Maciej **Gonet**

ZROZUMIEĆ EXCELA

VBA – MAKRA I FUNKCJE

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/zrexvb
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-5694-8

Printed in Poland.

Spis treści

Rozdział 5.

Funkcje definiowane przez użytkownika

W zastosowaniach naukowych najczęściej wykorzystuje się Visual Basic do definiowania funkcji, które mają uzupełnić dostępny standardowo zestaw o funkcje realizujące specyficzne potrzeby obliczeniowe. Wstępne informacje o funkcjach podano w rozdziale 1., w podrozdziale „Makra i funkcje", dalej w punkcie „Definicje podprogramów", a prosty przykład funkcji zamieszczono w tym samym rozdziale, w podrozdziale „Prosta funkcja zdefiniowana w VBA". Teraz deklarowanie i definiowanie funkcji zostanie omówione bardziej szczegółowo.

Sposoby przekazywania argumentów funkcji

Jeśli argument funkcji (lub procedury **Sub**) jest zmienną, to może być przekazany do funkcji wywołującej dwoma sposobami: przez **odwołanie** (używa się również określeń referencja lub wskazanie) lub przez **wartość**. Pierwszy sposób jest w VBA domyślny, gdyż jest szybszy i bardziej efektywny, jeśli chodzi o gospodarkę pamięcią, ale nie zawsze może być użyty. Jeśli chcemy przekazać argument przez wartość, należy poprzedzić jego nazwę w deklaracji funkcji słowem kluczowym **ByVal**. Jeśli natomiast chcemy zaakcentować chęć posłużenia się odwołaniem, możemy analogicznie użyć słowa **ByRef**, jednak nie jest to konieczne, gdyż jest to ustawienie domyślne. W przypadku odwołania, do funkcji wywołującej przekazywany jest faktycznie adres zmiennej i w związku z tym wszystkie zmiany dokonane na tej zmiennej będą widoczne również po wyjściu z funkcji (ogólniej: procedury). W przeciwieństwie do tego, gdy argument jest przekazywany przez wartość, tworzona jest kopia argumentu o tej samej wartości; procedura operuje na tej kopii, a oryginalna zmienna zachowuje swoją niezmienioną wartość.

Przeanalizujmy to na prostym przykładzie.

```
Sub Sumuj()
   Dim a
   a = 5
   Debug.Print Pomnóż(3, a) + a     'z ByVal 20, bez ByVal 30
   Debug.Print Pomnóż(5, a) + a     'z ByVal 30, bez ByVal 150
End Sub
```

```
Function Pomnóż(x, ByVal y)
   y = x * y
   Pomnóż = y
End Function
```

Ze względu na utrudnioną analizę kodu należy unikać sytuacji, gdy wywołanie funkcji powoduje „skutki uboczne", jak w powyższym przykładzie zmianę wartości zmiennej ***a*** w razie pominięcia słowa **ByVal**.

Przy przekazywaniu argumentów przez odwołanie należy upewnić się, że typy powiązanych zmiennych są identyczne, w przeciwnym razie wystąpi błąd kompilacji *ByRef Argument Type Mismatch*. Jeżeli argumenty są przekazywane przez wartość, typ zmiennej zostanie automatycznie dopasowany, jeśli to będzie możliwe.

Należy zwrócić uwagę na specyfikę przekazywania argumentów, gdy są nimi odwołania do obiektów, najczęściej zakresów. Jeśli parametr procedury jest zadeklarowany jako **Range** lub **Object**, to nie ma znaczenia, czy użyjemy wcześniej **ByVal** lub **ByRef**. Nie można utworzyć lokalnej kopii obiektu w procedurze i w związku z tym operacje odnoszące się do obiektu są od razu widoczne w tym obiekcie. Natomiast jeśli parametr procedury jest zadeklarowany jako **Variant** lub inny typ standardowy, a jako argument przekażemy odwołanie do zakresu (lub innego obiektu), to sytuacja jest bardziej złożona. Jeśli użyjemy słowa **ByVal**, to jest tworzona lokalna zmienna o wartości pobranej z zakresu, a sam zakres nie jest modyfikowany. Jeśli jawnie lub przez domniemanie użyjemy słowa **ByRef**, a argument zostanie przekazany w formie — na przykład — **Range**("C5") lub [C5], to wartość zakresu również tym razem zostanie tylko skopiowana do zmiennej lokalnej. Jedynie w przypadku, gdy parametr procedury jest zadeklarowany jako **Variant**, a argument zostanie przekazany w formie zmiennej obiektowej (przez nazwę), to wówczas faktycznie ta zmienna obiektowa będzie modyfikowana (czyli zachowanie będzie takie, jak przy użyciu deklaracji **As Range** albo **As Object**).

Poniższy listing ilustruje ten najbardziej nietypowy przypadek.

```
Private Sub ChangeValue(ByRef a)   ' As Variant
   ' Jeśli zmienna a jest zadeklarowana jako wariant, to na ogół przekazywana jest wartość
   ' a obiekt tylko wtedy, gdy argumentem będzie nazwana zmienna obiektowa
   Debug.Print a
   a = 35
   Debug.Print a
   Debug.Print [A6]    ' wartość komórki zmieniona, jeśli argumentem jest obiekt
End Sub

Sub TestChV()
   Dim b As Range
   Set b = [A6]
   ChangeValue b        ' b ->  argumentem jest obiekt, [A6],  Range("A6") -> argumentem jest wartość
End Sub
```

Następna komplikacja dotyczy tablic. Całe tablice (deklarowane po słowach **Dim**, **ReDim** lub **Static**) muszą być przekazywane przez referencję, natomiast pojedyncze elementy tablicy można przekazywać przez wartość. Jeżeli chcemy przekazać taką tablicę do innej funkcji przez wartość (aby uniknąć modyfikacji oryginalnej tablicy), to mamy do wyboru dwa sposoby: albo w wywołaniu

nazwę tablicy umieścimy w nawiasach, albo utworzymy kopię oryginalnej tablicy w zmiennej typu **Variant** i tę kopię przekażemy jako argument do dalszego przetwarzania. Sytuację ilustruje poniższy listing:

```
Sub TestDim()
   Dim a(1 To 3) As Long
   a(1) = 1: a(2) = 4: a(3) = 9
   ' Wariant 1
   Modyfikuj a          ' tablica a zostanie zmodyfikowana
   ' Wariant 2
   ' Modyfikuj (a)      ' tablica a nie zostanie zmodyfikowana
   ' Wariant 3
   ' Dim kopia
   ' kopia = a
   ' Modyfikuj kopia     ' tablica a nie zostanie zmodyfikowana
   Debug.Print a(1); a(2); a(3)
End Sub

Private Sub Modyfikuj(t)
   Dim u, v, i
   For i = LBound(t) To UBound(t)
      v = t(i)
      t(i) = v + u
      u = v
      Debug.Print t(i);
   Next i
   Debug.Print
End Sub
```

Słowa **ByVal** lub **ByRef** mogą być używane wraz ze słowem **Optional**, natomiast nie mogą być używane do tablic **ParamArray** (patrz podrozdział „Użycie tablicy parametrów").

Opcjonalne parametry funkcji

W deklaracji funkcji można określić zarówno typy parametrów, jak i typ wyniku. Jak w przypadku deklaracji zmiennych, służy do tego słowo kluczowe **As** lub znak deklaratora. Brak deklaracji typu oznacza typ **Variant**.

