Obsah

Předmluva

Otevíráte knížku, která je prvním členem řady učebnic programování a která vznikla jako reakce na současný nedostatek kvalitních učebnic z tohoto oboru. Podíváte-li se v současné době po odborné literatuře z oblasti výpočetní techniky, najdete řady příruček k nejrůznějším produktům, ale s výjimkou vysokoškolských skript se prakticky nesetkáte s učebnicí programování, která by se vás snažila dovést od naprostých začátků až k profesionálnímu přístupu. To platilo v době, kdy vyšlo první vydání této knihy, a platí to beze změny dodnes. Tato situace byla také jedním z hlavních důvodů, proč redakce časopisu Computerworld otevřela kurs programování Cesta k profesionalitě, z nějž vychází tato učebnice a převážně i následující díly.

Žádná z našich knih však není pouhým souhrnem jednotlivých pokračování zmíněného seriálu. Od doby, kdy tento seriál vycházel, se mnohé změnilo, a to nejen pokud jde o hardware (dnes se to zdá neuvěřitelné, ale v časopisecké verzi byly ještě aktuální poznámky pro programátory, kteří neměli ve svém počítači pevný disk), ale i pokud jde o oba probírané programovací jazyky. Navíc vzhledem k náhlému ukončení kursu ze strany redakce Computerworldu neobsahovala časopisecká verze všechnu zamýšlenou látku. Proto bylo nutno řadu míst upravit a doplnit. Přitom jsme využili i některých publikovaných i nepublikovaných částí seriálu *Kurs C/C++*, který vycházel v časopisu Bajt. Samozřejmě jsme také odstranili chyby, na které jsme přišli nebo na které nás čtenáři a známí upozornili.

V našich knihách jsme si položili za cíl seznámit čtenáře s moderními technikami a technologiemi programování a s možnostmi jejich použití v jazycích C++ a Pascal. Všechny jsou zaměřeny na práci na osobních počítačích kompatibilních se standardem IBM PC. Programy, které budete vytvářet v této knize, jsou určeny pro operační systém MS DOS.

Při psaní těchto knih jsme předpokládali, že čtenářem bude průměrný středoškolák bez jakýchkoli předběžných znalostí programování. Pro studium první z nich – tedy u té, kterou právě čtete – stačí, znáte-li základy práce s operačním systémem, tj. víte-li, co je to soubor, umíte-li soubory kopírovat, přejmenovávat a mazat, víte-li, co je to adresář, umíte-li vytvořit nový adresář a nastavit jej jako aktuální. V dalších dílech pak předpokládáme znalost látky na úrovni dílů předchozích.

Tato kniha i navazující díly jsou určeny hlavně budoucím programátorům, kteří potřebují mocný a efektivní jazyk pro vývoj a následnou údržbu svých programů. Jazyk, který jim vloží do rukou silné vývojové prostředky, ale zároveň je také bude chránit před řadou vlastních chyb. Proto jsme za profilový jazyk kursu zvolili jazyk C++, o němž si myslíme, že je v současné době pro profesionální programování nejvhodnější.

Paralelně s jazykem C++ budeme vykládat i jazyk Pascal, protože i tento jazyk je dnes často používán k vývoji profesionálních aplikací. Navíc se tento jazyk často používá k výuce informatiky na středních školách. Proto si myslíme, že pro skutečného profesionála je nezbytná alespoň povrchní znalost obou těchto jazyků.

Vedle toho se domníváme, že velká řada programátorů jak v Čechách, tak i na Slovensku byla odchována jazykem Pascal a někteří z nich mají zájem přejít na perspektivní C++. Jsme přesvědčeni, že možnost porovnat řešení příkladů ve známém jazyku s řešeními v jazyku, který se chtějí naučit, přispěje jak k rychlejšímu osvojení nového jazyka, tak k jeho hlubšímu zvládnutí.

Součástí této knihy je i doprovodná disketa, na níž najdete vedle zdrojových textů probíraných příkladů také pomůcky nezbytné pro odladění a spuštění příkladů.

Probírané příklady jsme odladili s použitím překladačů Borland C++ 3.1 a Turbo Pascal 7.0, tedy nejrozšířenějších dosovských verzí překladačů obou jazyků.

Co najdete v této knize

Kniha *Základy algoritmizace* je určena naprostým začátečníkům v programování, kteří však již zvládli základní znalosti práce s PC pod operačním systémem DOS. Všechny ostatní informace nutné pro návrh, vývoj, kódování, ladění, testování a provoz programů se vám budeme snažit v této knížce poskytnout.

Cílem naší učebnice je seznámit vás se základními algoritmickými konstrukcemi, při jejichž aplikaci není nezbytné použití dat (tedy proměnných). To výrazně zjednoduší jejich výklad.

V první kapitole si vysvětlíme základní možnosti integrovaného vývojového prostředí obou překladačů. Omezíme se ovšem pouze na věci naprosto nezbytné, nejzávažnější možnosti, bez jejichž úspěšného zvládnutí nelze programovat. Podrobný výklad všech možností obou překladačů lze najít v příručkách, které se s těmito překladači dodávají, nebo v nápovědě.

V následujících dvou kapitolách se seznámíme se způsobem, kterým pak budeme v průběhu tohoto i následujících dílů definovat pravidla zápisu nejrůznějších konstrukcí probíraných programovacích jazyků, a s robotem Karlem – vyučovací pomůckou, která nás bude provázet celým prvním dílem. Povíme si o jeho vzniku i o hlavních výhodách, které jeho použití při výuce programování přináší.

Ve čtvrté kapitole se nejprve naučíme přeložit, sestavit a spustit program. Pak si povíme i něco o ladění a jeden chybný program si spolu opravíme.

V páté kapitole se seznámíme se základními prvky obou jazyků. Poznáme identifikátory, klíčová slova a oddělovače. Zároveň si vysvětlíme význam a použití komentářů.

Šestá kapitola bude věnována procedurám. Naučíme se definovat je a povíme si i některé obecné zásady jejich tvorby.

V sedmé kapitole nahlédneme do světa velkých projektů. Vysvětlíme si, jak je třeba složité úlohy rozdělit na řadu úloh jednodušších, snáze zvládnutelných, a hned si to samozřejmě na řadě příkladů vyzkoušíme.

V osmé kapitole si rozebereme výhody a nevýhody dvou základních přístupů k řešení rozsáhlejších projektů: metodu shora dolů a metodu zdola nahoru. Zároveň si charakterizujeme situace, kdy je která z nich výhodnější.

Devátá kapitola nás seznámí s pojmem modul. Naučíme se vytvářet složitější programy, které sestávají z několika modulů.

V desáté kapitole se konečně dostaneme k výkladu algoritmických konstrukcí. Tato kapitola je věnována výkladu obou základních typů cyklů (iterace).

V jedenácté kapitole si pak budeme vyprávět o jednotlivých používaných podobách rozhodování (selekce).

Ve dvanácté kapitole poznáme funkce. Seznámíme se i s vedlejšími efekty funkcí.

Třináctou kapitolu jsme věnovali tolik proklínanému příkazu skoku a podobným konstrukcím.

Čtrnáctá kapitola bude zasvěcena rekurzi. Vysvětlíme si v ní používání tohoto algoritmického obratu, který se sice v řadě kursů programování nevysvětluje a programátoři jej proto většinou moc nepoužívají, je však nesmírně mocný a řešení řady úloh výrazným způsobem zjednodušuje.

V patnácté kapitole si pak ukážeme, jak lze rekurze s výhodou použít při řešení úloh metodou pokusů a oprav.

V poslední kapitole si shrneme, co jsme se naučili. Najdete tu také úplný přehled vlastností robota Karla.

Někteří z vás si možná všimli, že koncepce této učebnice se od běžných učebnic programování poněkud liší. Možná uslyšíte námitky proti umístění podprogramů na počátek celého výkladu (bývalý totalitní šéfredaktor časopisu Elektronika dokonce na toto téma řekl R. Pecinovskému, že asi v životě nečetl žádnou učebnici logiky – dodnes jsme nezjistili, jakou to mělo mít souvislost). Je sice pravda, že běžné učebnice programování si nechávají podprogramy až na konec výkladu, ale toto uspořádání (tj. podprogramy na konci) je dáno skutečností, že autorům se nechtělo vymýšlet příklady, které by byly dostatečně složité, aby použití podprogramů opravňovaly, a přitom tak jednoduché, aby je bylo možno zařadit před výklad ostatních partií. Kromě toho, hlavním cílem většiny učebnic není naučit čtenáře programovat, ale naučit je používat daný programovací jazyk. A učebnice končící výkladem podprogramů se přece jenom píše snadněji.

Stručně o dalších dílech

Na knihu *Základy algoritmizace* navazují díly *Práce s daty 1*, *Práce s daty 2*, *Objektové programování 1* a *Objektové programování 2*.

V knize *Práce s daty 1* (Grada 1997) se naučíte používat konstanty a proměnné, seznámíte se s výrazy, naučíte se definovat a používat funkce a procedury s parametry, pracovat s poli, zvládnete vstupní a výstupní operace v obou jazycích a v poslední kapitole najdete také příklad, využívající náhodných čísel.

Na tuto knihu bezprostředně navazuje *Práce s daty 2* (Grada 1997), v níž se seznámíte s ukazateli a s jejich použitím, naučíte se používat strukturované datové typy, pracovat se soubory a s datovými proudy a seznámíte se s nástroji pro podmíněný překlad a s používáním maker. V tomto dílu se také naučíte pracovat s borlandskou grafickou knihovnou pro DOS. Kniha o práci s daty končí rozsáhlejším příkladem, ve kterém naprogramujeme známou hru piškorky.

V *Objektovém programování 1* (Grada 1996) se seznámíte se základy objektového programování v obou jazycích. Naučíte se definovat a používat objektové typy, seznámíte se s možnostmi přetěžování operátorů v C++, poznáte další prostředky pro správu paměti a seznámíte se s dynamickými datovými typy (seznamy apod.) a s iterátory.

Kniha *Objektové programování 2* (Grada 1996) vás naučí odvozovat nové objektové typy od těch, které již používáte, pomocí dědičnosti a seznámí vás s další důležitou vlastností objektů – s polymorfismem. Najdete tu také podrobnosti o objektových datových proudech jazyka C++, naučíte se, jak si ušetřit práci použitím šablon, seznámíte se s výjimkami, bez nichž nelze rozumně používat dnešní překladače C++ nebo Delphi, s dynamickou identifikací typů a s prostory jmen. Mezi příklady najdete např. jednoduchoučký dosovský grafický editor nebo šablonu procedury, kterou lze využít jak ke třídění polí, tak i ke třídění seznamů.

Terminologie

Pro jazyk C budeme občas používat označení „Céčko“, neboť se s ním lépe zachází než se samotným písmenem. Podobně budeme používat přídavná jména „pascalský“, „céčkovský“, „borlandský“, „pascalista“, „céčkař“ apod., přestože proti nim mají někteří jazykoví korektoři výhrady.

Typografické konvence

V textu této knihy používáme následující konvence:

while	Tučně píšeme klíčová slova.
rekurze	Tučně píšeme nově zaváděné termíny a také pasáže, které chceme z jakýchkoli důvodů zdůraznit.
main	Kurzivou píšeme identifikátory, tj. jména proměnných, funkcí, typů apod. Přitom nerozlišujeme, zda jde o jména standardních součástí jazyka (např. knihovních funkcí) nebo o jména definovaná programátorem.
Dialog box	Kurzívou také píšeme anglické názvy.
ALT+F4	Kapitálky používáme pro vyznačení kláves a klávesových kombinací.
`Poloz();`	Neproporcionální písmo používáme v ukázkách programů a v popisu výstupu programů.
Options|Save	Bezpatkovým písmem vyznačujeme příkazy z nabídek a dialogových oken.

Části výkladu, které se týkají pouze Pascalu, jsou po straně označeny jednoduchou svislou čarou.

Části výkladu, které se týkají pouze C++, jsou po straně označeny dvojitou svislou čarou.

K této knize lze zakoupit doplňkovou disketu, na níž najdete úplné zdrojové texty všech příkladů uvedených v knize i některých dalších, na které v knize jen odkazujeme. Kromě toho obsahuje modul Karel, bez něhož nelze příklady v této knize spustit.

1. Základy práce s IDE

Před vlastním programováním byste měli nejprve zvládnout dvě věci: základy práce s operačním systémem (předpokládáme, že ty jste už zvládli) a základy práce s použitým vývojovým prostředím; tomu bude věnována tato kapitola.

Na jaře 1990 uvedla firma Borland na trh první verzi nového hitu, Turbo C++ 1.0, a s ním i nové integrované vývojové prostředí, nazvané IDE (zkratka slov *Integrated Development Environment* – integrované vývojové prostředí).

Toto prostředí pak s drobnými omezeními použila i v Turbo Pascalu 6.0 a samozřejmě (tentokrát již bez omezení, ale naopak rozšířené) v dalších verzích překladačů jazyka C++ a Pascalu. Zavedením IDE postavila firma Borland v té době komfort programátorské práce na novou, kvalitativně vyšší úroveň. Borlandské vývojové prostředí bylo ve srovnávacích recenzích překladačů různých firem hodnoceno jako nejpropracovanější a nejsnáze zvládnutelné. Zkušenosti ukázaly, že recenze měly pravdu; pro vývoj dosovských aplikací je Borland C++ patrně dodnes nejlepším nástrojem. Mimo jiné i proto jsme borlandské překladače zvolili za základ tohoto kursu.

Mezi charakteristické rysy IDE patří:

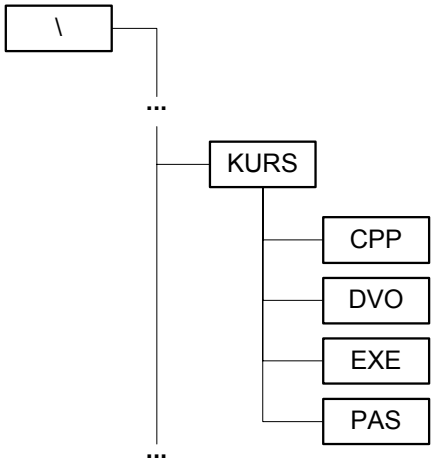
- ✧ prakticky libovolný počet vzájemně se překrývajících, přemísťovatelných oken, jejichž velikost lze operativně měnit,
- ✧ násobná editační okna umožňující nejen současnou editaci několika souborů, ale i současné zobrazení a editaci několika míst téhož souboru,
- ✧ možnost přenosu částí textu mezi jednotlivými soubory prostřednictvím jeho dočasného uložení do **schránky** (*clipboard* – v českém překladu borlandských manuálů se používal termín „nástěnka“),
- ✧ rozsáhlá, podrobná, kontextově citlivá nápověda s řadou příkladů, přičemž libovolnou část lze zkopírovat do vlastního textu,
- ✧ podpora myši (minimální požadovaná verze ovladače myši je uvedena v dokumentaci),
- ✧ propracovaná dialogová okna pro nastavení zvoleného režimu práce,
- ✧ makrojazyk umožňující rozšíření editoru o další funkce,
- ✧ možnost rychlého vyvolání dalších předdefinovaných programů – jejich seznam, způsob volání i předávané parametry si může uživatel upravit podle vlastních požadavků,

- ✧ u Turbo C++ a Borland C++ možnost nastavit pro každý modul vlastní překladač spolu s některými parametry překladu,
- ✧ u Turbo C++ a Borland C++ od verze 2.0 editační funkce *undo* a *redo*, implementované nejjednodušším (z uživatelského hlediska), ale také nejvýhodnějším způsobem, jaký si lze představit,
- ✧ u Turbo C++ a Borland C++ od verze 3.1 a u Turbo Pascalu od verze 7.0 možnost vizuálního rozlišení jednotlivých tříd prvků tvořících zdrojový text programu.

Pokusme se nyní alespoň ve stručnosti popsat nejzákladnější postupy, které budete při vývoji vlastních programů potřebovat. V této kapitole se omezíme pouze na otázky související s vlastní tvorbou zdrojových textů. O možnostech ladění programů a přípravy rozsáhlých projektů si povíme později, až je budeme při práci potřebovat.

1.1 Organizace disku

Než začneme vlastním výkladem, musíme si nejprve připravit v počítači prostředí pro další práci. Vytvořte si na pevném disku adresář *KURS* a v něm čtyři podadresáře: *EXE*, *DVO*, *CPP*, *PAS*. (Pokud se nechystáte sledovat programy v obou jazycích, stačí, vytvoříte-li si adresář pouze pro jazyk, který vás zajímá.) Do těchto adresářů si překopírujte soubory ze stejnojmenných adresářů na doprovodné disketě (o souborech v adresáři *CONFIG* si budeme povídat po chvíli). Z adresáře *KURS* pak budete spouštět používaný překladač, který máte již jistě nainstalovaný a cestu k němu uloženou mezi ostatními v systémové proměnné *PATH*.



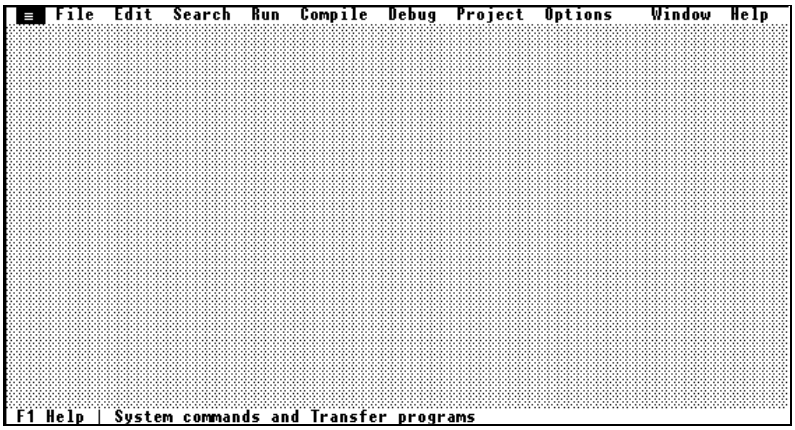
Obr. 1.1 Doporučená struktura adresářů

V dalším textu budeme předpokládat, že jste si instalovali překladač do adresáře *\B*. Pokud používáte překladače obou jazyků, Turbo Pascalu i Borland C++, můžete je do tohoto adresáře instalovat oba.

Nyní si nastavte adresář *KURS* jako aktuální a spusťte překladač. Ten by se měl po spuštění ohlásit prázdnou pracovní plochou, na níž bude svítit okno s údaji o provozované verzi. Toto okno odstraníme stisknutím kterékoli klávesy.

1.2 Systém ovládání – nabídky a horké klávesy

Prohlédněme si nejprve toto základní prostředí a podívejme se, co nám nabízí:



Obr. 1.2 Základní prostředí

První řádka obrazovky slouží jako vodorovně uspořádaná nabídka základních možností. Budeme jí říkat **základní nabídka** nebo **základní menu** – v originálních manuálech najdete název *menu bar*, tedy nabídkový pruh. Tato řádka je zároveň branou pro přístup ke všem specializovanějším nabídkám. Na základní nabídku se dostanete prakticky z kteréhokoli okna pouhým stiskem klávesy F10. Provedete-li to, zjistíte, že se **některá z položek nabídky zvýraznila**. Toto zvýraznění můžete přesouvat pomocí vodorovných šipek z položky na položku.

Stiskněte nyní (tj. při zvýraznění některé z položek základní nabídky) klávesu ENTER a pod zvýrazněnou položkou se vám rozbalí spouštěcí (roletová) nabídka (*pull down menu*). Všimněte si, že pokud je již rozbalena některá roletová nabídka, rozbalí se při přesunutí na jinou položku základní nabídky prostřednictvím vodorovných šipek automaticky spouštěcí nabídka té položky, na niž jste se přesunuli.

Pokud je položka v nabídce následována trojicí teček (. . .), znamená to, že při jejím vyvolání otevře systém dialogové okno. Pokud je položka následována trojúhelníčkem namířeným vpravo (➢), znamená to, že při jejím vyvolání otevře systém vynořovací nabídku (*popup menu*), v níž bude třeba požadavek blíže specifikovat. Pokud položka není následována žádným z těchto dvou symbolů, znamená to, že po jejím zadání systém přímo vykoná požadovanou akci.

Ještě jedna věc stojí u roletových nabídek za povšimnutí: u některých položek jsou uvedeny klávesy nebo kombinace kláves, pomocí nichž lze danou funkci vyvolat přímo, bez nutnosti projít systémem nabídek. Jak se budete se

systémem postupně seznamovat, zapamatujte si tyto **klávesové zkratky** neboli **horké klávesy** (*hot keys*), protože s jejich pomocí můžete svou práci podstatně zrychlit. Dodejme ještě, že horkými klávesami pro aktivování položek základní nabídky jsou klávesy s počátečními písmeny položek stisknuté při nastaveném přeřaďovači ALT.

Podívejme se nyní na poslední řádku, které říkáme **stavová řádka** (*status line*) a která většinou slouží jako rychlá nápověda. Pokud se pohybujete v nějaké nabídce, najdete ve stavové řádce nejprve informaci o tom, že po stisku klávesy F1 obdržíte podrobnější nápovědu, a za touto informací následuje stručné, jednořádkové vysvětlení funkce zvýrazněné položky. V průběhu práce pak stavový řádek obsahuje nabídku několika horkých kláves s nejpravděpodobněji požadovanými funkcemi – viz předchozí nabídku klávesy F1.

Položku v jakékoli nabídce můžete aktivovat několika způsoby:

1. Při aktivované nabídce:

   ✧ Stiskněte písmeno, které je v názvu položky zvýrazněno.

   ✧ Najeďte pomocí kurzorových šipek se zvýrazňovacím blokem na zvolenou položku a stiskněte klávesu ENTER.

   ✧ Najeďte kurzorem myši na zvolenou položku a stiskněte levé tlačítko (u základní nabídky to můžete provést i tehdy, není-li právě aktivována).

2. Kdykoliv:

   ✧ Stiskněte patřičnou horkou klávesu.

   ✧ Je-li odpovídající horká klávesa vyznačena ve stavovém řádku, je možno na ni najet myší a danou funkci vyvolat stiskem levého tlačítka myši.

Pro čtenáře, kteří si ještě nezvykli na anglické texty nabídek a nápověd, je v tabulce 1.1 přeložena charakteristika všech položek základní nabídky.

Položka	Význam
≡	Systémové příkazy (horkou klávesou je ALT+MEZERA)
File	Práce se soubory (otevření, zavření, vytvoření, uložení, tisk apod.)
Edit	Mazání, přesuny a kopírování bloků textu
Search	Vyhledávání částí textu a pohyb po souborech
Run	Spuštění a krokování programu, zadávání parametrů v příkazovém řádku programu

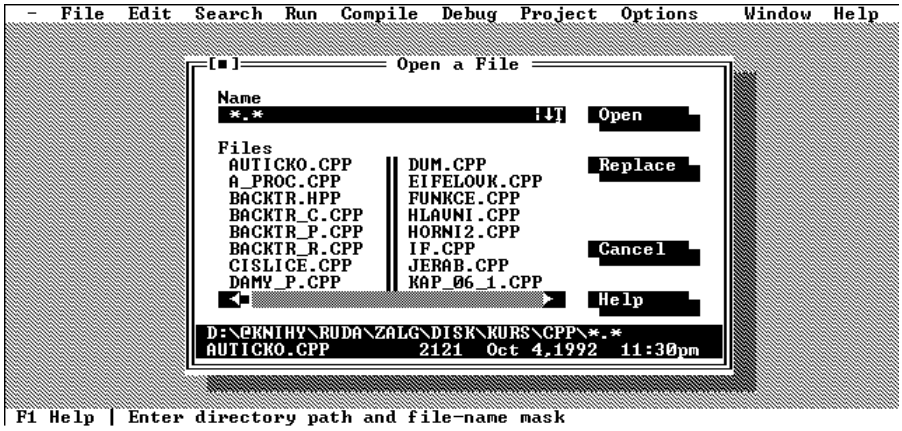
Položka	Význam
Compile	Překlad a sestavení programu
Debug	Vyhodnocování výrazů, modifikace dat, nastavování zarážek (breakpointů) a sledovaných výrazů (watch)
Options	Nastavení volitelných parametrů systému
Window	Příkazy pro práce s okny
Help	Nápověda

Tab. 1.1 Položky základní nabídky

1.3 Otevření souboru – dialogové okno

První, co bychom po spuštění překladače měli udělat, je otevřít editační okno a načíst do něj soubor. Stiskneme tedy F10 (vstup do základní nabídky), F (File – práce se soubory; jak víme, můžeme postup zkrátit pomocí ALT+F) a O (Open – otevři soubor; jak si můžete všimnout, horkou klávesou pro tuto službu je klávesa F3).

Systém nám nabídne **dialogové okno** (*dialog box*), abychom mohli snadno označit soubor, jehož načtení požadujeme.



Obr. 1.3 Dialogové okno

Než soubor doopravdy načteme, podívejme se nejprve podrobněji na vyvolané dialogové okno. Toto okno obsahuje několik typických bloků, s nimiž se budete setkávat i v ostatních dialogových oknech. Mezi jednotlivými bloky se

pohybujeme buď pomocí tabulátoru, anebo stiskem přeřaďovače ALT a zvýrazněného písmene z názvu bloku, který chceme aktivovat.

Kromě toho máme samozřejmě možnost aktivovat blok tak, že do něj najedeme kurzorem myši a stiskneme její levé tlačítko.

Vstupní řádka

Vstupní řádka (vstupní blok) (*input line*, *input box*) slouží pro vstup jména hledaného souboru, přičemž při jeho zadávání je možné použít i žolíky (*wildcards*) „*“ a “?“. Po zapsání požadované specifikace ji předáme stiskem ENTER. Ikona se šipkou dolů, kterou najdete vpravo od tohoto bloku, naznačuje možnost vyvolat seznam několika naposledy zadávaných hodnot stiskem klávesy „šipka dolů“ nebo odťuknutím[1] této ikony myší. Tento seznam nedávno zadaných hodnot se obvykle nazývá **historie** nebo **seznam historií** (*history list*).

V okně na obr. 1.3 se vstupní řádka jmenuje Name a zvýrazněné první písmeno nám říká, že ji lze aktivovat stiskem ALT+N; po otevření okna je aktivní.

Seznam

Seznam (*list box*) obsahuje seznam souborů, které vyhovují specifikaci zadané ve vstupním bloku. V seznamu se pohybujeme pomocí kurzorových kláves (nebo myši) a zvolenou položku vybíráme stiskem ENTER. Při použití myši vybíráme položku, na niž ukazuje kurzor myši, dvojklikem (dvojstisk) levého tlačítka (tj. dvěma stisky rychle za sebou – mezní dobu mezi dvěma po sobě jdoucími stisky téhož tlačítka, která odlišuje dvojklik stisk od dvou „nezávislých“ stisků, lze nastavit).

V našem okně se seznam jmenuje Files a aktivuje se automaticky po zadání jména souboru v bloku Name.

Informační blok

Informační blok (*information panel*, *infopanel*) nabízí uživateli některé dodatečné informace – v našem případě se nachází ve spodní části dialogového okna, nemá jméno a obsahuje podrobné informace o vybraném (zvýrazněném) souboru v seznamu.

[1] Tj. najedeme kurzorem myši na tuto ikonu a stiskneme její levé tlačítko.

Tlačítka

Tlačítka (*buttons*) jsou v našem okně čtyři:

Open
Otevře nové okno a načte do něj vybraný soubor. Tato možnost je nastavena jako implicitní, a proto je možno ji vyvolat pouhým stiskem ENTER. Lze ji také vyvolat stiskem ALT+O (ALT+PÍSMENO O) nebo označením pomocí myšího kurzoru a stiskem levého tlačítka myši.

Replace
Tato volba je možná pouze v případě, kdy jsme otevření souboru vyvolali z editačního okna. Zvolíme ji klávesovou zkratkou ALT+R, aktivací tlačítka Replace pomocí tabulátoru a následným stiskem ENTER, anebo pomocí myši. Tento příkaz nahradí obsah editačního okna vybraným souborem, přičemž byl-li původní obsah od posledního uložení změněn, systém se nejprve zeptá, nechceme-li jej před nahrazením uložit.

Cancel
Uzavře dialogové okno a vrátí se do předchozího režimu. Vše, co jsme v okně zadali, bude zapomenuto. Vyvolává se nejlépe (horkou) klávesou ESC, ale lze použít i myši nebo najetí tabulátorem a potvrzení.

Help
Poskytne nápovědu. Vyvolává se nejlépe klávesou F1, ale lze použít i myši nebo najetí tabulátorem a potvrzení. V jiných dialogových oknech najdeme ještě další typy bloků. S nimi se seznámíme, až je budeme potřebovat.

1.4 Uložení a přejmenování souboru

Požadovaný soubor jsme tedy načetli. Pokud v něm uděláme nějaké změny, je vysoce pravděpodobné, že budeme chtít jeho novou podobu uložit. Toho dosáhneme jednoduše stisknutím horké klávesy F2 nebo (pokud např. tuto klávesovou zkratku zapomeneme) příkazem File | Save.

Trochu komplikovanější situace vznikne, budeme-li chtít opravený soubor přejmenovat, přesněji – uložit pod novým jménem. I na tuto situaci je však IDE předem připraveno. Zvolíte File | Save as, IDE otevře dialogové okno, v němž do vstupního pole Name napíšete nové jméno, pod nímž chcete soubor uložit, a svoji volbu potvrdíte.

1.5 Práce s okny – ikony

Borlandské IDE je postaveno na práci s přemístitelnými okny, jejichž velikost lze měnit. Tato okna se mohou navzájem překrývat. V každém okamžiku může být aktivní pouze jediné okno.

Aktivní okno poznáme podle toho, že je orámováno dvojitou čarou, zatímco všechna ostatní okna jsou orámována pouze jednoduchou čarou. Kromě toho je aktivní okno na vrchu pomyslné hromady oken na obrazovce.

Všechna otevřená okna jsou číslována, přičemž prvních devět oken má svá pořadová čísla zapsána na pravém konci horního okraje rámečku. Tato okna můžeme z klávesnice aktivovat stiskem kombinace ALT+ČÍSLO OKNA: např. třetí okno aktivujeme klávesovou zkratkou ALT+3). Kterékoliv okno pak můžeme aktivovat myší tak, že najedeme na jeho viditelnou část (včetně rámečku) a stiskneme levé tlačítko myši. Z klávesnice pak můžeme aktivovat jedno okno po druhém postupným tisknutím klávesy F6, což je horká klávesa pro příkaz Window | Next. Tak sice můžeme projít jedno okno po druhém, až dojde na okno, které potřebujeme, ale rychlejší většinou bývá vyvolat si seznam oken (Window | List – horká klávesa je ALT+0, a tentokrát je to ALT+NULA) a vybrat si žádané okno ze seznamu.

Jak jsme již řekli, okna můžeme zvětšovat, zmenšovat a posouvat. Množinu akcí, které můžeme s daným oknem provádět, lze snadno určit podle řídicích obrazců, které najdeme na rámečku okna. Tyto řídicí obrazce slouží k vyvolávání různých funkcí prostřednictvím myši. IDE používá čtyři druhy řídicích obrazců: ikony (semigrafické znaky symbolizující spřaženou akci), horní hranu rámečku mezi ikonami, posunovací pravítka (někdy se jím říká posunovací lišty, posunovací proužky nebo rolovací lišty) a „velikostní růžek“. Všechny obrazce se aktivují tím, že na ně najedeme myším kurzorem a stiskneme levé tlačítko. Při aktivaci rámečku (tj. chceme-li posouvat okno po obrazovce), velikostního růžku a ukazatele pozice na posunovacím pravítku je třeba navíc držet levé tlačítko po celou dobu, co popojíždíme s myší, a pustit je až po dosažení žádaného výsledku.

Zavírací ikona

Zavírací ikona je malý plný čtverec vpravo vedle levého horního rohu rámečku. Tuto ikonu najdete u každého okna. Stiskneme-li na ní tlačítko myši, zavře se aktivní okno a jako aktivní se nastaví okno, které bylo aktivní předtím.

Pokud zavíráme editační okno a editovaný soubor se od posledního

uložení změnil, systém se nás před zavřením okna nejprve zeptá, zda tento soubor nechceme uložit. Stejného efektu dosáhneme z klávesnice stiskem ALT+F3 nebo příkazem Window | Close.

Důležité upozornění

Seznam všech zavřených editačních oken se ukládá, a to v Pascalu do konfiguračního souboru (.TP) a v C++ do projektového souboru (.PRJ). Kdykoliv si necháme zobrazit seznam oken (ALT+NULA nebo Window | List), najdeme v něm za otevřenými okny i okna zavřená. Pokud z tohoto seznamu nebudeme jednou za čas soubory odmazávat, velikost souboru poroste, a přesáhne-li 64 KB, systém se nám zhroutí.

Zvětšovací ikona

Zvětšovací ikona je šipka vzhůru vlevo od levého horního rohu rámečku. Při její aktivaci se dané okno zvětší přes celou obrazovku. Zároveň se však tato ikona přemění ve **zmenšovací ikonu** (šipka dolů tamtéž), při jejíž aktivaci se okno zmenší na původní velikost. Stejného efektu dosáhnete z klávesnice stiskem F5 nebo vyvoláním Window | Zoom.

Horní hrana rámečku

„Uchopíme-li“ okno myší za horní hranu rámečku (tj. najedeme na tuto hranu kurzorem, stiskneme levé tlačítko a budeme je držet), můžeme oknem pohybovat po obrazovce. Stejného efektu dosáhneme z klávesnice stiskem CTRL+F5 (horká klávesa pro Window | Size/Move) a následným posouváním okna pomocí kurzorových kláves.

Velikostní růžek

Okna můžete také zvětšovat a zmenšovat, i když ne všechny typy oken vám zvětšování a zmenšování umožní. Okna, která lze zvětšit či zmenšit, poznáme podle toho, že v pravém dolním rohu je jejich dvojitý rámeček přerušen a nahrazen jednoduchým. Toto přerušení slouží ke změně velikosti okna za pomoci myši a budeme mu říkat velikostní růžek.

