helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

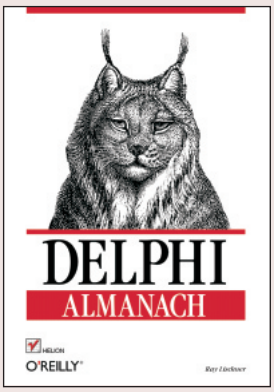
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

# Delphi. Almanach




Autor: Ray Lischner
ISBN: 83-7197-469-8
Tytuł oryginału: Delphi in a Nutshell. A Desktop Quick Reference
Format: B5, stron: około 560

Najnowszy produkt firmy Borland – Delphi 6 – jest kolejną wersją jednego z najpopularniejszych środowisk programistycznych typu RAD dla platformy Windows. Obsługując takie standardy, jak: XML, SOAP, XSL oraz serwisy sieci Web dla różnych platform Microsoft .Net oraz BizTalk z Sun Microsystems ONE zapewnia pełną skalowalność tego typu aplikacji. Delphi 6 zawiera też technologie: BizSnap (wspomagającą integrację działań B2B – Business-to-Bussines – poprzez tworzenie połączeń XML/SOAP Web Services), WebSnap oraz DataSnap, które pomogą użytkownikom tworzyć internetowe aplikacje Web Services, zarówno po stronie serwera, jak i klienta. Dzięki współpracy Delphi 6 z Borland Kylix – pierwszym środowiskiem programistycznym typu RAD dla Linuksa – jego użytkownicy mogą wykorzystać jeden kod źródłowy aplikacji dla platformy Windows i Linux. Kylix zawiera ten sam zestaw narzędzi, takich jak: formularz, Inspektor obiektów, debugger i kompilator. Ponadto biblioteka komponentów Kyliksa – CLX – została opracowana na wzór tradycyjnej biblioteki VCL.

Pierwsze rozdziały wprowadzają do Object Pascala oraz modelu obiektowego Delphi. W następnej kolejności omawiany jest kluczowy mechanizm środowiska – RTTI – często bardzo słabo dokumentowany w innych publikacjach, takich jak oficjalne pliki pomocy Delphi. Książka zawiera również rozdział poświęcony programowaniu współbieżnemu w Delphi oraz tworzeniu aplikacji wielowątkowych.

Sednem niniejszej pozycji jest kompletny, uporządkowany alfabetycznie spis elementów języka Delphi (słów kluczowych, procedur, funkcji, operatorów, zmiennych, stałych, klas, interfejsów). Dla każdego elementu języka podawana jest:

- składnia
- opis
- lista argumentów przyjmowanych przez funkcję lub procedurę
- wskazówki i porady – praktyczny opisu użycia danej cechy języka w rzeczywistym programie
- krótkie przykład
- wskazania pokrewnych elementów języka

Niezależnie od własnego doświadczenia w Delphi, książka ta jest pozycją, po którą programista będzie nieustannie sięgał podczas swojej pracy. Jej treść pozwoli poznać najistotniejsze cechy języka oraz będzie stanowić użyteczną pomoc przy rozwiązywaniu problemów we własnych programach.

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

# *Spis treści*

*Wstęp* ..... *7*

**Rozdział 1.** *Object Pascal* ..... *13*

Moduły ..... 13
Programy ..... 16
Biblioteki ..... 17
Pakiety ..... 19
Typy danych ..... 20
Zmienne i stałe ..... 37
Obsługa wyjątków ..... 38
Operacje na plikach ..... 43
Funkcje i procedury ..... 44

**Rozdział 2.** *Obiektowy model Delphi* ..... *49*

Klasy i obiekty ..... 49
Interfejsy ..... 74
Zliczanie referencji ..... 79
Komunikaty ..... 82
Zarządzanie pamięcią ..... 84
Obiekty starego typu ..... 90

**Rozdział 3.** *Informacja czasu wykonania* ..... *93*

Tablica metod wirtualnych ..... 93
Deklaracje publikowane ..... 96
Moduł TypInfo ..... 101
Metody wirtualne i dynamiczne ..... 110
Inicjalizacja i finalizacja ..... 111

Metody automatyczne ... 113
Interfejsy ... 114
W głąb RTTI ... 115

**Rozdział 4.** *Programowanie współbieżne* ... *119*
Wątki i procesy ... 119
Klasa TThread ... 128
Klasa TIdThread ... 134
Funkcje BeginThread i EndThread ... 137
Lokalny obszar wątku ... 137
Procesy ... 138
Transakcje Futures ... 146

**Rozdział 5.** *Kompendium języka* ... *155*

**Rozdział 6.** *Stałe systemowe* ... *513*
Kody typu wariantowego ... 513
Typy tablic otwartych ... 514
Offsety wirtualnej tablicy metod ... 515
Kody błędów czasu wykonania ... 516
Kody błędów CLX ... 518

**Rozdział 7.** *Operatory* ... *521*
Operatory unarne ... 521
Operatory mnogościowe ... 523
Operatory addytywne ... 524
Operatory porównania ... 526

**Rozdział 8.** *Dyrektywy kompilatora* ... *529*

**Dodatek A** *Narzędzia wiersza poleceń* ... *577*

**Dodatek B** *Moduł SysUtils* ... *595*
Błędy i wyjątki ... 595
Zarządzanie plikami ... 602
Operacje na łańcuchach ... 617

*Skorowidz* ... *673*

# 5

# *Kompendium języka*

Niniejszy rozdział zawiera kompendium języka. Tutaj znaleźć można każde słowo kluczowe, dyrektywę, funkcję, procedurę, zmienną, klasę, metodę oraz właściwość, będące częścią języka Object Pascal. Większość z tych elementów jest zdefiniowana w module `System`, natomiast niektóre w module `SysInit` oraz `Variants`. Moduły te są automatycznie włączane do każdego modułu Delphi. Należy pamiętać, że Object Pascal nie rozróżnia wielkich i małych liter, z jedynym wyjątkiem, dotyczącym procedury `Register` (dla zapewnienia zgodności z Builderem C++).

Dla wygody Czytelnika numery błędów czasu wykonania połączone zostały z odpowiednimi nazwami klas wyjątków. Moduły `System` oraz `SysUtils` przekształcają błędy w wyjątki. Często można usłyszeć, że wyjątki nie są częścią prawdziwego języka Object Pascal, jednak w obecnej wersji Delphi w dużej mierze przestało to być prawdą. Ze względu na powszechne użycie modułów `System` oraz `SysUtils` programiści Delphi znacznie lepiej posługują się wyjątkami niż numerami błędów.