Niektóre — lub nawet wszystkie — parametry funkcji można określić jako opcjonalne (nieobowiązkowe). Muszą one wystąpić jako ostatnie na liście parametrów. Nazwę takiego parametru w deklaracji poprzedzić trzeba słowem **Optional**. Jeżeli jest więcej niż jeden taki parametr, słowo to powinno się znaleźć przed każdym z nich. W deklaracji funkcji można podać jego wartość domyślną, która będzie użyta w razie braku argumentu w wywołaniu funkcji. Typ parametru opcjonalnego może być określony jawnie, może też pozostać nieokreślony (wtedy jest to typ **Variant**). Może to wyglądać na przykład tak:

```
Optional wsp As Double = 1
```

lub

```
Optional wsp# = 1
```

Jednak deklaracja

```
Optional wsp = 1#
```

ma inne znaczenie. W tym drugim przypadku parametr ***wsp*** jest typu **Variant**, a tylko jest inicjowany stałą typu **Double**. W wywołaniu lub treści funkcji można mu nadawać wartości innych typów.

Jeśli wartość domyślna parametru opcjonalnego nie została określona, a w wywołaniu wartość odpowiedniego argumentu nie została podana, to jeśli parametr był typu **Variant**, przyjmuje on specjalną wartość `Missing`, która nie może być użyta do obliczeń (ale może być zidentyfikowana — patrz niżej), a jeśli typ parametru był określony, przyjmuje on wartość, jaką domyślnie przyjmują zmienne określonego typu.

Parametr opcjonalny musi reprezentować pojedynczą wartość, na przykład nie może być jawną tablicą, ale może być zmienną typu **Variant** zawierającą tablicę.

Inaczej niż w przypadku funkcji Excela, pominięcie argumentu obowiązkowego w wywołaniu funkcji VBA zawsze prowadzi do błędu, natomiast w przypadku parametrów opcjonalnych nie ma znaczenia, czy pominięto tylko wartość argumentu, a pozostawiono separator, czy też pominięto argument wraz z separatorem. W obu przypadkach przyjmuje się wartość domyślną argumentu, a gdy ta nie została określona (ale typ parametru był określony) — wartość zerową lub pustą.

W kodzie funkcji można sprawdzać obecność lub brak argumentu opcjonalnego typu **Variant** za pomocą funkcji **IsMissing**. Funkcja ta zwraca **True**, jeśli argument pominięto i nie została określona wartość domyślna. Termin **IsMissing** należy rozumieć dosłownie — jeśli argument ma wartość **Null** lub **Empty**, to **IsMissing** zwraca **False**. W każdym innym przypadku funkcja również zwraca **False**. Także wtedy, gdy parametr miał określony typ, a argument pominięto (bo wtedy jest nadawana standardowa wartość domyślna — zerowa lub pusta). Parametru odpowiadającego pominiętemu argumentowi (dla którego funkcja **IsMissing** zwróciłaby **True**) nie można używać w obliczeniach, w razie użycia wyrażenie zwraca błąd. Można jednak przekazać wartość `Missing` do innej procedury lub nadać ją innej zmiennej pod warunkiem, że obie zmienne są typu **Variant**. Jest to specyficzna wartość, którą można odczytać tylko za pomocą funkcji **IsMissing**. Samo słowo `Missing` **nie jest** słowem kluczowym, nie ma więc możliwości nadania tej wartości przez przypisanie stałej, jak w przypadku wartości **Null** czy **Empty**. Parametrowi, któremu nie nadano wartości przy wywołaniu, można nadać wartość w treści procedury, ale wtedy — oczywiście — zmienna już nie będzie `Missing`. Oznacza to, że po nadaniu wartości nie można już sprawdzić, czy ta wartość została przekazana przy wywołaniu procedury, czy nadana w jej treści.

Alternatywą testu **IsMissing** jest test **IsError**. Brak argumentu generuje wartość błędu 448, któremu odpowiada opis *Named argument not found*. Opis ten nie jest w pełni adekwatny do sytuacji, ale czasem taki test może być wygodniejszy. Numer błędu odczytamy, stosując konwersję na tekst **CStr(*zmienna*)**. Wynikiem będzie tekst `Error 448`.

Użycie tablicy parametrów

Inny sposób przekazania zmiennej liczby argumentów do funkcji polega na użyciu tablicy **ParamArray** jako parametru. Może być tylko jedna taka tablica i musi występować jako ostatnia na liście parametrów. Jeśli zadeklarowano użycie tablicy **ParamArray**, nie można używać parametrów opcjonalnych. Po jej nazwie należy umieścić parę pustych nawiasów. Jest to tablica złożona z elementów typu **Variant** (czyli typu danych nie należy określać). Dolny indeks tej tablicy zawsze jest równy 0, niezależnie od ewentualnego użycia dyrektywy **Option Base 1**. Górny indeks można sprawdzić w kodzie naszej funkcji, używając funkcji **UBound**. Jeśli do tablicy parametrów nie przekazano żadnych argumentów, funkcja **UBound** ma wartość –1.

Na liście argumentów przy wywołaniu funkcji każdy argument składający się na tablicę **ParamArray** musi wystąpić oddzielnie, oddzielony przecinkiem (w arkuszu średnikiem) od następnych. Stanowią one kolejne elementy tablicy. Nie można użyć zakresu i oczekiwać, że poszczególne jego elementy zostaną potraktowane jako kolejne argumenty. Zakres może być użyty, ale stanie się on pojedynczym argumentem i kod funkcji musi wydobyć z niego poszczególne wartości. Przyjmując, że tablica **ParamArray** ma nazwę ***a***, za pomocą funkcji **IsMissing(*a*)** można sprawdzić, czy przy wywołaniu funkcji do tablicy **ParamArray** przekazano jakiekolwiek argumenty; jeśli przekazano choć jeden, obecność argumentu o numerze ***i*** można sprawdzić funkcją **IsMissing(*a*(*i*))**. Alternatywnie, podobnie jak w przypadku argumentów opcjonalnych, można wykorzystać test **IsError**.

Przy wywołaniu funkcji z poziomu VBA nie można przekazywać argumentów składających się na tablicę **ParamArray** jak argumentów nazwanych, to znaczy, że na przykład nie można użyć konstrukcji ***a***(1):=2.

Problem pustych argumentów

Używając funkcji z tablicą **ParamArray** należy zwrócić uwagę na użycie odwołań do **pustych komórek**, gdyż one również wywołują błąd. Normalnie funkcje agregujące radzą sobie dobrze z problemem pustych argumentów i pustych komórek. Można na przykład bezpiecznie napisać:

```
= ILOCZYN(3; ; 7)          lub          = ILOCZYN(3; pusta; 7)
```

gdzie `pusta` jest odwołaniem do pustej komórki. W obu przypadkach otrzymamy w wyniku 21.

Gdyby jednak użyć jako funkcji pośredniczącej funkcji UDF, która tylko pobierałaby argumenty i przekazywałaby je w formie tablicy do dalszego przetworzenia:

```
Function WyO(ParamArray x())
   WyO = x
End Function
```

to takie wywołania:

```
= ILOCZYN(WyO(3; ; 7))          lub          = ILOCZYN(WyO(3; pusta; 7))
```

zakończą się błędem #ARG!. Błąd dotyczy całego wywołania funkcji ***WyO***, a nie tylko pojedynczych elementów tablicy, a więc nie można go zamaskować funkcją JEŻELI.BŁĄD.

Sygnalizowany tu problem dotyczy tylko tablicy **ParamArray**, a nie argumentów opcjonalnych, poprzedzonych słowem **Optional**.

