Vnitřní programovací jazyk

Tato kapitola obsahuje systematický popis vnitřního programovacího jazyka **WinBase602**.

Tento jazyk se využívá nejen pro psaní programů uložených jako samostatné objekty v aplikaci, ale také pro zápis příkazů, podmínek a výrazů tvořících součást formulářů, menu a sestav.

Charakter programovacího jazyka

Programovací jazyk **WinBase602** je strukturovaný programovací jazyk velmi podobný ***Pascalu*** Hlavní odlišnosti od ***Pascalu*** spočívají v přidání nástrojů pro komunikaci s databází a pro vytváření uživatelského interface pod ***Windows***, a ve vypuštění nemnoha rysů, které by našly jen malé uplatnění při programování databázových aplikací. Kompletní přehled odchylek od standardního ***Pascalu*** naleznete na konci této kapitoly. Standardní procedury, funkce a metody, které se dají z jazyka volat, jsou popsány podle svého účelu v ostatních kapitolách tohoto manuálu. Jejich referenční popis najdete v elektronické nápovědě.

V dalším textu budeme předpokládat, že čtenář již zná alespoň základy programování v některém vyšším strukturovaném jazyce, jako jsou například Pascal, C, ADA či některý strukturovaný Basic. Tato kapitola není učebnicí programování.

Vnitřní jazyk versus SQL

Níže uvedená pravidla platí pro vnitřní programovací jazyk. Do programů je možné vkládat také příkazy jazyka SQL, pro nějž platí pravidla poněkud odlišná. Nejvýznamnějšími rozdíly jsou v prioritě operátorů, ve jménech typů, v zápisu data a času. Jazyku SQL je věnována samostatná kapitola.

Preprocesor a direktivy

PREPROCESOR je nástroj, který zpracovává program dřív, než se provádí jeho lexikální, syntaktická a sémantická analýza, a analyzuje tzv. DIREKTIVY vložené do zdrojového programu. Direktivy umožňují překládání pouze vybraných částí programu, definování a rozvíjení maker, vkládání jednoho programu do druhého nebo snadné zařazování SQL příkazů do programu.

Přestože popis preprocesoru logicky patří na začátek této kapitoly, patří jeho využití mezi pokročilé techniky. Při prvním čtení lze tuto sekci bez obav přeskočit.

Zápis direktiv v programu

Direktivy se zapisují (jako v jazyce ***C***) na zvláštní řádku ve tvaru:

```
#jméno_direktivy parametry
```

přičemž znak # musí být na začátku řádky - v prvním sloupci.

Alternativně lze direktivy zapisovat také způsobem používaným v ***Borland Pascalu*** a ***Delphi***, a to na začátku komentáře takto:

```
{$jméno_direktivy parametry ... }
```

Pokud nejde o direktivy **`define`** nebo **`sql`**, pak případný text, následující za direktivou uvnitř komentářových závorek, se ignoruje.

Jména direktiv lze psát velkými i malými písmeny.

Podmíněný překlad

Podmíněný překlad znamená, že se některé části programu uzavřené mezi direktivami překládají pouze při splnění určitých podmínek. Podmíněný překlad se využívá tehdy, když je potřeba mít v jednom zdrojovém programu více variant. Direktivy řídící podmíněný překlad vytvářejí v programu takového struktury:

```
#if podmínka1
... sekce A ...
#elif podmínka2
... sekce B ...
#else
... sekce C ...
#endif
```

Každá sekce mezi uvedenými direktivami může být tvořena libovolným počtem řádek programu. Z těchto sekcí se přeloží pouze (nebo nejvýše) jedna. Je-li splněna *podmínka1*, přeloží se sekce A. Pokud *podmínka1* není splněna a je splněna *podmínka2*, přeloží se sekce B. Jinak se přeloží sekce C.

Direktiva **`elif`** a následující sekce B se může opakovat libovolný počet krát, přeloží se sekce následující za první splněnou podmínkou. Direktivy **`elif`** a **`else`** a za nimi následující sekce B resp. C se nemusí použít vůbec. Není-li použita direktiva **`else`**, pak není-li splněna žádná podmínka, nepřeloží se žádná sekce. Konstrukce podmíněného překladu lze do sebe libovolně zanořovat bez omezení hloubky.

Podmínka smí mít jeden ze tří tvarů:

- *celé číslo* - podmínka je splněna, pokud má nenulovou hodnotu;
- *řetězec znaků* - podmínka je splněna, pokud je neprázdný;
- *dvojice celých čísel nebo dvojice řetězců spojena operátorem = nebo <>* - podmínka je splněna, pokud obě hodnoty jsou si rovny resp. nerovny. Při porovnávání řetězců se nepřihlíží k velikosti písmen.

V podmínce se mohou vyskytovat také identifikátory maker definované direktivou **define**.

Příklad**:**

```
//#define STATUS "DEMO"
#define STATUS "VYVOJ"

#if STATUS = "VYVOJ"
  Info_box("Stav programu", "Vývojová verze");
#elif STATUS = "DEMO"
  Info_box("Stav programu", "Demo verze");
#else
  Info_box("Stav programu", "Provozní verze");
#endif
```

ifdef, ifndef

Kromě direktivy **if** lze pro zahájení podmíněného překladu použít také direktivy:
`#ifdef` *ident* - podmínka je splněna, pokud je *ident* definován direktivou **define**,
`#ifndef` *ident* - podmínka je splněna, pokud *ident* není definován direktivou **define**.

Příklad:

```
Procedure Výpočet;
begin
#ifdef LADĚNÍ
  Info_box("Ladění", "Začátek procedury Výpočet");
#endif
.....
```

Řádek obsahující volání procedury **Info_box** se přeloží pouze tehdy, pokud je dříve v programu uvedena direktiva

```
#define LADĚNÍ
```

Definice maker

Pomocí direktivy **define** lze definovat makra, tedy specifikovat, že se v programu má určitý identifikátor nahrazovat jinými symboly. Direktiva ve tvaru:

```
#define ident rozvoj_makra
```

způsobí, že od tohoto místa v programu se všechny výskyty identifikátoru *ident* nahradí *rozvojem makra*. Rozvoj makra je posloupnost libovolných symbolů.

Makro s parametry

Definovat lze také makro s parametry ve tvaru:

```
#define ident(par1, par2, ...)  rozvoj_makra
```

V tomto případě se makro volá s parametry takto:

```
ident(sp1, sp2, ...)
```

a každé tatové volání se v programu nahradí rozvojem makra, v němž se každý výskyt *par1* nahradí *sp1*, výskyt *par2* nahradí *sp2* atd. Při volání makra je nutno uvést stejný počet parametrů, jako v definici makra.

V makrech se nerozlišuje mezi velkými a malými písmeny. Makra bez parametrů se používají zejména pří řízení podmíněného překladu. Pro označování konstantních hodnot je vhodnější používat místo nich deklarace konstant. Makra s parametry zjednodušují zápis opakujících se programových konstrukcí.

Předefinování a zrušení makra

Definici makra lze kdykoli změnit tak, že se použije nová direktiva **`define`** pro stejný identifikátor. Definici lze zrušit pomocí direktivy:

```
#undef ident
```

Příklad:

Budiž definováno makro:

```
#define CHYBA(text) begin Info_box("Chyba!", text);  halt end
```

Pokud v textu programu použijete zápis:

```
if i<0 then CHYBA("Nepřípustná vstupní hodnota");
```

přeloží se jako zápis:

```
if i<0 then begin
 Info_box("Nastala chyba", "Nepřípustná vstupní hodnota");  halt
end;
```

Předdefinovaná makra

Ve vnitřním programovacím jazyce jsou předdefinována tato makra:

`_WINBASE602_` - hodnotou je řetězec obsahující číslo verze klienta **WinBase602**, v současné verzi je to "6.0". Ve dřívějších verzích není makro použitelné, nelze tedy rozlišovat verze nižší než 6.0! V budoucích verzích však bude možné pomocí této direktivy překládat program pro různé verze **WinBase602**.