Każdy element tego rozdziału należy do jednej z kilku kategorii, opisanych poniżej:

*Dyrektywa*

Dyrektywa jest identyfikatorem, mającym specjalne znaczenie dla kompilatora, ale tylko w specyficznym kontekście. Poza tym kontekstem programista może swobodnie używać nazw dyrektyw jako zwykłych identyfikatorów. Edytor kodu źródłowego stara się pomagać programiście poprzez wytłuszczanie dyrektyw, gdy są one używane w odpowiednim kontekście, oraz drukowanie ich standardową czcionką, gdy mają charakter zwykłych identyfikatorów. Ponieważ jednak niektóre zasady języka dotyczące dyrektyw są bardzo złożone, czasem edytor po prostu myli się w swoich przewidywaniach.

*Funkcja*

Nie wszystkie funkcje są w rzeczywistości funkcjami; niektóre z nich są wbudowane w kompilator. Różnica nie jest na ogół istotna, ponieważ funkcje wbudowane wyglądają i działają tak, jak zwykłe funkcje, z tą różnicą, iż nie można pobrać ich adresu. Opisy zastosowane w tym rozdziale informują, które funkcje są wbudowane, a które nie.

*Interfejs*

Deklaracja standardowego interfejsu.

*Słowo kluczowe*

Słowo kluczowe jest zarezerwowanym identyfikatorem, którego znaczenie jest określane przez kompilator Delphi. Słów kluczowych nie można używać jako nazw zmiennych, metod lub typów.

*Procedura*

Podobnie jak w przypadku funkcji, niektóre procedury są wbudowane w kompilator, zatem nie można wyznaczyć ich adresu. Pewne procedury (np. `Exit`) zachowują się tak, jakby były wyrażeniami języka, ale nie są zarezerwowanymi słowami kluczowymi. Używa się ich w taki sam sposób, jak zwykłych procedur.

*Typ*

Niektóre z typów są wbudowane w kompilator, ale wiele z nich jest jawnie zdefiniowanych w module `System` lub `Variants`.

*Zmienna*

Większość zmiennych zdefiniowanych w języku Object Pascal to zmienne typów prostych, zawarte w modułach `System`, `SysInit` oraz `Variants`. Różnica pomiędzy tymi modułami polega na tym, że zmienne w module `System` są współdzielone przez wszystkie moduły wchodzące w skład aplikacji, natomiast `SysInit` oraz `Variants` są przekazywane każdemu modułowi jako odrębna kopia. Brak tej wiedzy może doprowadzić do wielu nieporozumień podczas programowania. Inne zmienne (`Self` i `Result`) są wbudowane w kompilator i posiadają specjalne przeznaczenie.

---

# *Abs (funkcja)*

## *Składnia*

```
function Abs(Number: typ numeryczny): typ numeryczny;
```

## *Opis*

Funkcja `Abs` oblicza i zwraca wartość bezwzględną z argumentu. Jest wbudowana w kompilator.

## *Wartość zwracana*

- Jeżeli argument jest typu całkowitego, `Abs` sprawdza, czy wartość jest ujemna i w razie potrzeby neguje ją. W zależności od argumentu, typem zwracanym jest `Integer` lub `Int64`.
- Dla wartości zmiennoprzecinkowych `Abs` zeruje bit znaku, pozostawiając bez zmian wszystkie pozostałe bity, ujemne zero i ujemna nieskończoność stają się dodatnim zerem i dodatnią nieskończonością. Nawet jeśli wartością nie jest liczba, wynikiem będzie oryginalna wartość z bitem znaku ustawionym na zero.
- Jeżeli argument jest typu `Variant`, Delphi konwertuje go na liczbę zmiennoprzecinkową i zwraca jej wartość bezwzględną typu zmiennoprzecinkowego (nawet jeśli wartość `Variant` ma charakter całkowity).

| Argument | Wartość zwracana |
|---|---|
| −∞ (minus nieskończoność) | +∞ (plus nieskończoność) |
| < 0 | – liczba |
| -0.0 | +0.0 |
| +0.0 | +0.0 |
| + ∞ (plus nieskończoność) | + ∞ (plus nieskończoność) |
| Niejawna wartość nie będąca liczbą | Oryginalna wartość z bitem znaku ustawionym na zero |
| Jawna wartość nie będąca liczbą | Oryginalna wartość z bitem znaku ustawionym na zero |

### *Patrz również*

`Double`, `Extended`, `Int64`, `Integer`, `Single`

## *Absolute (dyrektywa)*

### *Składnia*

```
var Deklaracja absolute Wyrażenie stałe;
var Deklaracja absolute Zmienna;
```

### *Opis*

Dyrektywa `absolute` nakazuje Delphi zapisanie zmiennej pod określonym adresem w pamięci. Adresem może być wartość numeryczna lub nazwa zmiennej. W drugim przypadku adres jest taki sam, jaki użyty został dla *`zmiennej`*. Dyrektywa `absolute` może być stosowana w odniesieniu do zmiennych globalnych i lokalnych.

### *Wskazówki i porady*

- Odradza się stosowanie dyrektywy `absolute`, o ile nie jest to naprawdę konieczne. W zamian należy stosować rekordy wariantowe, które są mniej błędogenne oraz łatwiejsze w czytaniu i rozumieniu.
- Dyrektywę `absolute` należy stosować zamiast rekordów wariantowych, jeżeli nie można w rozsądny sposób zmienić typu zmiennej. Na przykład z dyrektywy `absolute` może skorzystać procedura, która musi zmieniać sposób interpretacji swojego argumentu.
- Dyrektywa `absolute` z adresem numerycznym jest pozostałością z Delphi 1 i nie ma żadnego rzeczywistego zastosowania w nowszych, 32-bitowych systemach operacyjnych.

### *Przykład*

Przykład użycia dyrektywy `absolute` znaleźć można w opisie typów `Extended` oraz `Double`.

### *Patrz również*

`Record`, `Var`

# Abstract (dyrektywa)

## Składnia

```
Deklaracja metody wirtualnej; virtual; abstract;
procedure NazwaProcedury; virtual; abstract;
```

## Opis

Dyrektywa `abstract` stosowana jest w połączeniu z metodami wirtualnymi oraz dynamicznymi i oznacza, że metoda nie ma swojej implementacji. Kompilator rezerwuje miejsce w tablicy metod wirtualnych lub przypisuje numer metody dynamicznej. Metoda abstrakcyjna powinna zostać zaimplementowana w klasie potomnej. Dyrektywa `abstract` umieszczana jest za dyrektywami `virtual`, `dynamic` i `override`.