Aby wyeliminować ten problem, należy w treści funkcji testować argumenty funkcją **IsMissing**, aby wykryć pominięte argumenty i funkcją **IsEmpty**, aby znaleźć odwołania do pustych komórek. Zamiast testu **IsMissing** można użyć testu **IsError**; wówczas zamaskujemy nie tylko pominięte argumenty, ale również te, które zawierają błędy. Wartości te wystarczy zastąpić wartością **Empty**, aby funkcje agregujące i niektóre tablicowe prawidłowo interpretowały te wywołania. Oto przykładowy kod:

```
Function Wy(ParamArray x())
   Dim i As Long
   For i = 0 To UBound(x)
      If IsMissing(x(i)) Or IsEmpty(x(i)) Then x(i) = Empty
   Next i
   Wy = x
End Function
```

Tu ciekawostka. Pusta komórka jest interpretowana w kodzie funkcji jako wartość **Empty**, ale jest to odwołanie (**TypeName** = **Range**), jednak trzeba jeszcze raz przypisać jej wartość **Empty**, aby usunąć ślad po tym odwołaniu (**TypeName** = **Empty**).

Problem tablic wśród argumentów ParamArray

Takie rozwiązanie będzie wystarczające, jeśli argumenty przekazywane do tablicy będą tylko pojedynczymi wartościami. Jeśli będziemy chcieli przekazać również tablice i zakresy, to taki kod nie zda egzaminu, bo struktura tablicy wynikowej będzie „niestrawna" dla funkcji zewnętrznych. W takim przypadku należy zastosować kod, który „wydobędzie" wszystkie elementy składowe tablicy argumentów i utworzy z nich tablicę jednowymiarową, pomijając „po drodze" elementy puste.

```
Function Wy_(ParamArray x())
   Dim i As Long, el, elw, y()
   For Each el In x
      If Not (IsMissing(el) Or IsEmpty(el)) Then
         If IsArray(el) Then
            For Each elw In el
               i = i + 1
               ReDim Preserve y(1 To i)
               y(i) = elw
            Next elw
         Else
            i = i + 1
            ReDim Preserve y(1 To i)
            y(i) = el
         End If
      End If
```

```
    Next el
    Wy_ = y
End Function
```

Jeśli chcemy uniknąć powtarzania niemal identycznego fragmentu kodu dla tablic i wartości skalarnych, można zastosować „sztuczkę" z nadaniem pojedynczej wartości struktury tablicy, a następnie wykonać kod dla tablic. Odpowiednia instrukcja będzie miała postać:

```
If Not IsArray(el) Then el = Array(el)
```

Należy uwzględnić, że powyższy kod pomija elementy puste, więc liczba elementów w tablicy wynikowej może być mniejsza od liczby elementów danych. Jeśli zachowanie identycznej liczby danych jest istotne, należy zmodyfikować kod tak, żeby zamiast pomijania elementów pustych były w ich miejscach umieszczane wartości **Empty**. Jeśli liczba argumentów jest duża, każdorazowe rozszerzanie tablicy wynikowej o jeden element wydłuża czas wykonania funkcji. W takim przypadku warto rozważyć rozszerzanie tablicy wynikowej blokami o kilkadziesiąt czy kilkaset pozycji. Wydłuży to kod, bo trzeba za każdym razem sprawdzać, czy w tablicy jest jeszcze miejsce na nową wartość, ale przyspieszy wykonanie. Na końcu należy „odciąć" niewykorzystany fragment tablicy.

Deklaracja nagłówka funkcji o zmiennej liczbie parametrów

Nagłówek funkcji wykorzystującej tablicę parametrów może wyglądać na przykład tak:

```
Function Wielomian1(x As Double, ParamArray wsp()) As Double
```

Funkcja ta ma obliczać wartość wielomianu o współczynnikach określonych przez elementy tablicy ***wsp*** dla argumentu ***x***. Wywołanie tej funkcji w VBA może mieć postać:

```
Wielomian1(3.5,1,3,4,–2)
```

a w arkuszu:

```
Wielomian1(3,5;1;3;4;–2)
```

Funkcja obliczy wartość wielomianu x^3+3x^2+4x-2 dla $x = 3{,}5$. Zakładamy, że kod funkcji został tak napisany, że ***wsp***`(0)` jest współczynnikiem wielomianu przy najwyższej potędze x. Kod funkcji musi uwzględniać to, że liczba współczynników, określająca zarazem stopień wielomianu, może być dowolna.

Jeżeli współczynniki wielomianu są umieszczone w komórkach arkusza z zakresu `C7:F7`, to wywołanie z kodu VBA może mieć postać:

```
Wielomian1(3.5, Range("C7"), Range("D7"), Range("E7"), Range("F7"))
```

lub

```
Wielomian1(3.5, [C7], [D7], [E7], [F7])
```

a w arkuszu:

```
Wielomian1(3,5;C7;D7;E7;F7)
```

Gdybyśmy chcieli wywoływać tę funkcję, podając zakres zawierający współczynniki, nagłówek musiałby wyglądać inaczej, na przykład tak:

```
Function Wielomian2(x As Double, wsp) As Double
```

W tym przypadku ***wsp*** jest zmienną typu **Variant**, pod którą przy wywołaniu funkcji można podstawić zakres lub tablicę, tak jak to opisano w rozdziale 4., w punkcie „Tablice w zmiennych typu Variant". Wywołanie tej funkcji w sytuacji, gdy współczynniki wielomianu są umieszczone w komórkach arkusza z zakresu `C7:F7`, ma postać:

w kodzie VBA:

```
Wielomian2(3.5, Range("C7:F7"))        lub        Wielomian2(3.5, [C7:F7])
```

a w arkuszu:

```
Wielomian2(3,5;C7:F7)
```

Gdyby przy tej drugiej postaci funkcji trzeba było w wywołaniu podać współczynniki w formie liczbowej, wywołanie to mogłoby wyglądać tak:

w kodzie VBA:

```
Wielomian2(3.5,Array(1,3,4,–2))
```

a w arkuszu:

```
Wielomian2(3,5;{1;3;4;–2})        lub        Wielomian2(3,5;{1\3\4\–2})
```

Współczynniki wielomianu zostały umieszczone w stałej tablicowej o strukturze wiersza lub tablicy. Który zapis odpowiada której strukturze, to zależy od używanej wersji Excela. W wersji VBA użyto funkcji **Array** (opisanej w rozdziale 4., w punkcie „Funkcja Array"), która zwraca zmienną typu **Variant** o strukturze jednowymiarowej (wierszowej) tablicy złożonej z elementów wymienionych w wywołaniu. Każdy z tych zapisów powinien pozwolić na poprawne obliczenie wartości wielomianu, ale czy tak faktycznie będzie, to już zależy od treści funkcji. Ten aspekt omówiono w rozdziale 10., w podrozdziale „Obliczanie wartości wielomianu".

Nazwa funkcji `WIELOMIAN` jest w Excelu 2010+ wykorzystywana jako nazwa predefiniowana. Zatem, jeśli zdefiniowalibyśmy własną funkcję o takiej nazwie, nie mogłaby być ona użyta jako funkcja UDF.