`_LANGUAGE_` - hodnotou je řetězec znaků obsahující název zvoleného jazyka (jak jste si zvolili); pokud aplikace není vícejazyčná nebo pokud není zvolen žádný jazyk, pak je to prázdný řetězec

Příklad:

```
#if _LANGUAGE_ = "english"
       PřepočetNaPalce;
#elif _LANGUAGE_ = "čeština"
       PřepočetNaMetry;
#endif
```

Vložení programu do programu

Direktiva **`include`** umožňuje, aby se na určité místo programu vložil text uložený v databázi v samostatném objektu. Direktiva se zapisuje:

```
#include jméno_programu
```

nebo zkráceně:

```
#i jméno_programu
```

Příklad:

Nechť program MZDY má hlavní část pojmenovanou MZDY, deklarace proměnných má v části DEKL_PROM a deklarace podprogramů v části DEKL_PODPROG. Pak do hlavní části na místo, kam se zbylé dvě části mají při překladu vložit (nejspíše mezi deklarace typů a začátek těla programu) zapište:

```
#include DEKL_PROM
#include DEKL_PODPROG
```

nebo

```
{$I DEKL_PROM}
{$I DEKL_PODPROG}
```

Vložení příkazů SQL do programu

Jednořádkové příkazy

Pro vložení jednořádkového SQL příkazu do programu slouží direktiva **`sql`**. Zapisuje se takto:

```
#sql příkazy v jazyce SQL
```

Příklad:

```
#sql CALL ZrusitReservaci(:<limitRez)
```

nebo místo

```
if YesNo_box("Varování",syesno) then begin
   sdel := "DELETE FROM Registr WHERE cislo="+Int2str(cisadr);
   if SQL_execute(sdel) then Signalize;
...
```

lze psát

```
if YesNo_box("Varování",syesno) then begin
#sql DELETE FROM Registr WHERE cislo=:<cisadr
Signalize;
...
```

Víceřádkové příkazy

Pokud chcete zadat příkazy na více řádkách, použijte direktivy **`sqlbegin`** a **`sqlend`** takto:

```
#sqlbegin
příkazy SQL
......
#sqlend
```

Takto zadané příkazy SQL se provedou stejně, jako kdyby byly uvedeny jako parametr funkce **`SQL_execute`**. Zadáte-li více příkazů v jedné direktivě, všechny se provedou v jedné transakci. Výsledek provedení příkazů lze zjistit z funkce **`Sz_error`**. Informaci o případné chybě při jejich provádění lze zobrazit v okně pomocí funkce **`Signalize`**.

Příklad:

```
#sqlbegin
 BEGIN ATOMIC
   DECLARE i INT;
```

```
   SET i = 1;
   WHILE i <= 10 DO
    INSERT INTO Tab3 (cislo) VALUES (i);
    SET i = i + 1;
   END WHILE;
 END
#sqlend
```

Syntaktické vybarvování textu v direktivě `sql` a mezi direktivami **`sqlbegin`** a **`sqlend`** respektuje klíčová slova jazyka SQL.

Lexikální symboly jazyka

Na nejnižší syntaktické úrovni jazyka jsou tzv. LEXIKÁLNÍ ELEMENTY. Jsou vytvořeny ze znaků. Symboly lze rozdělit do sedmi druhů: na KLÍČOVÁ SLOVA, IDENTIFIKÁTORY, ČÍSLA, ZNAKOVÉ KONSTANTY, ŘETĚZCE ZNAKŮ, SPECIÁLNÍ SYMBOLY a ODDĚLOVAČE.

Klíčová slova

KLÍČOVÁ SLOVA jsou symboly s předem daným významem, který nemůže být v programu nijak změněn. Klíčovými slovy jsou např. **`BEGIN`**, **`CASE`**, **`TYPE`** apod. Klíčová slova lze psát jak malými tak i velkými písmeny resp. oba druhy písmen kombinovat. Velikost písmen nemá na význam vliv. Editor, v němž se programy píší, klíčová slova rozpozná a zvýrazní je modře.

Pro jazyk SQL existuje odlišná množina klíčových slov.

Celkový seznam vyhrazených slov je rozsáhý – naleznete ho v elektronické nápovědě.

Identifikátory

IDENTIFIKÁTORY se tvoří z písmen, číslic a znaku _ (podtržítko), přičemž musí začínat písmenem. *Identifikátory se musí lišit od klíčových slov*. Identifikátory jsou například: **`ALFA`**, **`I`**, **`Cena_zboží`**, **`NováCena`**, **`ODDĚLENÍ_56`**. V identifikátorech se nerozlišuje mezi velkými a malými písmeny.

Identifikátory s cizími znaky

Mimo to lze tvořit identifikátory z libovolných jiných znaků kromě obráceného apostrofu a psát je pak v programu uzavřené mezi dvojicí obrácených apostrofů. Tento druh identifikátorů se při běžném programování příliš nepoužívá, slouží zejména k odkazování na jména tabulek, dotazů a sloupců, s nimiž program pracuje a v nichž se smějí vyskytovat libovolné znaky. Identifikátory jsou tedy i zápisy: **`‘Seznam zaměstnanců‘`**, **`‘Platy > 12000‘`**, **`‘2. jméno‘`** apod.

Pozor !

V identifikátorech se bere v úvahu pouze prvních 14 znaků. Proto dva identifikátory, které se shodují v prvních 14 znacích, budou považovány za stejné.

Předdefinované identifikátory

Předdefinované identifikátory, které označují standardní objekty (procedury, funkce, konstanty) můžete v programu definovat v jiném významu. Tím ale zastíníte význam původní. Pokud například definujete proměnnou `Day`, nebudete moci v rozsahu její platnosti volat standardní funkci **`Day`**. Předdefinované identifikátory jsou v editoru barevně odlišeny.

Zápis čísel

Celá čísla

CELÁ ČÍSLA se zapisují jako posloupnosti číslic, přičemž před zápornými čísly se píše znak - (minus). Celá čísla musí ležet v rozsahu **-2147483647** až **2147483647**. Všechna celá čísla jsou hodnotami typu **Integer**.

Do typu **Short** patří pouze celá čísla z rozsahu **-32767** až **32767**.

Reálná čísla

REÁLNÁ ČÍSLA mohou obsahovat kromě celé části také desetinnou část za desetinnou tečkou (nikoli čárkou!) nebo desítkový exponent, případně obojí. Před desítkovým exponentem se uvede písmeno E a případně i znaménko + resp. -. Správně zapsaná reálná čísla jsou tedy např. **1.0, -538.045, 0.02, 56.871E-12, 7e5**. Reálná čísla jsou hodnotami typu **Real**.

Peníze

Čísla vyjadřující PENĚŽNÍ OBNOSY čili hodnoty typu **Money** zapisujeme se znakem `$` mezi celou a desetinnou částí, tedy např. **`120$`** nebo **`120$50`**. Kompilátor pak explicitně pozná, že se jedná o hodnotu typu **Money** a nikoli typu **Integer** nebo **Real**.

Zápis data a času

Zápis data

Datum se v programu zapisuje ve formátu:

`den.měsíc.rok` nebo *`den.měsíc.`*

Správně zapsané datum je tedy např. `27.4.1962` nebo `27.4.` . Pokud je *rok* vynechán, pak se míní běžný rok. Zápis roku se nesmí zkracovat, rok 99 není totéž co 1999. Celý zápis musí být na jedné řádce, uvnitř zápisu se nesmějí vyskytnout žádné mezery.

Zápis času

ČAS, tedy konstanta typu **Time**, se zapisuje stejně jako při vkládání hodnot do databáze prostřednictvím formuláře. Čas se tedy zapisuje ve formátu:

`hodiny:minuty:sekundy.tisíciny_sekundy` nebo

`hodiny:minuty:sekundy` nebo *`hodiny:minuty`*

Správným zápisem času je např. `17:48:56.908` nebo `17:48:56` nebo `17:48` . Celý zápis musí být na jedné řádce, uvnitř zápisu se nesmějí vyskytnout žádné mezery.

Pozor na záměnu proměnné typu **Time** se příliš úsporně zapsaným podmíněným výrazem. Zápis jako:

```
n := n>0?n*10:0;
```

kompilátor pochopí jako násobení číselné proměnné s proměnnou typu **Time** a ohlásí chybu. Aby tento podmíněný výraz byl správně vyhodnocen, musíte za dvojtečku vložit mezeru.

Timestamp

TIMESTAMP se zapisuje stejně jako při vkládání hodnot do databáze prostřednictvím formuláře. Zapisuje se ve formátu:

den.měsíc.rok hodiny:*minuty*:*sekundy* nebo
den.měsíc.rok hodiny:*minuty*

Zápisem timestampu je tedy např. **`27.4.1962 17:48:56`** nebo **`27.4.1962 17:48`**.

Znakové konstanty a řetězce znaků

Jednotlivé ZNAKY (literály typu **Char**) se v textu programu zapisují v apostrofech.

Příklad:

Znak A zapíšete jako: `'A'`.

Znak ' (apostrof) zapíšete jako 4 apostrofy za sebou: `''''`.