## Wskazówki i porady

- Jeżeli klasa potomna nie przesłania metody abstrakcyjnej, należy ją ominąć w deklaracji klasy lub zadeklarować z użyciem dyrektyw `override` i `abstract` (w takiej kolejności). Zalecana jest druga metoda, ponieważ w jasny sposób dokumentuje intencję programisty: nieimplementowania metody. W przeciwnym wypadku osoba czytająca kod programu może mieć trudności ze zrozumieniem, dlaczego metoda nie występuje w klasie.
- Jeżeli programista próbuje skonstruować obiekt na podstawie klasy zawierającej metodę abstrakcyjną, kompilator generuje ostrzeżenie. Zazwyczaj ostrzeżenie tego typu informuje o jednym z możliwych błędów: (1) programista zapomniał zaimplementować metodę abstrakcyjną w klasie potomnej lub (2) usiłuje stworzyć egzemplarz klasy bazowej, podczas gdy powinien stworzyć egzemplarz klasy potomnej.
- W przypadku stworzenia egzemplarza klasy bazowej i odwołania się do jednej z jej metod abstrakcyjnych, Delphi wywołuje procedurę `AbstractErrorProc` lub generuje błąd czasu wykonania o numerze 210 (`EAbstractError`).

## Patrz również

`AbstractErrorProc`, `class`, `dynamic`, `override`, `virtual`

# AbstractErrorProc (zmienna)

## Składnia

```
var AbstractErrorProc: Pointer;
procedure ProceduraObslugiBledu;
begin ... end;
AbstractErrorProc := @ProceduraObslugiBledu;
```

## Opis

Kiedy wywołana zostanie metoda abstrakcyjna obiektu pochodzącego z klasy bazowej (w której metoda ta nie jest implementowana), Delphi wywołuje procedurę wskazywaną przez zmienną

`AbstractErrorProc`. Jeżeli `AbstractErrorProc` zawiera `nil`, generowany jest błąd czasu wykonania o numerze 210 (`EAbstractError`). Jeśli wskaźnik jest różny od `nil`, powinien wskazywać punkt wejścia do bezargumentowej procedury obsługi błędu.

### *Wskazówki i porady*

- Moduł `SysUtils` przypisuje zmiennej `AbstractErrorProc` procedurę generującą wyjątek `EAbstractError`, zatem w większości aplikacji ustawianie tej zmiennej nie jest wymagane. W przypadku zdefiniowania własnej procedury obsługi dla błędów metod abstrakcyjnych, należy pamiętać o wygenerowaniu wyjątku; jeżeli procedura wykona się normalnie, Delphi zatrzyma program.

### *Patrz również*

`Abstract`, `AssertErrorProc`, `ErrorProc`, `ExceptProc`, `Halt`

## *AcquireExceptionObject (funkcja)*

### *Składnia*

```
function AcquireExceptionObject: Pointer;
```

### *Opis*

W momencie gdy klauzule `try-except` kończą obsługę wyjątku, Delphi automatycznie zwalnia obiekt wyjątku. Funkcja ta interpretuje obsługę wyjątku w taki sposób, aby obiekt wyjątku został przekazany do głównego wątku aplikacji, co zapobiega jego przedwczesnemu zwolnieniu przez Delphi. `AcquireExceptionObject` zwiększa licznik referencji obiektu wyjątku, więc nie jest on automatycznie zwalniany. Kiedy aplikacja wywołuje `AcquireExceptionObject`, odpowiedzialną za zwolnienie obiektu wyjątku staje się procedura `ReleaseExceptionObject`.

```
// Funkcja przechwytuje wyjątki i zapisuje w liście RaiseListPtr^
function AcquireExceptionObject: Pointer;
begin
  if RaiseListPtr <> nil then
  begin
  // Kiedy aplikacja wygeneruje wyjątek, zapisuje jego obiekt
  // w RaiseListPtr^.
  // Następnie ustawia referencję obiektu na
  // nil w zawartości wyjątku, aby zapobiec przedwczesnemu
  // zwolnieniu obiektu przez Delphi.
    Result := PRaiseFrame(RaiseListPtr)^.ExceptObject;
    PRaiseFrame(RaiseListPtr)^.ExceptObject := nil;
  end
  else
    Result := nil;
end;
```

### *Patrz również*

`ReleaseExceptionObject`

# AddModuleUnloadProc (procedura)

## Składnia

```
procedure AddModuleUnloadProc (Proc: TModuleUnloadProc);
procedure NazwaProcedury(HInstance: THandle);
begin ... end;
AddModuleUnloadProc(NazwaProcedury);
AddModuleUnloadProc(TModuleUnloadProcLW(Proc));
```

## Opis

Delphi przechowuje listę pakietów, z których składa się aplikacja. W chwili usuwania z pamięci pakietu wywoływana jest seria procedur, z których każda otrzymuje uchwyt biblioteki DLL. Programista może dodać własną procedurę do nagłówka listy poprzez przypisanie jej adresu do `AddModuleUnloadProc`. Procedury usuwania modułów z pamięci są również wywoływane podczas wychodzenia z aplikacji.

`AddModuleUnloadProc` jest rzeczywistą procedurą.

## Przykład

```
// Serwer grafiki zarządza zasobami graficznymi.
// Kiedy aplikacja ładuje zasób graficzny, serwer sprawdza jego głębię
// kolorów i jeśli jest ona wyższa niż wyświetlana, tworzy nową kopię
// obiektu graficznego z głębią kolorów, odpowiadającą głębi
// wyświetlanej, po czym zwraca nowy obiekt graficzny. Zastosowanie
// renderowania wysokiej jakości
// daje lepsze rezultaty w porównaniu do metody dopasowywania
// kolorów stosowanej przez Windows.
// Podczas usuwania z pamięci modułu niezbędne jest zwolnienie
// wszystkich jego zasobów

type
  PResource = ^TResource;
  TResource = record
    Module: THandle;
    Resource: TGraphicsObject;
  case Boolean of
    True: (Name: PChar;);
    False: (ID: LongInt);
  end;

var
  List: TList;

procedure ByeBye(HInstance: THandle);
var
  I: Integer;
  Resource: PResource;
begin
  for I:= List.Count-1 downto 0 do
  begin
    Resource := List[I];
    if Resource.Module = HInstance then
    begin
      List.Delete(I);
```

```
        Resource.Resource.Free;
        Dispose(Resource);
      end;
    end;
  end;

  initialization
    List:= TList.Create;
    AddModuleUnloadProc(ByeBye);
  finalization
    RemoveModuleUnloadProc(ByeBye);
    FreeAndNil(List);
  end.
```

### *Patrz również*

ModuleUnloadList, TModuleUnloadProcLW, PModuleUnloadRec, RemoveModuleUnloadProc, TModuleUnloadRec, UnregisterModule

---

# *Addr (funkcja)*

### *Składnia*

```
function Addr(var X): Pointer;
Addr(Zmienna)
Addr(Procedura)
```

### *Opis*

Funkcja Addr zwraca adres zmiennej lub procedury (funkcji). Typem zwracanym jest wskaźnik ogólnego przeznaczenia — Pointer. Nawet w przypadku użycia dyrektywy kompilatora $T lub $TYPEDADDRESS, Addr zwraca zawsze wskaźnik typu Pointer.