Przekazywanie zmiennej liczby parametrów pomiędzy funkcjami

Przekazywanie parametrów przez wartość

Należy pamiętać, że wewnątrz funkcji, która została zdefiniowana z tablicą parametrów **ParamArray** ***wsp***(), tablica ta jest jednowymiarową tablicą wierszową, zawierającą odczytane argumenty. I o ile wewnątrz tej funkcji do odczytania wartości argumentów możemy użyć na przykład instrukcji pętli **For Each** … **Next** (o różnych rodzajach pętli pisałem w rozdziale 3., w podrozdziale „Sterowanie wykonaniem kodu") w postaci:

```
Dim element
For Each element In wsp
   instrukcje
Next element
```

to w przypadku, gdy całą tablicę chcemy przekazać jako argument do innej funkcji, na przykład:

```
Wynik = MojaFunkcja(wsp)
```

to jeżeli ***MojaFunkcja*** oczekuje na argument typu **Variant**, nie można przekazać tablicy ***wsp*** przez referencję (wystąpi błąd kompilacji: *Invalid ParamArray use*), a jedynie przez wartość. Można to zrobić kilkoma sposobami. Jeśli deklaracja ***MojejFunkcji*** przewiduje przekazanie argumentu przez referencję, należy najpierw skopiować tablicę ***wsp*** do innej zmiennej typu **Variant** i dopiero tę zmienną przekazać do ***MojejFunkcji***.

```
Dim lista
lista = wsp
Wynik = MojaFunkcja(lista)
```

Można również utworzyć kopię przez wykorzystanie tablicy **Array**:

```
Wynik = MojaFunkcja(Array(wsp)(0))
```

Dwa powyższe sposoby odnoszą się też do sytuacji, gdy ***MojaFunkcja*** jest funkcją arkuszową typu agregującego, akceptującą dowolną liczbę argumentów, udostępnioną przez obiekt **Worksheet** ↪**Function** lub **Application**. W tym przypadku jednak wszystkie argumenty muszą reprezentować pojedyncze wartości.

Jeżeli możemy zadeklarować ***MojąFunkcję*** z parametrem przekazywanym przez wartość

```
Function MojaFunkcja(ByVal arg)
```

to można przekazać tablicę ***wsp*** wprost

```
Wynik = MojaFunkcja(wsp)
```

Jeżeli natomiast ***MojaFunkcja*** również została zadeklarowana z tablicą **ParamArray** ***wspn***(), można przekazać jej bezpośrednio tablicę ***wsp*** jako całość, ale należy mieć świadomość, że zostanie ona potraktowana jako pierwszy (a ściślej zerowy) element nowej tablicy parametrów. Dlatego w treści ***MojejFunkcji*** należy umieścić kod:

```
Dim element, lista
lista = wspn(0)
For Each element In lista
   instrukcje
Next element
```

Jeśli w łańcuchu argumentów przekazywanych z funkcji do funkcji tablica parametrów występowała kilkakrotnie, operację „rozpakowywania" argumentów należy powtórzyć. Odpowiedni przykład podano w rozdziale 6., w punkcie „Funkcje składające dowolne dane w tablice".

Oczywiście, należy pamiętać, że jeśli na liście argumentów ***wsp*** są argumenty grupowe (na przykład zakresy obejmujące kilka komórek), to nie zostaną one automatycznie „rozpakowane". Trzeba to zrobić odpowiednim kodem w treści funkcji przyjmującej dane, na przykład takim, jaki podałem wcześniej, w punkcie „Problem tablic wśród argumentów ParamArray".

Najbardziej skomplikowany jest przypadek, kiedy funkcja zadeklarowana z tablicą **ParamArray** jest funkcją predefiniowaną i nie ma dostępu do jej wnętrza albo z innych powodów nie można jej modyfikować. Argumenty do takiej funkcji muszą być przekazywane pojedynczo. Mogą to być elementy tablicy, ale nie można przekazać tej tablicy hurtem. Problemem, który trzeba rozwiązać, jest nieznana z góry liczba argumentów do przekazania. Musimy zadecydować, ile tych argumentów może być maksymalnie. To zależy od przeznaczenia funkcji i programista musi podjąć tę decyzję. Funkcje arkuszowe, wywoływane w VBA za pomocą obiektu **WorksheetFunction**, przewidują przekazywanie do 30 argumentów. Wydaje się, że jest to liczba wystarczająca do większości zastosowań, w wielu przypadkach może wystarczyć mniej. W treści funkcji zewnętrznej (tej, z której chcemy przekazać tablicę parametrów) musimy zadeklarować tablicę typu **Variant** o stałej liczbie elementów i przepisać do niej argumenty wprowadzone przy wywołaniu. Problemem są nadmiarowe elementy, których nie wykorzystujemy. W wyniku użycia instrukcji **ReDim Preserve** są one inicjowane wartością **Empty**. Jeśli takie dane zostaną przekazane do funkcji wewnętrznej, będą interpretowane tak, jak wartości zerowe. W niektórych przypadkach może to być akceptowalne (np. w funkcjach wykonujących operacje sumowania), ale w innych (np. w funkcji obliczającej średnią albo funkcji, która ma określoną liczbę argumentów) już nie. Problem można rozwiązać, nadając zbędnym elementom tablicy wartość **Null**. Wartości te będą ignorowane przez funkcję wewnętrzną, tak jakby ich nie było. Poniżej przedstawiam listing funkcji, która pozwala wywoływać dowolną funkcję z zestawu **WorksheetFunction** przez podanie tekstu jej angielskiej nazwy jako pierwszego argumentu, a następnie pozostałych niezbędnych argumentów:

```
Function Use(fname As String, ParamArray Data())
   Dim d() As Variant, n As Long   ' konieczne nawiasy przy deklaracji d()
   d = Data
   ReDim Preserve d(29)
   For n = UBound(Data) + 1 To 29
      d(n) = Null
   Next n
   Use = CallByName(WorksheetFunction, fname, VbMethod, _
         d(0), d(1), d(2), d(3), d(4), d(5), _
         d(6), d(7), d(8), d(9), d(10), d(11), _
         d(12), d(13), d(14), d(15), d(16), d(17), _
         d(18), d(19), d(20), d(21), d(22), d(23), _
         d(24), d(25), d(26), d(27), d(28), d(29))
End Function
```

Po przekazaniu wszystkich argumentów funkcji poszerzamy tablicę do wymaganego rozmiaru, uzupełniamy brakujące elementy wartościami **Null** i przekazujemy wszystkie elementy tablicy do funkcji **CallByName**, która umożliwia wywołanie metod obiektów VBA przez tekst nazwy (angielskiej). Opis tej funkcji podano w rozdziale 9., w podrozdziale „Funkcja CallByName — wywoływanie metod przez tekst ich nazwy". W wywołaniu funkcji **CallByName** można użyć obiektu **WorksheetFunction** lub obiektu **Application**, przy czym jeśli obiektem jest **WorksheetFunction**, a użyto argumentu, który nie jest obsługiwany w tym trybie, funkcja automatycznie przestawia się w tryb z kwalifikatorem **Application**. Różnice wyjaśniono w rozdziale 1., w punkcie „Użycie funkcji w wyrażeniach".