ŘETĚZCE ZNAKŮ (literály typu **String**) se v programu zapisují v apostrofech nebo uvozovkách. Uvnitř řetězce smějí být libovolné znaky. Pokud mají být apostrof nebo uvozovka obsaženy v řetězci, musí se napsat zdvojeně. Pokud řetězec znaků obsahuje jediný znak, pak musí být omezen uvozovkami, aby se dal odlišit od znakového literálu.

Příklady:

```
"To je řetězec"
'Reader''s Digest'
"A"
""
```

ASCII kódy

Do řetězce znaků lze zapisovat znaky také pomocí jejich ASCII-kódu. Potřebujete-li mezi znaky '>>' a '<<' v řetězci zapsat znaky s kódy 14 a 232 (dekadicky), napíšete řetězec takto:

```
'>>'#14#232'<<'
```

Takto můžete vložit libovolný počet speciálních znaků na libovolné místo řetězce s výjimkou jeho začátku. Na začátku řetězce musí být apostrof nebo uvozovka, proto, chcete-li začít speciálním znakem, předřaďte mu prázdný řetězec např. takto:

```
''#14'Tento řetězec začíná znakem, jehož kód je 14'
```

Jedno z možných využití je rozdělení řetězce v **`Info_boxu`** do více řádků:

```
Info_box('','první řádek'#10'druhý řádek');
```

nebo

```
str := date2str(today, 1)+''#10+time2str(Now, 1);
Info_box('Datum a čas', str);
```

Spojení řádek

Pokud se řetězec nevejde na jednu řádku, je možno ho na konci této řádky ukončit apostrofem nebo uvozovkou a na začátku další řádky apostrofem nebo uvozovkou pokračovat. Překladač obě takové části řetězce spojí.

```
'Tento řetězec zde '
'nekončí, nýbrž pokračuje.'
```

Při spojování obou částí řetězců překladač mezi ně **nevkládá** mezeru, takže je třeba ji uvést na konci prvního řetězce nebo na začátku druhého.

Zejména při vytváření SQL příkazů obsahujících řetězce může dojít k hromadění uvozovek (apostrofů), např. přiřazení

```
kod :="AA-99";
podm := "select * from KNIHY where KOD="""+kod+""""";
```

do proměnné `podm` přiřadí hodnotu

```
select * from KNIHY where kod="AA-99"
```

Binární konstanty

Konstantní hodnota typu **Binary** se zapisuje v hexadecimální notaci v apostrofech předcházených znakem X, tedy ve tvaru:

X´hexadecimální zápis´

Příklad: X´52A50F8C´ je zápis binární hodnoty délky 4 bajty.

Speciální symboly

Speciálními symboly jsou jednotlivé znaky a tyto dvojice znaků:

```
:=      !!      <=      <>      >=      @@      ##
.=      .=.     ..
```

Mimo to existují dvojznaky, které lze použít jako ekvivalenty některých speciálních symbolů nebo klíčových slov. Zde uvedeme, které to jsou, a více se o nich nebudeme zmiňovat.

Tyto ekvivalenty mají za cíl usnadnit život programátorům zvyklým na jazyk **C**:

!=	je ekvivalentní	**<>**
==	je ekvivalentní	**=**
||	je ekvivalentní	**OR**
&&	je ekvivalentní	**AND**
!	je ekvivalentní	**NOT**
/*	je ekvivalentní	**{**
***/**	je ekvivalentní	**}**

Další ekvivalenty vyžaduje jazyk SQL:

!<	je ekvivalentní	**>=**
!>	je ekvivalentní	**<=**

Oddělovače

Jediným významem oddělovačů je to, že od sebe oddělují ostatní symboly. Oddělovači jsou mezery, hranice řádek a komentáře.

Komentáře

Komentáře začínají znakem **{** nebo **/*** a končí znakem **}** nebo ***/**. Uvnitř komentáře smějí být libovolné znaky kromě znaku **}**. Komentář smí také začínat dvojznakem **//** a pak končí na konci řádky.

Pravidlo:

Mezi kterékoliv dva symboly v programu lze vložit libovolný počet libovolných oddělovačů. Dovnitř symbolů se nesmějí vkládat žádné oddělovače. Nejméně jedním oddělovačem je nutno navzájem oddělit identifikátory a čísla, pokud spolu sousedí.

Konstanty a jejich deklarace

Standardní konstanty

Konstanty označující hodnotu NULL

Sloupec v záznamu z tabulky může mít hodnotu NULL, která se ve formulářích a sestavách vypisuje jako prázdný řetězec. Proto pro označení této hodnoty je ve vnitřním programovacím jazyce pro každý typ zvláštní konstanta:

NONECHAR	NONEBOOLEAN	NONESHORT
NONEINTEGER	NONEREAL	NONETIME
NONEDATE	NONETIMESTAMP	NONEPTR
NONEMONEY		

Pro typy **String, Csstring** a **Csistring** (řetězce znaků) je hodnota NULL totožná s prázdným řetězcem, a proto ji lze zapsat (známým způsobem) jako dva apostrofy nebo dvoje uvozovky za sebou bez mezery.

Příklad

Je-li tedy např. RR sloupec typu **Real**, lze uvnitř příkazu WITH zpřístupňujícího vhodný záznam psát:

```
IF rr=NONEREAL THEN ...
```

Zvláštností vnitřního programovacího jazyka je, že konstanty `FALSE` a `TRUE` se smějí psát také jako `NE` a `ANO`.

Řada konstant specifických pro práci s databází je popsána v *Encyklopedii funkcí.*

Deklarace konstant

Deklarace konstant začínají vždy klíčovým slovem **CONST.**

Deklarace mají tento obecný tvar:

```
identifikátor_konstanty = konstantní_hodnota;
```

Konstantní_hodnotou smí být číslo (celé, reálné, peněžní částka, datum, čas) nebo znak. Konstanty typu celé číslo lze například použít pro označování zpráv, které programu předávají formuláře a menu.

Příklad:

```
CONST
 Žádost_skončit = 1001;
 Minimální_mzda = 2600$;
 Oddělovací_znak = ',';
 Konec_staré_daně = 31.12.1992;
 Pi = 3.1415926;
```

Jazyk neumožňuje definovat konstanty typů **`String`**, **`CSString`** ani **`CSIString`**. Místo nich lze použít makra.

Typy a jejich deklarace

Jednoduché typy

Množina jednoduchých typů jazyka odpovídá typům použitelným v tabulce.

Typ	Poznámky
Short	Celočíselný typ s rozsahem hodnot –32767 až 32767. Hodnota zabírá 2 bajty.
Integer	Celočíselný typ s rozsahem hodnot –2147483647 až 2147483647. Hodnota zabírá 4 bajty.
Money	Typ **Money** má hodnoty se **dvěma** desetinnými místy. Jeho rozsah hodnot je od $-1.4*10^{12}$ do $1.4*10^{12}$. Hodnota zabírá 6 bajtů.
Real	Typ **Real** zahrnuje reálná čísla v rozsahu absolutních hodnot přibližně $1.7*10^{-308}$ až $1.7*10^{-308}$, kladná i záporná. Přesnost výpočtů je 15-16 desetinných míst. Hodnota zabírá 8 bajtů paměti.
Char	Hodnotami typu **Char** jsou znaky. Hodnota typu **Char** zabírá 1 bajt.
Boolean	Typ **Boolean** má pouze dvě hodnoty, a to **True** a **False** (nepočítáme-li "nedefinovanou" hodnotu NONEBOOLEAN). Hodnota zabírá jeden bajt.
Date	Hodnotou typu **Date** je datum. Tato hodnota zabírá 4 bajty. Datum musí být po začátku našeho letopočtu.

Time	Hodnotou typu **Time** je čas. Tato hodnota zabírá 4 bajty. Čas se měří s přesností na tisíciny sekundy.
Timestamp	Hodnotou typu **Timestamp** je datum a čas. Tato hodnota zabírá 4 bajty. Čas udává s přesností na sekundy.
trecnum	Typ **trecnum** slouží pro absolutní čísla záznamů a je totožný s typem **Integer**
tobjnum	Typ **tobjnum** slouží pro uložení čísla objektu a je totožný s typem **Short**
tcateg	Typ **tcateg** slouží pro uložení kategorie objektu a je totožný s typem **Short**
window_id	Typ **window_id** se používá pro tzv. *handle* oken a je totožný s typem **Short** ve Windows 3.1 resp. s typem **Integer** ve Win32.

S ukazateli z databáze lze pracovat prostřednictvím typu **Integer**, neboť ukazatele se automaticky konvertují na celá čísla a naopak. Číselná hodnota ukazatele se rovná absolutnímu číslu záznamu, na nějž ukazatel ukazuje.