Podobne działanie do funkcji Addr ma operator @, z tą różnicą, że może zwrócić wskaźnik określonego typu (jeżeli włączona jest dyrektywa $T lub $TYPEDADDRESS).

Funkcja Addr jest wbudowana w kompilator.

### *Patrz również*

Pointer, $T, $TYPEDADDRESS

---

# *AllocMemCount (zmienna)*

### *Składnia*

```
var AllocMemCount: Integer;
```

### *Opis*

AllocMemCount przechowuje liczbę bloków pamięci zaalokowanych przez Menedżera pamięci Delphi.

### Wskazówki i porady

- Zmienna `AllocMemCount` nie jest w żaden sposób wykorzystywana przez Delphi — ma charakter wyłącznie informacyjny. Zmiana jej wartości nie ma żadnego celu i nie przynosi żadnej szkody.
- Programista tworzący własny Menedżer pamięci powinien sprawdzić zawartość `AllocMemCount` przed wywołaniem `SetMemoryManager`. Zmienna `AllocMemCount` powinna być wyzerowana. Jeżeli jest inaczej, oznacza to, że domyślny Menedżer pamięci przydzielił przynajmniej jeden blok pamięci. W takiej sytuacji Delphi może próbować zwolnić tę pamięć przy użyciu Menedżera pamięci programisty. Jeżeli Menedżer pamięci nie jest przygotowany do rozwiązania takiej sytuacji, najbezpieczniej jest zatrzymać program.
- We własnym Menedżerze pamięci zmiennej `AllocMemCount` można przypisywać liczbę aktualnie przydzielonych bloków pamięci.
- W przypadku stosowania bibliotek DLL, `AllocMemCount` może nie odzwierciedlać bloków pamięci przydzielonych innym modułom. Korzystając z modułu `ShareMem`, można wywołać jego funkcję `GetAllocMemCount` w celu obliczenia liczby bloków pamięci przydzielonych wszystkim modułom używającym `ShareMem`.

### Patrz również

`AllocMemSize`, `Dispose`, `FreeMem`, `GetHeapStatus`, `GetMem`, `GetMemory`, `New`, `ReallocMem`, `SetMemoryManager`

---

## AllocMemSize (zmienna)

### Składnia

```
var AllocMemSize: Integer;
```

### Opis

`AllocMemSize` przechowuje całkowity rozmiar w bajtach wszystkich bloków pamięci zaalokowanych przez Menedżera pamięci Delphi. Mówiąc inaczej, `AllocMemSize` reprezentuje rozmiar pamięci dynamicznej, używanej przez aplikację.

### Wskazówki i porady

- Zmienna `AllocMemSize` nie jest używana w żaden sposób przez Delphi. Zmiana jej wartości, chociaż bezcelowa, jest nieszkodliwa.
- We własnym Menedżerze pamięci zmiennej `AllocMemSize` można przypisywać aktualny rozmiar pamięci zaalokowanej przez menedżera.
- W przypadku stosowania bibliotek DLL, `AllocMemSize` może nie odzwierciedlać bloków pamięci przydzielonych innym modułom. Korzystając z modułu `ShareMem`, można wywołać jego funkcję `GetAllocMemSize` w celu znalezienia liczby bloków przydzielonych wszystkim modułom używającym `ShareMem`.

### *Patrz również*

`AllocMemCount`, `Dispose`, `FreeMem`, `GetHeapStatus`, `GetMem`, `GetMemoryManager`, `New`, `ReallocMem`, `SetMemoryManager`

## *And (słowo kluczowe)*

### *Składnia*

*Wyrażenie boolowskie* `and` *Wyrażenie boolowskie*
*Wyrażenie wartości całkowitej* `and` *Wyrażenie wartości całkowitej*

### *Opis*

Operator `and` wykonuje iloczyn logiczny, jeżeli jego operandy są typu boolowskiego, lub iloczyn bitowy, jeżeli operatory mają wartość całkowitą. Operandy całkowite mogą być dowolnego typu całkowitego, łącznie z `Int64`. Iloczyn logiczny zwraca fałsz (`False`), jeżeli wartość któregokolwiek operandu jest fałszem, i prawdę, jeżeli oba operandy mają wartość prawdziwą (`True`).

### *Wskazówki i porady*

- W przeciwieństwie do standardowego języka Pascal, jeżeli operand po lewej stronie ma wartość fałszywą, Delphi nie oblicza wartości operandu po prawej stronie, ponieważ wynikiem całego wyrażenia logicznego jest fałsz. Programista może wymusić obliczanie wartości obu operandów przy użyciu dyrektyw kompilatora `$B` lub `$BOOlEVAL`.
- Operator `and`, działający na wartościach całkowitych, porównuje ze sobą kolejne pary bitów obu operandów i ustawia wynik na zero, jeżeli dowolny z nich jest zerem, lub jeden, jeżeli oba bity są jedynkami. Jeżeli jeden z operandów jest mniejszy od drugiego, zostaje powiększony przez dodanie zer na najbardziej znaczących bitach (z lewej strony). Wynikiem operacji jest liczba o rozmiarze większego z operandów.

### *Przykłady*

```
var
  I, J: Integer;
  S: string;
begin
  I:= $F0;
  J:= $8F;
  WriteLn(I and J); // Wypisuje 128 ($80)
  // Zasada skracania operatora AND w następnym przykładzie
  // uniemożliwia Delphi odwołanie się do nieistniejącego elementu
  // łańcucha — S[1].
  S:= '';
  if (Length(S) > 0) and (S[1] = 'X') then
    Delete(S, 1, 1);
end;
```

### *Patrz również*

`Boolean`, `ByteBool`, `LongBool`, `Not`, `Or`, `Shl`, `Shr`, `WordBool`, `Xor`, `$B`, `$BOOlEVAL`

# AnsiChar (typ)

## Składnia

```
type AnsiChar = #0..#255;
```

## Opis

Typ `AnsiChar` reprezentuje 8-bitowy rozszerzony znak ANSI. W bieżącej wersji Delphi typ `Char` jest zgodny z `AnsiChar`, jednak w kolejnych wersjach definicja tego typu może zostać zmieniona. `AnsiChar` pozostanie typem 8-bitowym, niezależnie od definicji typu `Char`.

## Patrz również

`AnsiString`, `Char`, `WideChar`

# AnsiString (typ)

## Składnia

```
type AnsiString;
```

## Opis

Typ `AnsiString` jest długim łańcuchem (ze zliczaniem referencji), zawierającym znaki typu `AnsiChar`. Domyślnie Delphi traktuje typ `string` jako synonim typu `AnsiString`. Jeżeli jednak programista użyje dyrektywy kompilatora `$H-` lub `$LONGSTRINGS`, typ `string` stanie się takim samym typem, jak `ShortString`.

