתם משתמשים חייבות להימצא בתיקיית הפרויקט או בתיקיה שנמצאת בנתיב החיפוש (אותו אפשר להגדיר עבור כל פרויקט בדף Directories / Conditionals שבתיבת הדו-שיח Options של הפרויקט).

מתכנתי C++ צריכים לשים לב לכך שההצהרה uses אינה זהה להנחיה include. הצהרת uses מייבאת רק את חלק הממשק המהודר-מראש של היחידות שפורטו. חלק היישום של היחידה מעובד רק בעת הידור היחידה עצמה. היחידות אליהן אתם מפנים יכולות להיות בפורמט המקור (PAS) או בפורמט מהודר (בדלפי הוא מכונה DCU), אך ההידור חייב היה להתבצע במהדר פסקל מאותה גרסה.

הממשק של היחידה יכול להגדיר מספר רכיבים שונים, ביניהם פרוצדורות, פונקציות, משתנים גלובליים וטיפוסי נתונים. ביישומי דלפי, סביר להניח שטיפוסי נתונים הם השימוש הנפוץ ביותר. בכל פעם שאתם יוצרים טופס, דלפי מציבה טיפוס נתונים חדש מסוג מחלקה ביחידה. עם זאת, כאמור, יחידות בדלפי לא משמשות רק לאחסון של הגדרות טפסים. אתם יכולים להמשיך להשתמש ביחידות מסורתיות עם פרוצדורות ופונקציות, וביחידות עם מחלקות שאינן קשורות לטפסים או לרכיבים חזותיים אחרים.

יחידות והיקף

בשפת פסקל, יחידות הן המפתח לכימוס ולנראות, והן חשובות ככל הנראה אף יותר ממילות המפתח private ו-public במחלקות[54]. ההיקף (Scope) של מזהה (כגון משתנה, פרוצדורה, פונקציה או טיפוס נתונים) הוא חלק הקוד בו המזהה נגיש. הכלל הבסיסי הוא שלמזהה יש משמעות רק בתחום ההיקף שלו – כלומר, רק בתוך הבלוק בו הוא מוכרז. לא ניתן להשתמש במזהה מחוץ להיקפו. הנה מספר דוגמאות:

- **משתנים גלובליים סמויים**: אם אתם מכריזים על מזהה בחלק היישום (implementation) של יחידה, לא תוכלו להשתמש בו מחוץ ליחידה, אך כן תוכלו להשתמש בו בכל בלוק ופרוצדורה שמוגדרים בתוכה. הקצאת הזכרון עבור משתנה זה נעשית ברגע שהתוכנית מתחילה ונשמרת עד סופה. אפשר להשתמש בחלק האתחול (initialization) של היחידה כדי לספק למשתנה ערך התחלתי ספציפי.

- **משתנים מקומיים**: אם אתם מכריזים על משתנה בתוך בלוק שמגדיר רוטינה או מתודה, לא תוכלו להשתמש במשתנה זה מחוץ לאותה רוטינה. ההיקף של משתנה זה כולל את כל הרוטינה, כולל רוטינות פנימיות (אלא אם משתנה בעל שם זהה ברוטינה פנימית מסתיר את ההגדרה החיצונית). הזכרון עבור משתנה זה מוקצה במחסנית כאשר התוכנית מריצה את הרוטינה שמגדירה אותו. עם סיום הרוטינה, הזכרון במחסנית משוחרר אוטומטית.

- **משתנים גלובליים**: אם אתם מכריזים על מזהה בחלק הממשק (Interface) של היחידה, ההיקף שלו כולל כל יחידה אחרת שמשתמשת בזו שהכריזה עליו. משתנה זה משתמש בזכרון כמו משתנים מהקבוצה הראשונה והוא בעל אותו אורך חיים; ההבדל היחיד הוא בנראות שלו.

כל הכרזה בחלק הממשק של יחידה זמינה מכל חלק של התוכנית שכולל את אותה יחידה בסעיף uses שלו. משתנים של מחלקות טפסים מוכרזים באותו אופן, כך שניתן להפנות לטופס (ולשדותיו, למתודות, לתכונות [Properties] ולמרכיבים [Components] שלו) מקוד של כל טופס אחר. כמובן, הכרזה על כל המשתנים כגלובליים היא פרקטיקת תכנות גרועה. פרט לבעיות צריכת הזכרון המובנות מאליהן, השימוש במשתנים גלובליים הופך את התוכנית לקשה יותר לתחזוק ולעדכון. בקיצור, עליכם להשתמש במספר הקטן ביותר האפשרי של משתנים גלובליים.