Strukturované typy

Strukturovanými typy proměnných ve **WinBase602** jsou ŘETĚZEC ZNAKŮ, typ POLE, typ ZÁZNAM a typ SOUBOR.

Typy řetězec znaků

Pro ŘETĚZCE ZNAKŮ jsou k dispozici 3 typy: **String**, **CSString** a **CSIString** . Hodnotami těchto typů jsou řetězce znaků až do určité délky. Popisy typu řetězec znaků vypadají takto:

```
String[délka]
CSString[délka]
CSIString[délka]
```

Odlišnosti jednotlivých typů

Řetězce těchto tří typů se neliší svými hodnotami, ale pouze tím, jak se mezi sebou porovnávají.

Nejdelší řetězec, který se dá zapsat do proměnné takového typu, má délku stejnou jako *délka* uvedená v hranatých závorkách. Každý řetězec má na svém konci omezovací znak s hodnotou 0. Pro tento znak se při deklaraci řetězce rezervuje místo automaticky. Pokud budete řetězec indexovat a pracovat s jeho jednotlivými znaky, index prvního znaku je 1. Poslední index je o 1 větší než zadaná *délka*, ale do znaku s tímto indexem není dovoleno zapsat nic jiného než omezovací znak, tedy hodnotu `Chr(0)`.

Typ pole

Ze složek libovolného typu (kromě typu **File**) lze vytvořit typ POLE popisem:

```
ARRAY [dolní_mez .. horní_mez] OF typ_složek;
```

kde *dolní mez* a *horní mez* jsou celá čísla a *dolní mez* není větší než *horní mez*.

Vícerozměrná pole lze deklarovat tak, že vytvoříte pole polí, např.:

```
ARRAY [dolní_mez1 .. horní_mez1] OF
   ARRAY [dolní_mez2 .. horní_mez2] OF typ_složek;
```

Složky typu Char

Typ pole, jehož dolní mezí je 1 a jehož složky jsou typu **Char**, je kompatibilní s typem **String**. Pamatujte však na to, že při deklaraci pole se na jeho konec automaticky **nepřidá** znak pro omezovač řetězce. Proto chcete-li s polem 10 znaků pracovat stejně jako s řetězcem deklarovaným jako **`String[10]`**, musíte jej deklarovat jako **`ARRAY[1..11] OF Char.`**

Pozor !

Pokud chcete s polem znaků pracovat jako s celkem, tedy např. jej celý vypsat na obrazovku, zapsat do databáze nebo předložit jako parametr kterékoli proceduře, pak za posledním platným znakem řetězce musí být znak s kódem 0. Tuto vlastnost mají řetězce přečtené z databáze resp. z formuláře a je nutno ji zachovat operacemi, které budete nad polem resp. řetězcem znaků provádět.

Řetězce ve **WinBase602** jsou obdobou (až na indexaci od 1) řetězců zvaných `PChar` implementovaných v modulu `Strings` v ***Borland Pascalu*** nebo ***Delphi***, nikoli klasických řetězců používaných od nejstarších verzí ***Turbo Pascalu***. V nulté složce řetězce není uložena délka. Nultá složka řetězce ve **WinBase602** neexistuje.

Typ záznam

Typ ZÁZNAM lze definovat popisem

```
RECORD skupina ... skupina END;
```

kde *skupina* má tuto strukturu

```
identifikátor, ... , identifikátor : typ;
```

V typech záznam, které slouží k popisu struktury proměnného kurzoru, lze *skupinu* zapsat také v jednom z tvaru:

```
identifikátor, ... , identifikátor : MULTI typ;
identifikátor, ... , identifikátor : FLEXIBLE;
identifikátor, ... , identifikátor : MULTI FLEXIBLE;
```

Slovo `FLEXIBLE` označuje (kterýkoli) typ proměnné velikosti, slovo `MULTI` označuje multiatribut. Takto deklarované složky se vztahují výlučně k databázovým sloupcům a slouží k popsání struktury databázového záznamu proměnného kurzoru. Deklarujete-li proměnnou typu záznam, pak s jejími složkami, v jejichž popisu je slovo `FLEXIBLE` nebo `MULTI`, nelze pracovat.

Příklad:

```
TYPE rec = RECORD
       datum : Date;
       obs : ARRAY[1..28] OF Boolean;
       kdo : ARRAY[1..28] OF String[16];
       co : ARRAY[1..28] OF String[30];
     END;
```

Typ soubor

Pro práci s *textovými soubory* uloženými mimo databázi pod správou operačního systému slouží typ SOUBOR. Tento typ je pouze jediný a označuje se klíčovým slovem `FILE`. Jiné než textové soubory nejsou implementovány. Pro manipulaci se soubory slouží procedury a funkce **`Reset`**, **`Rewrite`**, **`Close`**, **`Seek`**, **`Eof`**, **`Filelength`**, **`Read`**, **`Write`** a **`Writeln`**.

Příklad:

Funkce, která zkontroluje existenci souboru

```
function IsThere(fil : string[90]) : boolean;
{************************************}
var
  f : file;
begin
  IsThere := false;
  if Reset(f,fil) then begin
    Close(f);
    IsThere := true;
  end;;
end;
```

Poznámka:

Potřebujete-li pracovat s jinými než textovými soubory, deklaruje a používejte ve svém programu souborové funkce z ***Windows*** API, např. **`CreateFile, ReadFile, WriteFile, CloseHandle`** atd.

Typ ukazatel

Typ UKAZATEL se definuje popisem

```
^ typ
```

kde *typ* je ten typ, na nějž ukazatel ukazuje.

Typ ukazatel se používá k práci s dynamicky alokovanými proměnnými. Nemá žádný vztah k databázovým soupcům typu ukazatel a není s nimi kompatibilní.

Nespecifikovaný typ

Ve **WinBase602** existuje zvláštní předdefinovaný tzv. NESPECIFIKOVANÝ typ označený identifikátorem **Untyped**. Slouží k operacím s objekty, jejichž typ není při překladu programu znám.

Deklarujete-li proměnnou nespecifikovaného typu (říkáme ji také **beztypová proměnná**), pak může nabývat hodnot všech nestrukturovaných typů a typů řetězec znaků. Beztypová proměnná je například parametrem funkce **C_max**, která hledá v databázových záznamech maximální hodnotu sloupce a vrací ji, aniž by v programu musel být znám její typ.

Příklad:

```
table
  Zaměstnan;
var
  u : untyped;
  str : string[100];
  rec : trecnum;
begin

// zjištění maximálního platu
  if not C_Max(Zaměstnan,'PLAT','',u) then
  begin
// zjištění čísla záznamu s max. platem
    rec := Look_up(Zaměstnan,'PLAT',u);
    if rec > -1 then begin
      str := Zaměstnan[rec].jméno+" "+Zaměstnan[rec].příjmení;
      str := "Nejvyšší plat má "+str;
      Info_box("",str);
    end;
  end;
end.
```

Deklarace typů

Výše uvedené typy není nutno deklarovat. Deklarace vlastních typů začíná klíčovým slovem TYPE a pokračuje deklaracemi ve tvaru:

```
identifikátor_typu = popis_typu;
```

Časté použití je např. při deklarování proměnných typových kurzorů.

Příklad:

Deklarace typového kurzoru

```
type
  ctyp = record
    autorx : csistring[10];
    den : date;
    _od,_do : time;
```

```
    popis : csistring[30];
    verejne : boolean;
  end;
var
  zapiscur : cursor ctyp;
```

Proměnné a jejich deklarace

Deklarace proměnných začínají klíčovým slovem VAR a mají tento tvar:

```
identifikátor, ... , identifikátor : typ;
```

Příklad:

Příklad deklarací proměnných:

```
var
  i, j, m : integer;
  r : real;
  jméno : array [1..20] of char;
  soubor : file;
  unt : untyped;
```

Proměnná typu kurzor

Speciálním případem je deklarace proměnné typu KURZOR neboli tzv. PROMĚNNÉHO KURZORU**.** Proměnná typu kurzor slouží k práci s databázovými záznamy, které budou vybrány způsobem specifikovaným až při běhu programu. Použití proměnného kurzoru je detailně popsáno v části *Přístup k záznamům v databázi.*

Deklarace proměnného kurzoru má tvar buď:

```
VAR identifikátor : CURSOR;
```

nebo

```
VAR identifikátor : CURSOR popis_typu_záznam;
```

Při použití prvního tvaru deklarujete BEZTYPOVÝ KURZOR, o němž při překladu ještě není známo, jaké sloupce budou v jeho záznamech. Typy sloupců se vyhodnocují a typové konverze provádějí až při běhu programu. Běh je proto poněkud pomalejší, ale použití kurzoru je maximálně flexibilní. V druhém případě definujte KURZOR TYPOVÝ.