Změna velikosti okna se provádí obdobně jako posouvání: velikostní růžek uchopíme myší a přesouváme. Zbytek okna přitom stojí na místě. Stejného efektu dosáhneme z klávesnice stiskem CTRL+F5 (horká klávesa pro Window | Size/Move) a následným posouváním hranic okna pomocí kurzorových kláves při stisknutém přeřaďovači SHIFT.

Posouvací pravítka

Posledním řídicím obrazcem jsou posouvací pravítka, která u editačních oken zabírají část rámu sousedící s velikostním růžkem. Najdeme je však nejen u oken, ale i u seznamových bloků v dialogových oknech. Posouvací pravítka sestávají ze tří částí: dvou krajních posouvacích ikon a z vlastního pravítka s táhlem zobrazujícím pozici kurzoru v rámci celého souboru nebo seznamu. Umístíme-li kurzor myši na některou z posouvacích ikon a stiskneme-li levé tlačítko, posune se obsah okna (seznamu) o jeden řádek nebo sloupec daným směrem. Umístíme-li kurzor myši někam na svislé pravítko mimo kurzor a stiskneme-li levé tlačítko, dosáhneme u svislého posunu stejného efektu jako při stisku kláves PGUP a PGDN a u vodorovného posunu podobného efektu, jenže ve vodorovném směru. Uchopíme-li táhlo na pravítku myší a posuneme-li jej do nové pozice, přesune se obsah okna (seznamu) tak, aby pozice kurzoru v souboru (seznamu) odpovídala pozici kurzoru na editačním pravítku.

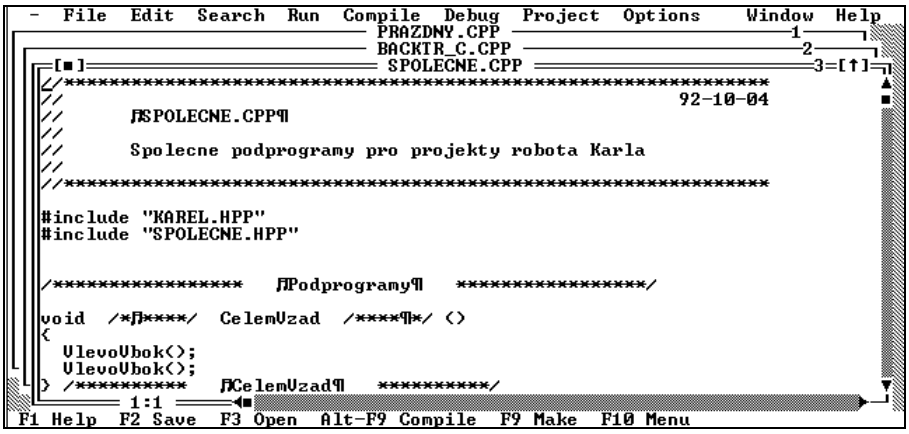
Závěr

Na závěr se ještě zmíníme o možnostech uspořádání editačních oken na obrazovce. IDE nabízí dvě možná předdefinovaná uspořádání: dělené (dlaždičkové) uspořádání a kaskádní uspořádání. První můžete nastavit příkazem Window | Tile a druhé příkazem Window | Cascade. Při obou uspořádáních IDE rozdělí pracovní plochu na dvě části; do horní přesune editační okna a do spodní části ostatní okna.

Při dlaždičkovém uspořádání se IDE pokusí uspořádat okna v obou částech tak, aby se navzájem pokud možno nepřekrývala (dovoluje-li to jejich minimální možná velikost) a aby všechna zaujímala přibližně stejnou plochu. Při kaskádním uspořádání budou mít všechna okna společný pravý dolní roh, přičemž spodní okno bude největší a každé další o jeden řádek a jeden sloupec menší. Předdefinovanými uspořádáními můžete ale také pohrdnout a uspořádat si okna na obrazovce podle své aktuální potřeby.

Nám se pro ovládání myší nejvíce osvědčila všechna okna stejně velká, navzájem posunutá o jeden řádek a jeden sloupec. Hlavní výhodou tohoto uspořádání je, že vždy z každého okna vidíte alespoň kousek, a proto je lze pomocí myši snadno aktivovat. Pascal však svým uživatelům znepříjemňuje život tím, že všechna jeho okna mají stín, který někdy nepříjemně zakrývá viditelné části spodních oken, takže si je pak musíte domýšlet. IDE překladačů jazyka C++ se chová naštěstí rozum- rozumněji: stín mají pouze okna,

která jsou viditelná pouze po dobu, kdy jsou aktivní, tj. dialogová okna, chybová a varovná okna apod.



Obr. 1.4 Stupňovité uspořádání oken v Borland C++

1.6 Základní editační příkazy, schránka

Editor v borlandských integrovaných prostředích navazuje na způsob ovládání, s nímž kdysi vyrukoval textový procesor WordStar. Jím zavedené konvence se postupem doby natolik vžily, že je začali ve svých editorech a textových procesorech používat i další výrobci, a tyto konvence představovaly u nás až do příchodu Windows de facto standard[2].

Většinu editačních příkazů lze v IDE zadat několika způsoby: Za prvé klasickým wordstarovským stylem pomocí písmenných kláves aktivovaných při stisknutém přeřaďovači CTRL, za druhé pomocí kláves z kurzorového a editačního pole a za třetí prostřednictvím myši.

Abychom mohli zestručnit další výklad, zavedeme si pro klávesy numerického, kurzorového a editačního pole a pro další pomocné klávesy značky uvedené v tabulce 1.2.

Značka	Klávesa	Značka	Klávesa
INS	INSERT	←	ŠIPKA VLEVO

[2] Borland C++ 3.1 podporuje také ovládání IDE pomocí klávesových zkratek, používaných ve Windows. Zadáme-li příkazy Options | Environment | Preferences, objeví se dialogové okno Preferences, v němž v bloku Command Set zaškrtneme možnost CUA.

DEL	DELETE	→	ŠIPKA VPRAVO
HOME	HOME	↑	ŠIPKA NAHORU
END	END	↓	ŠIPKA DOLŮ
PGUP	PAGE UP	TAB	TABULÁTOR
PGDN	PAGE DOWN	BS	BACKSPACE

Tab. 1.2 Značky pro kurzorové a editační klávesy

Editační příkazy

Klávesy v kurzorovém poli (šipky), v editačním poli (INS, DEL, HOME, END, PGUP, PGDN) a v numerickém poli se v IDE chovají nestandardně. Není-li stisknut přeřaďovač SHIFT, chovají se klávesy v kurzorovém a editačním poli dle očekávání a klávesy v numerickém poli jako při vypnutém režimu NUMLOCK (nastavení NUMLOCK nemá v IDE na chování těchto kláves žádný vliv). Je-li stisknut přeřaďovač SHIFT, ignorují se ostatní přeřaďovače a posunem kurzoru zároveň nastavujeme okraje bloku – viz dále.

Ve shrnutí základních editačních příkazů v následujících tabulkách nejsou jednotlivé akce popsány podrobně, protože si jejich přesný význam můžete zjistit pokusně sami. Tyto tabulky by vám měly posloužit hlavně pro získání rámcového přehledu o možnostech editoru zabudovaného v IDE[3].

U některých funkcí jsou v následujících tabulkách uvedeny dva možné způsoby vyvolání. V prostředním sloupci najdete standardní volání použitelné v obou IDE (pokud v tomto sloupci nic není, nelze tuto akci v Pascalu užít), v pravém pak volání, které lze použít pouze v C++ – podrobnosti najdete v podkapitole o zvláštnostech tohoto jazyka.

Akce	**V obou IDE**	**V BC++**
Posuň kurzor o jeden znak vlevo	←	
Posuň kurzor o jeden znak vpravo	→	
Posuň kurzor o jedno slovo vlevo	CTRL+←	
Posuň kurzor o jedno slovo vpravo	CTRL+→	
Vrať kurzor na počátek řádku	HOME	
Přesuň kurzor na konec řádku	END	
Přesuň kurzor o 1 řádek nahoru	CTRL+↑	

[3] V těchto tabulkách nejsou uvedeny editační příkazy, používané v MS Windows, i když je překladače Borland C++ 3.0 a 3.1 již podporují.

Akce	V obou IDE	V BC++
Přesuň kurzor o 1 řádek dolů	CTRL+↓	
Vrať kurzor na začátek 1. řádku okna	CTRL+HOME	
Přejdi na poslední řádek okna	CTRL+END	
Přejdi na počátek bloku	CTRL+Q, B	ALT+PGUP
Přejdi na počátek bloku	CTRL+Q, K	ALT+PGDN
Přejdi o stránku vzhůru	PGUP	
Přejdi o stránku dolů	PGDN	
Přejdi na počátek souboru	CTRL+PGUP	
Přejdi na konec souboru	CTRL+PGDN	
Roluj o řádek vzhůru (text se posune dolů)	CTRL+W	CTRL+↑
Roluj o řádek dolů (text se posune dolů)	CTRL+Z	CTRL+↓
Vrať se na minulou pozici kurzoru	CTRL+Q, P	

Tab. 1.3 Příkazy pro pohyb kurzoru

Akce	V obou IDE	V BC++
Smaž znak vlevo od kurzoru	BS	
Smaž znak, na němž je kurzor	DEL	
Smaž text vlevo od kurzoru k počátku slova		CTRL+F11
Smaž text vpravo od kurzoru do konce slova	CTRL+T	CTRL+F12
Smaž počátek řádku (od počátku ke kurzoru)		ALT+F11
Smaž zbytek řádku (od kurzoru do konce)	CTRL+Q,Y	ALT+F12
Smaž řádek	CTRL+Y	
Vlož řádek	CTRL+N	
Obnov řádek (jen v Pascalu)	CTRL+Q, P	
Přepni režim vkládání/přepisování	INS	
Změň funkci tabulátoru	CTRL+O, T	CTRL+TAB
Vlož tabulátor	TAB	
Zpět (vrať poslední akci)	ALT+ BS	

Akce	V obou IDE	V BC++
Znovu (zopakuj poslední akci)		ALT+SHIFT+BS

Tab. 1.4 Příkazy pro vkládání a mazání

Akce	V obou IDE	V BC++
Označ počátek bloku	CTRL+K, B	ALT+HOME
Označ konec bloku	CTRL+K, K	ALT+END
Zkopíruj blok na pozici kurzoru	CTRL+K, C	ALT+INS
Přesuň blok na pozici kurzoru	CTRL+K, V	ALT+DEL
Zkopíruj blok do schránky (*clipboard*)	ALT+INS	
Přesuň blok do schránky	SHIFT+DEL	
Vlož blok ze schránky	SHIFT+INS	
Smaž blok	CTRL+K, Y	
Smaž blok (nefunguje při editaci schránky)	CTRL+DEL	
Zvýrazni/zamaskuj blok	CTRL+K, H	
Vytiskni blok	CTRL+K, P	
Posuň blok o sloupec vpravo	CTRL+K, I	ALT+→
Posuň blok o sloupec vlevo	CTRL+K, U	ALT+←
Přečti blok ze souboru	CTRL+K, R	
Zapiš blok do souboru	CTRL+K, W	

Tab. 1.5 Příkazy pro práci s bloky

Akce	V obou IDE	V BC++
Následující znak vlož do textu (používá se pro vkládání řídicích znaků)	CTRL+P	
Nastav i-tou značku (záložku), i = 1 – 9. Používáte-li národní klávesnici, musíte spolu s číslicovou klávesou stisknout SHIFT	CTRL+K, *i*	
Přejdi na i-tou značku	CTRL+Q, *i*	
Ulož soubor	F2	
Najdi daný text	CTRL+Q, F	ALT+Q
Najdi daný text a nahraď jej novým	CTRL+Q, A	ALT+A

Akce	V obou IDE	V BC++
Opakuj poslední vyhledávání/nahrazování	CTRL+L	
Najdi odpovídající zavírací závorku (lze použít i pro ostatní druhy závorek)	CTRL+Q, [	ALT+↓
Najdi odpovídající otevírací závorku	CTRL+Q,]	ALT+↑

Tab. 1.6 Další důležité příkazy

Kromě popsaných způsobů vyvolání daných akcí často existují i další možnosti; najdete je v nápovědě, (která je ovšem anglicky) nebo v českém manuálu. Některé termíny a akce z předchozího souhrnu však přece jen potřebují podrobnější vysvětlení:

Hranice slov

Slovo je v IDE definováno jako posloupnost alfanumerických znaků (tj. písmen a číslic) ohraničená některým z ostatních znaků. Protože však naše znaky s diakritikou patří mezi ty „ostatní znaky“, nemusí v českých či slovenských komentářích odpovídat naše představa o hranicích slova představě počítače.

Vyhledávání a nahrazování textu

Při vyhledávání a nahrazování textu IDE nejprve otevře dialogové okno, v němž se vás zeptá na hledaný a nahrazující text, a umožní vám nastavit některé parametry následující akce. V tomto dialogovém okně najdeme dva nové typy bloků: blok voleb (*check box*) a blok přepínačů (*radio buttons*).

Blok voleb

V bloku voleb bývá několik voleb, z nichž každou můžeme nastavit (aktivovat, povolit), nebo potlačit (zakázat). Stav volby je znázorněn v hranatých závorkách vlevo vedle textu volby. Je-li v závorce x, je volba nastavena, je-li tam mezera, je potlačena. Změnu nastavení můžeme provést několika způsoby:

- ✧ stiskem zvýrazněného znaku volby při nastaveném přeřaďovači ALT.
- ✧ aktivací bloku, najetím řádkového kurzoru na měněnou položku a stiskem mezery (**ne** ENTER, **ale mezera!**),
- ✧ najetím kurzoru myši na měněnou položku a stiskem levé klávesy.

Blok přepínačů

V bloku přepínačů bývá naopak uvedeno několik možných hodnot jedné volby, z nichž může být nastavena vždy jen jedna. Hodnota, která je v danou chvíli aktivní, je označena černou tečkou v kulatých závorkách vlevo vedle textu. Ostatní hodnoty mají v závorkách mezeru. Pro nastavení hodnoty volby platí stejná pravidla jako pro nastavovací blok, až na to, že při druhém postupu není nutno mačkat mezeru, protože hodnota se nastaví automaticky ve chvíli, kdy na ni najedeme řádkovým kurzorem.

Abychom vyšli vstříc těm, kteří se studiem angličtiny teprve začínají, uvádíme v tabulce 1.7 volné překlady významu textů vztahujících se k jednotlivým blokům obou těchto dialogových oken.

Anglický text	**Překlad a vysvětlení**
Text to find	Vstupní řádka s hledaným textem. IDE nabídne slovo, na němž je kurzor. Klávesou → lze do tohoto textu připojit další část řádku
New text	Vstupní řádka obsahující text, kterým má být nalezený text nahrazen
Options	Dialogové okno s některými volbami
Case sensitive	Rozlišující velká a malá písmena
Whole words only	V prohledávaném souboru hledá jen celá slova, shodná se zadaným textem
Regular expression	Speciální možnosti – zájemce odkazujeme na manuál
Prompt on replace	IDE bude před každým nahrazením žádat o souhlas
Direction	Přepíná směr vyhledávání
Forward	Vpřed
Backward	Vzad
Scope	Rozsah prohledávaného textu
Global	Hledáme v celém souboru
Selected text	Hledáme pouze v označeném (zvýrazněném) bloku
Origin	Výběr počátku vyhledávání
From cursor	Od pozice kurzoru
Entire scope	Od počátku prohledávané oblasti
OK	Tlačítko, po jehož stisku se provede jedno vyhledávání nebo nahrazení podle zadaných voleb

Tab. 1.7 Překlad voleb v dialogových oknech Find a Replace

Označování bloků

Blok můžeme označit způsobem, popsaným v tabulce 1.5. To znamená, že kurzorem najedeme na začátek blo- bloku, stiskneme CTRL+K, B, pak se

přesuneme na konec bloku a stiskneme CTRL+K, K. Vedle toho nabízí IDE ještě druhý způsob: najedeme kurzorem na jeden konec bloku, pak se stiskne přeřaďovač SHIFT a při stisknutém přeřaďovači dovedeme kurzor pomocí kláves z kurzorového nebo editačního pole na druhý konec označovaného bloku; pak klávesu SHIFT pustíme. IDE přitom průběžně zvýrazňuje okamžitou podobu bloku.

Pamatujte, že jakmile je stisknut přeřaďovač SHIFT, ignorují se při používání kláves kurzorového, editačního a numerického pole ostatní přeřaďovače.

Přesun a kopírování textu mezi soubory

Přesuny a kopírování textu z jednoho souboru do druhého se v IDE řeší prostřednictvím schránky[4] (*clipboard*). Text, který chceme přesunout, resp. zkopírovat do jiného souboru, nejprve označíme jako blok a přesuneme (SHIFT+DEL), resp. zkopírujeme (CTRL+INS) jej do schránky. Máte-li otevřené okno s obsahem schránky, můžete se přesvědčit, že se náš blok „přilepil" za dosavadní obsah schránky a zůstal zvýrazněn jako blok. (Toto okno lze otevřít pomocí Edit | Show clipboard.) Texty ve schránce můžeme libovolně editovat, přičemž nesmíme zapomenout, že do cílového souboru se nepřesouvá celý obsah schránky, ale pouze vyznačený blok. Budeme-li tedy chtít vložit jiný text než ten, který je v danou chvíli zvýrazněn, musíme jej označit jako blok. Abychom mohli označený blok vložit do cílového souboru, přejdeme do okna s cílovým souborem, najedeme kurzorem na místo, kam jsme chtěli text přesunout, a stiskem SHIFT+INS tam blok ze schránky vložíme.

Hledání odpovídajících závorek

Tuto funkci oceníte zejména v rozsáhlejších programech, kdy nadbytečnost některé závorky zjistí překladač na úplně jiném místě, než kde k chybě doopravdy došlo. Tento příkaz vám umožní najít ke každé kulaté ((nebo)), hranaté ([nebo]), složené ({ nebo }) a v C++ ještě navíc k úhlové (< nebo >) a komentářové (/* nebo */) závorce její odpovídající protějšek. Na to by však stačil jeden příkaz, protože směr vyhledávání je z podoby závorky jasný (ostatně akce jsou při vyhledávání párových závorek u obou příkazů totožné).

[4] Clipboard – překlad podle slovníku: pracovní deska s klipsem na přidržování papírů. V časopisecké verzi jsme používali termín „zápisník", v českých překladech borlandských manuálů najdete termín „nástěnka". Ve Windows je k disposici obdobné zařízení, které se česky označuje **schránka**, a toto označení jsme převzali i do naší knihy.

Dva příkazy jsou zavedeny proto, aby bylo možno vyhledávat odpovídajícího partnera i k uvozovkám (") a apostrofům ('), u nichž musí směr pátrání definovat uživatel.

Tabulátory

Mezi programátory panuje spor o to, zda je při psaní programů lepší, aby se po stisku tabulátoru kurzor přesunul na pozici danou následující tabulační zarážkou, anebo na pozici definovanou počátkem nejbližšího dalšího slova na nejbližším předchozím řádku. IDE povoluje oba režimy a přepíná mezi nimi na povel CTRL+O, T. V C++ je při použití konfiguračního souboru z doprovodné diskety možno přepínat i pomocí CTRL+TAB. Uživatelé C++ však mají ještě jednu možnost: nechat režim přepnut na odsazování definované tabulačními zarážkami a pro odsazování definované předchozím řádkem používat klávesu SHIFT+TAB.

1.7 Specifika prostředí C++

IDE překladačů jazyka C++ je komplexnější a oproti Pascalu poskytuje řadu dalších možností zejména při tvorbě rozsáhlejších projektů. Z nabízených možností si zde povíme pouze o možnosti nahrání rozšiřujícího konfiguračního souboru a o možnosti definice přímo spustitelných programů (*transfer programs*).

Konfigurační soubory

Od verze 3.0 nabízejí překladače možnost nastavení řady parametrů přímo z IDE. K nastavení významu jednotlivých kláves spolu s novými editačními příkazy je třeba služeb překladače maker, který přeloží váš zdrojový kód maker a upraví podle něj informace uložené v konfiguračním souboru *TCCONFIG.TC*.

Abyste mohli využít rozšíření možností editoru o zjednodušené vyvolávání některých často používaných akcí a o některé další rozšiřující funkce (oboje viz tabulky v podkapitole 1.6), jsou pro vás na doprovodné disketě připraveny konfigurační soubory, v nichž je kromě rozšíření služeb editoru nastaveno i jiné vybarvení objektů IDE, které (alespoň podle názoru R. P.) umožňuje jednotlivé objekty IDE lépe rozlišit. Tyto konfigurační soubory se všechny jme-

nují *TCCONFIG* a jejich přípona pak definuje to, pro jaký monitor a jakou verzi překladače jsou určeny.

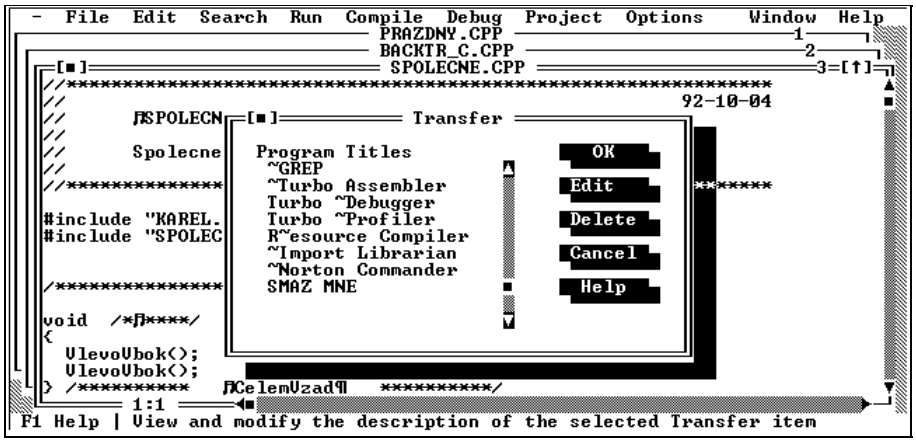
Chcete-li využít této možnosti, nahrajte si z doprovodné diskety jeden z konfiguračních souborů, které jsou tam pro vás připraveny v adresáři *CONFIG*. O tom, který z připravených konfiguračních souborů je pro vás ten pravý, se dozvíte z textového souboru *NEJDRIVE.CTI*. Vyberte si soubor, který odpovídá vaší konfiguraci, a uložte jej do adresáře *KURS* pod jménem *TCCONFIG.TC*. Budou-li vám úpravy vyhovovat, můžete konfigurační soubor přemístit z adresáře KURS do adresáře, v němž máte uložen překladač (přesněji přepsat tímto souborem původní konfigurační soubor), a využívat tak nových možností i v jiných projektech.

1.8 Transfery

Příjemným rozšířením, s nímž bychom vás chtěli seznámit, jsou programy spustitelné přímo z IDE neboli transfery. Jako transfery můžeme definovat soubory *.BAT*, *.COM* a *.EXE*. Jako transfer nelze použít interní příkaz operačního systému – např. DIR nebo CLS. Chceme-li takový příkaz spouštět z IDE, musíme vytvořit dávkový soubor (*.BAT*), z kterého jej vyvoláme, a tento dávkový soubor pak zařadit mezi transfery.

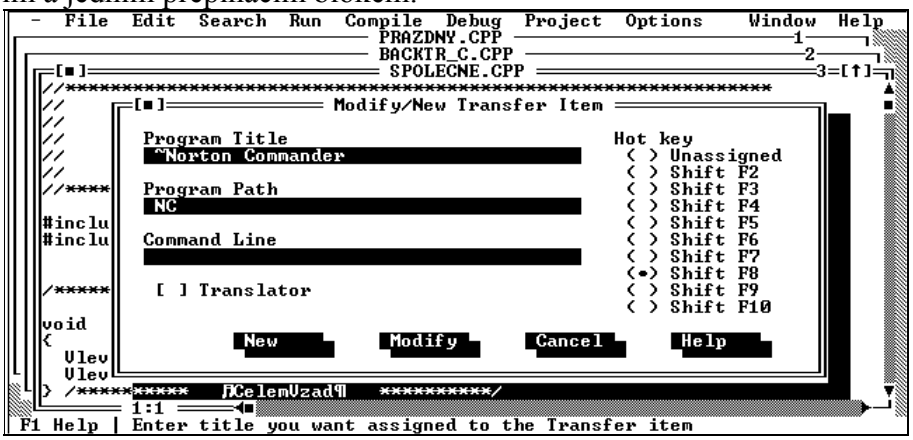
Transfery v Borland C++

Zvolte v základní nabídce volbu Options a v jejím roletovém menu si vyberte volbu Transfer. Systém otevře dialogové okno se seznamem definovaných transferů. Pokud jste si nahráli rozšiřující konfigurační soubor z doprovodné diskety, najdete mezi nimi i transfer pojmenovaný Smaž mne. Najeďte na něj řádkovým kurzorem a stiskem D (horká klávesa pro Delete) jej smažte.



Obr. 1.5 Okno transferů

Nyní najeďte na transfer ~Norton Commander a stiskem ENTER potvrďte volbu Edit. Systém otevře další dialogové okno (že se původní neztratilo, poznáte, když s novým oknem popojedete) s třemi vstupními, jedním nastavovacím a jedním přepínacím blokem.



Obr. 1.6 Okno pro úpravu transferu

Program Title

Ve vstupní řádce Program Title se zadává text, který se objeví jako položka v nabídce (menu) vyvolávané stiskem kombinace ALT+MEZERA, přičemž tilda (znak ~) oznamuje, že následující znak bude v nabídce transferů zvýrazněn, a bude tedy možno s jeho pomocí danou položku zvolit.

Program Path

Do vstupní řádky Program Path zadáváme název programu. Pokud cesta k programu není součástí systémové proměnné *PATH*, je třeba název programu zadat kompletní, tj. včetně disku a cesty. Doporučuji vám však zadat kompletní cestu vždy. Pokud na svém počítači nepoužíváte Norton Commander nebo jej máte v jiném adresáři, budete asi tuto položku opravovat.

Command Line

Do vstupního bloku Command Line zadáváme parametry, které programu zadáváme v příkazovém řádku. V našem případě je blok prázdný, pouze žádáme systém, aby se nás před vyvoláním programu zeptal, zda nechceme modifikované soubory nejprve uložit.

Translator

Volba Translator slouží zařazení transferu mezi překladače aplikovatelné na editované projekty – prozatím ji nebudeme používat.

Hot Key

Blok přepínačů Hot Key slouží k definici horké klávesy, kterou můžeme program přímo vyvolat, aniž bychom museli procházet stromem nabídek.

New

Tlačítkem New, které je implicitní (tj. reaguje na ENTER), ukládáme vytvořený záznam jako novou položku do seznamu transferů (před položku editovanou).

Modify

Tlačítkem Modify vytvořený záznam ukládáme místo původní podoby editované položky.

Transfery v Turbo Pascalu 7.0

Také Turbo Pascal umožňuje spouštět přímo z IDE jiné programy, ale pouze ve verzi 7.0. Zde se však pro tyto programy používá označení *Tools* – nástroje.

Zadáme-li v základní nabídce příkaz Options | Tools..., objeví se dialogové okno podobné jako na obr. 1.5. Bude se lišit pouze titulkem (*Tools* místo *Transfer*) a bude obsahovat navíc tlačítko New (nový nástroj). Jinak je postup přidání nového transferu podobný jako v C++, proto jej nebudeme podrobně popisovat.

Tyto nástroje lze pak volat ze základní nabídky Tools nebo pomocí přiřazené horké klávesy.

Uložení změn

Chceme-li, aby změny v nastavení prostředí zůstaly v platnosti i při příštím spuštění překladače, musíme je uložit v Borland C++ příkazem Options | Save ... při nastavené možnosti Environment. V Turbo Pascalu 7.0 použijeme příkaz Options | Save \...\BIN\BP.TP.

1.9 Příprava překladače

Předpokládáme, že jste si již zkopírovali soubory z doprovodné diskety do stejnojmenných adresářů, nastavili jako aktuální adresář *KURS* a spustili svůj překladač.

Specifika Pascalu

Otevřete soubor *PRAZDNY.PAS*, který najdete v adresáři *PAS* (úplná cesta zní *\KURS\PAS*). Tento soubor pro nás bude v prvých kapitolách výchozím souborem pro tvorbu našich programů. Abyste si jej zachovali pro další použití, uložte načtený text do souboru pod jménem např. *\KURS\PAS\1PAS.PAS*. Udělejte to tak, že stiskem ALT+F otevřete nabídku File a v ní zvolíte akci Save as. Systém otevře dialogové okno, v němž se nás zeptá na název, pod nímž chceme text z aktuálního editačního okna uložit. Napište tedy do vstupního bloku *PAS\1PAS.PAS* (jsme v adresáři *\KURS*) a svoji volbu potvrďte. Nyní si vyzkoušejte své znalosti editoru a nahraďte v souboru všechny znaky @ (šnek neboli zavináč) jménem souboru, tj. textem „*1PAS*“. Takto opravený soubor uložte (F2).

Zbývá ještě nastavit konfiguraci překladače.

1. Otevřete dialogové okno Options | Compiler a zkontrolujte, zda jsou:

   ✧ nastaveny volby Force far calls, Extended syntax, Debug information a Local information (tj. zda je v příslušných závorkách X),

   ✧ potlačeny volby Overlays allowed a Complete boolean eval.

2. Nastavené volby potvrďte stiskem tlačítka OK.

3. Otevřete dialogové okno Options | Linker a zkontrolujte, zda je přepínač Map file nastaven na Off a zda je přepí- přepínač Link buffer nastaven na

Memory. Obě tato nastavení zrychlují práci překladače. Nastavené volby potvrďte stiskem tlačítka OK.

4. Otevřete dialogové okno Options | Directories. Najdete v něm čtyři vstupní řádky, z nichž tři budou pro nás zajímavé.

   ✧ Vstupní řádka EXE & TPU directory definuje adresář, do nějž se ukládají soubory s přeloženými programy, a my do něj zadáme text EXE (jsme v adresáři *KURS* – můžete ale zadat i \KURS\EXE).

   ✧ Vstupní řádka Unit directories definuje adresář, v němž bude překladač hledat zdrojové texty, a my do něj zadáme text PAS nebo \KURS\PAS. Nastavené volby potvrdíme stiskem tlačítka OK.

5. Otevřete dialogové okno Options | Debugger a zkontrolujte v něm, zda je nastavena volba Integrated a zda je přepínač Display swapping nastaven na Smart. Nastavené volby potvrďte stiskem tlačítka OK.

6. Otevřete dialogové okno Options | Environment | Preferences a zkontrolujte, zda jsou nastaveny všechny volby v bloku Auto save (editované soubory, vzhled obrazovky i nastavení jednotlivých voleb se bude před spuštěním programu a před ukončením práce překladače automaticky ukládat) a zda je přepínač Desktop file nastaven na hodnotu Current directory. Nastavené volby potvrďte stiskem tlačítka OK.

7. Otevřete dialogové okno Options | Environment | Editor a zkontrolujte, zda jsou nastaveny volby Create backup file (vytvářej záložní soubory s příponou *.BAK*) a Autoindent mode (při přechodu na následující řádek se kurzor nastaví na pozici, na níž je první znak řádku předchozího). Nastavení obou těchto voleb považujeme za užitečné. Nastavení ostatních voleb může být věcí vkusu. Nastavené volby potvrďte stiskem tlačítka OK.

8. Používáte-li Turbo Pascal 6.0, otevřete dialogové okno Options | Environment | Save Options, zadejte do vstupní řádky text \KURS\TURBO\TP a nastavení potvrďte stiskem tlačítka OK. Používáte-li Turbo Pascal 7.0, uložte nastavení stejně jako v případě transferů.

Specifika C++

Otevřete soubor *PRAZDNY.CPP*, který najdete v adresáři *CPP* (úplná cesta zní *\KURS\CPP*). Tento soubor pro nás bude v prvních kapitolách výchozím souborem pro tvorbu našich programů. Abyste si jej zachovali pro další použití, uložte načtený text do souboru např. pod jménem *\KURZ\CPP\1CPP.CPP*. Udělejte to tak, že stiskem ALT+F otevřete na-

nabídku File a v ní zvolíte akci Save as. Systém otevře dialogové okno, v němž se nás zeptá na název, pod nímž chceme text z aktuálního editačního okna uložit. Napište tedy do vstupního bloku *CPP\1CPP.CPP* (jsme v adresáři *\KURS*) a svoji volbu potvrďte.

Nyní si vyzkoušejte své znalosti editoru a nahraďte v souboru všechny znaky @ (šnek) jménem souboru, tj. textem *1CPP*. Takto opravený soubor uložte (F2).

Kromě datového souboru si musíte připravit ještě tzv. projektový soubor (zkráceně projekt), který překladač potřebuje, aby věděl, že kromě vámi napsaného programu má použít ještě jiné zdroje – v našem případě půjde o soubor *KAREL.OBJ*, který najdete na doprovodné disketě. Na doprovodné disketě máte i projektové soubory, avšak není vyloučeno, že některá nastavení v nich nebudou odpovídat vaší konfiguraci. Konkrétně se jedná o nastavení adresářů, v nichž má systém hledat potřebné soubory. Ukážeme si proto, jak na to:

Nejprve zvolte v základním menu volbu Options a v jejím roletovém menu volbu Directories. Systém otevře dialogové okno s třemi vstupními řádkami.

V první jsou uvedeny cesty, odkud systém načítá některé zdrojové programy. Najdete zde text `C:\B\INCLUDE;CPP`. Jeho první část (před středníkem) by měla ukazovat cestu k adresáři, v němž jsou tzv. hlavičkové soubory dodávané se systémem. Je-li cesta nastavena správně, najdete v adresáři, do nějž cesta ukazuje, větší množství souborů s příponou *.H* (pokud jste tam sami nepřidali soubory další). Navíc tento adresář obsahuje podadresář *SYS*, v němž je ještě několik dalších souborů s příponou *.H*. Pokud tomu tak není, musíte si odpovídající adresář na disku sami najít a nastavit cestu na něj. Druhá část (za středníkem) označuje adresář `CPP`, do nějž budeme ukládat zdrojové programy my.