Podobny sposób można zastosować w przypadku funkcji UDF współpracujących z funkcjami arkuszowymi, gdy chcemy użyć funkcji jako argumentu, a równocześnie chcemy, aby funkcja arkuszowa odczytała to jako pominięcie argumentu. W takim przypadku należy przekazać wartość **Null**. Przykładem może być funkcja arkuszowa ADRES. Ostatnim (opcjonalnym) jej argumentem jest nazwa arkusza. Jeśli ten argument zostanie podany, funkcja generuje tekst adresu kwalifikowanego, na przykład Arkusz2!B5. Jeśli argument zostanie pominięty, otrzymamy tekst adresu bez kwalifikatora, na przykład B5. Jeśli jako argument zostanie przekazany tekst pusty, funkcja wygeneruje tekst adresu w postaci !B5. Taka postać adresu jest dopuszczalna tylko przy tworzeniu nazw, natomiast użyta bezpośrednio w arkuszu spowoduje błąd. Aby tego błędu uniknąć, a równocześnie nie pisać wariantowych wywołań funkcji ADRES, można (w sytuacji, gdy nie chcemy adresu kwalifikowanego) nadać funkcji UDF wartość **Null**. Podobną metodę można zastosować w przypadku dwóch ostatnich argumentów funkcji PRZESUNIĘCIE i generalnie tych funkcji, które mają na końcu więcej niż jeden argument opcjonalny. Funkcje zwykłe, które mają tylko jeden taki argument, traktują wartość **Null** jak zero.

Przekazywanie parametrów przez referencję

Omówione dotychczas sposoby przekazywania parametrów pomiędzy funkcjami sprowadzały się do przekazywania wartości. Mechanizm zastosowany w przypadku tablicy **ParamArray**, choć w zasadzie powinien działać przez referencję, zawiera jednak ograniczenia, które utrudniają napisanie kodu w taki sposób, aby funkcje zewnętrzne mogły zmodyfikować zawartość argumentów przekazywanych w tablicy **ParamArray**.

Należy również wziąć pod uwagę, czy argumentami przekazanymi poprzez tablicę **ParamArray** są tablice, czy zakresy komórek. Zacznijmy od analizy poniższych przykładów, które zilustrują możliwe przypadki.

Przykład 5.1.

W tym przykładzie dane są odwołaniami. W pierwszym kodzie użyto zmiennej typu **Variant**.

```
Sub AddOne1(ParamArray rg())
   Dim e
   For Each e In rg
      e = e + 1
      Debug.Print e;
   Next e
```

```
    Debug.Print
    Debug.Print Join(rg, " ")    'wartości niezmienione
End Sub
Sub TestAddOne1()
    AddOne1 [A1], [B1], [C1]
End Sub
```

Ze względu na użycie zmiennej ***e*** typu **Variant**, wartości ***e*** modyfikowane są tylko lokalnie, oryginalne wartości w komórkach nie zmieniają się. W drugim kodzie występuje jawne odwołanie do wartości obiektów zakresów, dzięki temu wartości w komórkach zostały zmienione.

```
Sub AddOne2(ParamArray rg())
    Dim e      'musi być Variant
    For Each e In rg
        e.Value = e.Value + 1
        Debug.Print e;
    Next e
    Debug.Print
    Debug.Print Join(rg, "; ")    'wartości zmienione
End Sub
Sub TestAddOne2()
    AddOne2 [A31], [B31], [C31]
End Sub
```

Przykład 5.2.

Kolejne dwa warianty kodu będą dotyczyć tablic jako argumentów.

```
Sub AddOne3(ParamArray tb())
    Dim e      'musi być Variant
    For Each e In tb
        e = e + 1
        Debug.Print e;
    Next e
    Debug.Print
    Debug.Print Join(tb, "; ")    'wartości niezmienione
End Sub
Sub TestAddOne3()
    Dim tabl
    tabl = Array(3, 7, 10)
    AddOne3 tabl(0), tabl(1), tabl(2)
    Debug.Print Join(tabl, "; ")    'wartości niezmienione
End Sub
```

W tym przypadku, podobnie jak w pierwszym wariancie kodu w przykładzie 5.1, w zmiennej ***e*** tworzone są tylko lokalne kopie elementów tablicy; w efekcie oryginalna tablica pozostaje bez zmian.

```
Sub AddOne4(ParamArray tb())
    Dim i As Long
    For i = 0 To UBound(tb)
        tb(i) = tb(i) + 1
        Debug.Print tb(i);
    Next i
    Debug.Print
    Debug.Print Join(tb, "; ")          'wartości zmienione
End Sub
```

```
Sub TestAddOne4()
   Dim tabl
   tabl = Array(3, 7, 10)
   AddOne4 tabl(0), tabl(1), tabl(2)
   Debug.Print Join(tabl, "; ")       'wartości zmienione
End Sub
```

W czwartym wariancie kodu odwołujemy się indywidualnie do poszczególnych elementów tablicy ze wskazaniem jej nazwy. W tym przypadku wywołanie przez referencję działa zgodnie z założeniem.

Przykład 5.3.

Z bardziej złożonym przypadkiem będziemy mieć do czynienia, gdy chcemy wykorzystać uniwersalną procedurę pośredniczącą zadeklarowaną z tablicą **ParamArray**. Proszę przeanalizować poniższy przykładowy kod:

```
Sub ModyfikujTablice()
   'modyfikuje elementy tablic a(), b() i c()
   Dim a, b, c
   a = Array(12, 1)    'Array(12, 1, 3)
   b = Array(5, –7)
   c = Array(10, 8, –4)
   Modyfikuj a, b, c
   Debug.Print "a(): "; Join(a) 'pokaż zawartość tablic
   Debug.Print "b(): "; Join(b)
   Debug.Print "c(): "; Join(c)
End Sub

Sub Modyfikuj(ParamArray y())
   'uniwersalna procedura modyfikująca
   Dim i As Long, x
   x = y 'nie można bezpośrednio użyć y
   'podejmij działanie w zależności od liczby elementów w pierwszym argumencie y(0)
   If UBound(y(0)) > 1 Then
      Dodaj x, 1
   Else
      Dodaj x, 2
   End If
   'kopiuj wyniki do tablicy ParamArray
   ' y = x        'to nie działa na tablice zewnętrzne, tylko lokalnie
   For i = 0 To UBound(y)
      y(i) = x(i)
   Next i      'to działa na tablice zewnętrzne
End Sub

Sub Dodaj(y, k As Long)
   'inkrementuje o k wszystkie zagnieżdżone elementy y
   Dim i As Long, j As Long
   For i = LBound(y) To UBound(y)
      For j = LBound(y(i)) To UBound(y(i))
         y(i)(j) = y(i)(j) + k
   Next j, i
End Sub
```

Źródłem problemu jest niemożność bezpośredniego przekazania przez referencję tablicy **ParamArray** w procedurze ***Modyfikuj***. Przez podstawienie ***x*** = ***y*** tworzymy kopię tablicy ***y***, ale już nie przez referencję, tylko przez wartość. Procedura ***Dodaj*** działa na tej kopii i wszelkie zmiany są wprowadzane do zmiennej ***x***, a nie do ***y***. Zwykłe przypisanie zwrotne całej tablicy ***y*** = ***x*** znowu odbywałoby się przez wartość i tablica ***y*** miałaby nowe wartości tylko wewnątrz procedury ***Modyfikuj***, a po jej opuszczeniu tablice ***a***, ***b*** i ***c***, przekazane w tablicy **ParamArray** ***y*** miałyby wartości niezmienione. Obejście tego problemu polega na przypisaniu elementów tablicy ***x*** do ***y*** pojedynczo, w pętli. W tym przypadku nowe wartości ***x*** są kopiowane pod stare adresy tablic ***a***, ***b*** i ***c***, czyli przez referencję.