Popis typu záznam v deklaraci proměnného kurzoru

Uvedete-li v deklaraci proměnné typu kurzor *popis_typu_záznam*, pak tím sdělíte překladači, jaké sloupce bude obsahovat odpověď na dotaz zpřístupněná tímto kurzorem. **WinBase602** předpokládá, že tato informace se zakládá na pravdě a nijak ji neověřuje.

Uvedený typ záznam musí obsahovat stejný počet složek jako dotaz, který položíte při otevírání tohoto kurzoru, a typy složek si musí po řadě odpovídat. Pokud je některým typem **String, CSString, CSIString** nebo **Binary**, pak i jeho délka se musí shodovat v definici tabulky, z níž sloupec pochází, a v typu záznam.

Pokud se v dotazu vyskytuje také sloupec s hodnotou proměnné velikosti, pak v příslušejícím typu záznamu uvedete místo něj typ `FLEXIBLE`. Před typem označujícím multiatribut uvedete slovo `MULTI`.

Uvědomte si, že těmito pravidly je omezeno použití proměnného kurzoru. Tím, že k jeho deklaraci připojíte typ, podstatně zúžíte množinu dotazů, jejichž odpověď může kurzor nést. Pokud v části `SELECT` položeného dotazu uvedete sloupce jiných typů nebo v jiném počtu, než specifikuje typ záznam, obdržíte nesmyslné výsledky, a může dojít i k pádu aplikace.

Typ záznam uvedený v deklaraci kurzoru určuje jména, kterými budete v programu odkazovat na sloupce záznamů v odpovědi (viz dále). Tato jména překryjí jména zadaná v klauzuli `SELECT` v dotazu.

Proměnná typu formulář

Proměnná typu formulář se deklaruje:

```
VAR identifikátor_formuláře : FORM jméno_návrhu_formuláře;
```

Její využití je popsáno v kapitole o programování uživatelského rozhraní aplikace.

Dynamicky alokované proměnné

Deklarujete-li proměnnou typu ukazatel, pak můžete pomocí funkce **`New`** dynamicky alokovat proměnnou doménového typu. Takovou proměnnou lze později dealokovat funkcí **`Dispose`**. Dynamicky alokované proměnné nezaniknou po skončení programu, nýbrž až při zavření projektu.

Deklarace tabulek a pevných kurzorů

Deklarace tabulky

Deklarace tabulek se skládá z klíčového slova TABLE, jednoho nebo více identifikátorů tabulek oddělených čárkami, a je ukončena středníkem. Identifikátor každé tabulky musí byt totožný se jménem, pod nímž je tabulka definována v databázi.

Příklad:

```
TABLE Personal, Práce, 'Mzdové listy';
```

Deklarovat je nutno ty tabulky, jejichž identifikátory hodláte v programu používat. Není třeba deklarovat tabulky, jejichž jména se vyskytnou pouze v řetězcích, např. v příkazech SELECT při otevírání proměnných kurzorů.

Deklarace pevného kurzoru

Pevný kurzor je nástrojem, který umožňuje programu manipulovat s obsahem odpovědi na dotaz *uložený v databázi*. Pomocí pevného kurzoru program položí tento dotaz a poté pracuje s odpovědí na něj stejným způsobem jako s tabulkou.

Deklarace pevných kurzorů se skládá z klíčového slova CURSOR, z jednoho nebo více identifikátorů kurzorů oddělených čárkami, a je ukončena středníkem. Každý identifikátor kurzoru musí být totožný se jménem, pod nimž je dotaz uložen v databázi.

Příklad:

```
CURSOR Důchodci, Mat_dovol, 'Plat > 12000';
```

Deklarace procedury a funkce

Deklarace procedury má tvar:

```
PROCEDURE identifikátor (skupina_par ... skupina_par); blok;
```

Deklarace funkce má tvar:

```
FUNCTION identifikátor (skupina_par ... skupina_par) : typ;
blok;
```

Část deklarace procedury resp. funkce uvedena před *blokem* se nazývá HLAVIČKA procedury resp. funkce.

Závorky a parametry v nich deklarované mohou být z hlavičky vypuštěny - pak jde o proceduru resp. o funkci bez parametrů.

Příklad:

funkce **Rand**, která vrací pseudonáhodná čísla mezi 0 a 1.

```
FUNCTION Rand : Real;
CONST
    c1 = 13849;
    c2 = 27181;
    c3 = 65536;
BEGIN
    Seed := (c1+(c2*seed)) MOD c3;
    Rand := Seed/c3;
END;
```

Konstrukce *skupina_par* deklaruje skupinu parametrů stejného typu a vypadá takto:

identifikátor, ... ,identifikátor : *typ*;

nebo

VAR *identifikátor, ... ,identifikátor* : *typ*;

V prvním případě jsou parametry předávány hodnotou, v druhém případě se předávají odkazem, předává se tedy adresa hodnoty parametru.

Předáváte-li referencí parametr typu řetězec, pak můžete uvést jako *typ* zkrácený zápis `STRING` bez udání délky. Funkce pak bude akceptovat jako skutečný parametr řetězcovou proměnnou libovolné délky. Programátor pak ručí za to, že ve funkci nepřiřadí do takto předaného řetězce hodnotu s délkou přesahující jeho meze.

Blok tvořící součást deklarace procedury nebo funkce má stejnou strukturu jako celý program se dvěma výjimkami:

- nemůže v sobě obsahovat deklarace dalších procedur, funkcí, tabulek ani pevných kurzorů;
- není ukončen tečkou.

Typ, uvedený na konci hlavičky funkce, je typem jejího výsledku. Nesmí to být strukturovaný typ.

Deklarace podprogramů nelze ve **WinBase602** do sebe vnořovat, uvnitř podprogramu nemůže být tedy deklarován jiný podprogram.

Proměnnou typu `CURSOR` (i ostatní strukturované typy vyjma řetězců) lze předávat do podprogramu pouze odkazem (referencí), nikoli hodnotou. Databázové sloupce lze do podprogramu předávat pouze hodnotou, nikoli odkazem.

Předsunutá deklarace procedur a funkcí

V některých programech je účelné volat podprogram před jeho deklarací. Nezbytné je to při použití *nepřímé rekurze*. V takových případech je nutno před voláním funkce umístit tzv. *předsunutou deklaraci* a později podprogram deklarovat se *zkrácenou hlavičkou*.

Předsunutá deklarace procedury a funkce má tvar:

```
PROCEDURE  identifikátor (skupina_par ... skupina_par);
FORWARD;
FUNCTION  identifikátor (skupina_par ... skupina_par) : typ;
FORWARD;
```

Deklarace se zkrácenou hlavičkou vypadá takto:

```
PROCEDURE  identifikátor;   blok;
FUNCTION  identifikátor;  blok;
```

Deklarace externích procedur a funkcí

Vnitřní programovací jazyk **WinBase602** umožňuje volat i procedury a funkce implementované v externích knihovnách typu DLL.

Lze volat pouze ty procedury a funkce, které vyhovují volací konvenci *stdcall* (v jazyce ***C*** musí být označeny klíčovým slovem `_stdcall`, v ***Borland Pascalu*** a ***Delphi*** `stdcall`) a musí mít tzv. *C-linkage*, tedy musí být exportovány bez informace o typech parametrů (v jazyce ***C++*** musí být označeny `extern "C"`).

Všechny takovéto procedury a funkce musí být v programu před místem svého prvního použití deklarovány. Deklarace má jeden z těchto dvou tvarů:

```
hlavička procedury nebo funkce;
EXTERNAL 'jméno_knihovny'

hlavička procedury nebo funkce;
EXTERNAL 'jméno_knihovny' NAME 'jméno_funkce_v_knihovně';
```

Pokud je jméno funkce v programu a v knihovně stejné, použijete první, zkrácený tvar, jsou-li různé, použijete druhý tvar. V praxi je nutno jméno funkce v knihovně uvádět pouze tehdy, pokud je delší než 14 znaků nebo pokud ve svém programu již máte stejně pojmenovaný objekt.

Pokud jméno dynamické knihovny uvedete bez přípony, předpokládá se přípona DLL. Upozorňujeme, že některé knihovny mají přípony jiné než DLL (např. OCX).