V druhém vstupním bloku najdete text `C:\B\LIB`. Jedná se o cestu ke knihovním souborům dodávaným se systémem. Pokud jste upravovali adresář hlavičkových souborů, je velmi pravděpodobné, že budete muset opravit i adresář knihoven. Je vysoce pravděpodobné, že adresář *LIB* bude mít stejný rodičovský adresář jako adresář *INCL* z minulého odstavce. Pokud je tomu jinak, pravděpodobně do systému někdo zasahoval a musíte jej proto požádat o individuální konfiguraci.

2. Pravidla zápisu syntaxe jazyka

Syntax jazyka definuje ČSN 369001/7–1987 jako „pravidla pro vytváření přípustných řetězců jazyka, užívaná bez ohledu na jejich význam“. Oborová encyklopedie výpočetní a řídicí techniky nám říká, že syntaxí se nazývá „souhrn formálních pravidel, která určují strukturu přípustných kombinací symbolů v daném jazyce“. Pokud bychom netrvali na tak učeném tónu, asi bychom řekli, že syntaktická pravidla jsou pravidla, jak se to píše.

Přesná definice syntaxe je při programování naprostou nezbytností. Přirozený jazyk však často připouští několik výkladů téže věty. Proto bývá v manuálech a učebnicích programovacích jazyků zvykem definovat syntax jazyka ještě nějakou jinou, exaktnější formou. V učebnicích a manuálech Pascalu se pro tento účel používají velmi často syntaktické diagramy, v manuálech a učebnicích jazyka C++ se zase většinou dává přednost textovému zápisu. Aby byl popis obou jazyků jednotný, budou v této knížce všechny syntaktické definice uváděny v textové podobě.

Než však přejdeme k popisu pravidel obou zápisů syntaxe jazyka, musíme si nejprve vysvětlit dva pojmy. **Neterminální symbol** je něco, co má pro překladač ještě nějakou vnitřní strukturu, tj. něco, co má definovanou syntax. Naproti tomu **terminální symbol** je symbol, který je pro překladač již dále nedělitelný – takový lexikální atom. Každému je asi jasné, že terminálním symbolem bude znak, ale existují i terminální symboly, které mají více znaků. Mezi ně patří tzv. **klíčová slova**, což jsou slova, která mají v daném jazyce předem daný pevný význam (např. klíčové slovo **begin** označuje v Pascalu začátek nějaké konstrukce), a víceznakové speciální symboly (např. symbol **:=** označující v Pascalu přiřazení).

Textový zápis syntaxe daného neterminálního symbolu (programové konstrukce) budeme vytvářet podle následujících pravidel:

- ✧ Na prvním řádku bude kurzivou uveden název definovaného symbolu (programové konstrukce) ukončený dvojtečkou nebo dvěma rovnítky. Dvojice rovnítek se použije v případě, kdy mezi jednotlivými prvky v definici **nesmějí** být mezery, kdežto dvojtečka se použije v případě, kdy mezery mezi jednotlivými prvky v definici být mohou a v případě možného sloučení sousedních prvků dokonce být musejí .
- ✧ Za touto dvojtečkou (resp. dvojicí rovnítek) může být vysvětlující komentář, jehož syntax bude odpovídat syntaxi jazyka, jehož programovou kon-

strukci definujeme[5].

- ✧ Na dalších řádcích pak budou uvedeny jednotlivé alternativní možnosti. Pro zvýšení přehlednosti je vůči názvu symbolu odsadíme o 2 znaky.
- ✧ Pokud se zápis dané možnosti nevejde na jeden řádek, bude pokračovací řádek označen tučným znakem + v prvním sloupci. Pro zvýšení přehlednosti budou pokračovací řádky odsazeny oproti prvnímu řádku dané možnosti, a to také o 2 znaky.
- ✧ Pokud se bude v definici vyskytovat větší počet jednoduchých alternativ, mohou být uvedeny na jednom řádku (s případnými pokračujícími řádky), který bude uvozen tučným **1 z:** (tedy jedna možnost z uvedených).
- ✧ Uvnitř definic jednotlivých alternativ budou používána následující pravidla:

  a) Terminální symboly, tj. symboly, které se v nezměněné podobě mohou objevit v programu, se zapisují **tučně**.

  b) Neterminální symboly, tj. symboly, které mají někde svoji vlastní syntaktickou definici, se zapisují ***kurzívou***.

  c) Volitelné části, tj. části, které v daném místě mohou, ale nemusejí být vloženy, se vkládají do hranatých závorek ***[a]***.

  d) Za části, které se mohou opakovat, se píše index $_{opak}$.

Jako příklad bychom si mohli uvést syntaktickou definici věty běžného jazyka.

Věta:
    SZVP [[,] Slovo] $_{opak}$ *Zakončení*

Slovo:
    SZVP
    SZMP

Zakončení:
    1 z: . ? !

SZVP== *// Slovo začínající velkým písmenem*
    Velké_písmeno [malé_písmeno] $_{opak}$

[5] V Pascalu uzavíráme komentář mezi složené závorky { a }, v C++ začíná komentář dvěma lomítky a pokračuje do konce řádku.

SZMP== *// Slovo začínající malým písmenem*
Malé_písmeno [malé_písmeno]$_{\text{opak}}$

Velké_písmeno:
1 z: **_ A Á Ä B C Č D Ď E É Ě F G H I J K L Ľ Ĺ M N Ň**
+ **O Ó Ô Ö P Q R Ř Ŕ S Š T Ť U Ú Ů Ü V W X Y Ý Z Ž**

Malé_písmeno:
1 z: **_ a á ä b c č d ď e é ě f g h i j k l ľ ĺ m n ň**
+ **o ó ô ö p q r ř ŕ s š t ť u ú ů ü v w x y ý z ž**

K předchozí definici doplníme pár poznámek:

- ✧ V této definici věty platí, že mezi čárkou a předcházejícím, resp. následujícím slovem může, ale nemusí být mezera, protože lze vždy jednoznačně určit, kde slovo končí, resp. začíná (čárka nemůže být součástí slova).
- ✧ Pokud nejsou jednotlivá slova v definici věty oddělena čárkou, musejí být oddělena alespoň jednou mezerou, protože bychom jinak nemohli poznat, kde první slovo končí a druhé začíná (výjimkou je případ, kdy druhé slovo začíná velkým písmenem, ale tuto možnost pro zjednodušení velkoryse přehlédneme).
- ✧ Dvojice rovnítek v definicích *SZVP* a *SZMP* jednoznačně určuje, že mezi jednotlivými písmeny tvořícími slova nesmí být mezera.
- ✧ Znak + v definicích malého a velkého písmena označuje pokračovací řádek definice položky, a není proto zařazen mezi definované znaky.
- ✧ Komentáře k definicím jsme zapsali způsobem obvyklým v C++ (komentář je vše od dvojice lomítek do konce řádku).

3. Robot Karel

V této knize se budeme učit vytvářet algoritmy pomocí robota Karla. Podrobné povídání o této vyučovací pomůcce najdete v dodatku, na který budeme průběžně odkazovat. Zde si zatím řekneme jen tolik, kolik je nezbytné, abychom mohli začít programovat.

Opusťme již ale obecné řeči a povězme si něco o robotu Karlovi. Robot Karel (přesněji řečeno svět robota Karla) je vyučovací pomůcka, kterou využijeme v průběhu následujícího výkladu základů algoritmizace. Svět robota Karla je navržen tak, abychom v něm zejména v počátečních fázích výuky mohli demonstrovat probíranou látku na co nejjednodušších programech a minimalizovat přitom rozsah témat, která je nutno vysvětlit, přestože s probíranou látkou přímo nesouvisejí.

Robot Karel se „narodil“ v sedmdesátých letech na Standfordské univerzitě v USA, kde ho jeho autor V. A. Pettis začal používat jako vyučovací pomůcku ve vstupních kursech programování. Karel měl Pettisovi pomoci odnaučit studenty zlozvykům, které mohli získat během svých předchozích programátorských snažení, a vštípit jim zároveň základy moderního přístupu k programování. Jméno dostal po našem spisovateli Karlu Čapkovi, a proto se i v Americe jmenuje Karel, a nikoliv Charles. U nás se jeho používání rozšířilo v polovině osmdesátých let v různých zájmových útvarech zabývajících se programováním a koncem osmdesátých let se tato vyučovací pomůcka dostala i do školních osnov.

Základní postavou celého systému je robot Karel, který žije na dvorku (někteří říkají ve městě), který si můžeme představit jako šachovnici, kolem níž je postavena zeď. Zdi mohou být i na jednotlivých polích této šachovnice. Karel se umí po dvorku (městě) pohybovat a umí na jeho jednotlivá políčka pokládat značky a opět je z nich sbírat. Na počátku výuky rozumí pouze čtyřem povelům (budeme jim říkat **primitiva**) a umí otestovat sedm základních (primitivních) podmínek. My budeme učit Karla vykonávat složitější příkazy a prostřednictvím toho se sami učíme programovat.

Robot Karel se tedy po dobu studia této knížky stane i vaším společníkem. Budeme pro něj postupně sestavovat programy, které budou řešit zadané úlohy. Nejprve spolu, ale řada úloh bude určena i pro vaše vlastní řešení. U těchto úloh by vám ukázková řešení měla sloužit spíše pro porovnání než jako nápověda.

Zápis pozice

Abychom se mohli stručně a jednoznačně domluvit o tom, co Karel má za úkol, zavedeme si následující označení. Když budeme určovat, co Karel má či nemá vykonat, budeme velice často hovořit o jeho pozici. Tuto pozici nám bude charakterizovat výraz

```
r / s : z - Směr
```

kde

`r` je číslo řádku, na němž se Karel nachází,

`s` je číslo sloupce, v němž se Karel nachází,

`z` je počet značek na políčku pod Karlem – pokud nebude důležitý, nebudeme jej ani uvádět,

`Směr` je označení směru, do nějž je Karel natočen. Někdy budeme pro zkrácení místo plného označení směru (Východ, Západ, Sever, Jih) používat pouze jednopísmenné zkratky (V, Z, S, J).

Při standardním nastavení je Karel na počátku v pozici `0/0:0-V`, tj. stojí v nultém řádku a nultém sloupci (v levém dolním rohu dvorku), nemá pod sebou žádnou značku a je otočen na východ. Tuto pozici budeme nazývat **standardní pozice**.

Základní příkazy

Již jsme si řekli, že na počátku rozumí Karel pouze čtyřem povelům:

1. **Krok**, po němž se přesune na sousední políčko ve směru, do nějž je natočen. To se mu samozřejmě může podařit pouze v případě, že před ním není zeď. V opačném případě ohlásí chybu.
2. **VlevoVbok**, po němž se otočí o 90 stupňů vlevo.
3. **Polož**,[6] po němž položí na políčko, na němž právě stojí, další značku. Pokud však již byl na tomto políčku maximální povolený počet značek (v našich úlohách to bude většinou 9), ohlásí chybu.
4. **Zvedni**, po němž pod sebou zvedne jednu značku. Není-li na políčku, na němž Karel právě stojí, žádná značka, ohlásí chybu.

[6] Pro zvýšení čitelnosti budeme jména příkazů pro robota Karla uvádět s diakritickými znaménky, i když ve vlastním programu tato písmena používat nelze. To znamená, že příkaz **Polož** se bude v programu jmenovat **Poloz** atd.

Kromě těchto čtyř základních primitiv „zná“ Karel ještě pomocný příkaz **Krát**, který jsme přidali proto, že nám může v řadě případů ušetřit mnoho psaní. Syntax příkazu je pro oba jazyky až na závěrečný středník shodná: do závorky za příkaz napíšeme počet opakování, čárku a jméno příkazu, který se má zopakovat tolikrát, kolik udává počet opakování. Formálně bychom ji mohli zapsat (*Ident_příkazu* je jméno příkazu, který chceme opakovat):

Příkaz_Krát:
    Krat (*Číslo*, *Ident_příkazu*)

Příkaz_Krát:
    Krat (*Číslo*, *Ident_příkazu*) ;

Dalším pomocným příkazem je příkaz **Čekej**, po němž počítač vykreslí na obrazovku aktuální stav Karlova dvorku, tj. Karlovu pozici a rozmístění zdí a značek, a poté zastaví provádění programu a počká na stisk libovolné klávesy. Pokud stiskneme klávesu ESC, počítač provádění programu ukončí, v opačném případě bude v plnění programu pokračovat.

Dvorek

Abychom se mohli co nejvíce soustředit na řešení vlastních úloh, musíme mít možnost připravit s minimální námahou prostředí, v němž bude Karel zadané úkoly plnit. Protože řada úloh má své specifické požadavky, které bychom pomocí čistě programátorských prostředků vysvětlených v této knize jen obtížně plnili, poskytuje systém možnost zadat veškeré parametry Karlova dvorku prostřednictvím textového souboru. Informace o současném stavu Karlova dvorku můžete v průběhu programu kdykoliv uložit do souboru a opět je z tohoto souboru načíst.

Soubor s informacemi o Karlově dvorku je obyčejný textový soubor (doporučuji vám pro něj příponu *.DVO*), který si můžete sami připravit libovolným textovým editorem. Musíte pouze dodržet základní formát, který je popsán v příloze v části **Dvorek**.

4. První kroky

Každý program (myslíme tím program v klasickém procedurálním jazyku, jakým je např. Basic, C++ nebo Pascal) se skládá ze dvou navzájem se prolínajících složek: z deklarací používaných dat a z posloupností příkazů. Jak deklarace dat, tak příkazy musíme umět zapsat na správné místo a správným způsobem.

Už jsme si řekli, že výklad práce s daty si ponecháme do dalšího dílu a po celý zbytek tohoto dílu se soustředíme pouze na příkazy. V této kapitole budeme ještě skromnější – veškeré naše úsilí soustředíme na část programu, která se nazývá **hlavní program**.

4.1 Prázdný program – překlad a sestavení

V kapitole o zásadách práce s IDE jsme si připravili konfigurační a projektové soubory pro další práci. Využijeme toho, že do připraveného prázdného programu nemusíme vůbec zasahovat, protože podpůrný program Karel je napsán tak, že při inicializaci hned nakreslí na obrazovku výchozí podobu Karlova dvorku. Zadáme proto počítači nejjednodušší možný program: prázdný program neobsahující žádný příkaz – ten vlastně již máme připraven. Stiskněte klávesu ALT-F9 (horká klávesa pro [Compile | Compile]), čímž počítači přikážete, aby program přeložil**.** Pokud jste ve zdrojovém souboru opravdu nic nezměnili, mělo by po úspěšném překladu být ve středu obrazovky rozsvíceno kompilační okno a v jeho posledním řádku by měl v Pascalu blikat nápis

```
Compile successful : Press any key
```

a v C++ nápis

```
Success : Press any key
```

Přeložený program však ještě není schopen spuštění. Z hlediska počítače se totiž ještě nejedná o program, ale pouze o modul, který se odvolává na řadu jiných modulů. Tento modul je třeba nejprve sestavit (spojit) s moduly, na něž se odvolává. Toho dosáhneme stiskem klávesy F9 (horká klávesa pro [Compile | Make]).

Poznámka 1

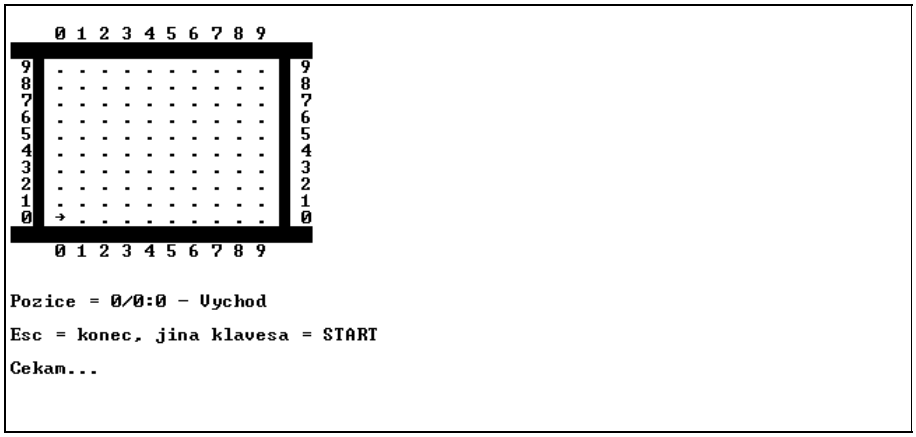
Tím, co je to vlastně ten modul a jaké má vlastnosti, se budeme podrobněji

zabývat později. Prozatím si řekneme, že modul je relativně samostatná část programu – např. to, co vytváříme. Dalším modulem je soubor KAREL.OBJ, resp. KAREL.TPU, který jste obdrželi na disketě. Kromě toho naše programy používají řadu dalších modulů, které jsou součástí systému a které se v případě potřeby připojí automaticky.

Poznámka 2

Turbo Pascal se při překladu a sestavování chová trochu nestandardně. Pokud překládaný modul obsahuje hlavní program (a to, jak poznáme dále, je v našem případě již splněno), překladač jej v rámci překladu zároveň sestaví (spojí) s volanými moduly ve výsledný program. V tomto směru mají pascalisté život jednodušší.

Jakmile program přeložíme a sestavíme, budeme ho asi chtít i spustit. Toho dosáhneme stisknutím klávesy CTRL-F9 (horká klávesa pro [Run | Run]). Lze očekávat, že po spuštění většinou pouze zjistíte, že program proběhl tak ďábelským tempem, že jste ani nestačili zaregistrovat, co že to počítač vlastně udělal. Snadná pomoc. Stiskem ALT-F5 vyměníte standardní obrazovku IDE za obrazovku, do níž váš program zaznamenával výsledky svého snažení. Obrazovka by měla vypadat tak, jak je znázorněno na obrázku 4.1.



Obr. 4.1 Výchozí stav Karlova dvorku

Stiskem jakéhokoli tlačítka se vrátíte zpět do IDE.

4.2 Hlavní program

Pokusme se nyní požádat Karla, aby něco dělal. To, co budeme vytvářet, bude tzv. **hlavní program**. V Pascalu je hlavním programem závěrečná část zdrojového textu ohraničená klíčovými slovy **begin** a **end**[7]. V C++ slouží ve funkci hlavního programu procedura nazvaná *main*. Co je to procedura si vysvětlíme později. Prozatím nám bude stačit, když budeme vědět, že ji v souboru *PRAZDNY.CPP* najdeme připravenou a že její tělo, tj. místo, kam budeme psát náš program, je ohraničeno složenými závorkami.

Znaky „{“ a „}“ (složené závorky) v C++ a klíčová slova **begin** a **end** v Pascalu budeme nazývat **příkazové závorky**. Pascalisty musíme zklamat zjištěním, že na rozdíl od C++ neexistuje v editoru pro Pascal způsob, jak k dané příkazové závorce najít její odpovídající protějšek.

Nejprve napíšeme jednoduchý program tvořený jediným příkazem *Polož*, po němž by měl Karel pod sebe položit značku. Příkaz vepíšeme do hlavního programu mezi příkazové závorky. V Pascalu bude mít tento program podobu:

```
{ ********* Hlavní program ********* }
begin
  Poloz;
end.
```

a v C++

```
/*************  Hlavní program  *************/
void  /*****/main /*****/ ()
{
  Poloz();
}/**********  main  **********/
```

Jak jste si mohli všimnout, v Pascalu zapisujeme příkaz obdobně jako v běžném jazyce (dokonce nezáleží ani na tom, zda používáme velká nebo malá písmena), pouze vynecháme diakritická znaménka (háčky, čárky, přehlásky a kroužky).

V C++ však

- ✧ záleží na velikosti písmen (*poloz*, *Poloz* a *poloZ* jsou tři různé příkazy),
- ✧ musíme za názvem příkazu zapsat prázdné kulaté závorky,
- ✧ součástí příkazu je závěrečný středník (dokud nenapíšeme středník, nejedná se o příkaz, ale o výraz – ale o tom až později).

[7] Za **end**, které ukončuje hlavní program v Pascalu, musí být tečka.

Poznámka
Středník se používá v obou jazycích, v každém má ale jiný význam. V Pascalu slouží k oddělování jednotlivých příkazů, a proto se před **end** *uvádět nemusí, protože tam již není co oddělovat. Naproti tomu v C++ se zapsáním závěrečného středníku teprve stává z výrazu příkaz. V C++ je tedy středník součástí příkazu.*

Stiskněte CTRL-F9, aby se program spustil. Všimněte si, že počítač pozná, že od posledního překladu došlo ve zdrojovém textu ke změně**,** a před spuštěním program nejprve přeloží a sestaví.

Obdobnou „inteligencí“ se vyznačuje i klávesa F9. Překladač se nejprve snaží zjistit, nebyl-li zdrojový text některého z modulů, které vstupují do hry, od posledního překladu pozměněn. Pokud byl, přeloží ho. Poté pátrá, zda byl od posledního sestavování některý z modulů znovu přeložen. Najde-li takový, znovu program z jednotlivých modulů sestaví. Zjistí-li naopak, že žádný z modulů nepotřebuje znovu přeložit standardní pozice a ani nebyl od posledního sestavování přeložen, odvodí z toho, že není proč se namáhat. Neudělá nic a pouze ohlásí, že program je *Up to date*, tedy že přeložený program odpovídá aktuálnímu stavu zdrojových souborů.

Vraťme se ale k našemu jednopříkazovému programu. Prohlédnete-li si nyní výslednou obrazovku (připomínáme: vyvoláte ji stiskem ALT-F5**)**, zjistíte, že se na dvorku nic nezměnilo. To proto, že značka je schovaná pod Karlem a o její přítomnosti se dozvíme pouze z informačního řádku, kde můžeme číst:

```
0/0:1 - Východ
```

4.3 Ladění

Zadejme nyní složitější program, při němž necháme Karla položit značku a poodejít ze své pozice, abychom se mohli podívat, zda tam po něm opravdu nějaká značka zbyla. Pak jej dovedeme zase zpět, necháme jej značku opět zvednout a opět s ním poodejdeme, abychom viděli, že ji doopravdy zvedl. Pascalský zdrojový text bude mít tvar

```
{**********  Hlavní program  **********}
begin
  Poloz;
  Krok;
  VlevoVbok;
  VlevoVbok;
  Krok;
  Zvedni;
  VlevoVbok;
  VlevoVbok;
  Krok;
```

```
end.
```

a zdrojový text pro C++ tvar

```
/************  Hlavní program  ************/
void  /*****/main /*****/ ()
{
  Poloz();
  Krok();
  VlevoVbok();
  VlevoVbok();
  Krok();
  Zvedni();
  VlevoVbok();
  VlevoVbok();
  Krok();
}/**********  main  **********/
```

Protože se již jedná o delší prográmek, v němž nemůžeme vyloučit překlep či jinou chybu, tak si dříve, než se podíváme, jak bude Karel program plnit, ukážeme, jaké máme prostředky pro nalezení a opravu případných chyb. Abyste si své chyby sjednotili, smažte v prvém příkazu první písmeno (zbude `oloz`) a v druhém řádku vynechte středník. Céčkaři si navíc v pátém řádku smažou obě závorky (zbude `Krok;`) a v předposledním (osmém) řádku vynechají otevírací závorku (zbude `VlevoVbok);`). Programy dostanou po těchto úpravách následující podobu:

```
{**********  Hlavní program  **********}
begin
  oloz;
  Krok
  VlevoVbok;
  VlevoVbok;
  Krok;
  Zvedni;
  VlevoVbok;
  VlevoVbok;
  Krok;
end.
```

a zdrojový text pro C++ tvar

```
/************  Hlavní program  ************/
void  /*****/main /*****/ ()
{
  oloz();
  Krok()
  VlevoVbok();
  VlevoVbok();
  Krok;
  Zvedni();
  VlevoVbok();
```

```
  VlevoVbok);
  Krok();
}/********** main **********/
```

Stiskneme-li nyní klávesu F9, počítač se pokusí program přeložit, a protože se mu to vzhledem k chybám nepodaří, nebude se o jeho sestavení ani pokoušet.

Reakce počítače na chyby se budou u C++ a Pascalu lišit, a proto si o nich povíme zvlášť.

Překladač Pascalu je koncipován tak, že po každé chybě zastaví překlad a nastaví kurzor na místo předpokládané chyby. První chybou je vynechané prvé písmeno v příkazu **Poloz**. Počítač se na tomto příkazu zastaví a oznámí vám, že takový identifikátor nezná. Vložíme tedy zpět vynechané písmeno a spustíme překlad znovu.

Tentokrát počítač skutečnou chybu přeběhne, nastaví kurzor na řádek následující za řádkem s vynechaným středníkem a oznámí nám, že na daném místě očekával středník. Vrátíme se tedy o řádek zpět, doplníme středník a spustíme další pokus o překlad. Pokud jste v programu neudělali žádnou neplánovanou chybu, měl by překlad tentokrát proběhnout úspěšně až do konce včetně následného sestavení.

Na rozdíl od překladače Turbo Pascalu umožňují překladače jazyka C++ nastavit počet chyb a počet varovných zpráv, při jejichž dosažení se další překlad zastaví. Tento počet se nastavuje v dialogovém okně [Options | Compiler | Messages | Display]. Počet chyb nastavujeme ve vstupním bloku Errors a počet varovných zpráv ve vstupním bloku Warnings. Implicitní nastavení dodávané firmou je 25 chyb a 100 varovných zpráv. Někteří programátoři preferují nižší počty – např. jeden z nás má na svém počítači nastaven počet chyb na 5 a počet varovných zpráv na 10.

Pokud jste program zapsali přesně tak, jak jsme vám doporučili, ohlásí vám překladač 3 chyby (*Error*) a jednu varovnou zprávu (*Warning*) a v posledním řádku kompilačního okna naleznete zprávu:

```
Errors : Press any key
```

Stiskneme nyní některou řadovou klávesu (tj. nikoliv přeřaďovač – nejlépe mezeru či *Enter*) a systém otevře okno zpráv (*Messages*), nastaví je jako aktuální, zapíše do něj všechna chybová hlášení a řádkový kurzor umístí na první zprávu. Je-li hlášení delší a nevejde-li se proto na řádek, můžeme po něm rolovat pomocí šipek vpravo a vlevo, případně pomocí kláves HOME a END. Je-li chybových hlášení více (a to v našem případě je), můžeme se po

nich pohybovat pomocí šipek vzhůru a dolů a pomocí kláves PAGEUP (PGUP) a PAGEDOWN (PGDN).

Pokud se zvýrazněná chybová zpráva týká překladu (to je náš případ), snaží se systém souběžně s vaším pohybem v okně zpráv označovat v editačním okně místa, k nimž se daná chyba či varovná zpráva vztahuje (není-li otevřeno editační okno se souborem, v němž byla nalezena chyba, nic se neoznačuje). My se pak můžeme stiskem klávesy ENTER přesunout do editačního okna se souborem, v němž byla nalezena chyba. Není-li toto okno ještě otevřeno, systém je otevře; ve volbách editoru si můžeme nastavit, zda se má otevřít nové okno (je-li přepínač Source tracking v dialogovém okně [Options | Environment | Preferences] nastaven na hodnotu New window), nebo zda se má přepsat obsah naposledy otevřeného aktuálního okna (přepínač Source tracking v dialogovém okně [Options | Environment | Preferences] nastaven na hodnotu Current window). V tomto okně najdete kurzor na předpokládaném místě chyby. Po vstupu do editačního okna s podezřelou částí programu se chybové hlášení zopakuje v posledním řádku editačního okna a tam setrvá do prvního stisku klávesy.

Stiskněte tedy ENTER a počítač vám nastaví kurzor na řádek s první chybou a bude vám tvrdit, že uvedená funkce musí mít prototyp. To může znamenat dvě věci: buď jste opravdu pouze zapomněli napsat prototyp (není tomu tak – co to je prototyp si však povíme někdy později), nebo taková funkce vůbec neexistuje – to je náš případ. Doplníme proto chybějící první písmeno a stiskem klávesy ALT-F8 se přesuneme na předpokládané místo další chyby.

Stejně jako v Pascalu i překladač C++ tuto chybu přeběhne a umístí kurzor na následující řádek. Na rozdíl od Pascalu však bude tvrdit, že našel chybu ve výrazu. (Podíváte--li se na to, co jsme si říkali o funkci středníku v C++, zjistíte, že má vlastně pravdu). Vrátíme se tedy o řádek výše, doplníme chybějící středník a pomocí ALT-F8 se přesuneme na další chybu.

Zde nám počítač oznámí, že našel neočekávanou uzavírací závorku – jinými slovy, že k nalezené uzavírací závorce chybí příslušná závorka otevírací. Doplníme ji a půjdeme dále.

Tentokrát vám již počítač neohlásí žádnou chybu, ale pouze vás varuje, že příkaz na dané řádce nebude mít žádný efekt. A má pravdu, protože pokud v jazyku C++ neuvedeme ve volání příkazu za názvem příkazu závorky, vyvoláme tím jinou činnost, která v tuto chvíli opravdu nemá žádný efekt. Doplníme tedy chybějící závorky a pokusíme se o nové přeložení a sestavení programu. Pokud jste v programu neudělali žádnou další chybu, mělo by

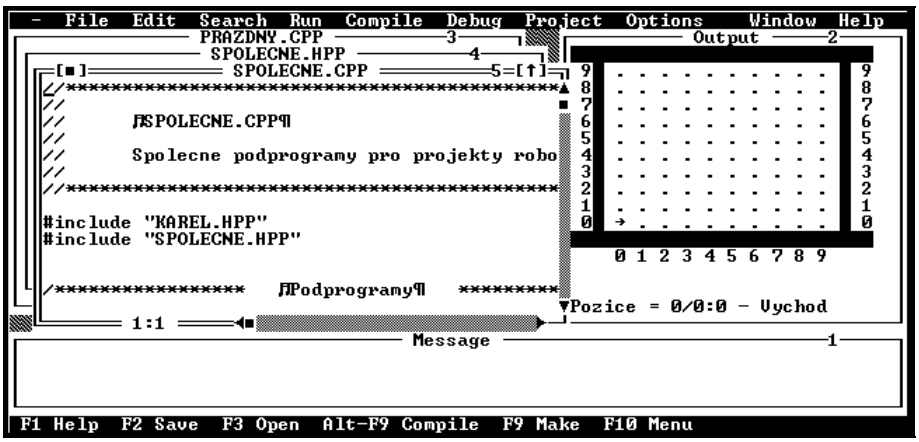
tentokrát proběhnout úspěšně.

Program jsme již úspěšně přeložili a sestavili. Nyní si jistě přejete vidět, jak krásně vám chodí. Spustíte-li jej však standardním CTRL-F9, proběhne program takovým tempem, že vůbec nestihnete zaznamenat, co přesně provedl.

Chceme-li sledovat běh programu krok za krokem, musíme jej spustit klávesou F8 (horká klávesa pro [Run | Step_Over], při ovládání myší můžeme tuto klávesu „odcvaknout“ i dole v systémovém řádku) a po každém dalším stisku této klávesy počítač provede jeden krok našeho programu. Rozdíl mezi těmito dvěma možnostmi spočívá v tom, že v prvním případě (CTRL-F9) program normálně odstartujeme, kdežto v druhém případě (F8) chceme po počítači, aby jej prováděl po jednotlivých příkazech, resp. řádcích (platí to, co je delší).

Po spuštění programu klávesou F8 vykoná počítač některé přípravné práce a zastaví se před prvním příkazem našeho programu. A protože se tento příkaz chystá právě vykonat, tak jej zvýrazní. Stiskneme tedy znovu F8 a vykonáme jej. Obrazovka problikne a počítač zvýrazní další řádek našeho programu. My bychom však chtěli vědět, jaký byl výsledek vykonání prvního řádku.

Máme dvě možnosti: buď si necháme ukázat pomocí ALT-F5 obrazovku jako před chvílí, anebo to uděláme rafinovaněji – necháme zobrazit výstupní okno *Output* tím, že stiskneme F10, W a O, a jeho umístění a velikost upravíme tak, aby nám to co nejlépe vyhovovalo.



Obr. 4.2 Obrazovka s otevřeným výstupním oknem

Řešení s výstupním oknem je výhodnější, protože se nemusíme pořád přepínat tam a zpět, a proto u něj zůsta- zůstaneme. A aby nám zůstalo

výstupní okno celé viditelné, nebudeme je ani opouštět a celý program projdeme postupnými stisky klávesy F8.

4.4 Variace dvorku

Podoba standardního dvorku je uložena v adresáři *DVO* v textovém souboru *DVOREK.DVO* vytvořeném podle zásad uvedených v minulé kapitole. Pokud bude třeba použít jiné konfigurace (rozměr, počáteční pozice Karla, podoby značek a samotného Karla), můžeme tento soubor načíst do editačního okna, opravit a uložit (nejlépe do stejného adresáře) pod novým jménem (bylo by vhodné dodržet příponu).

Skutečnost, že má pracovat s naším novým dvorkem, vysvětlíme systému tak, že v dialogovém okně [Run | Parameters] (Pascal), resp. [Run | Arguments] (C++) zadáme ve vstupním bloku název nově vytvořeného dvorku, např. *DVO\NOVY.DVO*. Při příštím spuštění programu bude program pracovat s touto novou podobou dvorku.

Příklady

1. Zkuste si přeložit a spustit svůj první program na dvorcích různé velikosti, s různým počátečním umístěním robota Karla a s jeho různými podobami.
2. Napište program, který na standardním dvorku (10 řádků, 10 sloupců, maximálně 9 značek na jednom poli) umístí do každého sloupce v nultém řádku tolik značek, kolik je číslo sloupce (do nultého nula, do prvního 1, do druhého 2 atd. až do devátého sloupce, v němž bude 9 značek). Po vykonání svěřeného úkolu se Karel přemístí o řádek výš, aby „nestínil“. Zkuste experimentovat s různými podobami jednotlivých počtů značek.

Těm, kteří si nejsou jisti, jak mají tento program napsat, napovíme, že pascalský hlavní program by měl mít tvar

```
{********** Hlavní program **********}
begin
  Krok; Poloz;
  Krok; Krat( 2, Poloz );
  Krok; Krat( 3, Poloz );
  Krok; Krat( 4, Poloz );
  Krok; Krat( 5, Poloz );
  Krok; Krat( 6, Poloz );
  Krok; Krat( 7, Poloz );
  Krok; Krat( 8, Poloz );
  Krok; Krat( 9, Poloz );
```

```
  VlevoVbok; Krok;
end.
```

5. Základní prvky jazyka

V této kapitole se seznámíme se základními syntaktickými prvky jazyka: identifikátory, klíčovými slovy, oddělovači a komentáři. Vlastního programování se její obsah bude týkat sice pouze okrajově, nicméně bez znalostí těchto prvků se programovat nedá.