[54] למעשה, אפילו ההשפעה של מילת המפתח private על מחלקה אינה נאכפת בתוך היקף היחידה שמכילה את המחלקה.

יחידות כמרחבי שמות

ההצהרה uses היא הטכניקה הסטנדרטית לגישה להיקף של יחידה אחרת. בנקודה זו תוכלו לגשת להגדרות של היחידה, אך יכול להיווצר מצב בו שתי יחידות אליהן אתם מפנים מכריזות על אותו מזהה; כלומר, ייתכן שיהיו לכם שתי מחלקות או שתי רוטינות בעלות אותו שם.

במקרה כזה תוכלו פשוט להשתמש בשם היחידה כתחילית לשם הטיפוס או הרוטינה שמוגדרים ביחידה. לדוגמה, תוכלו להפנות לפרוצדורה ComputeTotal שביחידה Totals מסוימת בשם Totals.ComputeTotal. הדבר לא אמור להתרחש פעמים רבות מדי, מכיוון שמומלץ בחום לא לבחור באותו שם לשני דברים שונים בתוכנית.

עם זאת, אם תסתכלו ב-VCL ובקובצי Windows תגלו שלחלק מהפונקציות של דלפי יש שם זהה לפונקציות API של Windows, שזמינות בעצמן מתוך דלפי (אם כי רשימת הפרמטרים שונה ברוב המקרים). הפרוצדורה הפשוטה Beep היא דוגמה לכך. אם תיצרו תוכנית דלפי חדשה, תוסיפו לחצן ותכתבו את הקוד הבא:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Beep;
end;
```

אז ברגע שתלחצו על הלחצן תשמעו צליל קצר. כעת, עברו להצהרת ה-uses של היחידה ושנו את הקוד מזה:

```
uses
  Windows, Messages, SysUtils, Classes, ...
```

לגרסה דומה מאד זו (שפשוט מעבירה את היחידה *SysUtils* לפני היחידה *Windows*):

```
uses
  SysUtils, Windows, Messages, Classes, ...
```

אם תנסו להדר כעת מחדש את הקוד, תקבלו שגיאת מהדר "Not enough actual parameters." ("אין די פרמטרים בפועל"). הבעיה היא שהיחידה Windows מגדירה פונקציית Beep אחרת בעלת שני פרמטרים. באופן כללי, מה שקורה בהגדרות של היחידה הראשונה שאתם כוללים בהצהרה uses עלול להיות מוסתר על ידי ההגדרות המקבילות ביחידות הבאות. הפתרון הבטוח הוא פשוט למדי:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SysUtils.Beep;
end;
```

קוד זה יהודר בלי קשר לסדר היחידות בתוך הצהרת ה-uses. בדלפי קיימות רק התנגשויות נוספות מעטות של שמות, פשוט מכיוון שקוד דלפי נמצא בדרך כלל בתוך מתודות של מחלקות, ומתודות בעלות שם זהה שנמצאות בתוך מחלקות שונות אינן יוצרות כל בעיה.

יחידות ותוכניות

יישום דלפי מורכב משני סוגים של קובצי קוד מקור: יחידה אחת או יותר, וקובץ תוכנית יחיד. אפשר לחשוב על היחידות כעל קובצי משנה, שהחלק העיקר של היישום – התוכנית – מפנה אליהם. בתיאוריה זה נכון. בפועל, קובץ התוכנית הוא לרוב קובץ שמופק אוטומטית ובעל תפקיד מוגבל. כל שעליו לעשות הוא להתחיל את התוכנית ולהפעיל את הטופס הראשי. את הקוד של קובץ התוכנית, או קובץ הפרויקט של דלפי (DPR), אפשר לערוך ידנית או באמצעות מנהל הפרויקטים (Project Manager) ומספר אפשרויות פרויקט (Project Options) הקשורות לאובייקט היישום ולטפסים.