Deklarace nejpoužívanějších funkcí z ***Windows*** API jsou obsaženy v souboru WINAPI32.PGM. Tento soubor naleznete po instalaci **WinBase602** ve stejném adresáři, jako její programy, a můžete jej importovat do své aplikace a poté includovat do programu.

Příklad:

Některé příklady deklarací:

```
FUNCTION GetDlgItem (Handle : window_id; ItemID : Integer)
   : window_id;
EXTERNAL "USER32.DLL";
```

```
FUNCTION IsDirectory (var FName:String) : Boolean;
EXTERNAL "WINSYS.DLL";

FUNCTION CreateBrush (Color : Integer) : Short;
EXTERNAL "GDI32.DLL" NAME "CreateSolidBrush";
```

Předání proměnné do externí funkce bez typové kontroly

Deklarujete-li některý formální parametr externí funkce zkráceným zápisem

```
VAR jméno
```

bez uvedení dvojtečky a typu, pak jako skutečný parametr budeme moci předat proměnnou libovolného typu.

Volání externích funkcí s parametry typu záznam

Deklarování těch externích funkcí, jejichž parametrem je struktura (proměnná typu záznam), vyžaduje přesné a bezchybné popsání této struktury ve vnitřním jazyce.

Pro každou strukturu, použitou jako parametr externí funkce, deklarujete ve vnitřním jazyce odpovídající typ záznam. Přitom jednotlivé složky záznamu musí mít odpovídající typy podle této tabulky:

typ v externí funkci	**typ ve vnitřním jazyce**
32-bitový celočíselný typ (int, long v C, Integer, Cardinal, LongInt v Pascalu)	Integer
16-bitový celočíselný typ (short v C, SmallInt, Word v Pascalu)	Short
8-bitový typ (Byte, ShortInt v Pascalu, char)	Char
64-bitový reálný typ (double)	Real
Řetězec znaků ukončený nulou, délky N včetně omezující nuly (char * v C, PChar v Pascalu)	String[N-1]

Ve vnitřním jazyce neexistují ekvivalenty pro jiné než 8-bajtové reálné typy a pro historický pascalský řetězec znaků obsahující délku v prvním bajtu.

Při deklarování externí funkce pamatujte na to, že strukturované parametry se zpravidla předávají odkazem, a nezapomeňte na klíčové slovo VAR v deklaraci těchto parametrů.

Příklad:

Chcete-li volat funkci **`GetWindowRect`** patřící do ***Windows*** API, deklarujete:

```
TYPE rect = RECORD left, top, right, bottom : Integer END;
PROCEDURE GetWindowRect(id : window_id;  VAR r : rect);
EXTERNAL "USER32.DLL";
```

Struktura programu

Program ve vnitřním jazyce **WinBase602** se skládá z deklarační části a z těla.

Deklarační část je posloupností deklarací konstant, typů, proměnných, tabulek, kurzorů, procedur a funkcí uvedených v libovolném pořadí. Tělo je složeným příkazem (viz dále) a je ukončeno tečkou. Za touto tečkou již nesmí nic následovat.

Rozsahy platnosti identifikátorů

Identifikátory označují v programu konstanty, typy, proměnné, tabulky, pohledy, kurzory, procedury a funkce, parametry, složky záznamů. Každý identifikátor musí být deklarován předtím, než je poprvé použit.

Deklarace identifikátorů

Identifikátory se deklarují v deklaračních částech programu, procedur a funkcí. Přitom tabulky, pevné kurzory, procedury a funkce se smějí deklarovat pouze v deklarační části programu. Parametr je deklarován svým výskytem v hlavičce procedury nebo funkce. Složka záznamu je deklarována svým výskytem v deklaraci příslušného typu záznam.

Platnost deklarace

Je-li identifikátor deklarován v deklarační části programu, pak tato deklarace platí až do konce programu (s níže uvedenou výjimkou). Je-li identifikátor deklarován v proceduře nebo funkci, pak platí pouze uvnitř této procedury resp. funkce. V jedné deklarační části nesmí být dvakrát deklarován stejný identifikátor, identifikátory deklarované v proceduře (funkci) nesmí být stejné jako parametry této procedury (funkce). To však neplatí pro identifikátory složek záznamů, které se musí lišit pouze navzájem mezi sebou v rámci každého typu záznam.

Je-li v proceduře (funkci) deklarován identifikátor stejný jako identifikátor deklarovaný v programu, pak tato nová deklarace překryje uvnitř této procedury (funkce) původní deklaraci.

Přístup k proměnným

Potřebujete-li v programu pracovat s proměnnou, napíšete v příslušném kontextu prostě její identifikátor. Složitější případ nastává, pokud chcete pracovat s některou složkou strukturované proměnné.

Vyznačení prvku proměnné typu pole

K proměnné typu pole nebo řetězec znaků lze připojit index ve tvaru

```
[ výraz ]
```

kde *výraz* musí být celočíselného typu a jeho hodnota musí ležet v mezích daných deklarací pole (pro typy **String** v mezích 1 až délka řetězce plus 1).

Jsou-li tedy například pole a řetězec znaků deklarovány:

```
VAR
  P : ARRAY[1..10] OF Integer;
  T : String[20];
```

pak zápisy `P[1]`, `P[2]`, ..., `P[10]` označují složky pole, `T[1]`, ..., `T[20]` složky řetězce - v prvním případě čísla typu **Integer**, v druhém znaky. Složka `T[21]` smí obsahovat pouze omezovací znak řetězce.

Index lze také použít k výběru jednoho znaku z databázového slupce typu řetězec znaků. Opačný postup, tj. zápis znaku do sloupce typu řetězec pomocí indexu, možný není.

Příklad:

Do složky dvourozměrného pole s indexem 1,1 se zapíše hodnota:

```
pole[1][1] := ...
```

Pozor !

Překročení mezí pole je jednou z nejběžnějších a nejzhoubnějších programátorských chyb. Například pokus zapisovat hodnotu do složky `P[0]` nebo `P[11]` výše deklarovaného pole vede k přepsání paměti a v krajním případě i ke zhroucení aplikace.

Vyznačení složky proměnné typu záznam

K proměnné typu záznam lze připojit specifikaci složky ve tvaru:

.identifikátor_složky

Příklad

V záznamu deklarovaném:

```
VAR rec : RECORD
  p : Str;
  a, b : Integer;
  r : Real
 END;
```

označují zápisy `REC.P`, `REC.A`, `REC.B`, `REC.R` jednotlivé složky záznamu.

Přechod přes ukazatel

Od proměnné typu ukazatel lze přejít k objektu, na nějž ukazuje, připojením znaku ^.

Příklad

Je-li deklarován typ ukazatel TP a proměnná P tohoto typu:

```
TYPE TP = ^rec;
VAR P : TP;
```

pak zápisy jako `P^.A`, `P^.B` zpřístupňují složky záznamu, na nějž P ukazuje.

Výrazy a operátory

Ve výrazech lze používat proměnné, konstanty, čísla, data, hodnoty času, znaky a řetězce znaků, volání funkcí a speciální symboly s hodnotou definovanou kontextem jako #, ##, @, @@.

Precedence operátorů

Operátory v pořadí od nejvyšší priority jsou:

1. NOT
2. *, /, AND, DIV, MOD
3. +, - (unární i binární), OR
4. <, >, =, <=, >=, <>, .=, .=., ~

Při vyhodnocování výrazu je pořadí operací dáno v první řadě strukturou výrazu - části výrazu uzavřené v závorkách se vyhodnocují předem. Pak rozhodují výše uvedené priority operátorů. Operace se stejnou prioritou se vyhodnocují zleva doprava.

Na priority operátorů je nutno myslet při konstrukci výrazů. Předpokládejme, že **I** je typu **Integer**. Pak výraz:

```
I >= 10 AND I < 300
```

je chybný, protože operátor AND má vyšší prioritu než relace < a >=. Tuto podmínku je proto nutno zapsat ve tvaru:

```
(I >= 10) AND (I < 300)
```

Při vyhodnocování logických podmínek (tj. výrazů typu **Boolean**) se postupuje jen tak dlouho, dokud není zřejmé, jakou má výraz hodnotu. Např. ve výrazu:

```
cond AND fnc(I)
```

se funkce **`fnc`** vůbec nezavolá, pokud proměnná **`cond`** má hodnotu FALSE. Toto lze s výhodou využít v podmínkách cyklu jako:

```
WHILE NOT konec AND Get_message(msg) DO ...
```

Na beztypové proměnné nelze použít žádné operátory. Pokud chcete provést určitou operaci s hodnotou beztypové proměnné, musíte ji napřed přiřadit do proměnné vhodného typu.