Funkcja zwracająca wynik w postaci tablicy

Zazwyczaj funkcja zwraca w wyniku jedną wartość, jednak nic nie stoi na przeszkodzie, aby napisać taką, która będzie zwracać tablicę wartości. W deklaracji takiej funkcji można nie podawać typu wyniku (zastosowany zostanie typ **Variant**) albo po nazwie typu wyniku dać parę pustych nawiasów, na przykład zapis **As Double()** oznacza, że wynik będzie tablicą złożoną z liczb typu **Double**. W kodzie funkcji należy tę tablicę utworzyć, a w końcu przypisać ją do nazwy funkcji. Jeśli typ funkcji nie jest zadeklarowany (lub jest zadeklarowany jako tablica elementów typu **Variant**), można też przypisać wynik do nazwy funkcji bezpośrednio, używając funkcji **Array** lub wyrażenia zwracającego wynik w postaci tablicy. Można modyfikować rozmiary wynikowej tablicy przez użycie instrukcji **ReDim** do nazwy funkcji, ale nie można nadawać wartości pojedynczo, gdyż każde wywołanie nazwy funkcji, po którym następują nawiasy, jest traktowane jak wywołanie rekurencyjne.

```
Sub TestArr1()
   Debug.Print Join(Arr1)
End Sub
Function Arr1() As Variant()
   Arr1 = Array(3.5, 1.5, 5, 3, 4)
   ReDim Preserve Arr1(2)  '0 To
End Function
```

Aby wyświetlić wyniki takiej funkcji w arkuszu kalkulacyjnym, trzeba skonstruować odpowiednie wyrażenie tablicowe i zatwierdzić je klawiszami *Ctrl+Shift+Enter* lub odwoływać się do poszczególnych składowych wyniku za pośrednictwem funkcji arkusza `INDEKS`. Można również użyć wywołania takiej funkcji w wyrażeniu na analogicznych zasadach jak w przypadku funkcji standardowych. Pamiętać też należy, że jeśli funkcja daje w wyniku tablicę jednowymiarową, to wartości wynikowe będą tworzyły wiersz. Aby uzyskać kolumnę, trzeba umieścić wyniki w tablicy dwuwymiarowej, w której drugi indeks jest zawsze równy 1 albo użyć funkcji **Array** wraz z funkcją **WorksheetFunction.Transpose**. W tym drugim przypadku wynik musi być typu **Variant**.

Jeśli rozmiary tablicy zwracanej przez funkcję są określone w sposób sztywny w treści funkcji, użytkownik musi zadbać o zaznaczenie odpowiednio dużego zakresu na wynik. Jakie są konsekwencje niedopasowania rozmiarów zakresu i tablicy, opisałem w rozdziale 4., w punkcie „Wypełnianie zakresu zawartością tablicy". Możliwe jest również napisanie funkcji UDF, która uwzględni wielkość zakresu zaznaczonego przez użytkownika. Odczytanie wielkości tego zakresu w treści

funkcji jest możliwe dzięki właściwości **Application.Caller**, o której napisałem więcej w rozdziale 8., w punkcie „Identyfikacja komórki, z której wywołano funkcję UDF". Właściwość ta zwraca obiekt zakresu, z którego wywołano funkcję UDF. Jeśli jest to obiekt wielokomórkowy, można odczytać jego rozmiary poprzez odczyt właściwości **Application.Caller.Rows.Count** i **Application.Caller.Columns.Count**. Jeśli — przykładowo — funkcja ma zwrócić tablicę jednowymiarową, można sprawdzić w treści funkcji, czy użytkownik zaznaczył zakres wierszowy, czy kolumnowy i dostosować do tego orientację tablicy.

```
Function Wektor(Optional pocz As Long = 1, Optional kon)
   Dim Tabl(), i As Long, wk As Long
   Dim wiersze As Long, kolumny As Long
   wiersze = Application.Caller.Rows.Count
   kolumny = Application.Caller.Columns.Count
   wk = Application.Max(wiersze, kolumny)
   ReDim Tabl(1 To wk)
   If IsMissing(kon) Then
      kon = pocz + wk - 1
   Else
      wk = Application.Min(Abs(kon - pocz) + 1, wk)
   End If
   For i = 1 To wk
      Tabl(i) = pocz + Sgn(kon - pocz)*(i - 1)
   Next i
   For i = wk + 1 To UBound(Tabl)
      Tabl(i) = ""
   Next i
   If kolumny > wiersze Then
      Wektor = Tabl
   Else
      Wektor = Application.Transpose(Tabl)
   End If
End Function
```

Przykładowa funkcja wypełnia zaznaczony zakres komórek kolejnymi liczbami całkowitymi od argumentu ***pocz*** aż do wartości ***kon*** (jeśli została podana) lub wypełnienia wiersza lub kolumny, w zależności od zaznaczonego zakresu. Jeśli podano ***kon*** < ***pocz***, komórki są wypełniane malejąco. Jeśli zaznaczono zbyt duży zakres, nadwyżkowe komórki są wypełniane tekstem pustym, a nie kodami błędu, jak w przypadku funkcji arkuszowych.

Przeliczanie wartości funkcji

Gdy jest włączony tryb automatycznego przeliczania formuł w arkuszu (ustawienie domyślne), każda zmiana wartości w arkuszu wywołuje przeliczenie wszystkich komórek zawierających formuły zależne od komórek, które uległy zmianie. W przypadku funkcji arkuszowych sprawa nie budzi większych wątpliwości, gdyż wszystkie argumenty funkcji muszą być jawnie podane. Niektóre funkcje mają status ulotnych (są przeliczane przy każdym przeliczeniu arkusza). Inaczej rzecz się ma w przypadku funkcji UDF, których treść zapisano w VBA. Tu w treści funkcji mogą wystąpić odwołania do komórek lub nazw, które nie są wyraźnie wymienione na liście argumentów. W takiej sytuacji przy przeliczaniu arkusza Excel „nie wie", czy taka funkcja powinna

być przeliczona. Jednym z możliwych rozwiązań tego problemu jest umieszczenie na początku kodu funkcji dyspozycji:

```
Application.Volatile
```

Spowoduje to przeliczenie funkcji przy każdym przeliczeniu arkusza. Jest to jednak rozwiązanie niezbyt efektywne, bo może powodować wielokrotne zbędne przeliczanie. Lepiej zadbać o to, by wszystkie argumenty były jawnie wskazane w nagłówku funkcji. Wtedy łatwo ustalić, kiedy funkcja musi być przeliczona.

Jeśli w treści funkcji UDF nie użyto polecenia **`Volatile`**, funkcja jest przeliczana „na życzenie" dopiero po naciśnięciu klawiszy *Ctrl+Alt+F9* lub po edycji komórki z tą funkcją. Jeżeli natomiast je zastosowano, do przeliczenia wystarczy użycie klawisza *F9* lub edycja dowolnej komórki.

Sprawa jawnej specyfikacji argumentów nabiera szczególnego znaczenia, gdy funkcja ma być użyta w obliczeniach iteracyjnych. Od wersji 2003 zasady przeliczania arkusza zmieniły się i wszystkie argumenty funkcji, które podlegają zmianie, powinny być jawnie wymienione, w przeciwnym razie nawet w przypadku użycia polecenia **`Application.Volatile`** funkcja jest przeliczana tylko jeden raz, a nie przy każdej iteracji.

Przykład takiej sytuacji omówiono w rozdziale 7., w podrozdziale „Metoda Evaluate".