המבנה של קובץ התוכנית פשוט, לרוב, הרבה יותר מאשר המבנה של היחידות. הנה קוד מקור של קובץ תוכנית לדוגמה:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.PAS' {Form1};

begin
  Application.Initialize;
  Application.CreateForm (TForm1, Form1);
  Application.Run;
end.
```

כפי שאתם יכולים לראות, נמצאים כאן רק סעיף uses והקוד הראשי של היישום, בין מילות המפתח begin ו-end. הצהרת ה-uses של התוכנית חשובה במיוחד, מכיוון שהיא משמשת לניהול ההידור והקישור (linking) של היישום.

סיכום

יחידות היו הטכניקה של פסקל (של טורבו פסקל, למעשה, מכיוון שוירת' הוסיף את התפיסה הזו לשפת Modula-2[55]) לתכנות מודולרי. בעקבותיהן הגיעו אובייקטים ומחלקות, אך למרות זאת הן עדיין משחקות תפקיד מרכזי בכימוס, בהגדרה של מרחב שמות כלשהו ובמבנה הכללי של תוכניות דלפי. כמו כן, יחידות משפיעות על היקפים ועל הקצאות זכרון גלובליות.

[55] למידע נוסף, ראו http://www.modula2.org/modula-2.php וכן http://en.wikipedia.org/wiki/Modula-2

פרק 12: קבצים בשפת פסקל

אחד מהמאפיינים הייחודיים של שפת פסקל לעומת שפות תכנות אחרות הוא התמיכה המובנית שלה בקבצים. השפה כוללת את מילת המפתח file[56], שמציינת טיפוס בדומה ל-array או ל-record. אתם יכולים להשתמש ב-file כדי ליצור טיפוס חדש, ולהכריז בעזרתו על משתנים חדשים:

```
type
  IntFile = file of Integer;

var
  IntFile1: IntFile;
```

ניתן גם להשתמש במילת המפתח file בלי לציין טיפוס נתונים כדי להצהיר על קובץ נטול טיפוס. לחלופין, אפשר להשתמש בטיפוס TextFile שמוגדר ביחידה System לצורך הכרזה על קבצים של תווי ASCII. לכל סוג קובץ רוטינות מוגדרות מראש משלו, כפי שנראה בהמשך הפרק.

רוטינות לעבודה עם קבצים

מרגע שהכרזתם על משתנה קובץ, תוכלו להקצות לו קובץ ממשי במערכת הקבצים באמצעות המתודה AssignFile. הצעד הבא יהיה, בדרך כלל, לקרוא ל-Reset כדי לפתוח את הקובץ לקריאה מתחילתו, ל-Rewrite כדי לפתוח אותו (או ליצור אותו) לצורך כתיבה, או ל-Append כדי להוסיף פריטים חדשים בסופו מבלי למחוק את הקודמים. בסיום פעולות הקלט או הפלט, עליכם לקרוא ל-CloseFile.

לדוגמה, עיינו בקוד הבא (ההדגמה IntegersToFile), שפשוט שומר כמה מספרים לקובץ:

```
type
  IntFile = file of Integer;

var
  IntFile1: IntFile;
  n: Integer;

begin
  AssignFile (IntFile1, 'test.my');
  Rewrite (IntFile1);
  n := 1;
```

[56] שימו לב שמרכיב השפה file of אינו עובד בדלפי לדוט נט, מכיוון שהוא מוגבל לגודל הפיזי של טיפוסי הנתונים שהוא מנהל.

```
  Write (IntFile1, n);
  n := 2;
  Write (IntFile1, n);
  CloseFile (IntFile1);
end;
```

הפעולה CloseFile צריכה להתבצע, לרוב, בתוך בלוק finally כדי להימנע מהשארת קובץ פתוח במקרה שהקוד המטפל בו יעלה חריגה. למעשה, פעולות מבוססות-קבצים מפיקות או לא מפיקות חריגות בהתאם להגדרת המהדר I$. אם המערכת אינה מעלה חריגות, בדקו את הפונקציה הסטנדרטית IOResult כדי לראות אם דבר-מה השתבש:

```
  res := IOResult;
  if res = 0 then
    writeln ('File test.my created correctly')
  else
  begin
    write ('File test.my creation failed with error: ');
    writeln (res);
  end;
```

ישנם שני כללים שצריך לקחת בחשבון בקטע הקוד שלמעלה. הראשון הוא שלא צריך לקרוא ל-IOResult עבור כל פעולת פלט: לאחר כשלון, המערכת תתעלם מכל קריאות הפלט הבאות. הכלל השני הוא שהקריאה ל-IOResult מאפסת את ערכה, וזו הסיבה לכך שקראנו לה פעם אחת ושמרנו את התוצאה במשתנה אחר לצורך הדיווח (קריאה חוזרת ל-IOResult בקוד המדווח תיתן את התוצאה 0, שפירושה "אין שגיאות").

דלפי כוללת רוטינות נוספות רבות לניהול קבצים, חלקן ברשימה הבאה:

Append	FileClose	Flush
AssignFile	FileCreate	GetDir
BlockRead	FileDateToDateTime	IOResult
BlockWrite	FileExists	MkDir
ChangeFileExt	FileGetAttr	Read
CloseFile	FileGetDate	Readln
DateTimeToFileDate	FileOpen	Rename
DeleteFile	FilePos	RenameFile
DiskFree	FileRead	Reset
DiskSize	FileSearch	Rewrite
Eof	FileSeek	RmDir
Eoln	FileSetAttr	Seek
Erase	FileSetDate	SeekEof
ExpandFileName	FileSize	SeekEoln
ExtractFileExt	FileWrite	SetTextBuf
ExtractFileName	FindClose	Truncate
ExtractFilePath	FindFirst	Write

```
FileAge          FindNext          Writeln
```

לא כל הרוטינות הללו מוגדרות בשפת פסקל התקנית, אך רבות מהן היוו חלק מטורבו פסקל מימיה הראשונים. מידע מפורט על הרוטינות הללו תוכלו למצוא בקובצי העזרה של דלפי. כאן אציג בפניכם שתי דוגמאות פשוטות שמבוססות על קובצי טקסט ומדגימות כיצד אפשר להשתמש בכמה מהרוטינות הללו. הדוגמה השניה תכלול עיבוד של שורת פקודה, והיא תהיה אולי הדוגמה השלמה ביותר בספר כולו.

טיפול בקובצי טקסט

השימוש בפורמט הקובץ הטקסטואלי נפוץ ביותר. כפי שציינתי קודם, שפת פסקל מספקת תמיכה ספציפית מסוימת לקובצי טקסט, במיוחד טיפוס הנתונים TextFile שמוגדר ביחידה System. בדוגמה StringsToFile אני יוצר קובץ (יש להעביר את שם הקובץ כפרמטר) עם מעט תוכן טקסטואלי:

```
var
  OutputFile: TextFile;
  I: Integer;
  Filename: string;

begin
  filename := ParamStr (1);
  if filename = '' then
  begin
    writeln ('Missing file name');
  end
  else
  begin
    // output the text to a file
    AssignFile (OutputFile, FileName);
    Rewrite (OutputFile);

    for I := 1 to 10 do
      writeln (OutputFile, 'item ' + IntToStr (I));

    CloseFile (OutputFile);
    writeln ('done');
  end;

  readln;
end.
```

קובצי טקסט של פסקל אינם חייבים להיות מחוברים לקובץ פיזי: אפשר לקשור אותם ישירות למדפסת כך שהפלט יודפס במקום להישמר בקובץ. כדי לבצע זאת, השתמשו בפרוצדורה AssignPrn. לדוגמה, בקוד שלמעלה אפשר להחליף את השורה:

```
AssignFile (OutputFile, FileName);
```

בשורה:

```
AssignPrn (OutputFile);
```

ממיר קובץ טקסט

עד כה ראינו דוגמאות פשוטות של יצירת קבצים חדשים. בדוגמה הבאה נעבד קובץ קיים וניצור אחד חדש עם גרסה שונה של תוכנו. תוכנה זו, שנקראת Filter (מסנן), יכולה להמיר את כל התווים בקובץ טקסט לאותיות רישיות, להמיר לרישיות רק את האות הראשונה בכל משפט, או להתעלם מהחלק העליון של אוסף תווי ASCII (בעלי הערך 128 ומעלה).

התוכנה מקבלת כפרמטרים שני שמות קבצים (עבור קובץ הקלט וקובץ הפלט), וכן אחד מהדגלים הבאים:

-U (אותיות רישיות – Uppercase)
-C (התחלת משפט – Capitalize)
-R (סילוק סמלים – Remove symbols)

בשלב ראשון, התוכנית מנתחת את הפרמטרים של שורת הפקודה ומחפשת את הדגלים ושמות הקבצים:

```
for I := 1 to ParamCount do
begin
  if ParamStr(i) [1] = '-' then
    Flag := ParamStr(i) [2]
  else
    if inputFile = '' then
      inputFile := ParamStr(i)
    else
      outputFile := ParamStr(i);
end;
```

מכיוון שהפרמטרים הם חובה, התוכנית מריצה את המבחן הבא על פרמטרי הקלט לפני שהיא מבצעת הפעולה הדרושה:

```
if (inputFile = '') or (outputFile = '') or
  not (Flag in ['U', 'R', 'C']) then
begin
  writeln ('Missing or wrong parameters');
  readln;
  Exit;
end;
```

הקוד האמיתי של הדוגמה נמצא בשלוש רוטינות ההמרה, שנקראות בהתאם לפרמטרים. הרוטינות הללו נמצאות ביחידה משנית בשם FilterRoutines.pas. הקריאות מתבצעות מתוך הצהרת case, שמהווה חלק מהפרוצדורה DoConvert:

```
case Flag of
  'U': ConvUpper;
  'C': ConvCapitalize;
  'R': ConvSymbols;
end;
```

שוב, אתם יכולים לראות את קוד המקור כולו בתוכניות לדוגמה הזמינות להורדה עבור הספר. הפרוצדורה DoConvert מבצעת את רוב העבודה הקשורה לטיפול בקבצים: בפעם הראשונה היא פותחת את קובץ הקלט כקובץ בתים (קובץ שמאחסן נתונים כבתים – bytes – פשוטים) כדי שתוכל להשתמש בפרוצדורה FileSize שאינה זמינה עבור קובצי טקסט. לאחר מכן קובץ זה נסגר ונפתח מחדש כקובץ טקסט.

הרוטינה מנהלת את קובצי הקלט והפלט, ולאחר מכן קוראת לאחת משלוש רוטינות העיבוד. בואו ונסתכל כעת מקרוב על אחת מהרוטינות הללו. הפשוטה ביותר מבין השלוש היא ConvUpper, שממירה כל תו בקובץ הטקסט לאות רישית. הנה הקוד שלה:

```
procedure ConvUpper;
var
  Ch: Char;

begin
  while not Eof (FileIn) do
  begin
    Read (FileIn, Ch);
    Ch := UpCase (Ch);
    Write (FileOut, Ch);
  end;
end;
```

היא קוראת כל תו מקובץ המקור עד שהתוכנית מגיעה לסוף הקובץ (Eof). כל תו בודד מומר ומועתק לקובץ הפלט. לחלופין, אפשר לקרוא ולהמיר שורה אחת בכל פעם (כלומר, מחרוזת אחת כל פעם) בעזרת רוטינות לטיפול במחרוזות. הדבר יהפוך את התוכנית למהירה יותר במידה משמעותית. הגישה שנקטתי בה כאן היא סבירה רק במסגרת דוגמה ראשונית.

פרוצדורת ההמרה לסילוק של סמלים היא פשוטה מאד:

```
while not Eof (FileIn) do
begin
  Read (FileIn, Ch);
  if Ch < Chr (127) then
    Write (FileOut, Ch);
  ...
```

לעומתה, הפרוצדורה שממירה לרישיות רק את האות שבתחילת המשפט היא קוד מורכב באמת, אותו תוכלו למצוא בקובץ הדוגמה. ההמרה שם מתבססת על הצהרת case בעלת ארבעה ענפים:

- אם האות היא רישית, והיא האות הראשונה לאחר סימן פיסוק שמסיים משפט (כפי שמציין המשתנה הבוליאני Period), משאירים אותה כמות שהיא; אחרת, ממירים אותה לאות רגילה. המרה זו אינה מבוצעת על ידי פרוצדורה סטנדרטית פשוט מכיוון שאין פרוצדורה כזו עבור תווים בודדים: היא מבוצעת בעזרת פונקציה בסיסית שכתבתי ושמה LowCase.

- אם האות היא רגילה, היא מומרת לרישית רק אם היא נמצאת בתחילת משפט חדש.

- ❖ אם התו הוא סימן פיסוק מסיים (נקודה, סימן שאלה או סימן קריאה), Period מקבל את הערך True.

- ❖ אם התו הוא כל דבר אחר, הוא פשוט מועתק לקובץ היעד ו-Period מקבל את הערך False.

הפלט הבא מציג דוגמה לשינויים שמבוצעים על ידי התוכנית:

```
// inputtext.txt
this is a red brown fox. the fox is
under a tree. GOOD for the fox.

// outputtext.txt
This is a red brown fox. The fox is
under a tree. Good for the fox.
```

תוכנית זו רחוקה מלהיות מתאימה לשימוש מקצועי, אך היא מהווה צעד ראשון לבניית תוכנית המרת רישיות מושלמת. החסרונות העיקריים שלה הם המרה של שמות עצם פרטיים (שמתחילים, באנגלית, באות רישית) לאותיות רגילות, והפיכה לרישית של כל אות שנמצאת אחרי נקודה – אפילו אם זו סתם האות הראשונה בסיומת של שם קובץ.

סיכום

טיפול ישיר בקבצים בגישה המסורתית של שפת פסקל עדיין מהווה טכניקה מעניינת, אך אני ממליץ בחום להשתמש בזרימות (streams – המחלקה TStream והמחלקות הנגזרות ממנה) לטיפול בכל קובץ מורכב שהוא ב-Object Pascal. זרימות מייצגות קבצים וירטואליים, אותם ניתן למפות לקבצים פיזיים, לבלוקים בזכרון, לשקע (Socket) או לכל סדרה רציפה אחרת של בתים. מידע נוסף על זרימות תוכלו למצוא בקובץ העזרה של דלפי ובסדרת ספרי Mastering Delphi שלי.

ניהול קבצים הוא נושא רחב מאד, ואפשר לכתוב ספר שלם עליו אפילו אם מתמקדים רק בטכניקות המסורתיות של פסקל; אך לי נגמר המקום. למעשה, זהו הפרק האחרון של הספר.

אחרית דבר

לעת עתה, הפרק על הקבצים חותם את הספר. הרגישו חופשיים לשלוח אליי משוב כפי שהצעתי בהקדמה (באמצעות קבוצת הדיון או הדואר האלקטרוני), עם הערות ובקשות. אם בעקבות היכרות זו עם שפת פסקל אתם מעוניינים להעמיק בתכנות מונחה עצמים ב- Object Pascal בדלפי, תוכלו לקרוא על כך בספר מודפס מהסדרה Mastering Delphi שלי (כגון Mastering Delphi 7 או Mastering Delphi 2005), שראתה אור בהוצאת Sybex, ובספר Delphi 2007 Handbook שזמין באתר Lulu.com.

רכישת הגרסה המודפסת של ספר זה או של כל אחד מספריי האחרים (ישירות ב-Lulu או דרך הקישורים ל-Amazon שבאתר הבית שלי) היא הדרך הטובה ביותר לתמוך בכתיבה שלי ולעודד אותי לכתוב עוד על פסקל, דלפי ונושאים אחרים בעתיד.

דרך טובה אחרת לתרום היא לסייע לי בעדכונים, בתיקונים ובהצעות.

למידע נוסף על המהדורה העדכנית ביותר של Mastering Delphi, על הספר הנלווה Essential Delphi ועל ספרים אחרים שלי (וגם של מחברים אחרים) למתקדמים, עיינו באתר הבית שלי: http://www.marcocantu.com.

תכנות נעים בפסקל!

נספח: דוגמאות

להלן רשימת הדוגמאות שמהוות חלק מהספר "יסודות הפסקל" ושזמינות להורדה כקובץ zip יחיד (EPasCodev3.zip, שגודלו כ-30KB) בכתובת:

```
http://www.marcocantu.com/epascal
```

רשימת הדוגמאות לפי פרקים[57]:

- פרק 2: EssHello, EPExpressions
- פרק 3: EPConstants, EPRange, TimeNow, Variables
- פרק 4: Pointers
- פרק 5: CaseTest, ForTest, IfTest, LoopsTest
- פרק 6: OpenArr, OverDef
- פרק 7: StrRef, FmtTest, StringAndPChar
- פרק 8: NewMessageTest
- פרק 9: EnumTitles, StrParam
- פרק 10: VSpeed
- פרק 12: Filter, IntegersToFile, StringsToFile

[57] לדוגמאות נוספות וקוד מקור נוסף בשפת פסקל, אני מציע להסתכל בארכיון SWAG שנמצא, נכון לעכשיו, בכתובת http://www.bsdg.org/SWAG/index.html

אינדקס

"שלום, עולם!" (תוכנית) 13
@ (אופרטור) 21, 46
^ (אופרטור) 46
:= (אופרטור) 51
Programs = Algorithms + Data Structures 50
And (אופרטור) 21
ANSIChar 29
ANSIString 74
Append 101
Array 43
As (אופרטור) 21
Assigned 48
AssignFile 101
AssignPrn 103
Beep 99
Boolean 27, 28
Break 57
Byte 27
Callback, פונקציות 88
Camel-casing (מוסכמה לשיום מזהים) 15
Cardinal 27
Case 53
Cdecl 66
Char 27
Chr 29, 37
Chrome 4, 12
CloseFile 101
CmdLine 89
Comp (טיפוס נתונים) 32
Const (פרמטרים) 62
Continue 57
Currency (טיפוס נתונים) 32
Date (פונקציה) 34
DateTimeToStr 34
DayOfWeek 34
Dec 31
DecodeDate 34
Delphi 4, 11
Delphi 2007 Handbook (ספר) 107
Dispose 47
Div 21
DLL (Dynamic Link Library) 87

Double (טיפוס נתונים) 32
Downto 54
DPR 100
DWORD 36
Else 52
EncodeDate 34
End 59
EnumWindows 88
Eof 105
Exclude 22
Exit 57
Extended (טיפוס נתונים) 32
Fastcall 66
File 101
FileSize 105
Finalization 96
FloatToDecimal 37
FloatToStr 37
For (לולאה) 54
For-in 55
Format (פונקציה) 64, 79
שומרי מקום 80
FormatDateTime 34
Free Pascal 12
FreeMem 47
GetMem 47, 48, 83
GetTickCount 94
GetUserName 78, 87
GNU Pascal 32, 33, 86
Halt 57
High 31, 44
If (תנאי) 52
Implementation 96
In (אופרטור) 21
Inc 31
Include 22
Initialization 96
Int 37
Int64 28
Integer 27, 30
Interface (של יחידות) 96
IntToHex 37
IntToStr 37
IOResult 102
Is (אופרטור) 22
Kylix 11
LIFO 82
LongInt 27
LongWord 27, 28

Low 31, 44
Lulu.com 107
Lvalue 51
Mastering Delphi 4, 107
Math (יחידה) 33
MessageBox 71, 90
Mod (אופרטור) 21
New 47, 48
Nil 47
Not (אופרטור) 21
Now 34
Odd 31
Or (אופרטור) 21
Ord 31, 37, 42
Overload 70, 73
ParamCount 90
ParamStr 90
Pascal-casing (מוסכמה לשיום מזהים) 15
PChar 77
Pred 31
Random 56
Readln 20
Real (טיפוס נתונים) 32
Real48 32
Register 66
Repeat 55
Reset 101
Resourcestring 26
Result 61, 63
Rewrite 101
Round 37
RTTI 30
Rvalue 51
SetLength 76, 83
Shl (אופרטור) 21
ShortInt 27
ShortString 74
Shr (אופרטור) 21
Single (טיפוס נתונים) 32
SizeOf 30
Slice 63
SmallInt 27
Stdcall 66
Str 37
StrPas 37
StrPCopy 37
StrToFloat 37
StrToInt 37

Succ..... 31
SWAG (ארכיון קטעי קוד)..... 108
SysUtils..... 35
TDateTime..... 33