Aritmetické operace

K dispozici jsou operace `+`, `-`, `*`, `/`, `DIV` a `MOD`. Operace `DIV` a `MOD` musí mít oba argumenty celočíselného typu.

Ve výrazech lze libovolně kombinovat typy **Short** a **Integer**. Všechny operace nad těmito typy se provádějí jako pro typ **Integer** a mají výsledek typu **Integer**, s výjimkou operace dělení /, v níž je podíl vždy typu **Real.**

Je-li jedním argumentem typ **Real** je hodnota výrazu typu **Real**.

Pro typ **Money** lze používat stejné operace jako pro celočíselné typy, avšak s jednou výjimkou: nelze mezi sebou násobit ani dělit operacemi `DIV` ani `MOD` dvě hodnoty typu **Money**. Typ **Money** lze kombinovat ve výrazech s typy **Integer**, **Short** a **Real**. Výsledek operace s typem **Real** je typu **Real**, v ostatních případech je typu **Money**.

Nad celočíselnými typy lze (ve vnitřním jazyce, nikoliv v SQL) provádět i *bitové operace* sjednocení, průniku a negace pomocí operátorů `OR`, `AND`, `NOT`. Platí např.

```
10 OR 6 = 14, 10 AND 6 = 2, NOT 6 = -7.
```

Operace s datem

K hodnotě typu **Date** lze přičíst (resp. odečíst) celé číslo - interpretuje se to jako přičtení (resp. odečtení) **dnů**.

Dvě hodnoty typu **Date** lze od sebe odečíst. Rozdílem je počet dnů mezi oběma daty (tedy číslo typu **Integer**).

Při operacích nad daty se berou v úvahu přestupné roky, ale nepočítá se s reformou kalendáře, která proběhla v roce 1582. Operace s datem tedy probíhají správně vzhledem ke gregoriánskému kalendáři.

Operace s časem

K hodnotě typu **Time** lze přičíst (resp. odečíst) celé číslo - interpretuje se to jako přičtení (resp. odečtení) *tisícin sekundy*. Při překročení hranice dne není výsledek operace definován.

Dvě hodnoty typu **Time** lze od sebe odečíst. Rozdílem je počet tisícin sekundy mezi oběma časy (tedy číslo typu **Integer**). Předpokládá se, že oba časy patří do téhož dne, takže výsledek může být záporný.

Operace s timestampem

K hodnotě typu **Timestamp** lze přičíst (resp. odečíst) celé číslo - interpretuje se to jako přičtení (resp. odečtení) *sekundy*. Například přičtením konstanty 86400 se zvětší hodnota proměnné o jeden den.

Dvě hodnoty typu **Time** lze od sebe odečíst. Rozdílem je počet sekund mezi oběma údaji (tedy číslo typu **Integer**).

Počítejte s omezeným rozsahem typu Timestamp – rok 1990 až cca 2030.

Operace s řetězci

Řetězce se dají spojovat pomocí operátoru +. Výsledný řetězec však bude mít maximální délku 255 znaků. Pokud je součet délek spojovaných řetězců větší, bude jeho konec přesahující délku 255 znaků odříznut. Pro spojování delších řetězců použijte funkci jazyka **`StrInsert`**.

Řetězce se dají porovnávat v řadě relací uvedených níže.

POZOR:

Standardním procedurám a funkcím se parametr typu řetězec znaků předává zásadně odkazem. Proto nelze jako skutečný parametr použít výraz obsahující dva řetězce spojené operátorem +. Místo toho je nutno oba řetězce napřed spojit do třetího řetězce a ten pak předat jako parametr.

Příklad:

Nechť v tabulce TAB je sloupec MUŽ typu **Boolean** a DAT_NAR typu **Date**:

```
with TAB[1] do
   str := (MUŽ ? 'Narozen ' : 'Narozena ') +
        date2str(DAT_NAR,1);
Info_box('informace', str);
```

Relace

Operátory vyjadřující relace jsou <, >, =, <=, >=, <>, `.=`, `.=.`, `~`. Poslední tři je možno použít pouze na znakové řetězce.

Výsledkem relace je hodnota typu **Boolean**. V relaci lze použít operandy stejných typů, dále lze mezi sebou libovolně kombinovat typy **Short, Integer**, **Money** a **Real** a také různé typy řetězců navzájem.

Při porovnávání řetězců různého typu se postupuje tak, že způsob porovnávání určuje ten z nich, který má vyšší pořadí v tomto výčtu:

1. **String**
2. **CSString**
3. **CSIString**

Speciální řetězcové relace mají tento význam:

- `x .= y` řetězec y je prefixem řetězce x;
- `x .=. y` řetězec y je obsažen v řetězci x;
- `x ~ y` řetězce x a y se navzájem liší nanejvýš mezerami, diakritikou nebo velikostí písmen.

Podmíněný výraz

Vnitřní jazyk **WinBase602** obsahuje také tzv. podmíněný výraz převzatý z jazyka C. Jde o přibližnou obdobu podmíněného příkazu.

Podmíněný výraz má tento tvar:

```
výraz_1 ? výraz_2 : výraz_3
```

kde *výraz_1* je nepodmíněný výraz typu **Boolean** a *výraz_2* je nepodmíněný výraz stejného typu jako *výraz_3*. Přitom *výraz_3* může být i podmíněný.

Při vyhodnocování podmíněného výrazu se nejprve vyhodnotí *výraz_1*. Pokud má hodnotu TRUE, vyhodnotí se dále *výraz_2*, jinak se vyhodnotí *výraz_3*. Ten z nich, který se vyhodnotil, udává zároveň hodnotu celého podmíněného výrazu.

Příklad:

Absolutní hodnotu X lze s pomocí podmíněného výrazu zapsat takto:

```
X >= 0 ? X : -X
```

V hodnotové složce formuláře nebo sestavy může stát:

```
druh=1?"odeslané":druh=2?"došlé":"neznámý původ"
```

Příkazy

Přiřazovací příkaz

Přiřazovací příkaz má tvar:

```
proměnná := výraz
```

Při provádění přiřazovacího příkazu se nejprve zjistí, jaká proměnná je na jeho levé straně, pak se spočte hodnota výrazu na pravé straně a nakonec se hodnota výrazu přiřadí proměnné.

Konverze typu při přiřazení

Pokud typy výrazu a proměnné v přiřazovacím příkazu nejsou shodné, překladač v některých případech sám zařídí potřebnou konverzi typu tak, aby přiřazení bylo možno provést.

Bez omezení se provedou konverze mezi typy **Integer**, **Short** a **Money**[1]. Všechny tři tyto typy se také konvertují na typ **Real**. Hodnota typu **Real** se automaticky konvertuje na typ **Money**. Konverzi typu **Real** na typ **Integer** nebo **Short** musíte ale zařídit sami, například pomocí funkcí **Round** nebo **Trunc**.

Automatická konverze se dále provádí mezi databázovým typem ukazatel nebo obousměrný ukazatel a typem **Integer**. Konverzí se převádí ukazatel na záznam na absolutní číslo záznamu v tabulce a naopak.

[1] Při konverzi typu **Money** na celočíselný typ se odřízne jeho desetinná část. Nezaokrouhluje se.

Přímo přiřazovat lze také všechny typy řetězec znaků a pole znaků. Pokud však přiřazujete obsah pole znaků, je nezbytné, aby za posledním znakem v poli byl omezovací znak s hodnotou nula (`Chr(0)`). Lze navzájem přiřazovat řetězec znaků a znak. Ze znaku vzniká přiřazením do řetězcové proměnné řetězec délky jedna, při přiřazení řetězce do znaku se přenese pouze první znak.

Lze také přiřadit řetězec znaků do sloupce proměnné velikosti (např. typu **Text**) a naopak.

Pro jiné typy žádná automatická konverze neprobíhá. Je proto nutno buď příslušný převod naprogramovat nebo využít některou ze standardních konverzích procedur (jejich seznam je na konci kapitoly, detailní popis v elektronické *Encyklopedii funkcí*).

Beztypové proměnné

Pokud při přiřazení je na jedné straně beztypová proměnná, pak na druhé straně musí být hodnota resp. proměnná nestrukturovaného typu nebo typu řetězec znaků, případně také beztypová proměnná. Při přiřazení se provede vhodná konverze typu. Pokud se konverze nedá provést (např. konverze řetězce 'ABC' na typ **Integer**), pak se přiřadí hodnota NULL.

Složený příkaz BEGIN ... END

Složený příkaz sdružuje posloupnost po sobě následujících příkazů do jediného příkazu. Má tento tvar:

```
BEGIN příkaz; ... ; příkaz END
```

Provedení složeného příkazu znamená postupné provedení příkazů v něm obsažených.