5.1 Identifikátory

Identifikátor je odborný výraz pro název nebo jméno, kterým pojmenovanou věc identifikujeme. Prozatím jsme se v programech setkali pouze s identifikátory Karlových příkazů *Krok*, *VlevoVbok*, *Polož* a *Zvedni* a programátoři v jazyce C++ navíc s identifikátorem *main*. Brzy k nim přibudou i naše vlastní.

Syntaktická definice identifikátoru je v obou jazycích stejná:

Identifikátor:
    Písmeno [*PČ*] opak

PČ: // Písmeno nebo číslice
    Písmeno Číslice

Písmeno:
1 z: **_ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**
a b c d e f g h i j k l m n o p q r s t u v w x y z

Číslice:
1 z: **0 1 2 3 4 5 6 7 8 9**

Z uvedených definic vyplývá, že identifikátorem může být libovolná posloupnost alfanumerických znaků (tj. písmen a číslic) začínající písmenem, přičemž písmenem se rozumějí znaky anglické abecedy rozšířené o znak podtrženo ("_" – kód 95). V identifikátoru tedy nemůžeme používat znaky české nebo slovenské abecedy s háčky, čárkami či jiným folklorem. Délka identifikátoru je omezena pouze maximální povolenou délkou řádky (minimálně 256 znaků).

Až potud se oba jazyky shodují. Liší se však v počtu významných znaků, tj. počtu znaků sloužících k vlastní identifikaci (zbylé znaky se ignorují), a v tom, zda rozlišují velká a malá písmena.

Pascal velká a malá písmena v identifikátorech nerozlišuje – **ahoj**, **Ahoj** a **aHoj** jsou proto tři způsoby zápisu jednoho a téhož identifikátoru. Významné jsou pro něj první 63 znaky.

C++ naproti tomu velká a malá písmena v identifikátorech rozlišuje (pokud to sami nezakážete, a to nebývá zvykem), takže **ahoj**, **Ahoj** a **aHoj** jsou tři různé identifikátory. V C++ jsou významné všechny znaky.[8]

Pokud jste při nastavování parametrů konfigurace narazili v nabídce [Options | Compiler | Source] na volbu Indentifier Length, tak vězte, že je určena pro programátory v jazyce C a na C++ se nevztahuje.

5.2 Klíčová slova a ostatní předdefinované symboly

Klíčová slova (anglicky *key words*) jsou identifikátory, které si překladač rezervuje pro vlastní potřebu. V programu je tedy ve významu řadových identifikátorů samozřejmě nesmíme použít. Není jich málo. Připočteme-li k nim ostatní nepoužitelné identifikátory, je jich v Pascalu 57 (Turbo Pascal 6), a v Borland C++ dokonce 85.

Myslíme si, že je zbytečné opisovat zde jejich seznam, když si o nich prozatím stejně nic podstatného říci nemůžeme a budeme se s nimi seznamovat postupně. Každý si je může najít v nápovědě nebo v manuálu (např. *Turbo Pascal 6.0 – Programer's Guide*, str. 7; *Borland C++ – Programer's Guide*, str. 9 a 10).

Kromě klíčových slov používá každý jazyk ještě řadu dalších jedno- i víceznakových symbolů (+, -, **::**, **:=**, == atd.), jejichž podoba nás však již ve volbě identifikátorů nijak neomezuje, protože obsahují znaky, které v identifikátoru být stejně nesmějí.

5.3 Oddělovače – bílé znaky

Oddělovače jsou znaky, které – jak ostatně sám jejich název napovídá – od sebe v programu oddělují jednotlivé prvky jazyka. Základními oddělovači, které můžeme použít prakticky kdekoliv, jsou tzv. **bílé znaky** (anglicky *white spaces*), mezi něž patří mezera, oba tabulátory (vodorovný a svislý) a znaky

[8] Alespoň tak to požaduje norma jazyka. Většina překladačů ve skutečnosti omezuje délku identifikátorů nějakým hodně velkým číslem, které se prakticky nedá překročit.

pro přechod na nový řádek a novou stránku. Budeme-li tedy v dalším textu hovořit o tom, že někde musíme, smíme nebo nesmíme napsat mezeru, budeme tím rozumět, že v daném místě musíme, smíme nebo nesmíme napsat kterýkoliv z bílých znaků nebo – jak se dozvíte v příští kapitole – komentář.

Nyní se asi někteří z vás zeptají, co to je „prakticky kdekoliv“? Odpověď je jednoduchá:

Bílý znak smíme napsat kamkoliv s výjimkou vnitřku víceznakových symbolů,

přičemž víceznakovými symboly budeme rozumět jak symboly předdefinované v jazyku (z použitých jsou to např. komentářové závorky, **begin**, **end** a další klíčová slova Pascalu stejně jako identifikátor *main* nebo klíčové slovo **void** v jazyku C++), tak námi definované identifikátory (např. *Poloz*). Možnost vkládat do identifikátorů pro zpřehlednění mezery, jak to možná někteří znají z Paradoxu, ani Pascal ani C++ nepřipouštějí.

Poznámka (R. P.)
Kdysi jsem v mladickém nadšení psal učebnici programování ve verších. Možná že někteří z vás přivítají říkanky, které z ní budu vybírat pro připomenutí a snazší zapamatování některých vykládaných skutečností. Kdo básničky nemá rád, ať je prostě přeskakuje. Zde je tedy první, hovořící o možném umístění mezer:

Chceš-li v program název vtělit,
mezera jej nesmí dělit,
ani jiný bílý znak.
Smiř se s tím, už je to tak.

5.4 Komentáře

Komentáře neboli poznámky slouží programátorům k tomu, aby své programy zpřehlednili. Prakticky každý program je třeba po jisté době nějak upravit, a ne vždy je v danou chvíli jeho tvůrce po ruce. Proto je nutné psát programy tak, aby se v nich dokázali orientovat i jiní lidé než autoři. Kromě toho je více než jisté, že se v nekomentovaném programu nevyzná s jistým odstupem ani samotný jeho tvůrce.

Význam komentářů stručně shrnuje následující říkanka:

Má tvůj program stinné tváře?

Používej komentáře!
Zdržení při jejich psaní
je jen malé ve srovnání
s časem, který rychle zmizí,
až dnes jasné bude cizí.

Syntakticky jsou komentáře části textu uzavřené mezi takzvané **komentářové závorky**. Narazí-li překladač v programu na otevírací komentářovou závorku, ignoruje další text do té doby, dokud při jeho pročítání nenalezne uzavírací komentářovou závorku. Je zřejmé, že ve vlastním textu komentáře se nesmí vyskytnout zavírací komentářová závorka, protože by pak překladač ukončil komentář předčasně a následujícímu textu by pravděpodobně nerozuměl. Na druhou stranu však otevírací komentářová závorka vložená do textu komentáře počítač zmást nedokáže, čehož se hojně využívá při vymaskovávání některých pasáží z programu. Časem si tyto „fígle“ také ukážeme.

V našich programech budou komentáře použity mimo jiné ke zvýraznění některých textů. Využijeme toho, že vnitřek komentáře překladač ignoruje, takže se v něm mohou vyskytnout prakticky jakékoli znaky včetně řídicích znaků pro tiskárnu. Budeme-li proto chtít nějaký text zvýraznit, uvedeme v komentáři před ním řídicí znak pro zapnutí širokého tisku (pro většinu tiskáren znak CTRL-N s kódem 14) a v komentáři za zvýrazňovaným textem široký tisk opět vypneme (pro většinu tiskáren znak CTRL-T s kódem 20). Takto jsou připraveny soubory řady *PRAZDNY* a zároveň i všechny programy na doprovodné disketě (vytiskněte si některý soubor na tiskárně a uvidíte, jak tyto formátovací příkazy fungují). Příklady v textu knihy však tuto formátovací úpravu využívat nebudou.

Jak Pascal, tak C++ umožňují psát komentáře dvěma způsoby, které není možno slučovat (komentář otevřený jedním způsobem je nutno uzavřít stejným způsobem – druhý typ uzavírací komentářové závorky překladač ignoruje):

Jazyk	**Komentářová závorka**	
	Otevírací	**Uzavírací**
C++	/*	*/
	//	konec řádky
Pascal	{	}
	(*	*)

Tab. 5.1 podoby komentářových závorek

Borlandské překladače jazyka C++ navíc umožňují komentáře do sebe vnořovat (smysl to má samozřejmě pouze u prvého typu) a je na vás, zda si při nastavování voleb toto vnořování povolíte, nebo zakážete. My oba raději vnořené komentáře zakazujeme, protože jejich používání většinou přehlednost programu spíše snižuje.

O možnostech umístění komentářů do textu programu platí jedno velice jednoduché pravidlo: Komentář můžeme napsat všude tam, kam smíme napsat mezeru, a dokonce jej můžeme použít i místo mezery. Tedy stručně (a pro lepší zapamatování veršovaně):

S komentářem zacházejte tak,
jako kdyby to byl bílý znak!

Komentáře lze – jako ostatně všechno – dělat dobře a špatně. Špatné komentáře jsou takové, které pouze jinými slovy popisují to, co je v programu vidět na první pohled, a neobsahují žádnou dodatečnou informaci. Dobrý komentář se naproti tomu nezmiňuje o věcech, které lze snadno vyčíst z programu, a popisuje věci, které z vlastního programu na první pohled patrné nejsou, ale bez jejichž znalosti nelze program v plné hloubce pochopit – a tedy ani zodpovědně opravit či upravit. Dobrý komentář se tedy nesnaží odpovědět na otázku, **jak** to autor naprogramoval (to můžeme vyčíst z programu), ale spíše se snaží objasnit, **proč** je program napsán právě takto.

Aby vám byly tyto zásady jasnější, ukážeme si nejprve špatně komentovaný hlavní program z minulé kapitoly:

```
{********** Hlavní program **********}
begin Poloz;                    { Karel položí značku }
  Krok;                         { Popojde }
  VlevoVbok;                    { Otočí se }
  VlevoVbok;
  Krok;                         { Popojde }
  Zvedni;                       { Zvedne značku }
  VlevoVbok;                    { Otočí se }
  VlevoVbok;
  Krok                          { Popojde }
end.
(* Konec programu *)
```

a nyní program okomentovaný již podstatně lépe

```
/************* Hlavní program *************/
void /*****/ main /*****/ ()
```

```
{
  Poloz();                        // Položí značku
  Krok();                         // Odstoupí, aby ji bylo vidět
  VlevoVbok();                    // Udělá čelem vzad
  VlevoVbok();
  Krok();                         // Nakročí znovu na značku
  Zvedni();                       // a zvedne ji
  VlevoVbok();                    // Znovu čelem vzad
  VlevoVbok();
  Krok();                         // Odstoupí, aby bylo vidět výsledek
} /********** main **********/
```

6. Procedury

Dosud jsme v našich programech používali pouze příkazy, s nimiž se Karel „narodil“. Nyní si ukážeme postup, jak definovat příkazy vlastní. Tyto definice budeme nazývat **podprogramy**.

Podprogramy lze rozdělit na dvě skupiny: na **procedury** a **funkce**. V této kapitole si budeme vyprávět pouze o procedurách[9], protože většiny vlastností funkcí bychom stejně ještě neuměli využít. Vrátíme se k nim hned, jakmile se naučíme něco, v čem by se nám znalost funkcí mohla hodit.

6.1 Definice procedury

Podívejme se nejprve, jak má v jednotlivých jazycích vypadat definice procedury po formální stránce. Následující definice však zatím ještě nebudou úplné, protože obsahují jen to, co budeme v nejbližších několika kapitolách používat.

Definice procedury (měli bychom dodat „bez parametrů“) začíná v Pascalu klíčovým slovem **procedure**, za nímž následuje identifikátor definované procedury, středník, klíčové slovo **begin**, posloupnost středníky oddělených příkazů tvořících tělo procedury, klíčové slovo **end** a závěrečný středník. Formální definice vypadá takto:

Definice_procedury: *{Definice procedury – první přiblížení}*
procedure *Identifikátor*; **begin** *příkaz [;příkaz]* opak **end ;**

Programátoři v C++ se již s definicí procedury setkali, protože v C++ není hlavní program nic jiného než procedura se jménem *main*. Z podoby její syntaktické definice bychom si tedy mohli snadno odvodit, že definice procedury (měli bychom dodat „bez parametrů“) začíná v C++ klíčovým slovem **void**, za nímž následuje identifikátor definované procedury, prázdné kulaté

[9] Takto se podprogramy rozdělují ve velké většině programovacích jazyků. Ovšem jazyky C a C++ představují z tohoto pravidla výjimku – hovoří pouze o funkcích. To neznamená, že v C++ nelze psát procedury – to znamená, že v C++ se i procedurám říká funkce. Protože jde jen o terminologický zmatek a nic jiného, budeme i v C++ rozlišovat procedury a funkce. Proto budeme hovořit např. o proceduře *main*, i když v běžné literatuře najdete povídání o funkci *main*.

závorky, otevírací složená závorka, posloupnost příkazů tvořících tělo procedury a zavírací složená závorka. Formálně tuto definici popíšeme takto:

Definice_procedury: //Definice procedury – první přiblížení
void *Ident* **( ) { *[příkaz]*** opak **}**

Podprogramy jsou základní konstrukcí celého programování. Programátoři si je personifikují a tvrdí, že „podprogram **xxx** udělá **yyy**“ místo přesnějšího „při vyvolání podprogramu **xxx** udělá počítač **yyy**“. Protože je však tento slangový způsob vyjadřování kratší a ve složitějších případech i výrazně přehlednější, přidržíme se ho v našem kursu i my.

Pokusme se nyní nějaký vlastní podprogram vytvořit. Nejprve však jedno doporučení: při definici nového příkazu – procedury používejte prázdnou definici procedury. Jednu takovou prázdnou definici najdete na konci souborů *PRAZDNY.xxx* (a tím i právě používaného souboru) mezi řádky[10]:

```
#if 0           // Následuje prázdná definice podprogramu
```

a

```
#endif          // Konec prázdné definice podprogramu
```

Tuto prázdnou definici si zkopírujte na místo, kde chcete nový podprogram definovat, nahraďte znak podtržení názvem definovaného programu a do těla procedury (tj. mezi příkazové závorky) vepište požadovaný program.

Tento postup je sice nepatrně složitější než napsání čisté prázdné definice, pokud jej ale dodržíte, budou mít všechny podprogramy stejnou, dohodnutou podobu. Neradi bychom vám za každou cenu vnucovali podobu procedur, kterou používáme my, v každém případě vám ale doporučujeme zvolit takovou, která vám pomůže program co nejvíce zpřehlednit.

Když jsme se učili spouštět překlad a sestavení programu, potřebovali jsme v demonstračním programu Karla otočit o 180 stupňů. Pokusme se rozšířit Karlův repertoár o příkaz *ČelemVzad*, po němž Karel tento obrat provede.

Vložíme tedy **před** hlavní program prázdnou definici procedury a nazveme ji *CelemVzad* (jak již víme, programátoři v Pascalu nejsou povinni dodržovat velikost písmen, programátoři v C++ ano). Najedeme kurzorem mezi příkazové závorky a zapíšeme dvě volání příkazu *VlevoVbok* (programátoři v

[10] V obou souborech je použita stejná syntax, protože Pascal text za závěrečnou tečkou stejně ignoruje.

C++ nezapomenou na závorky). Nyní najedeme kurzorem do hlavního programu a vyměníme v něm obě sekvence dvou za sebou jdoucích příkazů *VlevoVbok* za příkaz *CelemVzad*. Výsledný program bude mít tedy v Pascalu podobu:

```
procedure (*****) CelemVzad (*****);
begin
  VlevoVbok;
  VlevoVbok;
end;
(********** CelemVzad **********)

{********** Hlavní program **********}
begin
  Poloz;
  Krok;
  CelemVzad;
  Krok;
  Zvedni;
  CelemVzad;
  Krok
end.
```

a zdrojový text pro C++ bude mít tvar

```
void /*****/ CelemVzad /*****/ ()
{
  VlevoVbok();
  VlevoVbok();
}
/********** CelemVzad **********/

/************* Hlavní program *************/
void /*****/ main /*****/ ()
{
  Poloz();
  Krok();
  CelemVzad();
  Krok();
  Zvedni();
  CelemVzad();
  Krok();
} /********** main **********/
```

Pomocí klávesy F8 nyní program odkrokujte a ověřte si, že dělá to, co má.

Označení

Než půjdeme dále, uděláme si malou dohodu. Abychom výklad trochu zpřehlednili a abychom v následujícím textu ušetřili trochu místa, budeme po několik následujících kapitol při popisu posloupností příkazů označovat použité příkazy jednopísmennými symboly:

VlevoVbok	***l***
Polož	***p***
Zvedni	***z***
Krok	***k***
ČelemVzad	***č***

Celé posloupnosti budeme uzavírat do úhlových závorek. Posloupnost příkazů v proceduře *ČelemVzad* bude popsána jako < ***l l*** >, posloupnost příkazů v hlavním programu před naprogramováním procedury *ČelemVzad* jako < ***k l l k z l l k*** > a po jeho začlenění jako < ***k č k z č k*** >.

Před čtením dalšího textu si zkuste ujasnit, jak byste definovali proceduru *VpravoVbok*.

Hotovo? Předpokládáme, že většina z vás jej definovala jako posloupnost tří příkazů *VlevoVbok*, tedy ve zkratce < ***l l l*** >. Někteří ale možná chtěli využít nově definovaného příkazu *ČelemVzad* a navrhli jej buď jako **< *č l*** > nebo < ***l č*** >. Všechny tři definice jsou správné. Ovšem správná by byla i posloupnost < ***č č l č*** >. Pravda, toto řešení asi není optimální (říkáme „asi“, protože mohou nastat situace, kdy optimální bude), protože obsahuje zbytečné příkazy, ale nemůžeme mu upřít správnost.

Nyní byste jistě dokázali vychrlit řadu obdobných neoptimálních definic, avšak ne každá z nich musí být nutně také správná. Zamyslete se např. nad posloupností < ***k č k l*** >. Většinou Karla správně otočí vpravo, ale běda, je-li při jejím volání před Karlem zeď. Pak musí program nutně skončit chybou.

Z uvedeného příkladu pro nás plynou následující poučení:

- ✧ **Není tak důležité, jak je ta která činnost definována, tj. jakým postupem se dosáhne cíle. Důležité je to, zda při plnění tohoto programu počítač za jakýchkoli okolností splní zadanou úlohu.**
- ✧ **Chyby v definici nemusejí být na první pohled patrné. Když nějaký program vytvoříme (stačí i malá část původního úkolu), musíme se ihned přesvědčit, zda jsme jej napsali správně, a musíme jej odladit.**

Anebo říkankou:

Když se k cíli dostat chceme,
bývá jedno, kudy jdeme.
Hlídej si však, abys hnedle
nezjistil, že cíl je vedle.

Využijeme toho, že program je velice jednoduchý, a na tomto příkladu si

ukážeme, jak se programy ladí. Napište podprogram *VpravoVbok* s posloupností příkazů < ***k č k l*** >. Jeho definici zkuste nejprve umístit před definici procedury *CelemVzad*. Zjistíte, že překladač se bouří a hlásí chybu. Pamatujte si:

Deklarace podprogramu musí vždy předcházet jeho použití

a věžte, že definice podprogramu je zároveň jeho deklarací. O jiných způsobech deklarace podprogramů si povíme později.

Přemístěte proto definici příkazu *VpravoVbok* mezi definici příkazu *ČelemVzad* a hlavní program a v hlavním programu nahraďte druhé volání příkazu *CelemVzad* voláním příkazu **VpravoVbok** (v našich zkrácených záznamech budeme příkaz *VpravoVbok* označovat symbolem ***r***, takže hlavní program bude nyní popsán posloupností < ***p k č k z r k*** >).

Podívejme se nyní, jak budeme program ladit. Nejprve ho zkusíme obyčejně přeložit, sestavit a spustit. Program skončí předčasně s hlášením o chybě v příkazu *Krok*. Podprogram *Krok* jste nenaprogramovali, a proto ho nemůžete ani upravit. Víte však, že chyba není přímo v tomto podprogramu, ale je způsobena jeho špatným použitím, špatným voláním. Předvedeme si proto několik postupů, jak se dopídit skutečné příčiny chyby.

6.2 Krokování

Krokování prostřednictvím klávesy F8 již známe. V dalším textu mu budeme říkat **hrubé krokování**. Asi nám však při něm začne záhy vadit, že kvůli testování jednoho příkazu na konci programu musíte krokovat i všechny příkazy předchozí. Pomoc je poměrně jednoduchá. Najedeme kurzorem na řádek s příkazem *VpravoVbok*, stiskneme klávesu F4 (horká klávesa pro [Run | Go to Cursor]) a systém provede program až po řádek, na němž je kurzor, a tam se zastaví.

Stiskneme-li nyní F8, nedozvíme se opět nic jiného, než že ve chvíli, kdy jsme chtěli vykonat příkaz *VpravoVbok*, došlo k chybě při provádění příkazu *Krok*. U tak jednoduchého příkladu, jaký zrovna řešíme, je viník okamžitě patrný, ale kdyby byl příkaz *VpravoVbok* definován složitěji, nemuselo by být na první pohled zřejmé, kde v něm k chybnému vyvolání příkazu *Krok* dochází. Nezbude nám proto, než použít **jemné krokování**, které zařídí klávesa

F7. Po jejím stisku se ponoříme o „hladinu níže“ a začneme krokovat podprogram, který se právě chystáme vyvolat.

Spustíme tedy znovu svůj program a necháme jej dojít až k testovanému příkazu *VpravoVbok*. Potom stiskneme F7 a kurzor se přesune na počátek testovaného podprogramu – přesněji řečeno v C++ na jeho záhlaví a v Pascalu na jeho první **begin**. Systém tak oznamuje, že se chystá provést akce související se vstupem do tohoto podprogramu.

Nyní již můžeme pokračovat v hrubém krokování. Po prvním stisku F8 systém do podprogramu doopravdy vstoupí a připraví se vykonat první příkaz jeho těla. Kurzor se proto přesune na první řádek těla podprogramu, tj. na příkaz *Krok*.

Stiskneme znovu F8 a systém tento příkaz vykoná. Tedy přesněji, pokusí se vykonat. To se mu však nemůže podařit, protože v průběhu jeho provádění dojde k chybě. Řízení se nevrátí do IDE, protože program nám oznamuje chybu a ukazuje nám, v jaké situaci k ní došlo. Program čeká na stisk libovolné klávesy, načež se ukončí a řízení se vrátí zpět do IDE.

Nyní tedy víme, kde je chyba. Teď už zbývá pouze zapřemýšlet nad tím, jak ji odstranit. A to nebývá vždy snadné. To, že víme, kde chyba je, ještě zdaleka neznamená, že také víme, jak má vypadat správné řešení. Hlavní zásadou, která se nám musí zavrtat hluboko pod kůži, je:

Nebuďme při ladění zbrklí!

Pro ladění platí jeden velice nepříjemný, avšak o to pravdivější Murphyho zákon:

Každou opravou se do programu zanese alespoň jedna chyba.

Zbrklý programátor by v našem případě zajásal, že chybu našel, přebytečný příkaz *Krok* by vyhodil, a tím by vše ještě zkomplikoval. Program by sice podle počítače proběhl na první pohled bez chyby, ale při podrobnějším zkoumání bychom přišli na to, že vlastně dělá něco úplně jiného. Dovolte nám proto jednu vlastní murphyovskou definici:

Ladění programu je proces,
při kterém odhalujeme chyby zjevné
a vyrábíme z nich chyby skryté.

Proto vám ještě jednou připomínáme:

Nic neuspěchejte!
Chyby není potřeba JEN opravit,
chyby je třeba
OPRAVIT SPRÁVNĚ!

Než ukončíme tuto kapitolu, musíme si ještě říci něco o tom, co se stane, pokud budete chtít jemně krokovat příkaz, jehož zdrojový text nemáte k dispozici – vyzkoušejte si to a pokuste se krokovat např. příkaz *Krok*. Systém vám oznámí, že nemůže najít zdrojový text odpovídajícího modulu. Další postup se liší podle použitého jazyka:

Pascalisté nemají jinou šanci než prostě potvrdit, že berou tento fakt na vědomí, a dát si příště pozor. I když si ho nedají, nestane se nic horšího, než že je bude systém opět obtěžovat se svými upozorněními.

Naproti tomu C++ otevře dialogové okno, v němž nás požádá, abychom mu ukázali, kam se zdrojový text dotyčného modulu zatoulal. Pokud o něm víme, najdeme jej, předáme systému a můžeme v krokování pokračovat. Pro ty, co chybějící zdrojový text nemají, nabízí systém volbu Ignore, po níž bude všechny další pokusy o krokování podprogramů z tohoto modulu ignorovat.

Další postup je již pro oba jazyky společný: najedeme kurzorem na další příkaz, od nějž chceme v krokování pokračovat, a stiskem F4 sem systém „pozveme“. Nedodržíme-li tento postup a budeme-li místo toho v krokování pokračovat, bude nás Pascal pravděpodobně dále obtěžovat upozorněními na chybějící zdrojové texty a C++ se rozeběhne a nezastaví se dřív, dokud nenarazí na nějakou zarážku nebo na konec programu.

6.3 Zarážky a posloupnost volání

Nepříjemnou vlastností krokování je to, že při něm často příliš dlouho procházíme částmi programu, které jsou evidentně správné, abychom pak v zápalu boje přehlédli pasáž, v níž je chyba. O proběhnutí části programu bez

krokování pomocí klávesy F4 jsme v minulé podkapitole již hovořili. Nyní si ukážeme druhou možnost – možnost použít tzv. zarážku (*breakpoint*)[11].

Místo, kde se má program při běhu zastavit, označíme tak, že na dotyčný řádek najedeme kurzorem a stiskneme CTRL-F8 (horká klávesa pro [Debug | Toggle Breakpoint]). Stejným způsobem nastavenou zarážku opět zrušíme.

Zarážky nastavíme všude tam, kde jsou v programu podezřelé pasáže, jejichž průběh bychom např. chtěli krokovat. Budeme pak mít jistotu, že při běhu programu žádná podezřelá část programu neunikne našemu pozornému zraku. S výjimkou Turbo C++ 1.0 si dokonce IDE dokáže nastavení zarážek zapamatovat mezi jednotlivými sezeními, a tak když program spustíme druhý den, je mimo jiné i nastavení stopek totožné s nastavením, při němž jsme program opouštěli.

Pro hledání evidentních chyb v programu jsme pro vás připravili pomocné prostředky. Najděte si definici podprogramu *Chyba*. Tuto definici najdou programátoři používající C++ na počátku souboru PRAZDNY.CPP a programátoři používající Pascal v souboru nazvaném *CHYBA_.PAS*.

Vidíte, že tato definice má prázdné tělo, takže nic nedělá. Jejím jediným účelem je umožnit vám najít přesně místo vzniku chyby. Podívejme se, jak toho můžete dosáhnout.

Najedeme na ni kurzorem (pascalisté najedou na **begin**) a nastavíme na ni zarážku (připomínáme: CTRL-F8). Nyní spustíme pomocí CTRL-F9 program s původní podobou příkazu *VpravoVbok*. Program se rozběhne, narazí na chybu při plnění příkazu *Krok* a tuto skutečnost nám oznámí (C++ ještě počká, až stisknete nějakou klávesu).

Program by jistě chtěl ještě běžet dál, ale je napsán tak, že po odhalení chyby vyvolá proceduru **Chyba**, a v ní my máme nastavenu zarážku. Předá proto řízení zpět IDE, které otevře okno se souborem, v němž je zarážka (**CHYBA_.PAS**, resp. **HLAVNI.CPP**), a nastaví na ni kurzor.

My tedy nyní víme, že v programu nastala chyba, ale nevíme bohužel přesně kde (alespoň se tak tváříme). Abychom to zjistili, vyvoláme stiskem CTRL-F3 (horká klávesa pro [Window | Call Stack] v Pascalu a [Debug | Call Stack] v C++) na obrazovku okno, v němž je zapsána posloupnost volání podprogramů. V horním řádku okna najdeme jméno podprogramu, v němž došlo k přerušení běhu programu – v našem případě *Chyba*. Pod ním bude jméno

[11] Doslovný překlad termínu *breakpoint* je „bod přerušení“. To je ale poněkud zdlouhavé – připomíná to hokejové „postavení mimo hru na červené čáře“, než to vyslovíte, uteče celá třetina – a proto budeme používat kratší, ale možná i výstižnější označení **zarážka**.

podprogramu, který daný podprogram zavolal (*_Chyba* – údajů v závorkách si nevšímejte, povíme si o nich později), pod ním jméno podprogramu (v našem případě *Krok*), který jej zavolal, a tak dále, až v nejspodnějším řádku najdeme jméno hlavního programu (v Pascalu jméno programu, v C++ *main*).

Stiskneme šipku dolů a v okně se místo prvního řádku zvýrazní druhý. Pokračujeme dále dolů přes podprogramy, o jejichž správnosti jsme přesvědčeni (nic jiného nám ostatně nezbývá, stejně je nemůžeme opravit), tj. přes podprogramy *Chyba*, *_Chyba* a *Krok*, až narazíme na jméno podprogramu, kterým si již tak jisti nejsme, neboť jsme ho napsali sami – v našem případě *VpravoVbok*.

Do tohoto podprogramu se rozhodneme podívat. Řádkovým kurzorem jsme na jeho jméno již najeli, takže zbývá jeho výběr potvrdit (myší na ně můžete najet rovnou a vybrat je dvojitým stiskem).

IDE pak nastaví jako aktuální to okno, v němž je soubor s podprogramem, který jste si ze seznamu právě vybrali (pokud takové okno není, otevře je), a nastaví kurzor ve zvoleném podprogramu na řádek, který v něm byl vykonáván naposledy (předtím, než se program zastavil na zarážce). Tedy v našem případě na řádku, při jehož vykonávání došlo k chybě.

Tento postup můžeme libovolněkrát opakovat s různými podprogramy z posloupnosti (samozřejmě s výjimkou Karlových původních příkazů, které prozatím nemáme ve zdrojovém tvaru), a tak analyzovat celou posloupnost volání a rozhodnout se, kde že by mohla být chyba.

Pokud si omylem vybereme některé z Karlových primitiv, dostaneme se do stejné situace, jako kdybychom chtěli tento příkaz krokovat, a budeme i stejně reagovat. To znamená pascalisté vezmou zprávu na vědomí a céčkaři požádají systém, aby jejich příští podobné požadavky ignoroval.

Příklady

Naprogramujte procedury (příkazy), které Karla postupně „naučí“ všechny možné tahy šachového krále, přičemž směr natočení Karla před tahem a po něm bude stejný. Procedury nazvěte (skutečné názvy budou bez diakritiky) *ÚkrokVlevo*, *ÚkrokVpravo*, *Zpět*, *ŠikmoVlevo*, *ŠikmoVpravo*, *VlevoZpět* a *VpravoZpět*. Svá řešení si můžete porovnat s řešeními v souborech *APROC.PAS* a *A_PROC.CPP*.

6.4 Procedury a příkaz Krát v Pascalu

Použijeme-li jméno naší vlastní procedury v programu v Pascalu v příkazu *Krát*, může se stát, že překladač ohlásí chybu *Invalid procedure or function reference*. Přesný důvod této chyby nebudeme zatím rozebírat, řekneme si pouze, jak ji odstranit.

Tato chyba znamená, že jsme před překladem nenastavili v IDE volbu Force far calls v dialogovém okně Options | Compiler. Pokud z nějakého důvodu nechceme tuto volbu nastavit, můžeme před definici (nebo deklaraci, pokud předchází definici) této procedury zapíšeme komentářovou direktivu {$F+} a za ni {$F-}. Například takto:

```
{$F+}
Procedure Vlajka; forward;
{$F-}
```

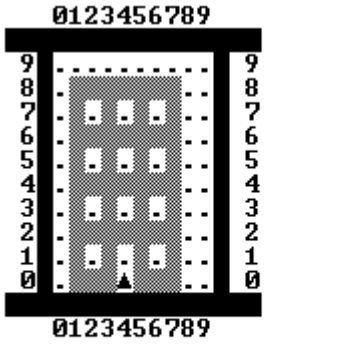
Mezi těmito dvěma direktivami můžeme zapsat i celou skupinu funkcí.

V Turbo Pascalu 7.0 můžeme tyto direktivy nahradit klíčovým slovem **far**, které zapíšeme za hlavičku funkce:

```
Procedure Vlajka; far; forward;
```

7. Zásady řešení složitých úloh

Představte si, že máme za úkol vytvořit podprogram *Dům*, při jehož plnění Karel vyznačkuje domek podle obrázku 7.1.



Obr. 7.1 Dům

Pokuste se tento podprogram napsat a odladit sami. Pokud použijete pouze doposud napsané podprogramy, bude výsledná definice velice nepřehledná, a to i v případě, když do ní vložíte na klíčová místa komentáře. A ještě nepřehlednější pak bude její ladění.

Uvážíme-li, že se vlastně jedná o řešení velice jednoduchého problému, bude všem asi ihned jasné, že tudy cesta k řešení složitějších problémů nevede. Přesto tento způsob řešení mnoho začátečníků jen velice nerado opouští. Jednou z největších začátečnických chyb totiž je pokoušet se naprogramovat zadanou úlohu ihned, tak řečeno z jedné vody na čisto. Některé programovací jazyky – např. klasický Basic – k takovému „hurá systému“ dokonce svými vlastnostmi přímo vybízejí.

Zkušení programátoři vám však potvrdí, že takovýto přístup je vlastně ze všech nejhorší. Během několika málo chvil je program tak složitý, že se v něm záhy neorientuje ani sám autor. V důsledku toho je program po dokončení plný chyb, které právě vzhledem k jeho složitosti a nepřehlednosti nejde ani pořádně najít, natož pak opravit.