TextFile..... 101, 103
TextToFloat..... 37
THandle..... 86
Then..... 52
Time..... 34
Trunc..... 37
TVarData..... 93
TVarRec..... 64
UINT..... 36
Uses..... 97
Val..... 37
Var..... 24, 61, 62
Variant (טיפוס נתונים)..... 92
VCL..... 86
While..... 55
WideChar..... 29
WideString..... 74
With..... 57
Word..... 27
Writeln..... 20
www.marcocantu.com..... 107
Xor (אופרטור)..... 21
אופרטורים..... 21
קדימות..... 21
של קבוצות..... 22
אותיות רישיות..... 15
בדיקת טווחים..... 41
ביטויים..... 20
בית (Byte)..... 27
בית אורך (Length byte)..... 75
בלוג (מרקו קנטו)..... 6
ברירת מחדל, פרמטרים של..... 71
דלפי..... 4, 11
ה-API של Windows..... 86
הבלטת תחביר..... 17
הטלה..... 36
היקף (Scope)..... 98
הכרזות חיצוניות..... 87
הכרזות מקדימות..... 67
המרה..... 25
הנחיות מהדר..... 15
העמסה (Overloading)..... 70
הערות..... 14
הצהרות..... 18, 50
השמה..... 51

וירת', ניקלאוס 10
וריאנט (טיפוס רשומה) 45
זכויות יוצרים (לספר זה) 5
זכרון 82
זליגה (leak) 47
זכרון גלובלי 82
טורבו פסקל 10
טיפוס פרוצדורלי 68
טיפוסי נתונים 27
טיפוסי נתונים מוגדרים על ידי המשתמש 39
טיפוסים לא-משוימים 39
טיפוסים מונים (Enumerated) 41
טיפוסים ממשיים 32
טיפוסים סודרים 27
טיפוסים ספציפיים ל-Windows 36
ידיות (handles) 86
יחידות (Units) 96
כללי תאימות טיפוסים 40
מדפסת 103
מוזיאון (לתוכנות Borland) 11
מוסכמות קריאה 66
מחסנית 82
מחרוזות 74
ארוכות 75
בסיום null 77
של פסקל 75
מילות מפתח 18
מימוש הפניה (Dereferencing) 47
מיקרוסופט 10, 11
מערך 43
מערכים דינמיים 83
מערכים מבוססי-אפס 44
מערכים פתוחים 63
בעלי טיפוס משתנה, כפרמטרים 64
כפרמטרים 80
מצביעים 46
מרחבי שמות 99
משוב (על הספר) 5
משתנה בוליאני 27, 28
משתנים 24
מתודה 66
נקודה-פסיק (;) 16, 50
נקודתיים-שווה (אופרטור) 51
סימנים לבנים 15
ספירת הפניות 75
עיגול הבנקאים 38
עיצוב נאה (Pretty-printing) 13
ערכים ליטרליים 19
ערמה (heap) 83

פונקציות.....60
מצביעים ל.....68
פסקל.....10
פרוצדורה.....60
פרמטרים
const.....62
out.....62
העברה.....61
הפניה.....61
ערך.....61
קבועים.....25
בעלי טיפוס, ניתנים להשמה.....26
קבוצה (set).....42
קבצים.....101
טיפוסים.....48
קוד מקור (לספר זה).....5
רוטינה.....60
רשומה (Record).....45
שורת הפקודה, פרמטרים של.....89
תאריך, פונקציה להחזרת ה.....34
תודות.....6
תווים.....29
קבועים.....29
תוכניות לדוגמה
CaseTest.....53
DoubleHello.....67
DynArr.....84
EPConstants.....25
EPExpressions.....20
EPRange.....29
Filter.....104
FmtTest.....80
ForTest.....55
IfTest.....52
IntegersToFile.....101
LoopsTest.....55
OpenArr.....64, 65
OverDef.....70, 71
Pointers.....47
ProcType.....69
StringAndPChar.....78
StringsToFile.....103
StrRef.....76
TimeNow.....35
Variables.....24
VariTest.....92
VSpeed.....94
תוכנית.....100
תחביר.....13

תחילית T 39
תכנות מונחה עצמים 11
תנאי, הצהרות 51
תת-טווח 40