`AnsiString` jest przechowywany jako wskaźnik do rekordu, ale zamiast wskazywać początek tego rekordu, wskazuje początek jednego z jego pól (o nazwie `Data`), przechowującego faktyczne dane. Zawartość łańcucha poprzedzają pola `Lenght` i `RefCount`.

```
type
  // Poniżej znajduje się logiczna struktura typu AnsiString,
  // bez związku z jego rzeczywistą implementacją.
  TAnsiString = record
    RefCount: LongWord;
    Length: LongWord;
    Data: array[1..Length+1] of AnsiChar;
  end;
```

## Wskazówki i porady

- Delphi zarządza czasem życia łańcuchów `AnsiString` poprzez zliczanie referencji. Jeżeli zajdzie taka potrzeba, licznikiem referencji można manipulować przy użyciu procedur `Initialize` i `Finalize`.
- Przypisanie łańcucha do zmiennej typu `AnsiString` powoduje skopiowanie wskaźnika do tego łańcucha i zwiększenie licznika referencji. Ponieważ Delphi stosuje semantykę kopiuj-przy-zapisie, programista może traktować nową zmienną w taki sposób, jakby posiadała

własną kopię łańcucha. Jeżeli zawartość łańcucha, którego liczba referencji jest większa niż jeden, zostanie zmieniona, Delphi automatycznie utworzy unikatową kopię łańcucha i do niej wprowadzi modyfikacje.

- Każdy łańcuch pamięta swoją długość jako oddzielną wartość typu całkowitego. Długość łańcucha może zostać ustawiona przez wywołanie `SetLength`. Delphi automatycznie wpisuje na koniec łańcucha znak `#0` (ale nie uwzględnia go w jego długości), umożliwiając tym samym zrzutowanie go na typ `PChar`, wymagany przez funkcje API Windows oraz inne wzorowane na języku C.

### *Patrz również*

`AnsiChar`, `Finalize`, `Initialize`, `Length`, `PChar`, `SetLength`, `SetString`, `ShortString`, `String`, `WideString`, `$H`, `$LONGSTRINGS`

## *AnsiToUtf8 (funkcja)*

### *Składnia*

```
function AnsiToUtf8(const S: string): UTF8String;
```

### *Opis*

Funkcja konwertuje łańcuch `S` typu `string` (zapisany w formacie ANSI) na łańcuch zgodny z typem `UTF8String`. Specyfiką kodowania UTF-8 (8-bitowy format `Unicode`) jest możliwość przesyłania kodów ANSI oraz ASCII praktycznie bez zmian. Tylko kody o numerach większych niż 127 podlegają modyfikacji. Dzięki temu np. polskie teksty nieznacznie powiększają swoją objętość.

### *Patrz również*

`AnsiChar`, `PChar`, `PUCS4Chars`, `ShortString`, `String`, `WideString`, `$H`, `$LONGSTRINGS`, `UTF8String`, `Utf8ToAnsi`, `Utf8ToUnicode`, `UTF8Decode`, `Utf8Encode`, `UnicodeToUtf8`.

## *Append (procedura)*

### *Składnia*

```
procedure Append(var F: TextFile);
```

### *Opis*

Procedura `Append` otwiera istniejący plik tekstowy do zapisu, ustawiając się na jego końcu. Wszelkie dane są dopisywane na końcu pliku.

Procedura `Append` jest wbudowana w kompilator. Przykład jej użycia znaleźć można w opisie procedury `AssignFile`.

### Błędy

- Jeżeli przed `Append` nie została wywołana procedura `AssignFile`, Delphi zgłasza błąd wejścia-wyjścia o numerze 102.
- Jeżeli z jakiegoś powodu plik nie może zostać otwarty, jako błąd wejścia-wyjścia zgłoszony zostaje kod błędu Windows.

### Wskazówki i porady

- Procedura `Append` otwiera jedynie pliki tekstowe — nie można używać jej do otwierania plików określonego typu lub innych plików binarnych. Aby dopisać dane do pliku binarnego, należy wywołać procedurę `Reset`, a następnie przejść na koniec pliku.

### Patrz również

`AssignFile`, `CloseFile`, `Eof`, `IOResult`, `Reset`, `Rewrite`, `TextFile`, `$I`, `$IOCHECKS`

---

## ArcTan (funkcja)

### Składnia

```
function ArcTan(Number: typ zmiennoprzecinkowy): Extended;
```

### Opis

Funkcja `ArcTan` zwraca arcustangens z liczby `Number` (w radianach). `ArcTan` jest funkcją wbudowaną.

### Wskazówki i porady

- Delphi automatycznie konwertuje argumenty typu `Integer` i `Variant` na typ zmiennoprzecinkowy. Aby przekonwertować argument typu `Int64` na typ zmiennoprzecinkowy, należy dodać `+0.0`, np. `125 + 0.0`.
- Jeżeli wartością `Number` jest dodania nieskończoność, rezultatem jest π/2 (lub mówiąc dokładnie, przybliżenie tej liczby); jeżeli `Number` jest ujemną nieskończonością, rezultatem jest przybliżenie liczby -π/2.
- Jeżeli `Number` jest niejawną wartością nienumeryczną, wynikiem jest `Number`.
- Jeżeli `Number` jest jawną wartością nienumeryczną, `ArcTan` zgłasza błąd czasu wykonania nr 6 (`EInvalidOp`).

### Patrz również

`Cos`, `Sin`

# *Array (słowo kluczowe)*

## *Składnia*

```
type Nazwa = array[typ indeksu] of typ bazowy;          // tablica
                                                        // statyczna
type Nazwa = array[typ indeksu, ...] of typ bazowy;     // tablica
                                                        // statyczna
type Nazwa = array of typ bazowy;                       // tablica
                                                        // dynamiczna
Nazwa: array of typ bazowy       // tablica otwarta jako parametr
                                 // funkcji/procedury
Nazwa: array of const            // wariantowa tablica otwarta jako parametr
                                 // funkcji/procedury
```

## *Opis*

Delphi posiada kilka różnych typów tablic: tablice statyczne, dynamiczne i otwarte.