Podmíněný příkaz IF

Podmíněný příkaz má tvar:

```
IF výraz THEN příkaz1 ELSE příkaz2
```

nebo jen:

```
IF výraz THEN příkaz1
```

Při jeho provádění se nejprve vyhodnotí *výraz*, jenž musí být typu **Boolean**. Pokud má hodnotu `TRUE`, provede se *příkaz1*, pokud má hodnotu `FALSE`, provede se *příkaz2* (pokud existuje, jinak se neprovede nic).

Cyklus WHILE

Cyklus **`WHILE`** má tento tvar:

```
WHILE výraz DO příkaz
```

Výraz musí být typu **Boolean**. Dokud má tento výraz hodnotu `TRUE`, opakuje se provádění *příkazu* uvedeného za **`DO`**. Protože se hodnota výrazu počítá před provedením příkazu, nemusí se tento příkaz provést ani jednou.

Cyklus REPEAT

Cyklus **REPEAT** má tento tvar:

```
REPEAT příkaz; ... ; příkaz UNTIL výraz
```

Výraz musí být typu **Boolean**. Příkazy uvedené mezi **REPEAT** a **UNTIL** se opakují tak dlouho, dokud výraz nenabude hodnoty TRUE. Protože se hodnota výrazu počítá až po provedení příkazů, provedou se všechny příkazy nejméně jednou.

Cyklus FOR

Cyklus **FOR** má tvar:

```
FOR identifikátor := mez1 TO mez2 DO příkaz
```

nebo

```
FOR identifikátor := mez1 DOWNTO mez2 DO příkaz
```

Identifikátor musí označovat proměnnou typu **Integer** nebo **Short** (nesmí to být jméno databázového sloupce), *mez1* a *mez2* musí být výrazy téhož typu.

Jde opět o cyklus, v němž se opakuje provádění příkazu uvedeného za **DO**. Předtím se však vypočte hodnota výrazu *mez1* a přiřadí se proměnné, jejíž *identifikátor* je uveden za **FOR**. V prvním případě se po každém zopakování příkazu zvětší hodnota této proměnné o 1, vypočte se hodnota *mez2* a porovnají se. Pokud proměnná přesáhne hodnotu *mez2*, cyklus končí. V druhém případě se hodnota proměnné po každém opakování zmenšuje o 1 a končí se, jakmile klesne pod *mez2*. Příkaz se neprovede ani jednou, pokud je *mez1* > *mez2* v cyklu s **TO**, resp. *mez1* < *mez2 v* cyklu s **DOWNTO**.

Na rozdíl od standardního ***Pascalu*** se hodnota výrazu *mez2* počítá v každém oběhu cyklu.

Příkaz CASE

Příkaz **CASE** slouží k rozvětvení programu do mnoha větví na základě celočíselné hodnoty. Má tento tvar:

```
CASE výraz OF
konstanta, ... konstanta : příkaz;
konstanta, ... konstanta : příkaz;
 ...
ELSE : příkaz;
END
```

Větev začínající slovem **ELSE** se může vyskytnout na libovolném místě mezi ostatními větvemi, ale nejvýše jednou. Její výskyt není povinný. Každá konstanta smí být v příkazu **CASE** použita nejvýše jednou.

Při provádění příkazu **CASE** se nejprve vyhodnotí *výraz*. Jeho hodnota se porovná s uvedenými konstantami. Pokud se některá konstanta rovná hodnotě výrazu, pak se pro-

vede *příkaz* následující na touto konstantou. Pokud žádná konstanta nemá stejnou hodnotu jako výraz, pak má-li příkaz větev **ELSE**, provede se příkaz v této větvi, pokud větev **ELSE** schází, neprovede se nic.

Mezi konstantami v příkazu **CASE** lze uvádět jak čísla tak i identifikátory deklarované jako celočíselné konstanty.

Volání procedury

Volání procedury má tento tvar:

```
identifikátor (parametr , ... , parametr)
```

Identifikátor označuje volanou proceduru. Parametry uvedené v tomto příkazu odpovídají po řadě parametrům uvedeným v deklaraci procedury. Hodnotou lze proceduře předat libovolný výraz vyhovujícího typu (automaticky se provádějí stejné konverze jako v přiřazovacím příkazu, viz výše). Odkazem lze předat pouze proměnnou stejného typu, jaký má parametr. Výjimkou je předávání polí: při předávání pole odkazem postačí, když skutečný a formální parametr mají stejný typ složek a stejnou dolní mez.

Jako skutečný parametr předávaný odkazem lze uvést i řetězec znaků nebo volání funkce vracející řetězec znaků. Toto uvolnění pravidel standardního Pascalu zjednodušuje programování, ale jeho využití je podmíněno opatrností - procedura nesmí hodnotu takového parametru změnit.

Pokud procedura nemá žádné parametry, pak se v jejím volání neobjeví závorky ani hodnoty skutečných parametrů.

Jako proceduru lze zavolat i kteroukoli standardní funkci. V takovém případě se její hodnota ignoruje.

Příklad:

Standardní funkci **Read** lze volat jako proceduru:

```
Read(FI, I);
```

Příkaz WITH

Příkaz **WITH** má ve **WinBase602** pozměněný význam oproti ***Pascal*u**. Umožňuje specifikovat, z kterého záznamu které tabulky nebo kurzoru z databáze budou pocházet sloupce, jejichž jména se vyskytnou v jeho vnitřním příkaze. Syntaxe příkazu **WITH** je:

WITH *tabulka_nebo_kurzor*[*celočíselný výraz*] **DO** *vnitřní příkaz*

Například pokud tabulka MTAB má sloupec X, pak místo:

```
IF NOT mtab[i].deleted THEN
   mtab[i].x:=17.5;
```

lze psát:

```
WITH mtab[i] DO
  IF NOT deleted THEN x:=17.5;
```

Příkaz **WITH** se nedá použít k práci s sloupci proměnného kurzoru, jehož struktura není popsána v jeho deklaraci. Příkaz **WITH** se také nedá použít ke zpřístupnění složek *proměnné* typu záznam.

Příkaz RETURN

Příkaz **RETURN** slouží k zaslání zprávy programu z menu nebo formuláře. Používá se výhradně jako akce vyvolávaná stiskem tlačítka formuláře nebo výběrem položky menu. Syntaxe příkazu **RETURN** je:

RETURN *celočíselný_výraz*

Při provádění příkazu **RETURN** se vyhodnotí *výraz* a zašle jako zpráva programu. Program ve vnitřním jazyce obdrží tuto zprávu prostřednictvím volání funkce **Get_ext_message**. Program v externím jazyce ji obdrží jako parametr zprávy WM_COMMAND zaslané jeho hlavnímu oknu.

Hodnota *výrazu* v příkazu **RETURN** musí být buď mezi 1001 a 60000 nebo rovna -1.

Příkaz HALT

Příkaz **HALT** slouží k okamžitému ukončení běhu programu. Používá se zejména po zjištění chyb, které neumožňují pokračovat v běhu programu.

Příklad:

```
Err_mask(TRUE);
table[recnum].sl := 123;
IF Sz_error <> 0 THEN
BEGIN
    Info_box('Chyba', 'Nelze zapisovat do databáze');
    HALT
END;
```

Hlavní odlišnosti od jazyka Pascal

Přehled hlavních konstrukcí standardního *Pascal*u, které ve WinBase602 nejsou implementovány (standardem se rozumí norma ISO 7185 úroveň 0):

- hlavička programu;
- definice a použití návěští;
- konstanty typu řetězec znaků;
- typy interval, množina, výčtové typy, variantní typy záznam;
- jiné než textové soubory;
- typy pole indexované jinak než celými čísly;
- vnořování procedur a funkcí;

- parametry typu procedura nebo funkce;
- příkaz `FOR` s jinými než celočíselnými mezemi;
- příkaz `GOTO`;
- přístup ke složkám *proměnné* typu záznam pomocí příkazu `WITH`.

Přehled nových syntaktických konstrukcí jazyka WinBase602:

- beztypové proměnné;
- deklarace `TABLE, CURSOR`;
- proměnné typu kurzor a formulář;
- přístup k obsahu databáze;
- symbol # vyjadřující počet složek multiatributu nebo délku hodnoty sloupce proměnné velikosti;
- symboly @ a @@ označující číslo záznamu ve formuláři;
- symbol ## označující číslo běžné stránky v sestavě;
- symbol !! označující formulář, z něhož je podprogram zavolán;
- symboly .=, .=. a ~ vyjadřující relace mezi řetězci.