Tuto kapitolu proto věnujeme výkladu zásad, kterými bychom se měli řídit, budeme- -li chtít s co nejmenším úsilím vytvářet co nejkvalitnější programy nezávisle na jejich složitosti. Před vlastním rozborem nejdůležitějších zásad si nejprve povíme o typickém životním cyklu programu.

7.1 Životní cyklus programu

Pokusme se srovnat životní cyklus programu s životním cyklem člověka.

Početí – nápad
Představte si firmu zabývající se vývojem programového vybavení, která zrovna přemýšlí o nové oblasti, do níž by napřela své síly. Tu někdo přijde s nápadem, že by počítač mohl řídit např. splachování záchodů (nesmějte se, tento program kdysi vyhrál soutěž o nejvtipnější program roku a díky němu ušetří konstruktéři mrakodrapů miliony dolarů na jinak předimenzované kanalizaci). Nápad je vždy provázen nadšením a velikými plány, jak dokonalý a geniální program bude vytvořen a co vše při plnění daných úloh počítač zastane. Zda tomu opravdu tak bude, to ukáže teprve další vývoj.

Prenatální vývoj – analýza
Problém se záhy ukáže být složitějším, než se všichni zpočátku domnívali. Nadšení pomalu vyprchává a nastává seriózní práce. Zapojuje se do ní tým programátorů-analytiků (nebo také pouze jeden), kteří podrobí celý problém důkladné analýze a vypracují základní algoritmus řešení.

Porod – kódování
Program začíná spatřovat světlo světa. Analytici předali své výsledky programátorům, kteří dané algoritmy zapíší v nějakém programovacím jazyce. Pokud je tvůrcem programu jediný člověk, je v tuto chvíli plný nadějného očekávání. Nevnímá svět kolem sebe a usilovně ťuká program do počítače.

Výchova – ladění
Stejně jako každé novorozeně i program se hlásí na svět velikým křikem – samozřejmě křikem programátora. Jeho krásný a geniální program totiž nechodí (ostatně jako každé novorozeně) – mimochodem známá programátorská pravda říká, že „v každém programu je alespoň jedna chyba“. Nastává práce nejpracnější a nejúmornější – ladění. Je třeba zjistit, **proč** program nechodí. Musejí se nalézt a opravit všechny chyby. A protože platí další známá programátorská pravda, že „každou opravou se do programu zanese nejméně jedna chyba“, je ladění práce opravdu klopotná – ostatně jsme o něm již hovořili.

Dospělost – používání
Program funguje a dělá, byť často s chybami, to, co má. Bohužel, ne každý program se své dospělosti dožije.

Stáří – modifikace
Program sice stále vykonává to, co jsme po něm chtěli, ale s jeho vlastnostmi již dávno nejsme spokojeni. Nějaký čas program ještě vylepšujeme, ale jednou přijde doba, kdy nám bude jasné, že výhodnější než úprava stávajícího programu je napsání programu nového.

Důchod – archivace
Program dosloužil. Buď na nové úkoly již nestačil a byl nahrazen novým (sešel stářím), nebo se ho někomu podařilo omylem zničit (pracovní úraz).

7.2 Základní pravidla návrhu programu

Již na počátku kapitoly jsme hovořili o tom, že základním začátečnickým zlozvykem je usednout k počítači a začít psát program ihned poté, co obdržíme zadání. Doporučovaný postup je však právě opačný. Ing. Půža jej kdysi krásně charakterizoval slovy:

> „**Čím později usednu k programování, tím dříve jsem s programem hotov**."

Základním pravidlem veškerého programování je:

> **Každý poblém je třeba nejprve důkladně analyzovat. Rozmyslet si, co vlastně máme řešit a jak na to půjdeme. Teprve po této analýze můžeme přistoupit k vlastnímu programování.**

Nebo, pokud jsou vám sympatické veršovánky:

Co ti musí býti svato?
Nejdřív dumej, kudy na to!
Neboj se v problému vrtat,
ať nemusíš potom škrtat.

Návrh základní koncepce

Mnozí z vás se možná při čtení podkapitoly o životě programu domnívali, že o první fázi, o nápadu, jsme se zmiňovali pouze proto, aby analogie byla úplná. Není tomu tak. Přestože se tato fáze jeví na první pohled jako zdaleka nejjednodušší a naprosto bezproblémová, opak je pravdou. Patří totiž k nej-

obtížnějším fázím celého projektu a závisí na ní z velké části výsledek veškerého následného snažení.

Abychom mohli po počítači něco chtít, musíme napřed vědět, že to dokáže. Nápad, že počítač může dělat to či ono, může dostat pouze ten, kdo o schopnostech počítačů alespoň něco ví (hovoříme o realizovatelném nápadu). Samotný nápad však ještě není nic. Vhodným výchozím bodem pro tvorbu programu se může stát teprve tehdy, až jej podrobně specifikujeme jako zadání úkolu. Až přesně definujeme, co všechno má program umět a jak se bude chovat nebo jak bude reagovat (až mu předáme kompletní genetickou informaci).

V první fázi proto promyslíme koncepci programu, tedy to, na čem bude nejvíce záviset jeho prodejnost či neprodejnost. Řečeno veršem:

Je to úkol pro vědce
a nesmí se dělat v spěchu,
vždyť základní koncepce
rozhoduje o úspěchu.

A ještě jednu věc vám chceme doporučit: nestyďte se poučit jinde (jak se mezi programátory říká: „Opsat a doplnit vlastníma chybama“) – samozřejmě v rámci možností daných autorským zákonem. Vezměte si příklad z programátora, který

Veškeré své zkušenosti,
znalosti a odhodlání
použije zprvu skrytě –
jak to dělaj' druzí, shání.
Všude hledá poučení,
co kde jiný uměl kdesi,
co je hlavní, a co není,
a na co už netroufne si.
Nepřeceňujte své schopnosti a neklaďte si
příliš náročné cíle!

Hlavně poslední verš je důležitý. Než jich (těch příliš náročných cílů) dosáhnete, zaplaví konkurence trh programy sice méně dokonalými, avšak **HOTOVÝMI!!!**

7.3 Analýza

Po specifikaci úkolu, tj. ve chvíli, kdy již přesně víme, co má program dělat a jak se bude tvářit navenek, přichází na řadu jeho analýza. Během ní si vyjasníme způsoby a metody, kterými budeme úkol řešit.

Často musíme problém nejprve matematizovat, to znamená, že musíme najít jeho matematický popis. Z něj nám pak obvykle vyplyne, co všechno si musí počítač při řešení problému pamatovat, jak často se bude ke kterým údajům obracet, a řada dalších faktů, které ovlivní celé řešení.

Známe-li základní principy určující řešení úlohy, musíme se rozhodnout, jakým způsobem budeme všechny znalosti o problému i případné mezivýsledky v počítači reprezentovat – musíme vypracovat návrh datových struktur. Převážná většina úloh totiž nepracuje s čísly, ale s daty nenumerického charakteru, a tam nebývá otázka vnitřní reprezentace jednoznačná.

Fáze rozhodování o vnitřní reprezentaci zpracovávaných dat nás nemine ani při používání Karla. My jsme sice na počátku této knihy vyhlásili, že deklarace a použití dat v programu si díky Karlovi můžeme nechat až do příštího dílu, ale to ještě neznamená, že se nebudeme zabývat reprezentací zpracovávaných dat. Např. v podprogramu *Dům* z počátku této kapitoly je čára, kterou domek kreslíme, symbolizována (reprezentována) políčky, na nichž je položena značka. Podobným způsobem budeme o vnitřní reprezentaci dat hovořit ještě vícekrát.

Poslední fází analýzy je návrh algoritmu řešení. Jedním ze stěžejních úkolů této etapy řešení je správná dekompozice složitějších úloh na úlohy jednodušší. Jedná se vlastně o počítačovou aplikaci starého římského „**divide et impera**“ neboli česky „**rozděl a panuj**“[12].

I pro nás je velice výhodné rozdělit si složitý problém nejprve na řadu dílčích podproblémů, ty vyřešit a z jejich řešení pak sestavit řešení problému původního. Pokud se nám bude zdát některý z podproblémů stále příliš složitý, není nic přirozenějšího než jej opět rozdělit. Po několika úrovních postupného zjemňování problému se nakonec musíme dostat do situace, kdy jednotlivé podproblémy budou již natolik jednoduché, že naprogramovat je bude pro nás hračkou.

Při dekompozici problému nesmíme zapomínat na přesnou definici mezimodulového rozhraní. To znamená, že než přistoupíme k řešení jednotlivých

[12] V literatuře se často uvádí pod anglickým označením *divide and conquer*, rozděl a zdolej.

podproblémů, musíme si přesně ujasnit, jak spolu budou později jednotlivé podprogramy spolupracovat.

Pamatuj, že podprogramy
nepracují nikdy samy.
Chceš předejít mnohou zlost?
Dřív než kód, řeš součinnost!

Naopak je dobré zapomenout na snahu o maximální efektivitu výsledného programu. Pamatujte si, že **program musí nejprve chodit a teprve pak má smysl přemýšlet o tom, jak zařídit, aby chodil lépe**.

Rada, jež se bude hodit,
chceš-li program zpracovat:
Nejdřív musíš umět chodit a
teprv pak tancovat!

Příliš časná snaha o vrcholnou optimalitu výsledného programu vede většinou ke složitým nepřehledným programům, které se velice obtížně ladí. A mezitím co zápasíte s odladěním svého geniálního programu, jiná firma vesele uvede na trh konkurenční program, který sice není tak kvalitní, ale **je na trhu!** (téměř čítankovým příkladem je Text602). Když vy vítězně odladíte svůj program, málokoho již nalákáte na to, že je mnohem lepší. Vaši potenciální zákazníci již dávno investovali své peníze do konkurenčního programu a nechtějí o tyto své investice přijít.

Včasným uvedením programu na trh nejen získáte finance na jeho další vývoj, ale navíc veškeré optimalizace programu mohou vycházet přímo z požadavků zákazníků. To ve svém důsledku vede k tomu, že se nebudete pokoušet zbytečně vylepšovat vlastnosti, o jejichž vylepšení stejně nikdo nestojí, a soustředíte se na ty rysy programu, jejichž současný stav je jeho nejbolavějším místem.

Všechny zásady, kterými se moderní programování řídí, směřují především k tomu, aby byli tvůrci se svými programy co nejdříve hotovi. Zákazníci totiž ve snaze o co nejrychlejší nasazení programu již dávno přistoupili na zásadu:

Zdá se vám, že program pracuje pomalu?
Kupte si rychlejší počítač!

V souboji programových produktů nevyhrává v současné době většinou

rychlost a úspora paměti, ale především celkové schopnosti programu – tedy vlastnosti, o nichž se rozhoduje ve fázi návrhu koncepce (podle předchozí podkapitoly fáze početí). Typickým příkladem byl souboj dBase – FoxBase – Paradox. Přestože byla FoxBase výrazně rychlejší než dBase a v některých akcích i než Paradox, vzhledem k tomu, že vychází z prakticky stejné koncepce jako dBase, nedokázala se proti ní příliš prosadit (hovoříme o situaci ve světě, ne u nás). Naproti tomu Paradox postavil svoji koncepci na zcela jiné filozofii, kterou jeho uživatelé považují za mnohem výhodnější. Proto také Paradox získával na počátku devadesátých let v oblasti databázových produktů jednu pozici za druhou.

8. Návrh metodou shora dolů a zdola nahoru

V této kapitole si povíme o dvou hlavních přístupech k řešení problémů: o metodě shora dolů a metodě zdola nahoru. Učebnice programování dávají většinou jednoznačně přednost metodě shora dolů, jenže, jak víme, suchá je teorie, a zelený strom života. V běžné praxi se často vyskytnou úlohy, které je bezesporu výhodnější řešit právě zdola nahoru. Povíme si proto o obou přístupech a na závěr kapitoly se pokusíme zhodnotit jejich výhody a nevýhody.

Předem vás prosíme o trpělivost ve chvílích, kdy si budete myslet, že děláme z komára velblouda. Chceme vás se základními zásadami seznámit na jednoduchých příkladech, takže někdy budeme rafinovaně programovat věci, o nichž si budete myslet, že by se daly řešit z hlavy. V těchto jednoduchých příkladech asi ano, ale ve složitých již ne. Tvařte se proto s námi, že řešený problém je složitý, a pokoušejte se získané poznatky sami zobecňovat.

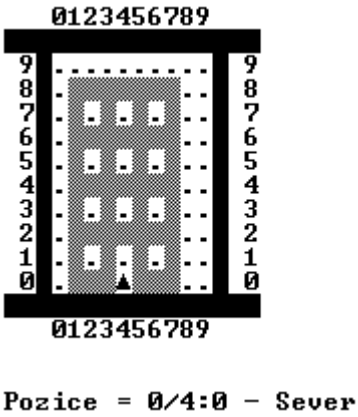
Poznámky

1. Jak jste si již mohli všimnout, programy v Pascalu a C++ jsou si doposud nebezpečně podobné. My jsme si toho všimli také, a proto budeme pro zjednodušení všechny ukázky programů v textu této kapitoly uvádět pouze v Pascalu.

2. Jak víte, označuje se ve standardním dvorku místo, kam Karel položil nějakou značku, číslicí vyjadřující počet položených značek. Chcete-li výsledný obrázek více přiblížit našemu, definujte si dvorek, v němž bude jedna položená značka zobrazena znakem s kódem 177. Pokud navíc nechcete, aby místa mezi jednotlivými políčky dvorku zůstala nezaplněna, zmenšete počet sloupců obrazovky, které zabírá sloupec dvorku, na jeden. Netroufáte-li si na tyto úpravy sami, použijte předdefinovaný dvorek DVO\SHORA.DVO.

8.1 Návrh metodou shora dolů – dekompozice

Metoda shora dolů vychází z přirozeného lidského postupu při řešení složitých problémů a je vlastně programátorskou aplikací známé římské zásady „rozděl a panuj“ (hovořili jsme o ní již ve stati 7.3.). Podívejme se znovu na obrázek domu, který má Karel vyznačkovat, a ukažme si, jak bychom mohli



Obr. 8.1 Dům, který má Karel na svém dvorku nakreslit

příkaz *Dům* naprogramovat.

Co máme udělat, přesně víme – zadání je dostatečně podrobné. Zrovna tak jasný je způsob reprezentace zpracovávaných dat: každý blok budeme reprezentovat jednou položenou značkou. Můžeme tedy rovnou přejít k dekompozici problému a pokusit se vyhmátnout vhodné podproblémy.

Při trochu pozornějším pohledu si jistě všimnete, že dům má čtyři stejná podlaží (řada oken a „strop“ nad nimi), pod nimiž je jakýsi základní sokl. Budeme-li chtít náš program ještě efektně zakončit a přesunout Karla do domovních dveří, mohli bychom jej napsat např. takto:

```
procedure (*****) Dum; (*****)
begin
  Krok;
  Sokl;
  Patro; Patro; Patro; Patro;
  DoDveri;
end; (********** Dum **********)
```

Pokud jste pozorně četli minulou kapitolu, jistě si pamatujete, že dříve než přistoupíme k řešení jednotlivých podproblémů, musíme si ujasnit, jak spolu budou jednotlivé části spolupracovat – v našem případě si musíme ozřejmit, kde jeden příkaz předá Karla příkazu následujícímu. Takže postupně:

Podprogram *Sokl* si převezme Karla v pozici `0/1:0-V` a předá jej tam, kde skončí, tedy v pozici `0/7:1-V`.

Podprogram *Patro* si převezme Karla na políčku, které je o jednu řádku

pod jeho východní (tj. pravou) hranou, přičemž Karel bude otočen na východ. Podprogram jej převede o řádek nahoru, nakreslí s ním dva řádky patra a předá jej na posledním pravém políčku horního řádku patra otočeného na východ.

Podprogram *DoDveří* si v pravém horním rohu domu převezme Karla otočeného na východ (tj. v pozici `8/7:1-V`) a předá jej v pozici `0/4:0-S`.

Nyní se již zdá být všechno jasné a zdá se, že bychom mohli přistoupit k programování jednotlivých podprogramů. To by však nebyl ten nejsprávnější postup. Proč? Protože bychom přišli o jednu z největších výhod postupu shora dolů: o **možnost kdykoliv spustit program**.

Jak to ale udělat, abychom mohli program spustit, když ještě není hotov, když jsme ještě nedefinovali všechny procedury? Nastupuje magie – do té doby, než jednotlivé podproblémy naprogramujeme, budou **skutečné definice** podprogramů zastupovat jejich **definice stínové**. Místo žádaných podprogramů (**vzorů**) prostě definujeme pouze jejich **atrapy.**

Atrapy podprogramů definujeme vždy co nejjednodušeji, ale přitom tak, aby alespoň vzdáleně simulovaly činnost, kterou budou později vykonávat jejich vzory. To je nutné proto, abychom mohli později stínovou definici podprogramu (tj. definici jeho atrapy) zaměnit za jeho skutečnou definici, aniž bychom museli dělat jakékoliv zásahy do jiných částí programu.

Vraťme se tedy k našemu příkladu a naprogramujme atrapy podprogramů *Sokl* a *Patro* tak, aby alespoň dovedly Karla z místa, kde ho převzaly, do místa, kde jej opět předají. Podprogram *DoDveří* žádnou stínovou definici nepotřebuje, protože by nám oproti skutečné definici stejně žádné zjednodušení nepřinesla.

Než budete číst dále, zkuste si tyto tři podprogramy nejprve napsat a odladit sami. Nezapomeňte, že každou definici musíte umístit **před** program, který ji bude používat!

Využijeme-li příkazů *ČelemVzad* a *VpravoVbok*, které jsme si naprogramovali v minulé kapitole, a funkci *Krát*, o níž jsme hovořili v úvodních poznámkách, mohly by definice těchto tří podprogramů vypadat následovně:

```
procedure (*****) Sokl; (*****)
begin
  Krat( 7, Krok);
end;
(********** Sokl **********)

procedure (*****) Patro; (*****)
begin
```

```
  VlevoVbok;
  Krok;
  VpravoVbok;
end;
(********** Patro **********)

procedure (*****) DoDveri; (*****)
begin
  CelemVzad;
  Krat( 3, Krok);
  VlevoVbok;
  Krat( 8, Krok);
  CelemVzad;
end;
(********** DoDveri **********)
```

Tak co? Zkusili jste vlastní definice, nebo jste opsali nabízené? A přišli jste na to, kde je v těch nabízených chyba? Máte poslední možnost projevit iniciativu a chybu odhalit samostatně. Své odhalení si můžete ověřit v následujícím odstavci.

Když jsme psali tuto pasáž a připravovali pro ni programy, napsali jsme v podprogramu *Patro* jeden *Krok* místo dvou. Po odhalení této chyby jsme ji nejprve opravili, ale pak jsme se rozhodli ji v programu ponechat, abyste viděli, že nikdo není chyb ušetřen (a my už vůbec ne).

Nepodléhejte sebeklamu a nikdy nevěřte tomu,
co jste právě napsali.
Každý podprogram vždy prověřte,
a to tak brzo, jak to jen jde.
Jinak se vám pravděpodobně stane,
že se vám s postupným zesložiťováním programu
budou přehlédnuté chyby za trest iniciativně měnit
z chyb zjevných v chyby skryté a
z chyb prostých v chyby záludné.

Poslechneme tedy předchozí radu a pokusíme se náš program hned prověřit. Hlavní program obsahující jediný příkaz *Dům* budeme velkoryse považovat za správný, a proto umístíme kurzor na první příkaz podprogramu *Dům* (tj. na příkaz *Sokl*) a klávesou F4 sem pozveme řádkový krokovací kurzor.

Pokud v programu není žádná chyba, program se přeloží, sestaví a spustí. Zastaví se až na označeném řádku. Pokud máte otevřené okno Output, objeví se na něm Karlův dvorek s Karlem v základní pozici. Stiskem klávesy F8 provedeme příkaz *Sokl* a zkontrolujeme, zda Karel skončil v pozici, v níž skončit měl.

Nyní bychom chtěli provést další příkaz, ale ouha! Nastává dilema. Jak jsme si řekli, při hrubém krokování se v každém kroku provede příkaz nebo řádek, na němž je krokovací kurzor, přičemž systém z těchto alternativ zvolí tu delší (zkuste si to).V našem případě je delší řádek, protože obsahuje čtyři příkazy. My však v současné době nepotřebujeme prověřit funkci kombinace čtyř příkazů, ale funkci jednoho příkazu. Jinak se i tak jednoduchá chyba, jakou jsme udělali např. my, zakuklí a přestává být průzračná.

Nabízejí se nám dvě možnosti: Buď přejdeme na jemné krokování, nebo program upravíme tak, aby byl každý příkaz na samostatném řádku. Protože prvé řešení má tu nevýhodu, že bychom museli zbytečně procházet vnitřkem příkazů *Patro* a mohli tak pro stromy přehlédnout les, zvolíme raději řešení druhé, tj. napsání každého příkazu na samostatný řádek.

Přerušíme proto běh programu (CTRL-F2**)**, upravíme zdrojový text a spustíme program znovu. Pokud jste zopakovali naši chybu, bez problému ji nyní již odhalíte, lokalizujete a opravíte.

První vrstvu máme tedy již naprogramovánu a odladěnu a můžeme se směle vrhnout na vrstvu druhou. Příkaz *DoDveří* je už naprogramován. Vybereme si tedy ze zbývajících dvou ten, kterým začneme. Příkaz *Sokl* je na první pohled jednodušší, a proto začneme s ním. Jistě vás napadlo využít toho, že sokl je sestaven ze dvou malých soklíků, a mohli bychom je tedy naprogramovat např.:

```
procedure (*****) Sokl; (*****)
begin
  Soklik;
  Prejdi;
  Soklik;
end;
(********** Sokl **********)
```

V této definici předpokládáme, že podprogram *Soklík* si převezme Karla na místě, kde má položit první značku, a předá jej na místě, kde položí značku poslední. Přitom bude Karel neustále otočen na východ. Z toho vyplývá i činnost požadovaná po podprogramu *Přejdi*.

Oba nové podprogramy jsou jednoduché, takže se nebudeme ani zatěžovat jejich stínovými definicemi a definujeme je hned doopravdy:

```
procedure (*****) Soklik; (*****)
begin
  Poloz; Krok;
  Poloz; Krok;
  Poloz;
end;
```

```
(********** Soklik **********)
procedure (*****) Prejdi; (*****)
begin
  Krok; Krok;
end;
(********** Prejdi **********)
```

Pokud jsme si toho nevšimli dříve, měli bychom se alespoň nyní zastavit u dvou věcí:

- ✧ V podprogramu *Soklík* se dvakrát opakuje posloupnost příkazů < ***p k*** >, která se jistě objeví i v dalších podprogramech, a proto by bylo vhodné ji definovat jako samostatný podprogram a nazvat jej např. *PoKrok*.
- ✧ I činnost definovanou podprogramem *Přejdi* bychom určitě potřebovali vykonávat častěji. Jméno podprogramu je však spojeno s řešením konkrétního úkolu a žádnou mnemotechnickou pomůcku pro zapamatování prováděné činnosti nenabízí. Asi by proto bylo výhodnější přejmenovat podprogram např. na *Dvojkrok*.

Upravíme tedy program podle obou předchozích doporučení (předpokládáme, že nový tvar definic uvádět nemusíme) a hned jeho funkci prověříme.

Nyní přejdeme k naprogramování podprogramu *Patro*. Považujme i tento podprogram za složitý a sestavme jej z podprogramů jednodušších:

```
procedure (*****) Patro; (*****)
begin
  VlevoVbok; Krok; VlevoVbok;              {O patro výš}
  Okna;
  VpravoVbok; Krok; VpravoVbok;            {O patro výš}
  Strop;
end;
(********** Patro **********)
```

Jak jste si jistě všimli, tentokrát již definice není konstruována jako čistá posloupnost volání nových podprogramů. Často sestává, že některá z posloupnosti činností, které má program realizovat, je natolik jednoduchá a specifická, že se nevyplatí ji definovat jako samostatný podprogram, ale je výhodnější ji na daném místě naprogramovat přímo. A to jsme právě udělali.

Takže si nyní program opište, definujte si samostatně stínové podoby podprogramů *Okna* a *Strop* (oba podprogramy si Karla převezmou na jednom konci a odevzdají jej neotočeného na druhém konci daného řádku patra) a ověřte si, že definice podprogramu *Patro* je správná.

Nyní se opět spustíme o vrstvu níže. Začneme podprogramem *Strop*, protože je jednodušší. Jeho definici můžeme napsat téměř bez přemýšlení:

```
procedure (*****) Strop; (*****)
begin
  Krat( 6,PoKrok );
  Poloz;
end;
(********** Strop **********)
```

Naprogramujte si ji a hned ji také prověřte.

Nyní nám již zbývá pouze podprogram *Okna*. Sami jistě vidíte, že jej lze realizovat opakovaným prováděním příkazů *Polož* a *Dvojkrok* nebo příkazů *PoKrok* a *Krok*. Naše funkce *Krát* však opakování posloupnosti několika příkazů zajistit neumí, a proto si definujeme pomocný podprogram *P2Krok*, jehož volání pak v podprogramu *Okna* několikrát zopakujeme:

```
procedure (*****) P2Krok; (*****)
begin
  Poloz;
  Dvojkrok;
end; (********** P2Krok **********)

procedure (*****) Okna; (*****)
begin
  Krat( 3, P2Krok );
  Poloz;
end;
(********** Okna **********)
```

Naposledy ještě funkci programu vyzkoušíme a jsme s úlohou hotovi. Výsledek si můžete prohlédnout v programu *DUM.CPP* nebo *DUM.PAS*.

Na závěr této podkapitoly si zopakujeme všechny důležité zásady, které jsme si postupně během vývoje podprogramu *Dům* prozradili.

- ✧ Žádný složitější problém neřešíme přímo, ale vždy se jej snažíme rozdělit na několik problémů jednodušších.
- ✧ Dříve než přistoupíme k řešení jednotlivých podproblémů, musíme si ujasnit, jak spolu budou odpovídající podprogramy spolupracovat – musíme si ujasnit mezimodulové rozhraní.
- ✧ Úpravu programů volíme takovou, aby se při krokování nedostaly do kolize naše požadavky s možnostmi ladicího programu – např. každý příkaz, který chceme vykonat samostatně, musí být napsán na samostatném řádku.
- ✧ Každý podprogram musíme vzápětí poté, co jej definujeme, také hned odladit. Abychom to mohli učinit, vytvoříme stínové definice všech použitých podprogramů, které ještě své skutečné definice nemají.
- ✧ Podprogramy ladíme vždy jako součást celého programu, protože

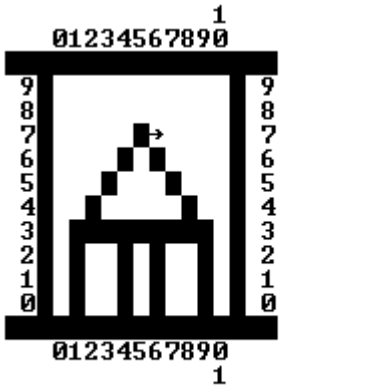
jedině tak můžeme zachytit nejen interní chyby podprogramu, ale i chyby ve vzájemné spolupráci podprogramu s ostatními částmi programu.

✧ Při analýze se snažíme vyhmátnout podproblémy, s nimiž se můžeme setkat i v jiných částech našeho programu, a naprogramovat je tak, abychom je mohli využít opakovaně.

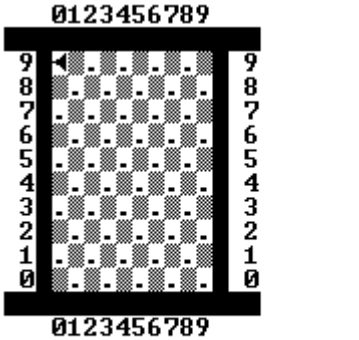
✧ Identifikátory podprogramů volíme tak, aby z nich bylo patrné, co podprogram dělá.

Úlohy

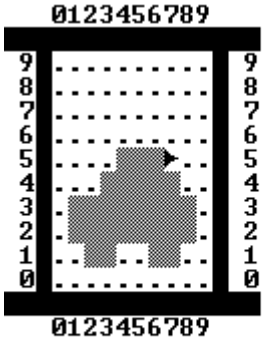
S využitím poznatků z předchozí kapitoly naprogramujte nakreslení následujících obrázků.



Obr. 8.2 Domek

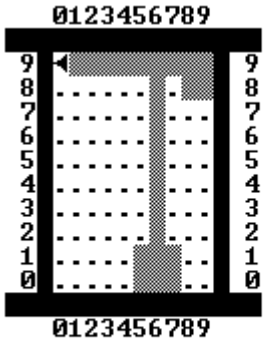


Obr. 8.3 Šachovnice



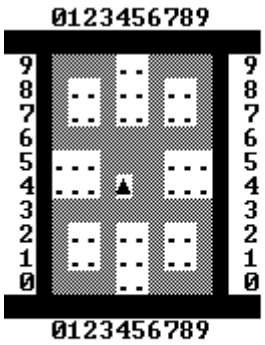
Pozice = 5/7:0 - Vychod

Obr. 8.4 Autíčko



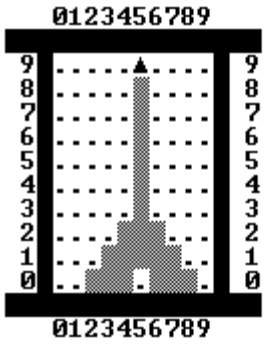
Pozice = 9/0:1 - Zapad

Obr. 8.5 Jeřáb



Pozice = 4/4:1 - Sever

Obr. 8.6 Kytka



Pozice = 9/5:1 - Sever

Obr. 8.7 Eifelovka

8.2 Návrh metodou zdola nahoru

Kromě návrhu programu metodou shora dolů, o kterém jsme hovořili dosud, se v praxi občas používá postup právě opačný, tj. návrh programu zdola nahoru. Učebnice moderního programování tento postup většinou zatracují, nicméně jsou situace (i když spíše výjimečné), kdy je postup zdola na-

nahoru přece jenom výhodnější.

Při postupu zdola nahoru vycházíme z úplně jiné filozofie řešení. Můžeme ji interpretovat tak, že při ní nám naopak připadá, že instrukční soubor použitého procesoru neodpovídá řešené úloze a my se pokoušíme vytvořit nad tímto instrukčním souborem soubor nový: soubor instrukcí, které nám umožní řešit naši úlohu snáze.

Vyřešme např. úlohu, při níž máme Karla naučit morseovku. Jak jsme si řekli v minulé kapitole, první, co si musíme vždy ujasnit, je koncepce programu. Příkaz „Naučte Karla morseovku!“ je příliš obecný. Musíme jej tedy konkretizovat.

Jednou z možností, které se nabízejí, je definovat pro každé písmeno podprogram, který toto písmeno na dvorku pomocí značek „vymaluje“. Hlavní program pak bude sestávat z posloupnosti příkazů – podprogramů malujících jednotlivá písmena, např. (v Pascalu):

```
A;B;E;C;E;D;A;
```

Druhou otázkou je, jak budeme Morseovy znaky zapisovat. Většinou se tyto znaky zapisují vodorovně, ale tento způsob zápisu je pro nás nevhodný, protože v zájmu čtvercové podoby dvorku je mezi každá dvě sousední pole vložena mezera. Výhodnější tedy pro nás asi bude zapisovat jednotlivé znaky svisle. A abychom měli život co nejjednodušší, budeme každý znak „vztyčovat“ do samostatného sloupce od nultého řádku směrem vzhůru.

Pokud bychom pracovali podle metody shora dolů, měli bychom nyní naprogramovat jednotlivé znaky. Kdybychom se touto cestou vydali, asi bychom v tomto jednoduchém případě nenarazili. Jenže jakmile by však byla úloha jen o trochu složitější, asi bychom těžko zaručili, že nebudeme vyvíjet dva podprogramy pro řešení dvou natolik podobných činností, že by pro jejich naprogramování stačil podprogram jeden.

Půjdeme na to proto z druhého konce: pokusíme se vytvořit takové základní procedury (tj. takový nový instrukční soubor), aby je mohly použít všechny podprogramy z vyšší vrstvy.

Všem je asi jasné, že základními podprogramy, které bychom měli vytvořit, je podprogram pro tečku a čárku a podprogram pro přechod z jednoho sloupce do sloupce dalšího – nazvěme je *Tečka*, *Čárka* a *Další*.

Dejme tomu, že se rozhodneme, že tečku znázorníme položením značky a čárku položením značky do dvou sousedních polí nad sebou. Mezi jednotlivými tečkami a čárkami jedno pole vynecháme. Abychom si programování dále co nejvíce usnadnili, přesuneme Karla po vyznačení tečky nebo čárky

do výchozí pozice pro další tečku či čárku.

