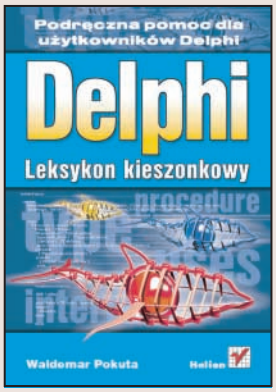
Delphi. Leksykon kieszonkowy



Autor: Waldemar Pokuta
ISBN: 83-7361-510-5
Format: B6, stron: 176

W dobie pracy pod presją czasu coraz popularniejsze stają się wizualne środowiska programistyczne, dzięki którym autorzy aplikacji mogą szybciej i efektywniej tworzyć nowe produkty. Wśród takich środowisk zasłużonym uznaniem cieszy się Delphi. Oparte na Pascalu środowisko umożliwia szybkie tworzenie różnego rodzaju aplikacji dla systemu Windows. Zaimplementowana w Delphi wersja Pascala, nosząca nazwę Object Pascal, została znacznie rozbudowana w porównaniu z oryginałem – posiada wiele możliwości, których nie znajdziemy w wersji standardowej. Z tego właśnie względu krótka i zwięzła „ściąga" zawierająca opis tego języka może okazać się niezwykle przydatna programistom.

Książka „Delphi. Leksykon kieszonkowy" jest taką właśnie ściągawką. Zawiera krótkie omówienia wszystkich klas, funkcji i procedur oferowanych przez Object Pascala oraz zwięzłe wprowadzenie do środowiska Delphi. Nie stanowi podręcznika programowania, ale raczej pomoc dla tych programistów, którzy poznali już podstawy Delphi i pogłębiają swoją wiedzę.

- Struktury danych
- Konwersja i formatowanie danych
- Operacje na ciągach tekstowych
- Funkcje matematyczne
- Sterowanie przebiegiem programu
- Operacje wejścia-wyjścia
- Elementy interfejsu użytkownika
- Obsługa multimediów

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 3. Konwersja i formatowanie

Przedstawione w rozdziale procedury i funkcje zawarte są w modułach: db, dbcommon, sysutils, classes i math.

Rzutowanie typów

Czasami potrzebne jest traktowanie wyrażeń tak, jakby należały do innego typu. Można otrzymać ten efekt przez rzutowanie typów. Np. `Integer('B')` pozwala traktować znak `'B'` tak, jakby był liczbą całkowitą. Składnia rzutowania jest następująca:

```
identyfikatorTypu(wyrazenie)
```

Jeżeli `wyrazenie` jest zmienną, wynik nazywany jest zmienną rzutowaną, w innym wypadku jest on wartością rzutowaną.

Wartość rzutowana

W rzutowaniu wartości zarówno `identyfikatorTypu` jak i `wyrazenie` muszą należeć do typu porządkowego, albo każde z nich musi należeć do typu wskaźnikowego. Przykłady rzutowania wartości:

```
Integer('B')
Char(33)
Boolean(0)
TColor(2)
Longint(@Bufor)
```

Wynik rzutowania otrzymywany jest poprzez obcięcie lub rozszerzenie wartości wyrażenia, przy czym zachowywany jest znak wyrażenia (+ lub −).

Instrukcja:

```
I := Integer('B');
```

przypisuje wartość `Integer('B')` — tzn. *66* — do zmiennej `I`.

Po rzutowaniu wartości nie można dopisywać kwalifikatora (wyrażenia po kropce). Rzutowanie wartości nie może występować po lewej stronie przypisania.

Zmienna rzutowana

Można rzutować dowolną zmienną na dowolny typ pod warunkiem, że ich rozmiary (zmiennej i typu) są równe i nie mieszamy typów całkowitych z rzeczywistymi (służą do tego specjalne funkcje omówione dalej). Przykłady rzutowania zmiennej:

```
Char(I)

Boolean(Liczba)
TMojZdefiniowanyTyp(MojaZmienna)
```

Rzutowanie zmiennej może występować po obydwu stronach przypisania, np.:

```
var c: char;
...
ShortInt(c) := 122;
```

przypisuje znak o kodzie *ASCII 122* (‘z’) do zmiennej `c`.