- Tablica statyczna jest tradycyjną tablicą języka Pascal. Indeks tablicy musi być typu porządkowego. Tablica może posiadać wiele indeksów (wymiarów). Rozmiar tablicy statycznej nie może zostać zmieniony w trakcie wykonania programu.
- Tablica dynamiczna jest tablicą o indeksie typu `Integer`, której rozmiar może się zmieniać w trakcie wykonania programu. Dolną granicą indeksu jest zawsze zero, natomiast granica górna jest ustawiana przez procedurę `SetLength`. Aby skopiować tablicę dynamiczną, należy wywołać procedurę `Copy`. Przypisanie tablicy dynamicznej skutkuje przypisaniem jedynie referencji do tablicy bez przepisywania jej zawartości. Do zarządzania czasem życia tablic dynamicznych Delphi używa mechanizmu zliczania odwołań. W przypadku tablic dynamicznych nie jest stosowana zasada kopiuj-przy-zapisie (jak ma to miejsce dla łańcuchów).
- Parametrem procedury (funkcji) może być tablica otwarta. Procedurze można przekazać dowolną statyczną lub dynamiczną tablicę. Delphi przekazuje dodatkowy, ukryty parametr, stanowiący górną granicę tablicy. Procedura (funkcja) nie może zmienić rozmiaru tablicy dynamicznej, przekazywanej jako tablica otwarta. Niezależnie od typu indeksu rzeczywistej tablicy, parametr w postaci tablicy otwartej używa typu `Integer` dla indeksu z zerem jako dolną granicą.
- Specjalnym typem tablicy otwartej jest wariantowa tablica otwarta, deklarowana jako `array of const`. Każdy element takiej tablicy jest konwertowany na rekord `TVarRec`. Wariantowa tablica otwarta jest najczęściej stosowana w procedurach i funkcjach przyjmujących zmienną liczbę argumentów (do których należy między innymi funkcja `Format` z modułu `SysUtils`).
- Tablica elementów typu `AnsiChar`, `Char` lub `WideChar` ma specjalny charakter, kiedy indeks typu całkowitego rozpoczyna się od zera. Delphi traktuje taką tablicę jako łańcuch lub długi łańcuch (o ile nie zostanie wyłączona dyrektywa kompilatora `$EXTENDEDSYNTAX` lub `$X`) z jednym wyjątkiem — tablicy znaków nie można przekazać do procedury przez zmienną łańcuchową. Referencję tablicy można przekazać przez argument do procedury, która przyjmuje parametr typu `PChar` lub `PWideChar`. Delphi automatycznie przekazuje adres pierwszego znaku w takiej tablicy.
- Tablice przechowywane są w porządku kolumnowym — najszybciej zmieniającym się indeksem jest zawsze pierwszy indeks od prawej strony.

### Przykłady

```
...
type
  TIntArray = array of integer;
var
  Counter: Integer;
  TestInfo: string;
  Ints: TIntArray;

implementation
{$R *.dfm}

// Dołączenie komunikatu do pliku logowania.
function Log(const Fmt: string; const Args: array of
             const):string; overload;
begin
  Result:=Format(Fmt, Args);
end;
// Dołączenie losowej liczby do tablicy dynamicznej.
// Ponieważ tablice dynamiczne i tablice otwarte używają tej samej
// składni, dla parametru tablicy dynamicznej trzeba zastosować
// specjalnie nazwany typ.
procedure AppendRandomInt(var Ints: TIntArray);
begin
  SetLength(Ints, Length(Ints) +1);
  Ints[High(Ints)] := Random(MaxInt);
end;
//----------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
  var I : Integer;
begin
//   ...
   for I := 1 to 10 do
     AppendRandomInt(ints);
   Form1.Edit1.Text := Log('Test: #%d: %s', [I, 'Login'])
                       +IntToStr(Ints[High(Ints)]);
end;

end.
```

### Patrz również

Copy, High, Length, Low, PAnsiChar, PChar, PWideChar, SetLength, Slice, Type, TVarRec, $EXTENDEDSYNTAX, $X

## As (słowo kluczowe)

### Składnia

```
Referencja obiektu as typ klasowy
Obiekt lub referencja interfejsu as typ interfejsu
```

### Opis

Operator as konwertuje referencję obiektu na inny typ klasowy lub referencję interfejsu na inny typ interfejsu. Po prawej stronie operatora powinien występować typ klasowy lub typ interfejsu.

### *Wskazówki i porady*

- Jeżeli obiekt lub referencja interfejsu zawiera `nil`, wynikiem będzie również `nil`.
- Zadeklarowana klasa obiektu musi być potomkiem lub rodzicem typu klasowego. Jeżeli referencja obiektu nie posiada odpowiedniego typu, kompilator generuje błąd. Jeżeli zadeklarowany typ jest poprawny, ale rzeczywisty typ obiektu w trakcie wykonania programu nie jest typem klasowym, Delphi generuje błąd czasu wykonania o numerze 10 (`EInvalidCast`).
- Użycie operatora `as` należy ograniczać jedynie do rzutowania typów referencji obiektów. Wyjątkiem od tej reguły są sytuacje, w których typ jest znany dzięki wcześniejszemu użyciu operatora `is`.
- Jeżeli typem docelowym jest interfejs, Delphi wywołuje metodę `QueryInterface`, przekazując jej jako pierwszy argument identyfikator GUID typu interfejsu. Jeżeli obiekt nie implementuje interfejsu, Delphi generuje błąd 23 (`EIntfCastError`).

### *Przykład:*

```
// Gdy dowolne pole wyboru jest zaznaczone, uaktywnij przycisk OK.
// Poniższa procedura obsługi zdarzenia może zostać użyta dla
// kilku pól wyboru jednocześnie.
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if (Sender as TCheckBox).Checked then
   OkButton.Enabled := True;
end;
```

### *Patrz również*

`Interface`, `Is`, `TObject`

---

## *Asm (słowo kluczowe)*

### *Składnia*

```
asm
  instrukcje assemblera
end;
```

### *Opis*

Słowo kluczowe `asm` rozpoczyna blok instrukcji assemblera.

### *Wskazówki i porady*

- Blok `asm` jest wyrażeniem, które można stosować wszędzie tam, gdzie dozwolone jest użycie wyrażenia lub bloku języka Pascal (np. w procedurze lub funkcji).
- Wewnątrz bloku assemblera można odwoływać się do zmiennych, a także skakać do etykiet zadeklarowanych w innym miejscu procedury. Nie należy korzystać z instrukcji skoku w obrębie macierzystej pętli (lub do innej pętli), chyba że jest to naprawdę niezbędne. Etykieta rozpoczyna się od znaku `@`, jest lokalna względem procedury i nie musi być deklarowana.

- Wbudowany assembler środowiska Delphi nie dorównuje najnowszym technologiom dostępnym na rynku, w związku z czym programista nie może bazować na pełnym zestawie instrukcji. Instrukcje można jednak kodować ręcznie przy użyciu dyrektyw `DB`, `DW` lub `DD`.
- Blok kodu `asm` może zmienić wartości rejestrów: `EAX`, `ECX` i `EDX`, ale musi pozostawić niezmienione wartości: `EBX`, `ESI`, `EDI`, `EBP` i `ESP`. Należy przyjąć zasadę, iż żaden z rejestrów nie zawiera znaczących wartości, jednak przy zachowaniu ostrożności poprzez rejestry można uzyskać dostęp do parametrów procedury. Należy zapoznać się z dyrektywami sterującymi sposobem wywoływania procedur (`cdecl`, `pascal`, `register`, `safecall` i `stdcall`), aby dowiedzieć się, w jaki sposób argumenty przekazywane są do podprogramów.
- Pisanie bezpośrednio w języku assembler nie ma większego wpływu na wydajność programu. Najczęstszym powodem stosowania bloków `asm` jest konieczność użycia instrukcji niezaimplementowanych w Delphi, takich jak polecenie `CPUID` przedstawione w poniższym przykładzie.