Tyto úvahy nás vedou k následujícím podobám podprogramů *Tečka* a *Čárka*:

```
procedure (*****) Tecka; (*****)
begin
  Poloz;
  Dvojkrok;
end;
(********** Tecka **********)

procedure (*****) Carka; (*****)
begin
  Pokrok;
  Poloz;
  Dvojkrok;
end;
(********** Carka **********)
```

Nesmíme zapomenout obě definice také ihned odladit – např. pomocí hlavního programu:

```
{********** Hlavní program **********}
begin
  VlevoVbok;
  Tecka;
  Carka;
  Tecka;
end.
```

Přejděme nyní k podprogramu *Další*. Při bližším pohledu záhy zjistíme, že tentokrát situace tak jednoduchá nebude. Prozatím totiž ještě neumíme Karla dovést z libovolné pozice ke zdi. Ponecháme proto dovedení ke zdi na každém písmenu (ono ví, jak daleko od zdi Karla odvedlo) a omezíme se pouze na vlastní přechod mezi sloupci: předtím předpokládáme, že Karla přebíráme u jižní zdi otočeného na jih. Definice je pak jednoduchá:

```
procedure (*****) Dalsi; (*****)
begin
  VlevoVbok;
  Krok;
  VlevoVbok;
end;
(********** Dalsi **********)
```

Odladíme ji (podobu ladicího programu necháme na vás) a tím máme připraveno vše, co potřebujeme. Můžeme se tedy pustit do programování vlastních písmen:

```
procedure (*****) A; (*****)
```

```
begin
  Tecka;
  Carka;
  CelemVzad;
  Krat( 5, Krok );
  Dalsi;
end;
(********** A **********)

procedure (*****) B; (*****)
begin
Carka;
Tecka; Tecka; Tecka;
CelemVzad;
Krat( 9, Krok );
Dalsi;
end;
(********** B **********)

procedure (*****) C; (*****)
begin
  Carka; Tecka; Carka; Tecka;
  CelemVzad;
  Krat( 10, Krok );
  Dalsi;
end;
(********** C **********)<F255D>
```

A už jsme narazili. Písmeno C se nám takto nakreslit nepodaří, protože Karel při kreslení druhé tečky narazí do zdi. Můžeme to sice obejít definicí:

```
procedure (*****) C; (*****)
begin
  Carka; Tecka; Carka; Poloz;
  CelemVzad;
  Krat( 8, Krok );
  Dalsi;
end;
(********** C **********)
```

ale tím se s nedostatečností dvorku nevyrovnáme, protože písmeno CH a větší část číslic se nám na dvorek nevejde po žádné úpravě. Nezbývá nám tedy, než se vrátit zpět a zcela změnit koncepci (je tu samozřejmě ještě možnost zvětšit dvorek, ale mysleme si, že velikost dvorku 10x10 je pro nás závazná).

Kdo zná tabulku znaků v počítačích PC (najdete ji v manuálu k operačnímu systému), ví, že v její horní polovině jsou mezi semigrafickými výplňovými znaky nejen znaky pro vyplnění celé plochy vyhrazené pro znak (jeden z nich s kódem 177 využíváme), ale i znaky pro vyplnění pouze horní nebo dolní poloviny této plochy. Nabízí se tedy možnost znázornit tečku tak, že vyplníme jenom jednu polovinu plochy, a čárku tak, že vyplníme tuto plochu

celou. Můžeme to ještě vylepšit o to, že mezera mezi tečkou a čárkou bude vždy polovinou plochy, takže čárka pak bude znázorněna buď vyplněním celé plochy (znak 219 – na rozdíl od „šedivého“ znaku 177 je tento znak plný), nebo vyplněním sousedících polovin dvou sousedních ploch (znak 220 pro dolní a 223 pro horní polovinu).

Upravme tedy dvorek tak, že zdi budou znázorňovány znakem 177, jedna položená značka znakem 223 (vyplněná dolní polovina), dvě značky znakem 219 (vyplněná celá plocha) a tři značky znakem 220 (vyplněná horní polovina).

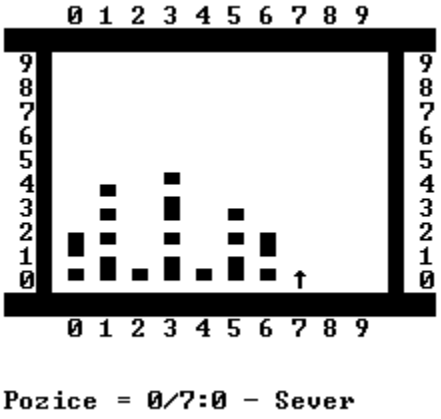
Při tomto přístupu se nám sice programování trochu komplikuje, protože budeme muset rozlišovat tečku znázorněnou v horní polovině znaku (horní tečku – *TH*) a tečku znázorněnou v dolní polovině znaku (dolní tečku – *TD*). Stejně budeme muset rozlišovat čárku začínající v dolní polovině znaku a vyplňující celý znak (dolní čárku – *CD*) a čárku začínající v horní polovině znaku a přesahující do dolní poloviny znaku následujícího (horní čárku – *CH*).

```
procedure (*****) TH; (*****)
begin
  PoKrok;
end;
(********** TH **********)

procedure (*****) TD; (*****)
begin
  Poloz; Poloz; PoKrok;
end;
(********** TD **********)

procedure (*****) CD; (*****)
begin
  Poloz; PoKrok;
end;
(********** CD **********)

procedure (*****) CH; (*****)
begin
  TH; TD;
end;
(********** CH **********)
```

Po těchto úpravách budou tedy definice prvních tří písmen vypadat následovně:

```
procedure (*****) A; (*****)
begin
  TH; CH;
  CelemVzad;
  Krat( 3, Krok );
```

```
  Dalsi;
end;
(********** A **********)
procedure (*****) B; (*****)
begin
  CH; TD; TD; TD;
  CelemVzad;
       Krat( 5, Krok );
```



Obr. 8.8 Slovo ABECEDA, které Karel napsal morseovkou

```
Dalsi;
end;
(********** B **********)

procedure (*****) C; (*****)
begin
  CH; TD; CD; TH;
  CelemVzad;
  Krat( 5, Krok );
  Dalsi
end;
(********** C **********)
```

Výsledný program najdete v souboru *MORSE.CPP* resp. *MORSE.PAS*. Podoba dvorku po napsání slova ABECEDA je na následujícím obrázku:

Příklad

Naučte Karla kreslit číslice podle schématu na následujícím obrázku:

Obr. 8.9 Číslice

Svá řešení si můžete porovnat s programem CISLICE.

8.3 Porovnání obou postupů

Než se začneme zabývat výhodami a nevýhodami obou uvedených postupů, zopakujeme si jejich charakteristiky. Vypůjčíme si slova dr. Suchomela, který tvrdí:

- ✧ U návrhu shora dolů vlastně říkáme: „Tato úloha je pro nás příliš složitá. Rozložíme ji na řadu úloh menších a ty na ještě menší, dokud nedostaneme soubor úloh, které jsou dostatečně jednoduché, abychom je mohli vyřešit.“
- ✧ U návrhu zdola nahoru místo toho říkáme: „Tento počítač mé úloze nevyhovuje, protože jeho základní operace jsou příliš základní. Použijeme proto základních operací a vytvoříme výkonnější počítač a operace tohoto počítače použijeme k vytvoření ještě výkonnějšího počítače, který již moji úlohu snadno vyřeší.“

  Anebo pro ty, kterým se líbí básničky:

Při postupu shora dolů
je úloha složitá.
Řada úloh jednodušších
snadněji se spočítá.
Při metodě zdola vzhůru instrukce si namnoze

rozšíříme, vylepšíme,
ať lépe slouží úloze.

Charakteristické rysy obou přístupů jsou shrnuty v tabulce, kterou jsem si vypůjčil od dr. Lexy. Tato tabulka neplatí pouze pro programování, a proto jsou v ní použity obecnější termíny **prvek**, což je náš podprogram, a **základna**, které jsme doposud říkali „soubor instrukcí“.

	Shora dolů	**Zdola nahoru**
Podstatou postupu je	analýza	syntéza
Nový prvek je nám	prostředkem	cílem
Že nový prvek půjde nad danou základnou vytvořit	nevíme jistě, spíše předpokládáme	víme jistě
Jaké bude mít nový prvek vlastnosti	víme přesně	to přizpůsobíme možnostem základny
Jak bude nový prvek zkonstruován	vyřešíme později	tím se musíme zabývat hned
konkrétní použití nového prvku	známe	nezajímá nás (může být libovolné)
Předpokládaná aplikace	v jednom projektu	v mnoha projektech
Návratnost vynaložené námahy	brzká, jednorázová	opožděná, postupná

Tab. 8.1 Porovnání metody shora dolů a zdola nahoru

K této tabulce přidáme ještě po jednom nebezpečí pro každou z metod:

U metody shora dolů nám hrozí, že dostatečně jasnozřivě neodhadneme opakovatelnost téhož prvku (viz např. podprogram *Přejdi* v prvním návrhu podprogramu *Sokl* z předminulé podkapitoly), a budeme proto tentýž anebo velice podobný problém programovat vícekrát.

U metody zdola nahoru nám naopak hrozí, že se možnosti námi vytvářeného nového souboru instrukcí (základny) nesetkají s požadavky řešeného úkolu – viz např. nutnost změny reprezentace dat při řešení morseovky.

9. Základy modulární výstavby programu

9.1 Prototypy podprogramů

Než se pustíme do výkladu modulární výstavby programu, chtěli bychom se vám svěřit s jednou z našich výhrad vůči klasickému stylu programování, a to s tím, že nás tento klasický styl nutí psát programy zezdola nahoru.

Zatím jsme si řekli, že v každém místě programu můžeme použít pouze ty funkce, které jsme již definovali. To znamená, že nejprve bychom měli naprogramovat nejzákladnější procedury, které již žádné další procedury nevyužívají, pak vždy procedury, které využívají pouze procedury již hotové, a na samém konci pak teprve hlavní program; v Pascalu tomu odpovídá i celková syntax programu.

Ve skutečnosti ale je nejvýhodnější pro řešení většiny úloh metoda shora dolů, při níž vymýšlíme program právě opačně. Ukažme si nyní, jak to musíme zařídit, abychom mohli uspořádat jednotlivé definice procedur v programu tak, jak nám to zrovna nejvíce vyhovuje.

Nejprve si musíme vysvětlit rozdíl mezi deklarací a definicí podprogramu. Jak vypadá definice podprogramu již víme. Deklarace podprogramu se od jeho definice liší tím, že vůbec nic neříká o tom, jak bude program plnit svůj úkol, ale pouze popisuje, jak bude komunikovat se svým okolím[13]. Její stále velmi zjednodušená, pro nás ale zatím naprosto dostatečná syntaktická definice má v Pascalu tvar

deklarace_procedury: { Pascal }
procedure *identifikátor*; [**forward**;]

Jak vidíte, syntax deklarace procedury má v Pascalu dva možné tvary: s klíčovým slovem **forward** a bez něj. Jejich použití závisí na tom, kde se daná

[13] V literatuře se také setkáte s rozlišováním **definiční deklarace** (ta proceduru nebo funkci definuje, říká, co bude dělat a jak) a **informativní deklarace** (ta pouze informuje překladač, že někde jinde je definována funkce či procedura s daným identifikátorem a případně dalšími vlastnostmi). Definiční deklarace popisuje implementaci procedury, prototyp – informativní deklarace – pouze její rozhraní, tedy způsob její komunikace s okolím. Znalost rozhraní překladači stačí ke správnému zpracování volání procedury.

deklarace vyskytuje. Deklarace, které se vyskytují v hlavě modulu (co to je hlava a tělo modulu, si řekneme za chvíli), klíčové slovo **forward** používat nesmějí, kdežto deklarace podprogramu, které se vyskytují v těle modulu, je naopak použít musejí.

V C++ má zjednodušená, ale pro nás postačující syntaktická definice deklarace procedury tvar:

deklarace_procedury: // C++
void *identifikátor* ( ) ;

Jak vidíte, od definice se liší tím, že místo těla následuje za hlavičkou procedury pouze středník.

Deklaracím procedur se v C++ říká prototypy. Tento výraz se nám líbí natolik, že jej budeme používat i v Pascalu.

Jediná komunikace, které jsou v současné době naše podprogramy schopny, je volání jednoho podprogramu jiným. K tomu, abychom podprogram správně zavolali, nám v současné chvíli stačí vědět, že se jedná o podprogram, a znát jeho jméno.

Deklarací podprogramu systému oznamujeme, že podprogram tohoto jména již někde existuje anebo bude existovat, a tím po systému vlastně nepřímo chceme, aby nám tento podprogram dovolil používat, byť na jeho definici ještě nenarazil.

Už asi tušíte, jak zařídit, abychom mohli procedury definovat v pořadí, které vyhovuje nám: na vhodném místě uvedeme prototypy všech procedur a od té chvíle máme v pořadí jejich definování naprosto volnou ruku.

Jediná drobná obtíž, která ještě zbývá, je pascalský hlavní program, který musí být podle syntaktických pravidel vždy na konci. Z důvodů, o nichž budeme hovořit v následujících částech této kapitoly, konstruujeme však většinou pascalské hlavní programy tak, že nás tento problém nepálí. Pokud by nám hlavní program na konci vadil, není nic jednoduššího než definovat zvláštní proceduru (pojmenujeme ji např. *Hlavni***)**, přemístit obsah hlavního programu do jejího těla. Hlavní program pak bude obsahovat jediný příkaz, volání procedury *Hlavní*, a takový hlavní program může být klidně i na konci souboru.

9.2 Modul a jeho části

Přejděme nyní k vlastnímu tématu kapitoly. Všechny programy, které jsme doposud vytvořili, byly velice jednoduché. Jistě jste si však všimli, že se nám v posledních několika programech stále opakují některé pasáže – konkrétně definice základních rozšiřujících příkazů *ČelemVzad*, *VpravoVbok*, *Dvojkrok* a *PoKrok*. Určitě vám již začala vrtat v hlavě otázka, zda opravdu musíme tyto jednou naprogramované procedury stále opisovat, zda bychom se s těmito společnými součástmi, které se stále opakují, nemohli vypořádat nějak elegantněji.

Takové řešení existuje a jmenuje se **modul**. Než si ale ukážeme jeho konkrétní použití na našem příkladě, povíme si o modulech něco všeobecně.

Modulem budeme nazývat logicky samostatnou, relativně uzavřenou část programu, naprogramovanou v samostatném souboru, resp. skupině souborů a překládanou jako jeden celek.

Přirovnáme-li program k celému světu, pak moduly budou v tomto světě představovat státy. Každý modul má nějaké prostředky – podprogramy (časem se dozvíme i o jiných typech prostředků). Stejně jako mezi skutečnými státy, i mezi moduly kvete čilý zahraniční obchod, při němž si moduly své prostředky navzájem poskytují. Každý modul může některé ze svých prostředků – nebo také všechny – **vyvézt** (**exportovat**), tj. nabídnout je ostatním modulům k použití, a naopak může od ostatních modulů některé prostředky **dovézt** (**importovat**), tj. použít je.

Základní pravidla mezimodulového zahraničního obchodu jsou následující.

- ✧ **Modul může dovážet pouze ty prostředky, které některý jiný modul nabídl k vývozu.** O prostředcích, které modul nenabídne k vývozu, ostatní moduly nevědí, a proto je také nemohou dovézt (moduly jsou slušné státy a špionáž nepěstují).
- ✧ **Modul může dovážet pouze ty prostředky, k jejichž dovozu se veřejně přizná**, tzn. že nějakým dohodnutým způsobem předem naznačí, že se chystá tyto prostředky dovážet.

Nad dodržováním těchto pravidel bdí překladač a sestavovací program. Modul obecně obsahuje následujících šest sekcí:

Export Tato sekce obsahuje deklarace a někdy i definice prostředků, které je modul ochoten vyvážet.

Import Tato sekce obsahuje informace o prostředcích, které se modul chystá dovážet.

V Pascalu uvádíme v sekci importu jména modulů, jejichž vyvážené prostředky bude modul dovážet. V dalším textu budeme proto někdy stručně hovořit o dovážených modulech.

V C++ uvádíme v sekci importu deklarace (a někdy i definice) dovážených prostředků – nejčastěji používanou formou je vložení hlavy modulu, který tyto prostředky vyváží, do zdrojového textu dovážejícího modulu.

Inicializace Popisuje činnost, která je nutná pro zdárnou práci modulu a která se provede ještě před spuštěním hlavního programu.

Vlastní tělo modulu Obsahuje definice všech prostředků modulu s výjimkou těch, jejichž definice již byly uvedeny v exportu.

Finalizace Popisuje činnost, která je nutná pro zdárné ukončení práce a která se provede po ukončení hlavního programu před předáním řízení operačnímu systému.

Moduly naprogramované v jazycích, které probíráme, budeme rozdělovat na dvě části: na **hlavu modulu**, která definuje export a tu část importu, která je potřeba pro definici exportu, a na **tělo modulu** obsahující všechny ostatní části včetně zbytku importu.

V C++ je hlava modulu vždy v samostatném souboru, kterému podle konvence dáme příponu .H nebo .HPP. Tomuto souboru budeme říkat **hlavičkový soubor** nebo **záhlaví** (mezi programátory se často setkáte i s termínem **hedr** z anglického *header*).

I když programátor dostane nějaký modul pouze v přeložené podobě, tj. jako relativní modul v souboru s příponou .OBJ, protože jeho autor nechce z nejrůznějších důvodů zdrojový text tohoto modulu uvolnit, zdrojový text záhlaví programátor dostat musí. Pokud chce totiž programátor v jiném modulu (nazvěme jej *MB*) dovézt některé z objektů, které tento modul (nazvěme jej

MA) vyváží, dosáhne toho tím, že do zdrojového textu modulu *MB* vloží záhlaví modulu *MA* – podrobněji si o tom povíme za chvíli.

V Pascalu bývá hlava i tělo modulu v jednom souboru. Když programátor dostane nějaký modul pouze v přeložené podobě, tj. pouze jako relativní modul v souboru s příponou .TPU (Turbo Pascal unit), protože jeho autor nechce z nejrůznějších důvodů zdrojový text tohoto modulu uvolnit, může zjistit přesnou podobu mezimodulového rozhraní nanejvýš z doprovodné dokumentace. Pascalský překladač totiž zdrojový tvar definice mezimodulového rozhraní nepotřebuje, protože si všechny potřebné informace dokáže vyhledat v souboru .TPU.

Někdy se nám zdá, že by stálo za úvahu akceptovat zvyky jazyka C++ a u modulů, které předáváme kolegům či zákazníkům v přeloženém tvaru, definovat záhlaví v samostatném souboru (dejme tomu s příponou .TPH), který ve vhodném místě do zdrojového souboru vložíme. K této otázce se za chvíli ještě vrátíme.

Ze zmínek o modulech v předchozích kapitolách kursu si asi pamatujete, že jsme několikrát hovořili o modulu KAREL, který jste dostali v přeložené podobě na doprovodné disketě. Podívejme se, co bychom mohli říci o jeho komponentách:

Export modulu KAREL si mohou uživatelé C++ prohlédnout v souboru KAREL.HPP.

V duchu našich předchozích úvah jsme vytvořili také hlavičkový soubor pro pascalskou verzi Karla, takže si můžete export modulu Karel prohlédnout v souboru KAREL.TPH.

Import modulu je jeho soukromá věc, a proto se o něm bez přístupu ke zdrojovému textu programu nemůžeme nic dozvědět.

Inicializace modulu KAREL připraví Karlův dvorek do výchozí podoby a vykreslí jej na obrazovku.

Finalizace modulu KAREL ještě jednou vykreslí aktuální podobu Karlova dvorku, oznámí konec programu a počká na stisk klávesy.

Tělo modulu je nám samozřejmě bez zdrojového textu nepřístupné. Víme o něm však, že obsahuje definice všech příkazů, které používáme.

9.3 Prázdný modul

Jak jsme si řekli již v kapitole 4.1, i naše programy jsou pro překladač moduly. Jsou však výjimečné tím, že obsahují hlavní program. Takovýmto modulům budeme v dalším textu říkat **hlavní moduly** (hlavní modul může být v každém programu pouze jeden), na rozdíl od modulů, které hlavní program neobsahují a kterým budeme říkat **řadové moduly**. Připravme si proto soubor PRAZDNY.CPP, resp. PRAZDNY.PAS tak, abychom z něj později snadno vytvořili jak hlavní, tak řadový modul.

Pascalisté si otevřou soubor PAS\PRAZDNY1.PAS (jsme v adresáři \KURS) a přejmenují jej na (uloží jej jako) PAS\PRAZDNY.PAS. Tím původní soubor PRAZDNY.PAS, který byl vytvořen pouze pro snadnou tvorbu hlavních programů, přepíší souborem, který umožňuje snadno vytvořit jak hlavní, tak řadový modul. Podívejme se nyní na tento soubor blíže.

Pomineme-li komentáře, začínaly všechny naše dosavadní programy klíčovým slovem **Program**. Toto klíčové slovo označuje v Turbo Pascalu modul, který obsahuje hlavní program. Řadový modul se od modulu s hlavním programem liší tím, že začíná klíčovým slovem **Unit**, za nímž následuje identifikátor modulu a středník. Proto je v souboru PRAZDNY.PAS za řádkem s příkazem **Program** řádek s příkazem **Unit** a podle toho, který modul zrovna vytváříme, smažeme vždy ten řádek, který nebudete potřebovat.

Podívejme se nyní na syntax řadového pascalského modulu. Můžeme ji popsat následovně:

Řadový_modul:
Unit *Identifikátor* ;
\+ [**Uses** [*identifikátor* [, *identifikátor*] $_{opak}$;]
\+ **interface** *rozhraní*
\+ **implementation**
\+ [**Uses** [*identifikátor* [, *identifikátor*] $_{opak}$;]
\+ *tělo_modulu*
\+ **begin** *inicializace* **end.**

Aby systém dokázal soubor se zdrojovým textem modulu najít na disku, musí být identifikátor modulu totožný s názvem souboru, v němž je zdrojový text modulu, a přípona jména souboru musí být *.PAS*.

Řadový modul se od modulu s hlavním programem, který nic vyvážet nemusí, liší také tím, že musí něco vyvézt. Kdyby nic nevyvezl, nemohl by ni-

kdo žádný z jeho prostředků použít, a nebyl by proto žádný důvod, aby se tento modul účastnil práce programu. A protože řadový modul nemůže běžet samostatně, nebyl by prostě k ničemu.

Za klíčovým slovem **Unit** musí proto následovat klíčové slovo **interface**, které uvádí část zdrojového textu, jež definuje export modulu, a za touto částí pak klíčové slovo **implementation**, které tuto část odděluje od vlastního těla modulu.

Část souboru mezi klíčovými slovy **interface** a **implementation** tedy tvoří hlavu modulu s definicemi mezimodulového rozhraní. Pokud byste chtěli tuto hlavu uvést v samostatném souboru, je pro vás v prázdném modulu připravena komentářová direktiva[14] (komentář, jehož prvním znakem je dolar) tvaru

```
{$I jméno_souboru}
```

která přikáže překladači, aby do zdrojového textu vložil místo této direktivy obsah uvedeného souboru. Pokud hlavu modulu do samostatného souboru umístit nehodláte, musíte tuto direktivu smazat.

Připravená direktiva předpokládá, že jméno souboru s hlavou modulu bude stejné jako jméno souboru se zdrojovým textem modulu a bude se od něj lišit pouze příponou .TPH (Turbo Pascal header).

Pokud váš modul hodlá dovážet prostředky vyvážené jinými moduly, musí být před všemi deklaracemi (tj. v hlavním modulu ihned za příkazem **program** a v řadových modulech vzápětí za jedním z klíčových slov **interface** a **implementation**) deklarován import modulu. Toho dosáhneme tak, že na dané místo napíšeme klíčové slovo **uses** následované seznamem dovážených modulů ukončeným středníkem.

Za klíčové slovo **interface** budeme definici importu umísťovat pouze v případě, když budeme chtít importované prostředky použít již při popisu exportu. V ostatních případech ji budeme umísťovat až za klíčové slovo **implementation**.

Zbývá nám ještě inicializace a finalizace modulu. Inicializační část modulu vypadá naprosto stejně jako vlastní hlavní program v modulu s hlavním programem. Je na konci, je ohraničena příkazovými závorkami **begin** a **end** a za **end** je tečka (jak jsme si již řekli, překladač ignoruje vše, co následuje za touto tečkou). Pokud modul žádnou inicializaci nepotřebuje, zůstane tato část prázdná.

[14] V programátorské hantýrce se tyto direktivy pro překladač občas nazývají „dolarové poznámky“.

S finalizací je to u Pascalu trochu složitější. Abyste mohli definovat finalizaci modulu stejně snadno jako programátoři v C++, připravili jsme pro vás v modulu KAREL proceduru *atexit*, kterou zavoláte v průběhu inicializace (samozřejmě pouze tehdy, pokud budete potřebovat v daném modulu nějakou finalizaci definovat), přičemž za její identifikátor napíšete do kulatých závorek identifikátor procedury, která bude mít finalizaci daného modulu na starosti.

Pro tento účel je v souboru připravena prázdná procedura, jejíž identifikátor je tvořen názvem vytvářeného modulu s předponou *Exit_*. Zároveň je v inicializační části tato procedura zařazena mezi procedury, které se mají vykonat po ukončení programu. Pokud nebudete finalizaci potřebovat, tak jednoduše prázdnou definici této procedury i její použití v inicializaci smažte.

Výslednou podobu souboru *PRAZDNY.PAS* ukazuje následující program. Budete-li nyní vytvářet nový modul, budete postupovat jako dosud, pouze navíc smažete řádky, které neodpovídají typu vytvářeného modulu a případným dodatečným požadavkům (např. nechcete-li finalizaci).

```
(*********************************************************
**@.PAS
**
**
**
*********************************************************)

Program Pascal;
Unit @;

interface
{$I @.TPH - Hlava modulu}

implementation
uses Karel, Chyba_;

(** Prototypy lokálních podprogramů **)
(************* Vyvážené podprogramy *************)
(************** Lokální podprogramy **************)
(********************** Finalizace **********************)

procedure (*****) Exit_@ (*****);
begin
end;
(********** Exit_@ **********)

(****************** Hlavní program ******************)
(******************** Inicializace ********************)
begin
atexit( Exit_@ );
end.

#if 0      // Následuje prázdná definice podprogramu
```

```
procedure (*****) _ (*****);
begin
end;
(********** _ **********)
#endif     // Konec prázdné definice podprogramu
(********************** @.PAS ********************)
```

Programátoři v C++ si otevřou soubor *PRAZDNY1.CPP* a přejmenují jej na *PRAZDNY.CPP*. Tím původní soubor *PRAZDNY.CPP*, který byl vytvořen pouze pro snadnou tvorbu hlavních programů, přepíší souborem, který umožuje snadno vytvořit jak hlavní, tak i řadový modul. Podívejme se nyní na tento soubor blíže.

V C++ se modul s hlavním programem odlišuje od ostatních modulů pouze tím, že obsahuje podprogram, který se jmenuje *main***,** takže v tomto směru nejsou žádné úpravy nutné. Ukážeme si, jak vypadají jednotlivé komponenty modulu v C++.

Hlavičkový soubor je tedy hlavou modulu a odpovídající soubor .CPP je pak jeho tělem. Vzhledem k povaze záhlaví (samostatný soubor) bývá častým zvykem, že hlavy řady modulů jsou společně v jednom hlavičkovém souboru.

Import modulu deklarujeme v C++ tak, že do zdrojového souboru včleníme pomocí příkazu **#include** hlavu modulu, jehož exportované prostředky dovážíme. Syntaktická definice příkazu[15] **#include** je

Příkaz_include:
 # include <*Jméno_souboru*>
 # include "*Jméno_souboru* "

Jak jste si jistě všimli, jméno vkládaného hlavičkového souboru můžeme uvést v lomených (úhlových) závorkách nebo v uvozovkách. Bez dalšího bližšího vysvětlování se dohodneme na tom, že jména hlavičkových souborů dodaných se systémem budeme uvádět v lomených závorkách a jména hlavičkových souborů, které jsme vytvořili sami, v uvozovkách.

Bývá dobrým zvykem, že modul pro jistotu doveze i svůj vlastní export. Umožní tak překladači zkontrolovat, zda skutečné vlastnosti vyvážených prostředků odpovídají vlastnostem uvedeným v exportu. Za dovozem modulu *Karel* proto najdete ještě řádek s příkazem

```
#include "@.HPP"
```

[15] Přesně vzato nejde o příkaz, ale o direktivu pro tzv. preprocesor – program, který zpracovává zdrojový text našeho programu před vlastním překladem. Za těmito direktivami se nepíše středník.

v němž znak @ zaměníte za název odpovídajícího souboru.

Skutečnost, že je určitý podprogram určen k inicializaci nebo finalizaci modulu, oznamujeme systému v Borland C++ pomocí speciálního příkazu[16] **#pragma**, jehož syntaktická definice je

Příkaz_pragma:
#pragma startup *Identifikátor_procedury*
#pragma exit *Identifikátor_procedury*

Možnost **startup** používáme při určování podprogramů pro inicializaci a volbu **exit** při určování podprogramu pro finalizaci.

Abychom při případném zadávání inicializace nebo finalizace modulu již nemuseli přemýšlet nad přesnou podobou klíčových slov v příkazu **#pragma**, zapíšeme do prázdného souboru oba příkazy s tím, že pokud je nebudeme potřebovat, tak je smažeme.

Poslední úpravou je úprava hlaviček podprogramů. Všimněte si, že do hlavičky prázdné definice podprogramu bylo vloženo klíčové slovo **static**. Takto je třeba označit všechny definice i deklarace procedur a funkcí, které daný modul nehodlá vyvážet, a které proto označíme za **lokální**.

Podrobnější rozebírání významu a funkce klíčového slova **static** je při současném stavu vašich znalostí ještě předčasné a budeme se mu věnovat až v příštím dílu. Zatím vám jenom poradíme, abyste si dali pozor, aby podprogramy, které hodláte z daného modulu vyvážet, byly definovány bez klíčového slova **static**. Naopak definice podprogramů, které budete používat pouze v těle daného modulu, by měly být tímto klíčovým slovem uvedeny.

Tím jsme s úpravami souboru *PRAZDNY.CPP* hotovi. Abychom měli usnadněnu práci i při tvorbě hlav modulů, vytvoříme obdobně soubor *PRAZDNY.HPP*.

```
//**********************************************************
//
// @.CPP
//
//
//
//**********************************************************
#include "KAREL.HPP"
#include "@.HPP"

// Tato procedura smí být definována pouze v jednom modulu
void Chyba() {}
```

[16] Opět jde ve skutečnosti o direktivu preprocesoru, a proto se za ní nedělá středník.

```
/******* Prototypy lokálních funkcí *******/
/****************** Hlavní program ******************/

void /*****/ main /*****/ ()
{
}
/********** main **********/

/************* Vyvážené podprogramy *************/
/************** Lokální podprogramy **************/

/******** Inicializace a finalizace ********/
#pragma startup Init_@
#pragma exit Exit_@

#if 0      // Následuje prázdná definice podprogramu
static void /*****/ _ /*****/ ()
{
}
/********** _ **********/
#endif     // Konec prázdné definice podprogramu

/********************** @.CPP **********************/
```

9.4 Řadový modul SPOLECNE

Vytvořme nyní modul[17] se základními rozšiřujícími příkazy. Otevřeme soubor *PRAZDNY.CPP*, resp. *PRAZDNY.PAS* a uložíme jej pod jménem *SPOLECNE.XXX*, kde *.XXX* znamená podle okolností *.CPP* nebo *.PAS*. Zapíšeme do hlavy modulu deklarace a do jeho těla definice příkazů *ČelemVzad*, *VpravoVbok* a *PoKrok*. Výsledná podoba programu v Pascalu bude následující:

```
(*********************************************************
 **
 **  SPOLECNE.PAS
 **
 **  Definice často používaných procedur
 ** *******************************************************)

Unit Spolecne;

interface
  procedure CelemVzad;
  procedure VpravoVbok;
  procedure PoKrok;

implementation
  uses Karel;
```

[17] V těchto výpisech je uvedena pouze podmnožina programů, které v modulech SPOLECNE najdete na doprovodné disketě.

```
(************* Vyvážené podprogramy *************)
procedure (*****) CelemVzad (*****);
begin
  VlevoVbok;
  VlevoVbok;
end;

(********** CelemVzad **********)
procedure (*****) VpravoVbok (*****);
begin
  VlevoVbok;
  VlevoVbok;
  VlevoVbok;
end;
(********** VpravoVbok **********)

procedure (*****) PoKrok (*****);
begin
  Poloz;
  Krok;
end;
(**********PoKrok **********)

(******************** Inicializace ********************)
begin
end.
(********************** SPOLECNE.PAS ********************)
```

Pokud se rozhodneme používat i v Pascalu hlavičkové soubory, bude soubor *SPOLECNE.TPH* mít tvar

```
(*********************************************************
 **
 **  SPOLECNE.TPH
 **
 **  Export modulu s často používanými procedurami
 ** *******************************************************)

Unit Spolecne;

interface
  procedure CelemVzad;
  procedure VpravoVbok;
  procedure PoKrok;

(********************** SPOLECNE.PAS ********************)
```

a vlastní modul bude

```
(*********************************************************
 **
 **  SPOLECNE.PAS
 **
 **  Definice často používaných procedur
 ** *******************************************************)
```

```
Unit Spolecne;

interface
{$I SPOLECNE.TPH}
implementation
{ ... a dále stejně jako předtím ... }
```

V C++ bude mít hlavičkový soubor *SPOLECNE.HPP* tvar

```
//********************************************************** //
// SPOLECNE.HPP
//
// Hlava modulu SPOLECNE
//
//**********************************************************
void CelemVzad();
void VpravoVbok();
void PoKrok();
/******************** SPOLECNE.HPP ********************/
```

a soubor *SPOLECNE.HPP* bude

```
//**********************************************************
//
// SPOLECNE.CPP
//
// Definice často používaných procedur
//
//**********************************************************
#include "KAREL.HPP"
#include "SPOLECNE.HPP"

/************* Vyvážené podprogramy *************/
void /*****/ CelemVzad /*****/ ()
{
  VlevoVbok();
  VlevoVbok();
}
/********** CelemVzad **********/

void /*****/ VpravoVbok /*****/ ()
{
  VlevoVbok();
  VlevoVbok();
  VlevoVbok();
}
/********** VpravoVbok **********/

void /*******/ PoKrok /*******/ ()
{
  Poloz();
  Krok();
} /************ PoKrok ************/

/**********************SPOLECNE.CPP**********************/
```

9.5 Jednoduchý dvoumodulový program

Vytvořený modul si hned vyzkoušíme – např. na programu *Kytka*, zadaném jako domácí úkol v podkapitole 8.1. Nejprve si ale povíme o vnořených procedurách.

Procedura lokální v proceduře

Vnořené (lokální) procedury jsou specialitou Pascalu. Tento jazyk totiž umožňuje definovat proceduru uvnitř jiné procedury. Takto definovanou proceduru nezná (a nemůže používat) nikdo jiný než procedura, v níž je daná procedura definovaná.

Pokud potřebujeme definovat proceduru, kterou bude používat pouze jedna procedura, můžeme tuto proceduru definovat jako vnořenou, a to tak, že její definici umístíme za hlavičku a před klíčové slovo **begin** (před tělo).