Wywołanie makroinstrukcji z kodu VBA

To, co napisałem powyżej o funkcjach UDF, odnosi się też w dużym stopniu do makroinstrukcji. Zasadnicza różnica polega na tym, że makroinstrukcje mogą zmieniać stan arkusza (wpisywać dane, zmieniać format komórek itp.), a funkcje UDF, w zasadzie, nie mogą — mają tylko zwrócić wynik obliczeń do miejsca wywołania (o wyjątkach od tej reguły piszę w rozdziale 16.). Ponadto makroinstrukcje, które zawierają parametry wywołania, mogą być wywołane wyłącznie z poziomu kodu VBA. Z poziomu arkusza można wywoływać tylko makroinstrukcje bez parametrów. Nie dotyczy to natomiast tych, które są wywoływane automatycznie do obsługi zdarzeń, ale one mają stały zestaw parametrów.

Wywołanie makroinstrukcji z poziomu kodu VBA może być zrealizowane kilkoma sposobami: albo przez wpisanie nazwy makroinstrukcji, a następnie wartości argumentów wywołania (jeżeli występują), albo przez użycie słowa **`Call`** poprzedzającego nazwę makroinstrukcji, albo za pomocą metody **`Run`**. Jeżeli użyto słowa **`Call`**, argumenty wywołania makra muszą być umieszczone w nawiasach; jeżeli użyto tylko nazwy makra bez **`Call`**, należy też pominąć nawiasy. Słowo **`Call`** może być też formalnie zastosowane do wywołania funkcji, jednak wówczas nie można przekazać wyniku funkcji — takie wywołanie może mieć sens tylko w szczególnych przypadkach. Inny sposób wywołania makra to użycie konstrukcji:

```
Application.Run "nazwa_makra", arg_1, arg_2, …
```

Nazwa makroinstrukcji jest podawana jako tekst w cudzysłowie, następnie podaje się argumenty wywołania oddzielone przecinkami. W analogiczny sposób można również wywołać funkcję:

```
wynik = Application.Run("nazwa_funkcji", arg_1, arg_2, …)
```

Istotne jest, że za pomocą metody **`Run`** można wywoływać tylko funkcje zdefiniowane w kodzie VBA, a nie funkcje wewnętrzne Visual Basica.

Makroinstrukcje można wywoływać również pośrednio, za pomocą metody **`Evaluate`**, zarówno z kodu, jak i z arkusza. Wywołuje się je podobnie jak funkcje. Ponieważ makroinstrukcja nie zwraca wyniku, nie należy używać nawiasów otaczających tekstowy argument metody, natomiast nawiasy są potrzebne wewnątrz argumentu tekstowego, niezależnie od tego, czy makro ma być wywołane z argumentami, czy bez. (O użyciu tej metody piszę w rozdziale 7., w podrozdziale „Metoda Evaluate" i w rozdziale 16., w podrozdziale „Funkcje UDF wywoływane w sposób pośredni".) Niestety, w tym przypadku obowiązują ograniczenia i makra wywoływane w ten sposób nie mogą wykonywać wszystkich zmian w arkuszu, ale jest to pewien sposób na obejście problemu przekazywania argumentów do makroinstrukcji wywoływanej z poziomu arkusza, gdyż tu można je przekazać podobnie jak do funkcji.

Funkcje użytkownika podobne do funkcji standardowych

Korzystając z funkcji standardowych dostępnych w arkuszu kalkulacyjnym, nie zastanawiamy się zwykle nad tym, że funkcja, która na przykład wymaga podania argumentu w postaci liczby rzeczywistej, może w istocie przyjąć jako argument stałą liczbową, stałą tablicową, adres pojedynczej komórki, wyrażenie dające w wyniku liczbę, nazwy reprezentujące wymienione wcześniej rodzaje danych, ale również argument tekstowy, który ma postać liczby (liczbotekst) i zostanie skonwertowany na liczbę. Argument może także przybrać formę tablicy, jeżeli funkcja zostanie użyta w odwołaniu tablicowym. Ta tablica również może mieć postać stałej tablicowej, zakresu komórek, wyrażenia dającego w wyniku zakres komórek lub tablicę wartości oraz nazwy reprezentującej jedną z wymienionych wcześniej postaci. Dla przykładu można spróbować wywołać w arkuszu funkcję `LN` z różnego rodzaju argumentami:

- `=LN(2)`
- `=LN({2})`
- `=LN("2")`
- `=LN(C3)`, przy założeniu, że w komórce `C3` umieszczono liczbę `2`
- `=LN(G2+1)`, przy założeniu, że w komórce `G2` umieszczono liczbę `1`
- `=LN(`***`arg`***`)`, przy założeniu, że nazwa ***`arg`*** została wcześniej zdefiniowana i reprezentuje jeden z typów danych wymienionych powyżej.

Można również użyć wywołania tablicowego — zaznaczywszy wcześniej trzy sąsiednie komórki (poziomo), trzeba wprowadzić do nich jedno z poniższych wyrażeń tablicowych i zatwierdzić przez *Ctrl+Shift+Enter*:

- `=LN({2;5;8,5})` (w Excelu 2010+ wprowadzamy `=LN({2\5\8,5})`)
- `=LN(C3:E3)`, przy założeniu, że w zakresie komórek `C3:E3` umieszczono liczby, na przykład `2`; `5` i `8,5`

- `= LN(C3:E3+1)`, przy założeniu, że w zakresie komórek `C3:E3` umieszczono odpowiednie liczby
- `= LN(arg_tabl)`, przy założeniu, że nazwa ***arg_tabl*** została wcześniej zdefiniowana i reprezentuje jeden z typów danych tablicowych wymienionych powyżej.

Niestety, nie wszystkie funkcje predefiniowane w Excelu akceptują tak różnorodne typy argumentów. Łatwo sprawdzić, że na przykład funkcje pochodzące z dodatku *Analysis ToolPak*, mimo że od wersji 2007 dodatek ten jest już zintegrowany z arkuszem, nie akceptują argumentów w postaci odwołań do zakresów wielokomórkowych.

Możemy to sprawdzić na przykładzie jednej z funkcji z tego dodatku: `SQRTPI` (od wersji 2010 występuje pod spolszczoną nazwą `PIERW.PI`). Funkcja ta oblicza pierwiastek z argumentu pomnożonego przez liczbę π. W razie użycia jej z argumentem w postaci tablicy funkcja może zwracać kod błędu `#ARG!`. Konkretny wynik zależy od wersji Excela i postaci argumentu. Przykładowo, Excel 2000 i 2003 nie akceptuje żadnych argumentów tablicowych, wersje 2007+ akceptują stałe tablicowe, ale nie przyjmują zakresów.

Jeżeli piszemy własną funkcję w VBA i chcemy, aby zachowywała się podobnie jak funkcje standardowe, musimy sami zadbać o to, aby funkcja akceptowała różne rodzaje danych. Jeżeli typ argumentu zostanie zadeklarowany jako **`Single`**, **`Double`** lub typ całkowitoliczbowy, w wywołaniu funkcji może wystąpić dowolny rodzaj danych, odnoszący się do pojedynczej wartości (z wyjątkiem stałej tablicowej). Jeżeli typ argumentu zostanie zadeklarowany jako **`Object`** lub **`Range`**, w wywołaniu funkcji może wystąpić tylko adres komórki lub nazwa tej komórki. Dotyczy to sytuacji, gdy w treści funkcji argument jest traktowany jak pojedyncza wartość.

Jeżeli chcemy, aby własna funkcja mogła być używana w odwołaniu tablicowym, akceptowała dane tablicowe i zwracała wynik w postaci tablicy, musimy w treści funkcji sprawdzić, jaki rodzaj danych reprezentuje argument i stosownie do tego zorganizować obliczenia. Załóżmy, że mamy przykładową funkcję `SQRTPI`, która pobiera jedną liczbę jako argument i zwraca jedną liczbę jako wynik. Chcemy zmodyfikować ją tak, aby mogła akceptować również argument w postaci tablicy i zwracać wynik w postaci tablicy o takich samych rozmiarach.