## *Przykład*

```
unit cpuid;
// Identyfikacja procesora
// Moduł definiuje funkcję GetCpuID, która używa polecenia CPUID do
// pobrania typu procesora. GetCpuID zwraca prawdę, jeżeli procesor
// posiada instrukcję CPUID i fałsz w przeciwnym wypadku.
// Starsze wersje procesorów 486 oraz przodkowie tej serii nie
// posiadają instrukcji CPUID.
// Copyright © 1999 Tempest Software, Inc.
interface
const
  VendorIntel = 'GenuineIntel';
  VendorAMD   = 'AuthenticAMD';
  VendorCyrix = 'CyrixInstead';
type
  TCpuType = (cpuOriginalOEM, cpuOverdrive, cpuDual, cpuReserved);
  TCpuFeature = (cfFPU, cfVME, cfDE, cfPDE, cfTSC, cfMSR, cfMCE,
      cfCX8, cfAPIC, cfReserved10, cfReserved11, cfMTRR, cfPGE, cfMCA,
      cfCMOV, cfPAT, cfReserved17, cfReserved18, cfReserved19,
      cfReserved20, cfReserved21, cfReserved22, cfReserved23, cfMMX,
      cfFastFPU, cfReserved26, cfReserved27, cfReserved28,
      cfReserved29, cfReserved30, cfReserved31);
  TCpuFeatureSet = set of TCpuFeature;
  UInt4 = 0..15;
  TCpuId = packed record
    CpuType: TCpuType;
    Family: UInt4;
    Model: UInt4;
    Stepping: UInt4;
    Features: TCpuFeatureSet;
    Vendor: string[12];
  end;

// Pobranie informacji o CPU i zapisanie jej w Cpuid.
function GetCpuID(var Cpuid: TCpuId): Boolean;
implementation
function GetCpuID(var Cpuid: TCpuId): Boolean;
asm
  // Sprawdzenie, czy procesor obsługuje instrukcję CPUID
  // Test zmienia wartości rejestrów EXC i EDX.
  pushfd
```

```
  pop ecx             // Załadowanie EFLAGS do ECX
  mov edx, ecx        // Utworzenie kopii EFLAGS w EDX
  xor ecx, $200000    // Ustawienie znacznika ID
  push ecx            // Próba ustawienia EFLAGS
  popfd
  pushfd              // Sprawdzenie, czy zmiana została zachowana
  pop ecx             // Pobranie nowej wartości EFLAGS do ECX
  xor ecx, edx        // Porównanie z EDX
  je @NoCpuID         // Jeżeli bity są sobie równe, procesor
                      // nie posiada instrukcji CPUID
  // Rozkaz CPUID istnieje. Odtworzenie oryginalnej wartości EFLAGS
  push edx
  popfd
  // Rozkaz CPUID zniszczy zawartość EAX, dlatego argument Cpuid
  // zachowany zostanie w ESI. Delphi wymaga rejestru ESI w chwili
  // zakończenia bloku ASM, dlatego niezbędne jest zapamiętanie jego
  // oryginalnej wartości.
  // Zapamiętywana jest również wartość EBX, którą zniszczy CPUID,
  // a która musi być zachowana.
  push esi
  push ebx
  mov esi, eax
  // Pobranie nazwy producenta, będącej złączeniem zawartości
  // rejestrów EBX, EDX i EAX, traktowanych jako 4-bajtowe tablice
  // znaków.
  xor eax, eax
  dw $a20f                                  // CPUID instruction
  mov BYTE(TCpuid(esi).Vendor), 12          // string length
  mov DWORD(TCpuid(esi).Vendor+1), ebx      // string content
  mov [OFFSET(TCpuid(esi).Vendor)+5], edx
  mov [OFFSET(TCpuid(esi).Vendor)+9], ecx
  // Pobranie informacji o procesorze
  dw $a20f               // rozkaz CPUID
  mov TCpuid(esi).Features, edx
  // Sygnatura procesora dzieli się na cztery części, z których
  // większość ma długość 4 bitów. Delphi nie posiada pól bitowych,
  // dlatego rekord TCpuid
  // używa bajtów do zapisywania tych pól. Oznacza to konieczność
  // pakowania czwórek bitów (ang. nibble) w bajtach.
  mov edx, eax
  and al, $F
  mov TCpuid(esi).Stepping, al
  shr edx, 4
  mov eax, edx
  and al, $F
  mov TCpuid(esi).Model, al
  shr edx, 4
  mov eax, edx
  and al, $F
  mov TCpuid(esi).Family, al
  shr edx, 4
  mov eax, edx
  and al, $3
  mov TCpuid(esi).CpuType, al
  pop ebx             // odtworzenie rejestrów EBX i ESI
  pop esi
  mov al, 1           // Zwrócenie wartości prawdziwej
  ret
@NoCpuId:
  xor eax, eax        // Zwrócenie fałszu jako oznaki braku instrukcji
                      // CPUID
end;

end.
```

### Patrz również

CDecl, Pascal, Register, SafeCall, StdCall

---

## Assembler (dyrektywa)

### Składnia

```
Nagłówek procedury; assembler;
```

### Opis

Dyrektywa `assembler` nie ma żadnego znaczenia. Istnieje dla zachowania zgodności z Delphi 1.

### Patrz również

Asm

---

## Assert (procedura)

### Składnia

```
procedure Assert(Test : Boolean [; const Message: string]);
```

### Opis

Procedura `Assert` ma za zadanie dokumentować i wymuszać pewne założenia, jakie programista robi podczas pisania kodu. `Assert` nie jest prawdziwą procedurą. Kompilator traktuje jej wywołanie w sposób specjalny, kompilując nazwę pliku i numer wiersza, w którym została umieszczona. Działanie takie ma pomóc programiście w lokalizacji usterek programu.

Jeżeli argument `Test` jest fałszem, Delphi wywołuje procedurę wskazywaną przez zmienną `AssertErrorProc`. Moduł `SysUtils` przypisuje tej zmiennej adres procedury, która generuje wyjątek `EAssertionFailed`. Jeżeli `AssertErrorProc` zawiera `nil`, wywoływany jest błąd czasu wykonania o numerze 21 (`EAssertError`).

Programista może dołączyć komunikat, który Delphi przekaże procedurze `AssertErrorProc`. W przypadku braku tego komunikatu używany jest komunikat domyślny, np. „`Assertion failed`."