Přiznáme se, že definice vnořených procedur nepovažujeme za příliš vhodné, protože se nám zdá, že program nezpřehledňují, ale naopak spíše znepřehledňují, neboť oddalují hlavičku procedury od jejího těla. Navíc lokální procedury **nelze používat v příkazu** *Krát.* Výklad procedur lokálních v jiné proceduře jsme proto zařadili pouze pro úplnost a kromě následujícího příkladu se této konstrukci budeme spíše vyhýbat.

```
(*********************************************************
 **
 ** KYTKA.PAS
 **
 **    Řešení rozdělené do dvou modulů; květina se skládá ze
 **    4 okvětních lístků, na pohled podobných vlajce
 *********************************************************)

program _Kytka;

uses Karel, Spolecne;

(**  Deklarace lokálních objektů  **)
{ Následující deklarace jsou uvedeny proto, abychom definice mohli
  zapsat v pořadí, v němž je při postupu shora dolu vymyslíme. }
procedure Obrazec;  forward;
{$F+}            { Proceduru Vlajka chceme použít v příkazu Krát  }
procedure Vlajka; forward;
{$F-}
procedure hlavni; forward;

(*********   Lokální podprogramy   *********)
procedure (*****)  hlavni  (*****);
begin
  Obrazec;
end;
```

```
(**********   hlavni   **********)
procedure (*****)   Obrazec   (*****);
begin
  Krat( 4, Krok );
  VlevoVbok;
  Krat( 4, Krok );
  Krat( 4, Vlajka );
end;
(**********   Obrazec   **********)
procedure (*****)   Vlajka   (*****);
procedure (*****)   L   (*****);       { Vnořená procedura }
begin
  Krat( 3, PoKrok );
  VlevoVbok;
end;
begin                                  { Tělo procedury Vlajka }
  VlevoVbok;                           { Patka }
  PoKrok;
  VpravoVbok;                          { Žerď }
  PoKrok;
  PoKrok;                              { L nelze použít v příkazu Krát }
  L;L;L;L;                             { Praporek }
  CelemVzad; Krok;                     { Přesun }
  VlevoVbok; Krok;
end;
(**********   Vlajka   **********)

(**************   Hlavní program   **************)
begin
  hlavni;
end.
(**********   Hlavní program  **********)

(********************** KYTKA.PAS ********************)
```

Pokud program sestává z více modulů, musíme v C++ každý z modulů, z nichž je program sestaven, uvést v projektu. Dosáhneme toho tak, že otevřeme okno projektu, umístíme řádkový kurzor na položku, před níž chceme nový modul zařadit, a stiskneme klávesu INS. Systém otevře dialogové okno podobné oknu pro otevírání souboru a my mu ukážeme, který soubor obsahuje tělo modulu. Systém pak tento soubor zařadí do projektu. Hlavní modul by u programu *Kytka* mohl vypadat následovně:

```
//***************************************************************************
//
// KYTKA.CPP
//
// Hlavní modul programu Kytka
//
//***************************************************************************

// Dovážené moduly
#include "KAREL.HPP"
```

```
#include "SPOLECNE.HPP"

// Tato procedura smí být definována pouze v jednom modulu
void Chyba() {}

/******* Prototypy lokálních funkcí *******/
static void Vlajka();
static void L();

/****************** Hlavní program ******************/
void /*****/main /*****/ ()
{
  Krat( 3, Krok );                    //Nástup do výchozí pozice
  VlevoVbok();
  Krat( 3, Krok );
  Krat( 4, Vlajka );                  //Vlastní vykreslení
}
/********** main **********/

/************** Lokální podprogramy **************/
static void /*****/ Vlajka /*****/ ()
{
  Krat( 3, PoKrok );
  Krat(4, L );
}
/********** Vlajka **********/

static void /*****/ L /*****/ ()
{
  Krat( 3, PoKrok );
  VlevoVbok();
}
/********** L **********/

/********************** KYTKA.CPP ***********************/
```

Všimněte si, že v tomto programu jsme z modulu *Společně* použili pouze příkaz *PoKrok*. Mohli bychom říci, že ostatní příkazy jsou v něm definovány zbytečně. Pokud bychom chtěli výsledný program optimalizovat, asi bychom je z něj vyňali (časem si ukážeme ještě jiné možnosti). Pokud nám však jde hlavně o to, abychom si při vývoji programů ušetřili práci, splňuje modul *Společně* naše požadavky bezezbytku. Naopak, budeme do něj postupně přidávat definice dalších příkazů, které se ukáží při vývoji programů užitečné.

Překlad více modulů

Na závěr si ještě ukážeme, jak se systém chová při překladu programu sestaveného z několika modulů. Jak již víme, jednotlivé moduly překládáme samostatně, a to tak, že aktualizujeme editační okno s tělem modulu a stiskneme ALT-F9.

Pokud chceme přeložit a sestavit celý program (např. stiskem klávesy F9),

systém se nejprve podívá, zda nebyl od posledního překladu upravován zdrojový text některého z modulů tvořících program. A nyní pozor:

Pokud systém zjistí, že v těle některého modulu byla provedena změna, přeloží jej ještě jednou. Pokud však zjistí, že změna byla provedena v jeho hlavě, přeloží všechny moduly, které daný modul importují!

V Pascalu se navíc pokaždé překládá hlavní program, protože na rozdíl od překladače jazyka C++, který má seznam modulů včetně seznamů jejich importů uložen v projektu, musí si překladač Pascalu pokaždé znovu vybudovat strom vzájemných závislostí z informací v příkazech **Uses**.

Přestože je pascalský překladač velice rychlý, neustálé překládání hlavního modulu může někoho obtěžovat. V tom případě stačí, bude-li hlavní modul pascalského programu pouze symbolický: Bude obsahovat jen hlavní program a v něm jediný příkaz – volání klíčové procedury jednoho z podřízených modulů. Teprve tato procedura pak obsahuje skutečný hlavní program.

Při takovémto uspořádání má hlavní modul pouze několik málo řádků, které jsou naprosto zanedbatelné, a tak můžeme tvrdit, že se vždy překládají pouze moduly, které se překládat musejí.

Toto uspořádání řeší zároveň i problém s umístěním hlavního programu na konci souboru, protože hlavní modul je zcela triviální a ostatní moduly již ekvivalent hlavního programu neobsahují.

10. Cykly

V této kapitole se budeme zabývat programovými konstrukcemi, kterým říkáme **cykly**, **smyčky** nebo **iterace**. Cykly použijeme v programu všude tam, kde potřebujeme nějakou činnost vícekrát opakovat. Ani náš příkaz **Krát** není ve skutečnosti ničím jiným než zakukleným cyklem.

V životě je cyklus velice častým způsobem chování. Jako cyklus můžeme chápat např. chůzi, povolování a utahování vodovodního kohoutku, ale také vyhledávání slov ve slovníku.

Cyklus se v nejobecnější podobě skládá z následujících pěti částí s přesně určeným pořadím:

Inicializace obsahuje akce, které se provedou před vlastním spuštěním cyklu.

Vstupní podmínka nám povoluje vstup do těla cyklu. Testuje se před každým vykonáním těla cyklu. Je-li splněna, tělo cyklu se vykoná ještě jednou, není-li splněna, provádění cyklu se ukončí a pokračuje se prvním příkazem za cyklem. Pokud není vstupní podmínka splněna od samého počátku, tělo cyklu se neprovede ani jednou.

Tělo cyklu obsahuje vlastní posloupnost příkazů, které se mají opakovat.

Modifikace obsahuje akce, které mění hodnoty testovaných podmínek. Tato část se samostatně vyskytuje pouze u cyklu s parametrem, o němž budeme hovořit v příštím dílu. V cyklech, o nichž si budeme povídat v této kapitole, bývají modifikující akce velice často součástí těla cyklu či některé z testovaných podmínek.

Výstupní podmínka nám povoluje cyklus opustit. Testuje se po každém provedení těla cyklu a případných modifikací. V některých jazycích provádění cyklu skončí, je-li výstupní podmínka splněna, v jiných naopak skončí, není-li tato podmínka splněna. Pokud výstupní podmínka neumožňuje cyklus ukončit, pokračuje se testováním vstupní podmínky.

Žádná z uvedených částí však není povinná. O tom, že může chybět modifikace, jsme již hovořili. Bezproblémová je i situace, kdy chybí inicializace a tělo cyklu. Problematická by však mohla být absence některé z podmínek. Pokud některá podmínka chybí, chová se cyklus tak, jako by její vyhodnocení vedlo k setrvání v cyklu. Protože se pak nejedná o klasickou podmínku, ale o podmínku vyjádřenou konstantou *ANO* nebo *NE*, budeme o takové podmínce říkat, že je **degenerovaná**.

Složený příkaz

Dříve než přistoupíme k vlastnímu výkladu cyklů, musíme povědět něco více o příkazech. Doposud jsme poznali jedinou podobu příkazu – volání procedury. V této kapitole se navíc seznámíme se dvěma podobami příkazu cyklu. Abychom však mohli využít všechny možnosti, které nám cykly nabízejí, musíme se seznámit ještě s jedním typem příkazu – se složeným příkazem.

V syntaktických definicích programových konstrukcí se velice často dozvíme, že na nějakém místě smíme zapsat příkaz (přesněji řečeno jediný příkaz). My však budeme potřebovat na daném místě použít několik příkazů. V tuto chvíli máme dvě možnosti: buď onu posloupnost příkazů definujeme jako tělo procedury, kterou v daném místě zavoláme, anebo ji uzavřeme do příkazových závorek, a tím z ní vytvoříme jediný složený příkaz.

Syntaktická definice složeného příkazu[18] je tedy v Pascalu:

Složený_příkaz:
begin *příkaz* [; *příkaz*] $_{\text{opak}}$ **end**

a v C++

Složený_příkaz:
{ [*příkaz*] $_{\text{opak}}$ **}**

Všimněte si, že i těla procedur mají podobu složeného příkazu.

V souvislosti se složeným příkazem bychom si měli něco povědět o grafické úpravě programu. Podle všeobecné konvence se příkazy, které se nacházejí uvnitř nějaké složitější programové konstrukce (např. složeného příkazu), odsazují. Až potud se všichni shodnou. Liší se však v konkrétní realizaci tohoto odsazování, tj. v umísťování příkazových závorek a v počtu znaků, o něž se bude odsazovat.

Než se vám pokusíme nabídnout přehled nejpoužívanějších způsobů grafické úpravy programů, chtěli bychom dopředu odpovědět všem, kteří si budou myslet, že se „nimráme v prkotinách“. Nevěřili byste, jakou železnou košili si na sebe přijetím nějaké standardní grafické úpravy programů navlékáte. Pokud si totiž na nějakou grafickou úpravu programů zvyknete, stanou se pro vás zanedlouho všechny programy, které tuto úpravu nebudou dodržovat, mnohem obtížněji čitelné. Do obdobných problémů se dostanete i tehdy, když

[18] Připomeňme si, že v Pascalu slouží středník k oddělení jednotlivých příkazů, kdežto v C++ je součástí příkazu.

po nějaké době zjistíte, že grafická úprava programů, kterou jste dosud používali, vám z nejrůznějších důvodů přestává vyhovovat a že byste ji potřebovali změnit.

Proto jsme se rozhodli, že vám nabídneme přehled nejpoužívanějších grafických úprav, abyste si je mohli vyzkoušet a pro jednu z nich se rozhodnout.

Klíčovou otázkou grafické úpravy zdrojových textů je způsob umísťování příkazových závorek. Podívejme se, s jakými přístupy se můžete v publikovaných programech setkat. Otevírací příkazová závorka (**begin**, resp. **{**) se umísťuje:

a) Na stejném řádku jako předcházející text konstrukce, k níž daný složený příkaz náleží. Výhodou je ušetřený řádek a snadné přidávání a ubírání příkazů na počátku těla, nevýhodou je špatná možnost kontroly souhlasu závorek.

b) Na následujícím řádku, stejně odsazená jako předchozí řádek. První příkaz těla složeného příkazu je na stejném řádku. Výhodou je ušetřený řádek a snadná kontrola přítomnosti závorky, nevýhodou je trochu pracnější přidávání a odmazávání řádků z počátku těla.

c) Na následujícím řádku stejně odsazená jako předchozí řádek. První příkaz těla složeného příkazu je na dalším řádku (samozřejmě již odsazený). Výhodou je snadné přidávání a ubírání příkazů na počátku těla a snadná možnost kontroly přítomnosti závorky, nevýhodou oproti předchozím způsobům je další zabraný řádek (na obrazovku se vejde méně, a to může být někdy docela protivné).

d) Na následujícím řádku, avšak odsazená oproti předchozímu řádku. První příkaz těla složeného příkazu je na dalším řádku odsazený stejně jako otevírací příkazová závorka. Výhody a nevýhody jsou obdobné jako u předchozího způsobu, jen kontrola přítomnosti závorky je trochu obtížnější.

e) Na následujícím řádku, avšak odsazená oproti předchozímu řádku. První příkaz těla složeného příkazu je na dalším řádku a je oproti otevírací příkazové závorce ještě jednou odsazený. Výhody a nevýhody jsou stejné jako u způsobu c. Dokonce bychom mohli říci, že možnost kontroly přítomnosti příkazových závorek je ještě snazší, avšak text programu „odjíždí“ zbytečně rychle doprava.

Umístění uzavírací příkazové závorky je již do značné míry dáno umístěním závorky otevírací. Uzavírací příkazové závorky (**end**, resp. **}**) se umísťují:

a) Na stejném řádku jako poslední příkaz těla. Takto sice můžeme ušetřit jeden řádek, ale vzhledem k nepřehlednosti zápisu a špatným možnostem kontroly se v současné době téměř nepoužívá. Pokud se používá, sdružuje se většinou s otevírací závorkou umístěnou podle bodu a.

b) Na řádku následujícím za posledním řádkem těla a stejně odsazená jako řádky s příkazy těla. Toto umístění se používá spolu s umístěním otevírací příkazové závorky podle odstavců a, b, c a d.

c) Na řádku následujícím za posledním řádkem těla a předsazená oproti řádkům s příkazy těla. Toto umístění se používá s kterýmkoliv z výše zmiňovaných umístění otevírací příkazové závorky.

Preferované přístupy se mohou lišit i podle použitého programovacího jazyka. V Pascalu se poměrně často setkáme s úpravou typu b-c, c-c a e-c (v borlandských firemních programech většinou najdeme úpravu typu c-c, v C++ pak s úpravami a-c, c-c, d-b a e-c (v borlanských programech najdeme první tři). Všechny vyjmenované úpravy jsou znázorněny v následujících ukázkách:

```
Procedure ProcBC;
begin  Příkaz1;
       Příkaz2;
end;

Procedure ProcCC;
begin
  Příkaz1;
  Příkaz2;
end;

Procedure ProcEC;
  begin
       Příkaz1;
       Příkaz2;
  end;
```

```
void ProcAC() {
  Příkaz1;
  Příkaz2;
}

void ProcCC()
{
  Příkaz1;
  Příkaz2;
}

void ProcDB()
  {
  Příkaz1;
  Příkaz2;
```

```
  }
void ProcEC()
  {
      Příkaz1;
      Příkaz2;
  }
```

K dalším grafickým úpravám patří i zvýrazňování některých částí zdrojového textu pomocí hvězdiček, vkládání řídicích znaků pro tiskárnu a další úpravy, které najdete v souboru *PRAZDNY.xxx*. Znovu opakujeme: Pokud vám kterékoliv z našich zvyklostí nevyhovuje, klidně si soubory upravte podle svého. Doporučujeme vám však potom zavedená pravidla důsledně dodržovat.

Funkce

Už dříve jsme si naznačili, že podprogramy se dělí na dvě skupiny – na **procedury** a **funkce**. Doposud jsme pracovali pouze s procedurami. Funkce se od procedur liší tím, že poté co vykonají předepsanou činnost, vrátí volajícímu programu nějakou vypočtenou hodnotu – výsledek. Zápis funkce v programu pak představuje hodnotu, kterou tato funkce vypočte a vrátí.

V tomto dílu se budeme prozatím setkávat pouze s funkcemi, které vracejí logické hodnoty *ANO* a *NE*.

Nejprve se naučíme používat předdefinované (někdy říkáme primitivní) funkce z Karlova prostředí (*Zeď*, *Značka*, *Sever*, *Východ*, *Jih*, *Západ* a *Stisknuto*), později (v kapitole 11) se naučíme vytvářet i funkce vlastní.

Význam předdefinovaných funkcí – podmínek – z Karlova prostředí je následující:

Zeď Vrátí *ANO* v případě, pokud je na políčku před Karlem zeď, jinak vrátí *NE*.

Značka Vrátí *ANO* v případě, pokud je na políčku pod Karlem alespoň jedna značka, jinak vrátí *NE*.

Sever Vrátí *ANO* v případě, pokud je Karel otočen směrem na sever, tj. nahoru, jinak vrátí *NE*.

Východ Vrátí *ANO* v případě, pokud je Karel otočen směrem na východ, tj. vpravo, jinak vrátí *NE*.

Jih Vrátí *ANO* v případě, pokud je Karel otočen směrem na jih, tj. dolů, jinak vrátí *NE*.

Západ	Vrátí *ANO* v případě, pokud je Karel otočen směrem na západ, tj. vlevo, jinak vrátí *NE*.
Stisknuto	Vrátí *ANO* v případě, pokud byl od posledního testu stisknut mezerník, jinak vrátí *NE*.

Podmínky

Další věcí, o níž si před vlastním výkladem cyklů musíme pohovořit, jsou podmínky. V úvodu kapitoly jsme si řekli, že opakování či opuštění cyklu je řízeno vstupní a výstupní podmínkou cyklu. Prozatím jsme si však neřekli, jak takovou podmínku v programu zapíšeme.

Podmínku budeme chápat jako **logický výraz**, tj. výraz, který může nabývat jedné ze dvou hodnot: *ANO* a *NE*. Pokud bude hodnota výrazu *ANO*, řekneme, že podmínka je splněna; bude-li hodnota výrazu *NE*, prohlásíme o ní, že splněna není. Syntaktická definice[19] pascalského logického výrazu je následující:

Logický_výraz:
    Logická_konstanta
    Zápis_logické_funkce
    (*Logický_výraz*)
    NOT *Logický_výraz*
    Logický_výraz **AND** *Logický_výraz*
    Logický_výraz **OR** *Logický_výraz*
    Logický_výraz **XOR** *Logický_výraz*
    Logický_výraz = *Logický_výraz*
    . . .

Syntaktická definice logického výrazu v C++ je obdobná:

Logický_výraz:
    Logická_konstanta
    Volání_logické_funkce
    (*Logický_výraz*)

[19] Toto je pouze podmnožina operací, které můžeme v logických výrazech použít (proto jsou v posledním řádku tři tečky). Se všemi se seznámíme až v dalším dílu, v němž budeme hovořit o práci s daty.

! *Logický_výraz*
Logický_výraz **&&** *Logický_výraz*
Logický_výraz || *Logický_výraz*
Logický_výraz **!=** *Logický_výraz*
Logický_výraz **==** *Logický_výraz*
. . .

V obou jazycích pak můžeme definovat

Logická_konstanta:
ANO
NE

Jistě jste si v syntaktických definicích logického výrazu všimli, že výraz může být kromě samotného zápisu logické konstanty nebo funkce tvořen také operacemi s jednoduššími výrazy. Tyto operace zapisujeme pomocí **operátorů**, jejichž význam popisují následující tabulky.

Pascal	**C++**	**Název operace**
not	!	Negace (není pravda, že)
and	&&	Konjunkce (a zároveň – musejí platit obě možnosti)
xor	!=	Disjunkce (musí platit právě jedna z možností)
or	||	Alternativa (musí platit alespoň jedna z možností)
=	==	Ekvivalence (musejí platit obě zároveň nebo žádná)

Tab. 10.1 Názvy základních logických operací

Pro čtenáře, kteří se dosud se základními logickými operacemi nesetkali, uvedeme ještě podrobnou tabulku hodnot jednotlivých operací pro různé kombinace hodnot operandů.

A	**B**	**! A**	**A && B**	**A != B**	**A || B**	**A == B**
ANO	*ANO*	*NE*	*ANO*	*NE*	*ANO*	*ANO*
ANO	*NE*	*NE*	*NE*	*ANO*	*ANO*	*NE*
NE	*ANO*	*ANO*	*NE*	*ANO*	*ANO*	*NE*
NE	*NE*	*ANO*	*NE*	*NE*	*NE*	*ANO*

Tab. 10.2 Význam základních logických operací

Pokud budete používat tyto operátory ve svých programech, nezapomeňte na

jejich priority (přednosti při vyhodnocování). Obdobně jako má v aritmetických výrazech unární minus (např. ve výrazu -1) přednost před ostatními operátory a jako má násobení přednost před sčítáním, má v logických výrazech negace přednost před všemi ostatními operátory a konjunkce má přednost před alternativou. Priorita disjunkce je v Pascalu mezi konjunkcí a alternativou (tj. jako v tabulce), v C++ mezi negací a konjunkcí.

Pokud se vám zdá, že si pravidla o prioritách nezapamatujete, nevadí. Pořadí operací lze v logických výrazech vždy vyznačit pomocí kulatých závorek a na priority pak můžeme klidně zapomenout.

10.2 Cyklus se vstupní podmínkou

Po předchozích nezbytných odbočkách se konečně dostáváme k vlastnímu tématu kapitoly, k cyklům. Hovoříme-li o cyklu se vstupní podmínkou, máme tím na mysli cyklus, který obsahuje pouze vstupní podmínku. Podle způsobu zápisu v obou našich programovacích jazycích mu také často říkáme **cyklus while**. Jeho syntaktická definice v Pascalu je jednoduchá:

```
Cyklus_while:                          {Cyklus se vstupní podmínkou}
        while Podmínka do Příkaz
```

V C++ není o nic složitější:

```
Cyklus_while:                          //Cyklus se vstupní podmínkou
        while (Podmínka ) Příkaz
```

Příkaz zde představuje tělo cyklu. Jak vidíte, ze syntaktické definice vyplývá, že tělo cyklu **while** se v obou jazycích skládá z jediného příkazu. Potřebujeme-li jich v těle více, použijeme složený příkaz. Připomeňme si, že vzhledem k umístění podmínky před tělem cyklu může nastat situace, kdy se **tělo cyklu neprovede ani jednou**.

Ukažme si použití tohoto cyklu přímo v programu. Naprogramujme příkaz (proceduru) *Vyber*, který způsobí, že Karel vysbírá všechny značky na políčku, na němž právě stojí.

Protože pod Karlem nemusí být žádná značka a protože zvednutí značky na místě, kde žádná značka není, vede k chybě, musíme před každým zvednutím značky testovat, zda pod Karlem vůbec nějaká značka je.

V Pascalu by tedy mohla mít definice procedury *Vyber* tvar

```
procedure (*****) Vyber; (*****)
begin
  while Znacka do
       Zvedni;
end;
(********** Vyber **********)
```

a v C++

```
void /*****/ Vyber /*****/ ()
{
  while( Znacka() )         // I zápis funkce v podmínce
  Zvedni();                 // musí být následován závorkami
}
/********** Vyber **********/
```

Než budete číst dále, nejprve si definici příkazu *Vyber* naprogramujte a pak si ji odkrokujte – např. v následujícím programu (uvedeme pouze pascalskou verzi):

```
(********** Hlavní program **********)
begin
  Krat( 5, Poloz );
  Vyber;
end.
```

Tak co? „Cyklil“ vám cyklus správně? Tak přejděme hned k dalšímu příkladu. Naprogramujme příkaz *KeZdi*, po němž Karel dojde k nejbližší zdi ve směru, do něhož je právě natočen.

Na první pohled vypadá úloha stejně jako ta minulá, ale opravdu jen na první pohled. Jak si můžete přečíst v kapitole 3 (*Karel*), podmínka *Zeď* nám vrací *ANO* v případě, když je před Karlem zeď. My bychom však potřebovali vykonat tělo cyklu, tj. příkaz *Krok*, naopak tehdy, když před Karlem žádná zeď není. Řešení je jednoduché a jistě vás již napadlo: potřebujeme testovanou podmínku negovat. Toho, jak víme, dosáhneme v Pascalu tím, že před testovanou podmínku napíšeme **NOT**, a v C++ tím, že před ni napíšeme vykřičník „**!**“.

V Pascalu by tedy mohla mít definice příkazu *KeZdi* tvar

```
procedure (*****) KeZdi; (*****)
begin
  while not Zed do Krok
end;
(********** KeZdi **********)
```

a v C++

```
void /*****/ KeZdi/*****/ ()
{
```

```
  while( !Zed() ) Krok();
}
/********** KeZdi **********/
```

Programátory používající jazyk C++ bychom chtěli upozornit, aby nezapomínali na dvojici kulatých závorek za identifikátorem volané funkce. V opačném případě se místo vyhodnocené podmínky dosadí adresa uvedené funkce (časem si to vysvětlíme podrobněji), tu systém vyhodnotí jako vždy pravdivou podmínku a podle toho se zachová.

V současné chvíli by pro vás bylo jistě hračkou naprogramovat si sami příkaz *NaSever*, který otočí Karla na sever, a příkaz *Domů*, který dovede Karla odkudkoliv do základní pozice, tj. do pozice `0/0-V`. Ověřte si na nich, že jste vše správně pochopili. Pokud si věříte, naprogramujte si samostatně i následující příklad, pokud ne, sledujte výklad a pokuste se pak daný příklad naprogramovat bez průběžného nahlížení do textu kursu.

Příklad

Naprogramujte příkaz (proceduru) *ZaplňNad*, který způsobí, že Karel položí po jedné značce na všechna políčka, která jsou nad řádkem, na němž právě stojí.

Tuto úlohu budeme řešit klasickým postupem shora dolů. Proceduru bychom mohli napsat např. takto (Pascal):

```
procedure (*****) ZaplnNad; (*****)
begin
  NaSever;
  while( not Zed )do
  begin
       Krok;
       ZaplnRadek;
       NaSever;
  end;
end;
(********** ZaplnNad **********)
```

V tomto programu předpokládáme, že příkaz *ZaplňŘádek* si převezme Karla kdekoliv na řádku, který má zaplnit, a předá jej na témže řádku. Příkaz si hned naprogramujeme a odladíme. Místo příkazu *ZaplňŘádek* si prozatím definujeme jeho atrapu. Podíváme-li se však podrobněji na jeho popis v předchozím odstavci, vidíme, že atrapa nemusí dělat vůbec nic, a stínová definice procedury může proto zůstat prázdná.

Po prověření funkce příkazu *ZaplňNad* se pustíme do definice

příkazu *ZaplňŘádek*. Jedna z jeho možných podob je (tentokrát pro změnu v C++):

```
void /*****/ ZaplnRadek /*****/ ()
{
  NaZapad();
  KeZdi();
  CelemVzad();
  while( !Zed() ) PoKrok();
  Poloz();
}
/********** ZaplnRadek **********/
```

Program je samozřejmě možné definovat i tak, že se Karel nevrací pokaždé na počátek zaplňovaného řádku. Zkuste si tuto variantu sami při definici příkazu *VyberHorní*, po němž Karel vybere všechny značky na řádku, na němž stojí, a na všech řádcích nad ním. Možné řešení najdete na doprovodné disketě v souboru *HORNI2.xxx*.

10.3 Cyklus s výstupní podmínkou

Druhou základní variantou cyklu je cyklus s výstupní podmínkou (rozuměj cyklus, který obsahuje pouze výstupní podmínku), který bývá v Pascalu často nazýván **cyklus repeat – until** nebo jen **cyklus until**. Jeho syntaktická definice je

Cyklus_s_výstupní_podnímkou: { Cyklus until }
repeat *Příkaz* [**;** *Příkaz*]$_{opak}$ **until** *Podmínka*

Tento **cyklus skončí, je-li** *Podmínka* **splněna**. Tělo cyklu tvoří příkazy mezi klíčovými slovy **repeat** a **until**.

V C++ se cyklus s výstupní podmínkou obvykle označuje jako cyklus **do – while** nebo jenom cyklus **do**.

Cyklus_s_výstupní_podnímkou: // Cyklus do
do *Příkaz* **while** (*Podmínka*) ;

Příkaz za klíčovým slovem **do** tvoří tělo cyklu; v případě potřeby zde použijeme složený příkaz. Formálně se tedy podobá pascalskému cyklu **until**, avšak pozor: Na rozdíl od Pascalu **tento cyklus skončí, není-li** *Podmínka* **splněna**. Tato rozdílná interpretace výstupní podmínky vede často k chybám

– zejména u programátorů, kteří mezi oběma jazyky přecházejí[20]. Proto jsme do modulu KAREL přidali i možnost ukončovat v C++ cyklus s výstupní podmínkou stejně jako v Pascalu. Jeho syntaktická definice je

Cyklus_until_v_C++:
do *Příkaz* **until** (*Podmínka*) ;

Tělo tohoto cyklu se (podobně jako tělo cyklu **do – while**) skládá z jediného příkazu; můžeme ale použít i příkaz složený. Tento cyklus skončí, je-li *podmínka* splněna. **Tento cyklus však není standardní součástí C++, a proto nedoporučujeme používat ho, pokud nepřecházíte mezi C++ a Pascalem.**

Z umístění podmínky je zřejmé, že tělo cyklu s výstupní podmínkou se provede vždy alespoň jednou. Tato vlastnost také určuje případy, kdy tento druh cyklu v programech používáme: když se o pokračování či ukončení cyklu můžeme rozhodnout až po provedení jeho těla.

Poměrně typickou úlohou vedoucí k použití cyklu s výstupní podmínkou je např. naprogramování procedury *SeberVpředu*, jež způsobí, že Karel sebere značku, která je někde před ním, a vrátí se zpět do své výchozí pozice.

Aby Karel svoji výchozí pozici po sebrání značky našel, měl by si ji před opuštěním označit. Aby si nespletl hledanou značku se značkou, kterou si označil svoji současnou pozici, musí před testem hledané značky výchozí pole opustit. Definice v Pascalu by tedy mohla vypadat takto:

```
procedure (*****) SeberVpredu; (*****)
begin
  Poloz;                    { Označ výchozí pozici }
  repeat                    { Najdi značku }
      Krok;
  until Znacka;
  Zvedni;                   { Zvedni ji }
  CelemVzad;                { Najdi výchozí pozici }
  repeat
      Krok;
  until Znacka;
  Zvedni;                   { Odstraň přidanou značku }
```

[20] Rozdílné zacházení s výstupní podmínkou v C++ naznačují použitá klíčová slova. Pascalské **until** znamená „do té doby, než“. Příkaz
repeat *Cosi* **until** *Podmínka*;
tedy říká „opakuj *Cosi* do té doby, než bude splněna *Podmínka*. Na druhé straně klíčové slovo **while**, použité v C++, znamená „pokud“. Příkaz
do *Cosi()*; **while**(*Podmínka*);
tedy říká „dělej *Cosi*, dokud je splněna *Podmínka*.

```
  CelemVzad;                { Zaujmi původní pozici }
end;
(********** SeberVpredu **********)
```

Jistě jste si všimli, že se nám část programu opakuje dvakrát. Takovou část je rozumné naprogramovat jako samostatnou proceduru, takže náš příkaz *SeberVpredu* přepíšeme do tvaru (tentokrát použijeme C++):

```
void /*****/ ZVpO() /*****/
//Zvedni vpředu a otoč se
{
  do
  {
      Krok();
  } while( ! Znacka() );
  Zvedni();
  CelemVzad();
}
/********** ZVpO **********/

void /*****/ SeberVpredu /*****/ ()
{
  Poloz();
  ZVpO();
  ZVpO();
}
/********** SeberVpredu **********/
```

10.4 Cyklus s oběma podmínkami

Náš výklad na počátku této kapitoly nevylučoval možnost cyklu s oběma podmínkami, tj. s plnohodnotnou (nedegenerovanou) vstupní i výstupní podmínkou. Pascal ani C++ však tuto algoritmickou konstrukci nepodporují, a proto se jí nebudeme podrobněji zabývat[21].

10.5 Nekonečný cyklus

Opakem cyklu s oběma plnohodnotnými podmínkami je cyklus, který má obě podmínky degenerované (mohli bychom tedy říci, že nemá ani vstupní ani výstupní podmínku), a který proto nelze opustit. Takový cyklus se nazývá **nekonečný** a je v praxi velice používaný.

Pascal ani C++ nekonečný cyklus ve svém repertoáru nemají, ale lze jej v nich snadno naprogramovat. Stačí, když do vstupní či výstupní podmínky

[21] Cyklus s oběma plnohodnotnými podmínkami podporují např. jazyky Edison a Forth.

vložíme konstantu: V C++ v cyklu **while** i **do** hodnotu *ANO*, v Pascalu do vstupní podmínky cyklu **while** hodnotu *ANO*, do výstupní podmínky cyklu **until** hodnotu *NE*.

Aby se život maličko zjednodušil, nabízí modul *KAREL* programátorům používajícím C++ konstrukci **loop**, která představuje nekonečný cyklus a jejíž syntaktická definice je

Cyklus_loop: // není standardní součástí C++
loop *Příkaz*

Zdůrazňujeme, že jde o nestandardní rozšíření, definované v modulu *KAREL* pouze v C++, neboť Pascal takováto rozšíření neumožňuje.

Příkladem použití nekonečného cyklu definovaného s využitím této konstrukce může být následující procedura *Okolo*, která způsobí, že Karel bude obcházet kolem stěn dvorku (podrobný popis příkazu *Rychlost* najdete v příloze).

```
void /*****/ Okolo /*****/ ()
//Obchází okolo stěn dvorku
{
  // Zpomalit, aby byl vůbec vidět
  // Rychlost vyberte od 0 do 16
  // U 0 čeká každá akce na stisk klávesy
  Rychlost( 10 );
  loop                // Nekonečný cyklus
  {
       while( !Zed() ) Krok();
       VlevoVbok();
  }
}
/********** Okolo **********/
```

11. Podmíněné příkazy (selekce)

V této kapitole si povíme, jak naprogramovat rozhodování. K tomuto účelu byl v programovacích jazycích zaveden podmíněný příkaz, který je podle řídicího klíčového slova nazýván často **příkaz if** a jehož syntaktická definice v Pascalu je

Příkaz_if: { Podmíněný příkaz }
if *Podmínka* **then** *Příkaz* [**else** *Příkaz*]

Definice v C++ má tvar

Příkaz_if: // Podmíněný příkaz
if (*Podmínka*) *Příkaz* [**else** *Příkaz*]

Prvnímu příkazu v definici, tj. příkazu, který v pascalské definici následuje za klíčovým slovem **then** a v „plusové“ definici za závorkou uzavírající testovanou podmínku, budeme říkat **podmiňovaný příkaz** nebo prostě **první příkaz**. Druhému příkazu v definici, tj. příkazu následujícímu za klíčovým slovem **else**, budeme říkat **alternativní** nebo **druhý příkaz**.