Można też przekształcić zmienną na typ proceduralny. Na przykład mając następujące deklaracje:

```
type Funkc = function(X: Integer): Integer;
var
  F: Funkc;
  P: Pointer;
  L: Integer;
```

można dokonać następujących przypisań

```
F := Funkc(P);     // Przypisuje wartość do zmiennej
                   // proceduralnej F
Funkc(P) := F;     // Wartość zmiennej proceduralnej
                   // przypisuje do P
L := F(L);         // Wywołuje funkcję używając
                   // zmiennej F
L := Funkc(P)(L); // Wywołuje funkcję używając
                   // zmiennej P
```

Po rzutowaniu zmiennej może wystąpić kwalifikator (np. pole rekordu):

```
type
  TBajtRek = record
    Lo, Hi: Byte;
  end;
var
  B: Byte;
  W: Word;
begin
  W := $4321;
  B := TBajtRek(W).Lo;
  TBajtRek(W).Hi := 0;
end;
```

Rzutowanie klas

Kontrolowanego rzutowania obiektów można dokonać operatorem `as`. Wyrażenie:

```
obiekt as klasa
```

zwraca wskaźnik na *`obiekt`*, który jednak jest traktowany jako obiekt typu podanego przez *`klasa`*. W czasie działania programu *`obiekt`* musi rzeczywiście być typu podanego przez *`klasa`* (lub jego potomkiem) albo mieć wartość `nil`. W przeciwnym przypadku wywoływane jest przerwanie. Typowe zastosowanie operatora `as` pokazuje przykład:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Sender as TButton do
    Caption := 'Naciśnięto';
end;
```

Aby wykorzystać operator `as` bezpośrednio w instrukcji, należy wpisać:

```
(Sender as TButton).Caption := 'Naciśnięto';
```

Konwersje typów liczbowych

Typ `Comp` można konwertować na `Double` i `Currency` (lub z powrotem) za pomocą:

```
function CompToDouble(acomp: Comp): Double;
procedure  DoubleToComp(adouble: Double; var result:
Comp);
function  CompToCurrency(acomp: Comp): Currency;
procedure  CurrencyToComp(acurrency: Currency; var
result: Comp);
```

Aby przekształcić format *BCD* (każda cyfra zapisana na czterech bitach) na `Currency` (walutowy) i z powrotem, wykorzystuje się funkcje:

```
function BCDToCurr(const BCD: TBcd; var Curr:
Currency): Boolean;
function CurrToBCD(Curr: Currency; var BCD: FMTBcd;
Precision: Integer=32; Decimals: Integer=4):
Boolean;
```

Przykładowe wykorzystanie funkcji `BCDToCurr`:

```
procedure TForm1.FormClick(Sender: TObject);
const
  bcd: TBcd = (Precision: 8; SignSpecialPlaces:
  128+4);
// 8 cyfr, 128-liczba ujemna, 4 miejsca po przecinku
var
  c: Currency;
```

```
begin
  bcd.Fraction[0] := 1*16+2; // liczba 12
  bcd.Fraction[1] := 3*16+4; // liczba 34
  bcd.Fraction[2] := 5*16+6; // liczba 56
  bcd.Fraction[3] := 7*16+8; // liczba 78
  BCDToCurr(bcd, c);
  caption := FloatToStr(c);  // wynik = -1234,5678
end;
```

Aby przekształcić typ `Extended` lub `Currency` na dziesiętny, można wykorzystać funkcję:

```
procedure FloatToDecimal(var DecVal: TFloatRec;
const Value; ValueType: TFloatValue; Precision,
Decimals: Integer);
```

Wynik umieszczany jest w zmiennej `DecVal` i składa się z tablicy cyfr, znaku liczby i pozycji przecinka. `ValueType` określa, czy przekształcamy typ `Extended`, czy `Currency`. `Precision` i `Decimals` kontrolują zaokrąglanie wyniku.

Typy złożone

Istnieje również kilka funkcji konwertujących proste typy na złożone (np. rekordowe).

Rekord (w niektórych językach nazywany strukturą) to zbiór elementów różnego typu. Każdy element nazywany jest polem.

Typ `TPoint` definiuje pozycję na ekranie. Jest to rekord zawierający dwa pola — współrzędne `X` i `Y`.

Definicja tego typu wygląda następująco:

```
type TPoint = record
  X: Longint;
  Y: Longint;
end;
```

Aby skonwertować dwie liczby całkowite na typ `TPoint`, należy użyć funkcji:

```
function Point(AX, AY: Integer): TPoint;
```

Funkcja ta służy do szybkiego wypełniania pól rekordu bez konieczności przypisywania wartości każdemu polu z osobna. Można również dzięki niej pominąć deklarację typu `TPoint` przy wywoływaniu funkcji, które jako parametr przyjmują zmienne tego typu:

```
Canvas.Polygon([Point(20, 20), Point(40, 20),
Point(140, 40), Point(250, 130)]);
```

Analogicznie rekord `TRect` zawiera współrzędne prostokąta na ekranie. Kolejne pola to współrzędne lewego górnego i prawego dolnego narożnika. Aby wypełnić ten rekord danymi, można użyć funkcji:

```
function Rect(ALeft, ATop, ARight, ABottom:
Integer): TRect;
function Bounds(ALeft, ATop, AWidth, AHeight:
Integer): TRect;
```

Druga z funkcji zamiast współrzędnych prawego dolnego rogu oczekuje szerokości i wysokości prostokąta.

Konwersja miar kątowych

W Delphi większość funkcji wykorzystujących miary kątowe jako jednostki przyjmuje radiany. Jeżeli chcielibyśmy np. wyświetlić obliczony kąt w stopniach lub innych jednostkach, to istnieje kilka funkcji konwertujących te wielkości:

```
function RadToCycle(Radians: Extended): Extended;
function CycleToRad(Cycles: Extended): Extended;
function RadToDeg(Radians: Extended): Extended;
function DegToRad(Degrees: Extended): Extended;
function RadToGrad(Radians: Extended): Extended;
function GradToRad(Grads: Extended): Extended;
```

Formatowanie napisów

Liczby

Aby przekształcić wartość typu rzeczywistego na napis, możemy użyć jednej z funkcji:

```
function FloatToStr(Value: Extended): string;
function CurrToStr(Value: Currency): string;
```

Jeżeli chcemy mieć większy wpływ na format utworzonego napisu, można też wykorzystać:

```
function FloatToStrF(Value: Extended; Format:
TFloatFormat; Precision, Digits: Integer): string;
function CurrToStrF(Value: Currency; Format:
TFloatFormat; Digits: Integer): string;
function FormatFloat(const Format: string; Value:
Extended): string;
function FormatCurr(const Format: string; Value:
Currency): string;
```

Aby sformatować napis do tablicy znaków, można zastosować funkcje:

```
function FloatToText(Buffer: PChar; const Value;
ValueType: TFloatValue; Format: TFloatFormat;
Precision, Digits: Integer): Integer;
function FloatToTextFmt(Buffer: PChar; const Value;
ValueType: TFloatValue; Format: PChar): Integer;
```

Sformatowane napisy nie są kończone zerem. Funkcja jako wynik zwraca liczbę znaków napisu.

Liczbę całkowitą możemy przekształcić na napis reprezentujący liczbę w systemie dziesiętnym lub szesnastkowym:

```
function IntToStr(Value: Integer): string;
function IntToHex(Value: Integer; Digits: Integer):
string;
```

Argumentem procedury `Str` może być liczba typu całkowitego lub rzeczywistego:

```
procedure Str(X [: Width [: Decimals ]]; var S);
```

Dodatkowo można sformatować wynik określając liczbę wszystkich cyfr i liczbę cyfr po przecinku. Przykład:

```
Str(Pi:5:6, napis);
```

Złożone formatowanie

Kilka poniższych funkcji używa jako argumentu napisu z wzorcem formatowania. Wzorzec ten może zawierać w sobie zwykły tekst i specyfikatory formatowania. Mają one następującą formę:

```
%[index:][-][width][.prec]type
```

Specyfikator rozpoczyna się od znaku `%`. Na końcu musi być określenie typu. Reszta jest opcjonalna i zależna od typu.

Specyfikator *`width`* określa minimalną szerokość napisu wynikowego. Aby to osiągnąć, do napisu mogą być dodane spacje. Domyślnie spacje są dodawane z lewej strony. Jeżeli jednak w formacie podano znak minusa (wyrównanie do lewej strony), spacje będą dodawane z prawej strony.

Specyfikator *`prec`* ma różne znaczenia dla różnych typów formatowania.

Specyfikator *`index`* zmienia porządek odwołań do tablicy wartości. Na przykład funkcja:

```
Format('%d %d %0:d %1:d', [10, 20]);
// formatuje napis: '10 20 10 20'.
```

Możliwe typy zostały wymienione w tabeli 3.1.

Tabela 3.1. Oznaczenia typów w złożonym formatowaniu napisów

Typ	Opis	Znaczenie specyfikatora prec
`d`	Dziesiętny. Argument musi być wartością całkowitą. Wartość jest konwertowana na napis składający się z cyfr dziesiętnych.	Jeżeli podano *prec*, to określa on minimalną liczbę cyfr w napisie (napis będzie lewostronnie dopełniany zerami).
`u`	Dziesiętny bez znaku. Analogiczny jak dziesiętny, ale bez znaku.	Podobnie jak wyżej.
`e`	Naukowy. Argument musi być wartością zmiennoprzecinkową. Wartość jest konwertowana do napisu postaci: `"-d.ddd...E+ddd"`.	Jeżeli podano *prec*, to określa on liczbę wszystkich cyfr w napisie wynikowym (domyślnie 15).
`f`	Klasyczny. Argument musi być wartością zmiennoprzecinkową. Wartość jest konwertowana do napisu postaci: `"-ddd.ddd..."`.	Jeżeli podano *prec*, to określa on liczbę cyfr po przecinku w napisie wynikowym (domyślnie 2).
`g`	Główny. Argument musi być wartością zmiennoprzecinkową. Wartość jest konwertowana do możliwie najkrótszego napisu w formacie naukowym lub klasycznym.	Jeżeli podano *prec*, to określa on liczbę znaczących cyfr w napisie wynikowym (domyślnie 15).
`n`	Numer. Argument musi być wartością zmiennoprzecinkową. Wartość jest konwertowana do napisu postaci: `"-d,ddd,ddd.ddd..."`. Formatowanie podobne do klasycznego z wyjątkiem obecności separatorów tysięcznych.	Jeżeli podano *prec*, to określa on liczbę cyfr po przecinku w napisie wynikowym (domyślnie 2).
`m`	Waluta. Argument musi być wartością zmiennoprzecinkową. Wartość jest konwertowana do napisu w postaci walutowej. Konwersja jest przeprowadzana z uwzględnieniem zmiennych globalnych: `CurrencyString`, `CurrencyFormat`, `NegCurrFormat`, `ThousandSeparator`, `DecimalSeparator` i `CurrencyDecimals`.	Jeżeli podano *prec*, to określa on liczbę cyfr po przecinku w napisie wynikowym (domyślnie 2).

Tabela 3.1. Oznaczenia typów w złożonym formatowaniu napisów — ciąg dalszy

Typ	Opis	Znaczenie specyfikatora prec
p	Wskaźnik. Argument musi być wartością wskaźnika (adres w pamięci). Wartość jest konwertowana na 8-znakowy napis składający się z cyfr szesnastkowych.	
s	Napis. Argument musi być znakiem, napisem lub wartością typu `PChar`. Znak lub napis są wstawiane w miejsce specyfikatora.	Jeżeli podano *prec*, to określa on maksymalną długość w napisie wynikowym.
x	Szesnastkowy. Argument musi być wartością typu całkowitego. Wartość jest konwertowana na napis składający się z cyfr szesnastkowych.	Jeżeli podano *prec*, to określa on minimalną liczbę cyfr w napisie (napis będzie lewostronnie dopełniany zerami).

Aby wykorzystać powyższe opcje formatowania, można użyć jednej z funkcji:

```
function Format(const Format: string; const Args:
array of const): string;
function FormatBuf(var Buffer; BufLen: Cardinal;
const Format; FmtLen: Cardinal; const Args: array of
const): Cardinal;
procedure FmtStr(var StrResult: string; const
Format: string; const Args: array of const);
```

Przykładowe wykorzystanie funkcji `Format`:

```
Caption := Format('Minęło już %d dni od daty twoich
urodzin', [Round(Now-
DateTimePicker_urodziny.Date)]);
```

Do formatowania napisów kończonych zerem (*null-terminated*) można wykorzystać funkcje:

```
function StrFmt(Buffer, Format: PChar; const Args:
array of const): PChar;
function StrLFmt(Buffer: PChar; MaxLen: Cardinal;
Format: PChar; const Args: array of const): PChar;
```

Odczytywanie napisów

Konwertowanie napisów na liczbę nie zawsze kończy się powodzeniem. W przypadku wystąpienia błędu generowany jest wyjątek.

Aby odczytać z napisu liczbę całkowitą (w postaci dziesiętnej lub szesnastkowej), można posłużyć się funkcjami:

```
function StrToInt(const S: string): Integer;
function StrToInt64(const S: string): Int64;
```

Aby uniknąć generowania wyjątku w przypadku błędu, można wykorzystać funkcje:

```
function StrToIntDef(const S: string; Default:
Integer): Integer;
function StrToInt64Def(const S: string; Default:
Int64): Int64;
```

Jeżeli nastąpi błąd, zwrócona będzie wartość domyślna.

W przypadku liczb zmiennoprzecinkowych można użyć funkcji:

```
function StrToFloat(const S: string): Extended;
function StrToCurr(const S: string): Currency;
```

Dla napisów kończonych zerem użyjemy funkcji:

```
function TextToFloat(Buffer: PChar; var Value;
ValueType: TFloatValue): Boolean;
```

Procedura:

```
procedure Val(S; var V; var Code: Integer);
```

może odczytywać zarówno wartości całkowite jak i zmiennoprzecinkowe. Jeżeli `Code` przybierze wartość różną od zera, to nastąpił błąd na pozycji określonej przez `Code`.

> ***Patrz także***
>
> Konwersje dotyczące daty i czasu znaleźć można w rozdziale 8. „Obsługa systemu".

Standard Unicode

Typ `WideString` reprezentuje napisy w formacie *Unicode* (każdy znak zapisany na 16 bitach). Odwołując się do funkcji *COM API*, często jesteśmy zmuszeni zmienić typ z `AnsiString` (`String`) na `WideString`:

```
function StringToWideChar(const Source: string;
Dest: PWideChar; DestSize: Integer): PWideChar;
```

Odwrotny efekt można uzyskać wywołując:

```
function WideCharToString(Source: PWideChar): string;
procedure WideCharToStrVar(Source: PWideChar; var
Dest: string);
function WideCharLenToString(Source: PWideChar;
SourceLen: Integer): string;
procedure WideCharLenToStrVar(Source: PWideChar;
SourceLen: Integer; var Dest: string);
```