### Wskazówki i porady

- Właściwe użycie procedury `Assert` polega na wyspecyfikowaniu warunków, jakie muszą być spełnione w celu prawidłowego działania kodu. Wszystkie programy czynią pewne założenia odnośnie wewnętrznego stanu obiektu, wartości lub poprawności argumentów procedury, wartości zwracanej przez funkcję. Dobrym rozwiązaniem odnośnie założeń jest to, aby sprawdzały one błędy programistów, a nie błędy użytkowników.

- Chociaż działanie funkcji `Assert` może zostać wyłączone przy użyciu dyrektyw kompilatora `$ASSERTIONS` lub `$C`, zazwyczaj nie ma takiej potrzeby. Wyświetlenie komunikatu „assertion error" jest niepokojącym symptomem dla użytkownika, niemniej jednak o wiele lepiej będzie, jeżeli program zakończy się dziwnym komunikatem, niż gdyby miał działać dalej i na przykład uszkodzić cenne dane użytkownika.

### *Przykład*

Niniejszy podrozdział zawiera kilka przykładów użycia procedury `Assert`, występujących w opisie innych elementów języka — patrz procedura `Move`, funkcja `TypeInfo`, funkcje `VarArrayLock` i `VarIsArray`.

### *Patrz również*

`AssertErrorProc`, `$ASSERTIONS`, `$C`

## *AssertErrorProc (zmienna)*

### *Składnia*

```
var AssertErrorProc: TAssertErrorProc;

procedure AssertErrorHandler(const Message, Filename: string;
                    LineNumber: Integer; ErrorAddr: Pointer)


AssertErrorProc:= @AssertErrorHandler;
```

### *Opis*

Kiedy założenie programu nie zostanie spełnione, Delphi wywołuje procedurę o adresie przechowywanym w zmiennej `AssertErrorProc`. Kompilator przekazuje tej procedurze komunikat oraz lokalizację wyrażenia `Assert`.

### *Wskazówki i porady*

- Programista może zaimplementować w tej procedurze dowolne funkcje programu, takie jak logowanie błędu, wysyłanie poczty do osoby odpowiedzialnej za sprawne działanie aplikacji itp. W przeciwieństwie do innych procedur obsługi błędu, `AssertErrorProc` pozwala na kontynuowanie programu od wyrażenia następującego za wywołaniem procedury `Assert`.
- Jeżeli `AssertErrorProc` wskazuje `nil`, Delphi generuje błąd czasu wykonania 21 (`EAssertError`).
- Moduł `SysUtils` przypisuje zmiennej `AssertErrorProc` adres procedury generującej wyjątek `EAssertError`.

### *Patrz również*

`AbstractErrorProc`, `Assert`, `ErrorProc`, `ExceptProc`

# Assign (procedura)

## Składnia

```
procedure Assign(var F: File; const FileName: string);
procedure Assign(var F: TextFile; const FileName: string);
```

## Opis

Procedura `Assign` wykonuje takie samo zadanie, jak `AssignFile`. W nowym kodzie zalecane jest stosowanie procedury `AssignFile` — `Assign` jest nazwą metody często stosowaną w Delphi, w związku z czym łatwo może dojść do nieporozumienia w kodzie. `Assign` nie jest rzeczywistą procedurą.

## Patrz również

`AssignFile`

# Assigned (funkcja)

## Składnia

```
function Assigned(const P: Pointer): Boolean;
function Assigned(Pbj: TObject): Boolean;
function Assigned(Method: TMethod): Boolean;
```

## Opis

Funkcja `Assigned` zwraca prawdę, jeżeli argument jest różny od `nil`, lub fałsz, jeżeli argument jest pusty (równy `nil`). `Assigned` nie jest rzeczywistą funkcją.

## Wskazówki i porady

- Argument może być wskaźnikiem, referencją obiektu lub metodą.
- Stosowanie funkcji `Assigned` zamiast bezpośredniego porównania wskaźnika z `nil`, skutkuje spadkiem wydajności programu.
- Jeżeli wskaźnik jest wskaźnikiem funkcji, użycie `Assigned` daje jasno do zrozumienia kompilatorowi, że intencją programisty nie jest wywołanie funkcji i porównanie jej wyniku z `nil`. Dlatego `Assigned` jest często stosowana do testowania wskaźników funkcji i metod.
- Wskaźnik metody składa się z dwóch części: wskaźnika kodu i wskaźnika danych. `Assigned` sprawdza jedynie bardziej znaczące słowo referencji kodu: jeżeli słowo to zawiera zero, referencja metody jest równa `nil`. Wskaźnik kodu jest ignorowany przez `Assigned`.

## Przykład

```
procedure TForm1.Button1Click(Sender: TObject);
var P: Pointer;
```

```
begin
  P := nil;
  if Assigned(P) then
    Edit1.Text:='wskaźnik nie dotyczy funkcji';
  GetMem(P, 1024);
  FreeMem(P, 1024);
  if Assigned(P) then
    Edit1.Text:=' testowana funkcja GetMem()';
end;
```

### *Patrz również*

`Nil`

---

## *AssignFile (procedura)*

### *Składnia*

```
procedure AssignFile(var F: File; const FileName: string);
procedure AssignFile(var F: TextFile; const FileName: string);
```

### *Opis*

`AssignFile` przypisuje nazwę pliku do pliku o określonym lub nieokreślonym typie lub do pliku tekstowego (przed otwarciem). `AssignFile` nie jest rzeczywistą procedurą.

### *Wskazówki i porady*

- Jeżeli przed wywołaniem jednej z procedur `Append`, `Reset` lub `Rewrite` nie zostanie wywołana procedura `AssignFile`, zgłoszony zostanie błąd wejścia-wyjścia o numerze 102.
- Delphi interpretuje pusty łańcuch jako konsolę. W aplikacji typu konsolowego do konsoli przypisywane są automatycznie pliki `Input` i `Output`. Próba użycia pliku konsolowego w aplikacji z interfejsem graficznym skutkuje błędem wejścia-wyjścia o numerze 105.

### *Przykład*

```
var
  LogFile: string = 'c:\log.txt';

// Dodanie komunikatu do pliku dziennika. Przykład w opisie słowa
// kluczowego Keyword demonstruje inną przeciążoną procedurę Log.
procedure Log(const Msg: string); overload;
var
  F: TextFile;
begin
  AssignFile(F, LogFile);
  // Próba otwarcia pliku, która powiedzie się tylko w przypadku, gdy
  // plik istnieje.
{$IOCHECKS Off}
  Append(F);
{$IOCHECKS On}
  if IOResult <> 0 then
    // Plik nie istnieje, należy go zatem utworzyć.
    Rewrite(F);
  WriteLn(F, Msg);
  CloseFile(F);
end;
```