Taka funkcja musi przyjmować argument typu **`Variant`** i sama zwracać wynik tego samego typu. W treści funkcji należy sprawdzić, czy argument jest pojedynczą liczbą, czy tablicą — w tym drugim przypadku trzeba jeszcze odróżnić zakres komórek od zwykłej tablicy. Do sprawdzenia, czy mamy do czynienia z tablicą, służy funkcja **`IsArray`**, natomiast za pomocą **`IsObject`** sprawdzimy, czy mamy do czynienia z zakresem. W różny sposób sprawdza się rozmiary zwykłej tablicy (służą do tego funkcje **`LBound`** i **`UBound`**) i zakresu (właściwości **`Rows.Count`** i **`Columns.Count`**). Osobna obsługa zakresów i tablic jest uzasadniona, jeśli funkcja ma odwoływać się do innych niż wartość właściwości zakresu. W tym przypadku potrzebna jest tylko wartość, więc można skopiować wartość zakresu do tablicy, stosując zwykłe przypisanie ***y*** `=` ***x***. Jeżeli ***y*** jest zmienną typu **`Variant`**, a ***x*** zakresem, to taka operacja spowoduje, że ***y*** będzie przechowywać wartości zawarte w zakresie ***x***, nie będąc obiektem.

Musimy jeszcze sprawdzić, czy argument tablicowy jest jednowymiarowy, czy dwuwymiarowy — od tego zależy sposób odwoływania się do jego elementów. Jeśli pierwotnym argumentem był zakres, to tablica będzie zawsze obiektem dwuwymiarowym, natomiast jeśli pierwotnym argumentem było wyrażenie zwracające tablicę lub stała tablicowa, to tablica zawierająca jeden wiersz jest obiektem jednowymiarowym, a inne tablice (w tym jednokolumnowa) mają strukturę dwuwymiarową.

Nie ma możliwości bezpośredniego sprawdzenia, czy tablica jest jednowymiarowa, czy dwuwymiarowa. Można to zrobić pośrednio, próbując odczytać drugi wymiar tablicy i sprawdzając, czy wystąpi błąd, czy nie. Metodę tę wykorzystano za pośrednictwem zmiennej ***TwoDim***, która przyjmuje wartość logiczną **True**, jeśli sprawdzany argument jest tablicą dwuwymiarową.

Korzystnie będzie oddzielić część ogólną procedury, która będzie wspólna dla różnych funkcji, od definicji samej funkcji poddawanej „opakowaniu". Aby to było możliwe, nazwa funkcji będzie przekazywana do dalszej obróbki w postaci tekstowej i po podstawieniu pojedynczej wartości argumentu wartość funkcji będzie obliczana za pomocą metody **Evaluate** (por. podrozdział „Metoda Evaluate" na początku rozdziału 7. i podrozdział „Funkcje UDF wywoływane w sposób pośredni" w rozdziale 16.).

Ponieważ funkcja, którą chcemy zmodyfikować, pochodzi z dodatku *Analysis ToolPak*, sposób odwołania do niej zależy od wersji arkusza, w której chcemy uruchomić funkcję. Jak już pisałem w rozdziale 1., w podrozdziale „Wyrażenia", w wersjach wcześniejszych — do 2003 — można nazwę funkcji poprzedzić kwalifikatorem [***atpvbaen.xls***], oznaczającym nazwę pliku biblioteki funkcji, którą trzeba wcześniej zainstalować; można również użyć samej nazwy funkcji. Od wersji 2007 nazwę funkcji przy bezpośrednim wywołaniu należy poprzedzić słowem **WorksheetFunction** lub **Application**, jednak w omawianym przykładzie, gdzie wywołanie jest pośrednie, a funkcja w postaci tekstowej jest obliczana za pomocą metody **Evaluate**, wystarczy sama nazwa funkcji.

Całość można zapisać w następujący sposób:

```
Function MySqrtPi(x)
   MySqrtPi = MyFunction("SqrtPi(x)", x)   'albo [atpvbaen.xls].
End Function

Function FVal(FName As String, x)
   If IsError(x) Then
      FVal = x
   Else
      If IsNumeric(x) Then x = Str(x) Else x = Chr(34) & x & Chr(34)
      FVal = Application.ThisCell.Parent.Evaluate(Replace(FName, "x", x))
   End If
End Function

Function MyFunction(FName As String, x)
   Dim i, j, y, TwoDim As Boolean
   If IsArray(x) Then
      y = x              'kopia x jako tablica
      TwoDim = False
      On Error Resume Next
```

```
        TwoDim = UBound(y, 2) > 0
        On Error GoTo 0
        For i = LBound(y) To UBound(y)
           If TwoDim Then
              For j = LBound(y, 2) To UBound(y, 2)
                 y(i, j) = FVal(FName, x(i, j))
              Next j
           Else : y(i) = FVal(FName, x(i))
           End If
        Next i
        MyFunction = y
     Else
        MyFunction = FVal(FName, x)
     End If
  End Function
```

Po zaproponowanej modyfikacji funkcje użytkownika będą działać bardzo podobnie do funkcji predefiniowanych Excela, z jedną różnicą. Gdy w zwykłym wywołaniu argumentem funkcji jest zakres, przypisanie faktycznego argumentu odbywa się na zasadzie rzutu prostokątnego (dokładniej o tym piszę w [A7]). Po zaproponowanej przeze mnie modyfikacji, w takim przypadku faktycznym argumentem będzie pierwsza wartość z zakresu, tak jakby użyto formuły tablicowej. Natomiast gdy argumentem jest wyrażenie, na przykład ***zakres****1, obowiązuje zasada rzutu prostokątnego, gdyż Excel wyznacza faktyczną wartość argumentu zanim zostanie on przekazany do funkcji.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Programowanie i makra w Excelu? Nic strasznego!

- Poznaj Visual Basic for Applications (VBA)
- Naucz się tworzyć makra i własne funkcje
- Zrozum Excela

Na rynku nie brakuje książek opisujących obsługę i zastosowania arkusza kalkulacyjnego MS Excel, żadna jednak nie wprowadzi Cię w tę tematykę tak skutecznie jak ta! Omiń rafy i białe plamy dokumentacji, skorzystaj z doświadczenia autora i śmiało wkrocz w świat niesamowitych możliwości Excela.

Poznaj konstrukcje języka VBA i naucz się przeprowadzać obliczenia za jego pomocą. Odkryj zastosowania formantów, zapanuj nad danymi opisującymi datę i czas, dowiedz się, jak radzić sobie z tekstami, wykorzystaj funkcje definiowane przez użytkownika. Przekonaj się też, do czego mogą Ci się przydać makra.

- Struktura i konstrukcje VBA
- Obiekty, właściwości i metody
- Instrukcje warunkowe i komunikacja
- Definiowanie i używanie tablic oraz funkcji
- Obsługa wyjątków i zdarzeń
- Zaawansowane struktury danych
- Operacje na datach, czasach i tekstach
- Operowanie kolorami i formatowanie warunkowe
- Graficzne elementy sterujące
- Funkcje makr programu Excel

Dowiedz się, jak wykorzystać Excela do zautomatyzowania swojej pracy!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-5694-8

9 788328 356948

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 99,00 zł