V následujících podkapitolách si ukážeme základní způsoby použití podmíněného příkazu v programech. Nejprve si jej předvedeme při programování jednoduchého rozhodnutí o tom, zda nějakou činnost provedeme či nikoliv, poté si vysvětlíme možnost volby ze dvou různých alternativ dalšího postupu a nakonec si povíme o možnosti volby z více než dvou alternativ.

11.1 Jednoduchý podmíněný příkaz

Jednoduchý podmíněný příkaz neobsahuje část začínající klíčovým slovem **else** (ze syntaktického popisu je vidět, že ji lze vynechat). Slouží nám k naprogramování činnosti, kterou program vykoná, je-li splněna určitá podmínka, a jinak neudělá nic.

Narazí-li počítač při plnění programu na jednoduchý podmíněný příkaz, vyhodnotí nejprve podmínku za klíčovým slovem **if**. Je-li splněna, provede podmiňovaný příkaz. Není-li splněna, neprovede nic a vykonávání programu pokračuje dalším příkazem.

Ukažme si vše na příkladu. Víme např., že je-li před Karlem zeď, vede příkaz *Krok* k chybě; stejně vede k chybě příkaz *Zvedni* v situaci, kdy pod Karlem

není žádná značka.

Naprogramujme proto procedury *OpatrnýKrok* a *OpatrněZvedni*, po nichž Karel provede požadovaný příkaz pouze v případě, že nepovede k chybě. V opačném případě bude příkaz ignorovat.

Definici procedury *OpatrnýKrok* si ukážeme v Pascalu, definici procedury *OpatrněZvedni* v C++. Mohou vypadat např. takto:

```
procedure (*****) OpatrnyKrok (*****);
begin
  if not Zed then Krok;
end;
(********** OpatrnyKrok **********)
```

```
void /*****/ OpatrneZvedni /*****/ ()
{
  if( Znacka() ) Zvedni();
}
/********** Zvedni **********/
```

Abychom si vyzkoušeli aplikaci jednoduchého podmíněného příkazu na nějakém trochu složitějším příkladě, definujeme proceduru *ŘízenýKrok*, po němž se Karel nejprve tolikrát otočí vlevo (přesněji udělá tolikrát *VlevoVbok*), kolik pod ním bude značek, a pak udělá krok ve směru, do něhož se právě natočil. Po Karlově odchodu by na políčku měl zůstat stejný počet značek, jaký tam byl před jeho příchodem.

Vzhledem k poslední podmínce nemůžeme úlohu řešit cyklem. Než se ji naučíme řešit pro libovolný počet značek, budeme předpokládat, že na daném políčku nebudou více než tři značky. Možné definice v obou jazycích jsou následující (všimněte si na nich použití složených příkazů):

```
procedure (*****) RizenyKrok (*****);
begin
  if Znacka then                    { Je aspoň 1? }
  begin
      VlevoVbok;                    { --1. otočka - vlevo vbok }
      Zvedni;                       { --Zvedáme 1. Značku }
      if Znacka then                { --Jsou aspoň 2? }
      begin
          VlevoVbok;                { ----2. otočka - čelem vzad }
          Zvedni;                   { ----Zvedáme 2. Značku }
          if Znacka then            { ----Jsou 3? }
               VlevoVbok;           { ------3. otočka - vpravo vbok }
                                    { Více značek nečekáme, proto už
                                      nic nezvedáme }
          Poloz;                    { ----Vracíme 2. Značku }
      end;
      Poloz;                        { --Vracíme 1. Značku }
  end;
```

```
end;
(********** RizenyKrok **********)
```

```
void /*****/ RizenyKrok /*****/ ()
{
  if( Znacka() )                        // Je aspoň 1?
  {
       VlevoVbok();                     // --1. otočka - vlevo vbok
       Zvedni();                        // --Zvedáme 1. značku
       if( Znacka() )                   // --Byly aspoň 2?
       {
            VlevoVbok();                // ----2. otočka - čelem vzad
            Zvedni();                   // ----Zvedáme 2. značku
            if( Znacka() )              // ----Byly 3?
                 VlevoVbok();           // ------3. otočka - vpravo vbok
                                        // Více značek nečekáme, proto už
                                        // nic nezvedáme
            Poloz();                    // ----Vracíme 2. značku
       }
       Poloz();                         // --Vracíme 1. značku
  }
}
/********** RizenyKrok **********/
```

Zkuste si nyní sami naprogramovat následující příkazy:

OpatrnýDvojkrok	Karel se pokusí udělat dvojkrok, avšak dává cestou pozor, aby nenarazil do zdi.
ZádyKeZdi	Karel se podívá, zda je na některém sousedním políčku zeď, a pokud ji najde, otočí se k ní zády. Pokud na žádném ze sousedních polí zeď není, zůstane ve své původní pozici.
PouzeDvojkrok	Liší se od příkazu *OpatrnýDvojkrok* tím, že v případě, kdy Karel nemůže udělat celý dvojkrok, neudělá nic (přesněji vrátí se do původní pozice).

Vzorová řešení všech těchto příkladů najdete na doprovodné disketě v souboru IF.xxx.

11.2 Úplný podmíněný příkaz

V běžném programování nevystačíme s možností nějakou činnost udělat nebo neudělat. Velice často se musíme na základě nějakých podmínek rozhodnout pro jednu z několika variant.

Úplný podmíněný příkaz, někdy nazývaný jednoduchá alternativa, je úplnou verzí podmíněného příkazu (tj. i s částí začínající klíčovým slovem **else**). Narazí-li na ni počítač při plnění programu, vyhodnotí opět nejprve podmínku za klíčovým slovem **if**. Je-li splněna, vykoná se první (podmiňovaný) příkaz, není-li splněna, vykoná se druhý (alternativní) příkaz.

Vezměme si například proceduru *Změň*, jež má naučit Karla změnit počet značek na políčku, na němž stojí. Tento příkaz nemůžeme obecně vyřešit tak, že budeme značky pouze přidávat, ani tak, že je budeme pouze ubírat. V obou případech by při mezním počtu značek nastala chyba.

Jednou z možností, jak tento úkol vyřešit, je zvednout značku v případě, že na políčku vůbec nějaká bude, a v opačném případě ji tam položit. Možná definice (opět v obou jazycích) je:

```
procedure (*****) Zmen (*****);
begin
  if( Znacka )then
       Zvedni                    { Byla tam }
  else
       Polož                     { Nebyla tam }
end;
(********** Zmen **********)
```

Pascalisté, POZOR! Jednou z nejčastějších chyb je, že zapomenete, že před else se v Pascalu nepíše středník!

```
void /*****/ Zmen /*****/()
{
  if( Znacka() )
       Zvedni();                 // Byla tam
  else
       Poloz();                  // Nebyla tam
}
/********** Zmen **********/
```

Protože používání složených příkazů již znáte, bude pro vás jistě hračkou definovat příkaz *Nahoru*, který předpokládá, že Karel je otočen na východ nebo na západ a na tento příkaz vystoupí ve stejném sloupci o řádek výše a otočí se na druhou stranu (vzpomeňte si na proceduru *Dům*, kde jsme něco podobného potřebovali). I jeho řešení najdete na doprovodné disketě v souboru *IF.xxx*.

Na závěr této podkapitoly vás ještě musíme upozornit na jedno nebezpečí, které na vás číhá při vnořování podmíněných příkazů:

Pamatujte, že překladač (a to jak Pascal, tak C++) spojí

sekci **else** s nejbližším předcházejícím nespárovaným **if**.

Jako příklad naprogramujeme proceduru *ZUZ* (Značka U Zdi), která způsobí, že Karel ve volném prostoru udělá krok, kdežto u zdi místo toho položí značku, avšak pouze za podmínky, že tam dosud žádná značka neleží. Na první pohled by se mohlo zdát, že pascalská definice by mohla vypadat následovně:

```
procedure (*****) ZUZ (*****);   { Chyba v příkazu if }
begin
  if( Zed )then
       if( not Znacka )then
            Poloz
  else
       Krok;
end;
(********** ZUZ **********)
```

To je však velký omyl, protože překladač nehledí na úpravu a tuto definici interpretuje jako program

```
procedure (*****) ZUZ (*****);   { Znamená totéž co předchozí ukázka }
begin
  if Zed then
  begin
       if( not Znacka )then
            Poloz
       else Krok;
  end;
end;
(********** ZUZ **********)
```

To je však naprosto špatně, protože ve chvíli, kdy je u zdi značka, udělá Karel krok do zdi a procedura skončí chybou. Aby překladač program interpretoval podle našeho přání, musíme definici upravit tak, že vnitřní podmíněný příkaz uzavřeme mezi příkazové závorky, čímž větev **else** naprosto jednoznačně přiřadíme vnějšímu **if**.

```
procedure (*****) ZUZ_2 (*****); { Tak je to správně }
begin
  if Zed then
  begin
       if not Znacka then
            Poloz
  end
  else
       Krok;
end;
(********** ZUZ_2 **********)
```

Pro jistotu ukážeme správné řešení i pro C++:

```
void /*****/ ZUZ_2 /*****/       // Tak je to správně
{
  if( Zed() )
  {
       if( !Znacka() )
           Poloz()
  }
  else
       Krok();
}
/********** ZUZ_2 **********/
```

11.3 Vícenásobná alternativa

V programech se často potřebujeme rozhodnout pro jednu z více než dvou možností. Některé programovací jazyky na tento případ pamatují vlastní programovou konstrukcí, většinou se však používá vnořování příkazů pro jednoduchou alternativu.

Lepší než zdlouhavé vysvětlování bude ukázat si vše hned na příkladu. Definujme proceduru *PoSvahu*, která bude simulovat chůzi po svahu vztyčeném k severu. Půjde-li Karel do kopce, tj. na sever, udělá na tento příkaz jeden krok. Půjde-li po vrstevnici, tj. na západ či na východ, udělá kroky dva, a půjde-li Karel ze svahu, tj. na jih, udělá na tento příkaz kroky tři.

Tato procedura může mít v Pascalu tvar

```
procedure (*****) PoSvahu (*****);
begin
  if Sever then
       Krok
  else
       if Zapad then
       begin
           Krok; Krok;
       end
       else
           if Jih then
           begin
                 Krok; Krok; Krok;
           end
           else if Vychod then
           begin
                 Krok; Krok;
           end;
end;
(********** PoSvahu **********)
```

a v C++

```
void /*****/ PoSvahu /*****/ ()
{
  if( Sever() )
       Krok();
  else
       if( Zapad() )
       {
            Krok(); Krok();
       }
       else
            if( Jih() )
            {
                 Krok(); Krok(); Krok();
            }
            else
                 if( Vychod() )
                 {
                      Krok(); Krok();
                 }
}
/********** PoSvahu **********/
```

Taková úprava zápisu však naprogramovaný algoritmus přece jenom trochu zatemňuje. Proto se pro větší názornost zapisují takto vnořené podmíněné příkazy v definici na stejnou úroveň – např.

```
procedure (*****) PoSvahu (*****);
begin
  if Sever then
       Krok
  else if Zapad then
  begin
       Krok; Krok;
  end else if Jih then
  begin
       Krok; Krok; Krok;
  end
  else if Vychod then
  begin
       Krok; Krok;
  end;
end;
(********** PoSvahu **********)
```

Programátoři používající C++ mohou ve svých programech využívat pro zpřehlednění zápisu místo **else if** i pouhé **ef** (i toto rozšíření je definováno v modulu *KAREL*):

```
void (*****) PoSvahu (*****);
{
  if( Sever() )
```

```
        Krok();
   ef( Zapad() )
        Dvojkrok();
   ef( Jih() )
        Krat( 3, Krok );
   ef( Vychod() )
        Dvojkrok();
}
/********** PoSvahu **********/
```

11.4 Složené podmínky

V syntaktických definicích podmínek jsme si mohli přečíst, že podmínky nemusejí být tvořeny pouze konstantami nebo voláními logických funkcí, ale že se může jednat také o složitější logické výrazy.

Abychom si to předvedli, naprogramujeme si proceduru **Cestuj**, která způsobí, že Karel bude cestovat po obrazovce, a na místech, kudy projde, změní počet položených značek. Karel bude chodit rovně a zahne vlevo v případě, že narazí na zeď, nebo že uživatel stiskne mezerník.

Pascalská verze procedury *Cestuj* by mohla mít tvar

```
procedure (*****) Cestuj; (*****)
begin
  while( ANO )do
  begin
       while( not Zed and not Stisknuto ) do
       begin
            Zmen;
            Krok;
       end;
       VlevoVbok;
  end;
end;
(********** Cestuj **********)
```

Její verze v C++ by mohla být

```
void /*****/ Cestuj /*****/ ()
{
  loop                      // Nekonečný cyklus
  {
       while( !Zed() && !Stisknuto() )
       {
            Zmen();
            Krok();
       }
       VlevoVbok();
  }
}
```

```
/********** Cestuj **********/
```

Jiným příkladem použití složené podmínky by mohla být procedura *PoSvahu*, s níž jsme se setkali v předchozí podkapitole. V této proceduře má Karel udělat stejnou akci (dva kroky), pokud šel na východ nebo na západ. Je tedy naprosto zbytečné programovat tyto příklady zvlášť. Jednodušší a přehlednější podoba této procedury je

```
procedure (*****) PoSvahu (*****);
begin
  if Sever then
       Krok
  else if Zapad or Vychod then { Je-li splněna alespoň jedna z podmínek}
  begin
       Krok; Krok;
  end
  else if Jih then
  begin
       Krok; Krok; Krok;
  end
end;
(********** PoSvahu **********)
```

Programátoři používající C++ mohou ve svých programech využívat pro zpřehlednění zápisu místo **else if** i pouhé **ef** (i toto rozšíření je definováno v modulu *KAREL*):

```
void (*****) PoSvahu (*****);
{
  if( Sever() )
       Krok();
  ef( Zapad() || Vychod() ) // Je-li splněna alespoň jedna z podmínek
       Dvojkrok();
  ef( Jih() )
       Krat( 3, Krok );
}
/********** PoSvahu **********/
```

Pokud si budete chtít na chvíli od programování odpočinout, můžete se bavit např. tím, že se budete snažit zaplnit co největší část Karlova dvorku značkami. Bude-li na vás počítač moc rychlý, zkuste jej zpomalit příkazem *Rychlost*.

12. Funkce

V minulých dvou kapitolách jsme se učili funkce používat. V této kapitole se naučíme tvořit funkce vlastní. V první podkapitole se nejprve naučíme vytvářet „čisté“ funkce a ve druhé podkapitole si pak vysvětlíme tvorbu funkcí s vedlejším efektem.

12.1 „Čisté“ funkce

Jak jsme si již řekli v kapitole o cyklech, funkce se od procedur liší tím, že volajícímu programu vracejí nějakou vypočtenou hodnotu. V našich případech to byla prozatím vždy jedna z hodnot *ANO* nebo *NE*. U funkcí, které se omezují na tyto dvě vracené hodnoty, zůstaneme až do konce této knihy.

Podívejme se nyní na úlohy, v nichž je vytvoření vlastních funkcí výhodné. Naprogramujme příkaz *Obcházej*, po němž bude Karel obcházet vyzděný obdélníkový objekt, který má po své levé ruce. Pokud bychom měli úlohu řešit pouze dosud známými prostředky, vypadala by definice v Pascalu přibližně takto:

```
procedure (*****) Obchazej (*****);
begin
  while( ANO )do                        { Nekonečný cyklus }
  begin
       VlevoVbok;                       { Podívej se vlevo }
       if( Zed )then                    { je-li tam zeď }
       begin                            { Je = }
           VpravoVbok;                  { otoč se zpět }
           Krok;                        { a popojdi podle ní }
       end else                         { Není = neotáčej se zpět,}
           Krok;                        { ale popojdi za roh }
  end;
end;
(********** Obchazej **********)
```

Při pozornějším pohledu byste ji jistě dokázali zjednodušit do tvaru:

```
procedure (*****) Obchazej2 (*****);
begin
  while( ANO )do                        { Nekonečný cyklus}
  begin
       VlevoVbok;                       { Pohleď vlevo }
       if( Zed )then                    { Je-li tam zeď, }
           VpravoVbok;                  { otoč se zpět }
       Krok;                            { Popojdi }
end;
```

```
(********** Obchazej2 **********)
```

Výsledný program je sice jednoduchý, ale těžko o něm můžeme prohlásit, že je zcela průzračný: ten, kdo by neznal zadání, by je z programu odvozoval s obtížemi. Kdybychom jej mohli napsat ve tvaru

```
procedure (*****) Obchazej3 (*****);
begin
  while( ANO )do                        { Nekonečný cyklus }
  begin
       while( VlevoZed )                { Jdi podle zdi }
           Krok;                        { až na její konec}
       VlevoVbok;
       Krok;                            { Tam popojdi za roh}
  end;
end;
(********** Obchazej3 **********)
```

byl by mnohem jasnější. Abychom jej takto mohli přepsat, musíme umět definovat podmínku *VlevoZeď* – a to se nyní naučíme.

Poznámka

V průběhu dalšího výkladu se vám bude možná zdát, že řešení s funkcemi je složitější než řešení bez funkcí. To je však způsobeno skutečností, že si základní principy použití funkcí předvádíme na velice jednoduchých příkladech. Na složitějších příkladech by sice byla patrná výhoda použití funkcí, zanikly by však podstatné podrobnosti.

Definice funkce v Pascalu je velice podobná definici procedury, najdeme tu však několik odchylek:

1. Aby překladač již dopředu poznal, že bude překládat funkci, nikoli proceduru, začíná definice funkce klíčovým slovem **function**.
2. Za identifikátorem funkce v hlavičce je nutno uvést i typ vracené hodnoty. V Pascalu se typ logických hodnot jmenuje **boolean**. Při práci se systémem Karel lze kromě toho použít i identifikátor *LogH*.
3. Již víme, že funkce se od procedury liší tím, že vrací volajícímu programu hodnotu. Tuto hodnotu však musíme někdy v průběhu vykonávání funkce definovat.

Syntaktický popis přiřazení funkční hodnoty v Pascalu je

Přiřazení_funkční_hodnoty:
Identifikátor_funkce := *Logický_výraz*

Syntaktický popis definice funkce je tedy následující:

Definice_funkce:
function *Identifikátor_funkce* **:** *Typ_vracené_hodnoty* **;**
+ **begin** *Příkaz* [**;***Příkaz*]$_{opak}$ **end** ;

V syntaktické definici bohužel není možno postihnout povinnost zabezpečit přiřazení funkční hodnoty. Na tuto povinnost však musíte myslet sami, protože **Turbo Pascal programátora nehlídá, zda funkční hodnotu opravdu přiřadí**. Pokud na přiřazení funkční hodnoty zapomeneme, vrátí naše funkce místo vypočtené funkční hodnoty nějaké smetí!

Poučeni se nyní pustíme do definice funkce *VlevoZeď*. Může vypadat například takto:

```
function (*****) VlevoZed (*****) : LogH;
begin VlevoVbok;                  {Otoč se čelem ke zdi}
  if( Zed )then                   {Je tam zeď? }
       VlevoZed := ANO            {Je - vrať ANO }
  else
       VlevoZed := NE;            {Není - vrať NE }
  VpravoVbok;                     {Otoč se zpět }
end;
(********** VlevoZed **********)
```

Využijeme-li toho, že přiřazovanou hodnotou nemusí být pouze konstanta *ANO* nebo *NE*, ale také hodnota vracená jinou funkcí, zjednoduší se nám definice funkce *VlevoZeď* do následující podoby:

```
function (*****) VlevoZed2 (*****) : LogH;
begin
  VlevoVbok;                      { Otoč se čelem ke zdi}
  VlevoZed := Zed;                { Je tam zeď? }
  VpravoVbok;                     { Otoč se zpět }
end;
(********** VlevoZed **********)
```

V C++ je mechanismus práce s funkcemi trochu jiný než v Pascalu. Abychom si jej vysvětlili, museli bychom hovořit o věcech, které se týkají práce s daty v programech; o těch budeme hovořit až v příštím dílu. Proto jsme do modulu Karel připravili několik berliček, které nám umožní se práci s daty ještě chvíli vyhnout. Jak tedy budeme definovat logické funkce v C++?

1. Místo klíčového slova **void**, které překladači oznamuje, že definovaná funkce nevrací žádnou hodnotu[22], a že ji tedy má chápat jako proceduru, zadáme identifikátor *bool*, kterým překladači oznámíme, že funkce bude vracet jednu z hodnot *ANO* nebo *NE*.
2. Jako první příkaz v definici funkce zadáme příkaz *Podminka();* který připraví vše potřebné pro nastavení a předání funkční hodnoty.
3. Vlastní přiřazení funkční hodnoty realizujeme příkazem *Hodnota(Vysledek);* kde za *Vysledek* dosadíme konstantu *ANO* nebo *NE* nebo volání logické funkce (nezapomeňte na prázdné kulaté závorky), jejíž vracená hodnota bude zároveň i výstupní hodnotou námi definované funkce.
4. Posledním příkazem v definici bude *Predej();* který zařídí vlastní předání hodnoty funkce volajícímu programu.

Syntaktický popis definice logické funkce v C++ je tedy následující:

Definice_funkce:

bool *Identifikátor* **( )** { **Podminka();**[*Příkaz*]$_{opak}$ **Predej();**}

V syntaktické definici bohužel není možno postihnout povinnost zabezpečit přiřazení funkční hodnoty. Naštěstí dokáže překladač C++ splnění této povinnosti ohlídat za vás, a pokud zapomenete přiřadit funkční hodnotu, ohlásí chybu.

Poučeni se nyní pustíme do definice funkce *VlevoZeď*. Jedna z jejích možných podob může být

```
bool /*****/ VlevoZed /*****/ ()
{
  Podminka();
  VlevoVbok();                  // Otoč se čelem ke zdi
  if( Zed() )                   // Je tam zeď?
       Hodnota( ANO );          // Je - vrať ANO
  else
       Hodnota( NE );           // Není - vrať NE
  VpravoVbok();                 // Otoč se zpět
  Predej(); }
/********** VlevoZed **********/
```

Využijeme-li skutečnosti, že vracet nemusíme pouze konstantu *ANO* či *NE*, ale také hodnotu vypočtenou jinou funkcí, zjednoduší se nám definice funkce do tvaru

[22] Připomeňme si, že v terminologii jazyka C++ se nerozlišují procedury a funkce, používá se pro ně společné označení *funkce*. Procedura je prostě funkce, která nevrací žádnou hodnotu.

```
bool /*****/ VlevoZed2/*****/ ()
{
  Podminka();
  VlevoVbok();                  // Otoč se ke zdi
  Hodnota( Zed() )              // Je tam zeď?
  VpravoVbok();                 // Otoč se zpět
  Predej();
}
/********** VlevoZed_ **********/
```

Po přečtení předchozího textu možná někteří z vás podlehli omylu, že přiřazení funkční hodnoty je definitivní záležitost, že když jednou funkci hodnotu přiřadíme, nemůžeme ji už změnit. Není tomu tak. Naopak, jedním z oblíbených postupů při programování funkcí je, že někde na počátku definice přiřadíme očekávanou výstupní hodnotu, pak si ověříme oprávněnost tohoto přiřazení a v případě potřeby ji samozřejmě opravíme.

Ukážeme si to na pascalské definici funkce *AspoňDvěZnačky*, která testuje, zda jsou pod Karlem alespoň dvě značky.

```
function (*****) AsponDveZnacky (*****) : boolean;
begin
  AsponDveZnacky := ANO;        { Předpokladáme, že tam jsou }
  if( not Znacka )then AsponDveZnacky := NE { Není tomu tak }
  else
  begin
       Zvedni;                  { Je pod první značkou ještě druhá?}
       if( not Znacka )then
           AsponDveZnacky := NE; { Není }
       Poloz;                   { Vrať zvednutou značku }
  end;
end;
(********** AsponDveZnacky **********)
```

Příklad

Na procvičení si naprogramujte funkci *ZnačkaPřed*, která vrátí *ANO* v případě, že na políčku před Karlem je značka (je-li tam zeď, nemůže tam být značka), a funkci *ZnačkaŠikmoVlevo*, která vrátí *ANO* v případě, že je značka na políčku, které je před Karlem úhlopříčně vlevo (je-li Karel v základním postavení `0/0-V`, jedná se o políčko `1/1`).

12.2 Funkce s vedlejším efektem

Ne vždy je nejlepším řešením funkce, která vrátí vše do stavu, v němž to přebírala. Často bychom naopak potře- potřebovali, aby funkce během

testování dané skutečnosti provedla ještě nějakou další akci. Takové funkce nazýváme **funkce s vedlejším efektem**. Přitom oním vedlejším efektem je vše, co funkce udělá kromě výpočtu a vrácení funkční hodnoty.

V literatuře o programování se používání funkcí s vedlejšími efekty obvykle nedoporučuje, ale sami víte, že suchá je teorie, a zelený strom života. Nicméně pamatujte si, že používání funkcí s vedlejšími efekty s sebou nese jisté riziko, které spočívá v tom, že na vedlejší efekt funkce snadno zapomeneme a potom se nestačíme divit.

Ukažme si vše nejprve na jednoduchém příkladě. Všichni asi znáte ze sobotních příloh novin hlavolamy nazývané královská procházka. Definujme funkci *DalšíTah*, která v okolí políčka, na němž král – Karel stojí, najde políčko, na němž ještě nebyl. Přitom předpokládáme, že všechna doposud navštívená políčka již Karel označil.

Pokud bychom tuto úlohu chtěli řešit bez využití vedlejších efektů funkcí, mohl by program v Pascalu vypadat např. takto:

```
procedure (*****) DalsiTah0 (*****);
{ Bez použití cyklů a funkcí s vedlejším efektem }
begin Poloz;               { Označ toto pole jako navštívené }
  if( ZnackaPred )then
  begin                    { Značka je před Karlem }
       VlevoVbok;          { = otoč se doleva }
       if( ZnackaPred )then
       begin               { Značka je i vlevo od Karla }
            VlevoVbok;     { = otoč se dozadu }
            if( ZnackaPred )then
            begin          { Značka je i za Karlem }
                 VlevoVbok; { = otoč se doprava }
            end;
       end;
  end;
{  Pokud Karel našel neoznačené (= nenavštívené) políčko,
  zůstal stát čelem k němu. }
  if( ZnackaPred )then
  begin                    { Značky jsou ve všech přímých směrech }
       if( ZnackaSikmoVlevo )then
       begin               { Značka je i šikmo vlevo před Karlem }
            VlevoVbok;     { = otoč se vlevo }
            if( ZnackaSikmoVlevo )then
            begin          { Značka je i šikmo vlevo za Karlem }
                 VlevoVbok; { = otoč se dozadu }
                 if( ZnackaSikmoVlevo )then
                      begin{ Značka je i šikmo vpravo za Karlem }
                           VlevoVbok; { = otoč se vpravo }
                      end;
            end;
       end;
{  Pokud Karel našel neoznačené (= nenavštívené) políčko,
```

```
  má je šikmo vlevo před sebou }
       if( not ZnackaSikmoVlevo )then
       begin                { Pole je neoznačené - jdi tam }
            Krok;
            VlevoVbok;
            Krok;
       end;
  end
  else                      { Pole před tebou je neoznačené - jdi tam }
       Krok;
end;
(********** DalsiTah0 **********)
```

Trochu složité, že? Pokud však nebudeme chtít zavádět umělé obraty, které sice zpřehlední zápis programu, ale zesložití jeho vnitřní vazby, asi se nám úloha jednodušeji naprogramovat nepodaří.

Ovšem v profesionální praxi se někdy podobným fintám nevyhneme. Zkuste si před čtením dalšího textu vymyslet, jak byste to „narafičili“, abyste nemuseli explicitně popisovat testování v každém směru, ale abyste mohli program zjednodušit např. použitím příkazu

```
Krat( 4, TestujPred );
```

Vymyšleno? Náš návrh je následující: Procedury *TestujPred* a *TestujSikmoVlevo* definujeme tak, že zvednou značku na výchozí pozici v případě, kdy bude testované políčko dosud nenavštívené, a otočí Karla vlevo v případě, když zjistí, že testované políčko již bylo navštívené. Pascalská verze takto postaveného programu by pak mohla vypadat následovně:

```
procedure (*****) DalsiTah1 (*****);
{ S použitím cyklu, ale bez použití funkcí s vedlejším efektem. }
begin
  Poloz;                    { Připrav signalizační značku }
  Krat( 4, TestujPred );
  if( not Znacka )then
  begin                     { Nenavštívené pole bylo nalezeno }
       Poloz;               { Označ pole jako navštívené }
       Krok;                { a přesuň se na nalezené }
  end                       { dosud nenavštívené pole }
  else                      { Značka nebyla odstraněna = v přímých }
  begin                     { směrech nebylo nalezeno nenavštívené pole }
       Krat( 4, TestujSikmoVlevo );
       if( not Znacka )then
       begin                { Bylo nalezeno nenavštívené pole }
            Poloz;          { Označ pole jako navštívené }
            Krok;           { a přesuň se na nalezené }
            VlevoVbok;      { dosud nenavštívené pole }
            Krok;
       end;
  end;
```

```
end;
(********** DalsiTah1 **********)

procedure (*****) TestujPred (*****);
begin
  if( Znacka )then
{ Nepřítomnost značky by naznačovala, že v některém z minulých testů
  již bylo nenavštívené pole nalezeno, a že je tedy další testování
  zbytečné. }
  begin              { Nenavštívené pole ještě nalezeno nebylo }
      if( ZnackaPred )then VlevoVbok
                     { Pořád nic = otoč Karla do dalšího směru }
      else
          Zvedni;    { Našli jsme je = dej o tom vědět }
  end;
end;
(********** TestujPred **********)

procedure (*****) TestujSikmoVlevo (*****);
begin
  if( Znacka ) then
  { Význam testování značky viz komentář k TestujPred<_>}
  begin              { Nenavštívené pole ještě nalezeno nebylo }
      if( ZnackaSikmoVlevo )then
          VlevoVbok  { Pořád nic = otoč Karla do dalšího směru }
      else
          Zvedni;    { Našli jsme je = dej o tom vědět }
  end;
end;
(********** TestujSikmoVlevo **********)
```

Obě testovací procedury vlastně plnily úlohu funkcí: cosi zjistily a nějakým předem definovaným způsobem zjištěnou informaci předaly. Pokud budeme považovat stav označkování výchozího políčka za výstupní hodnotu, mohli bychom o nich dokonce hovořit jako o pseudofunkcích s vedlejším efektem, protože kromě předání výstupní hodnoty (odstranění či neodstranění značky z políčka) provedly ještě další akci: v případě neúspěchu otočily Karla vlevo.

Nyní přidáme pravé funkce s vedlejším efektem. Místo funkcí *ZnačkaPřed* a *ZnačkaŠikmoVlevo* definujeme funkce *VolnoPřed* a *VolnoŠikmoVlevo*, které Karla v případě, že je testované políčko neoznačené, do výchozí pozice nevrátí. Jejich vedlejším efektem tedy bude, že v případě, že najdou doposud nenavštívené políčko, na něj přesunou Karla, a v případě, že je nenajdou, vrátí Karla na pole, kde si jej převzaly, avšak otočeného oproti výchozí pozici o 90 stupňů vlevo.

Podle tohoto zadání bychom výše uvedené funkce mohli naprogramovat např. následovně (definice uvádíme pro změnu v C++):

```
bool /*****/ VolnoPred /*****/ ()
{
```

```
  Podminka();
  Hodnota( ANO );               // Předpokládám volno
  if( Zed() )
       Hodnota( NE );           // Zeď = není volno
  else
  {                             // Není zeď – zkoušíme dál
       Krok();
       if( Znacka() )
       {                        // Už jsme tu byli – škoda
           Hodnota( NE );       // Není volno
           CelemVzad();         // Návrat zpět
           Krok();
           CelemVzad();
       }
       else                     // Pokud se nenašla značka, již na
           ;                    // tomto políčku zůstanu
  }
  Predej();
}
/********** VolnoPred **********/
```

Poznámka

V předchozí ukázce je na konci programu prázdná větev **else**. *Tato větev je zde uvedena pouze kvůli přehlednosti algoritmu; v programu bychom ji mohli klidně vynechat. Všimněte si v ní samotného středníku označujícího prázdný příkaz, tj. příkaz, který nedělá nic. Kdyby tam tento středník nebyl, překladač by za tělo této větve považoval první příkaz následující za klíčovým slovem* **else**. *V našem příkladě mezi klíčovým slovem* **else** *a koncem složeného příkazu již žádný další příkaz není. Pokud by byla funkce napsána v Pascalu, bylo by i bez středníku vše v pořádku, protože Pascal prohlásí za alternativní příkaz, tj. za tělo větve* **else** *ono nic, které mezi* **else** *a uzavírací příkazovou závorkou (***end***) najde. (Jak víme, v Pascalu středník k příkazu nepatří a slouží pouze k oddělení příkazů.) Překladač C++ by však při absenci tohoto středníku ohlásil chybu, protože podle syntaktické definice musí být za* **else** *příkaz a on tam žádný nenajde. (V C++ je prázdný příkaz tvořen samotným středníkem a nic jednoduššího C++ nezná).*

```
bool /*****/ VolnoSikmoVlevo /*****/ ()
{
  Hodnota( NE );            // Předpokládej obsazeno
                            // Opačný předpoklad než v minulé verzi
  if( !Zed() )              // Zeď = opravdu obsazeno
  {                         // Není zeď – zkoušej dál
       Krok();
       VlevoVbok();
       if( Zed() )
       {                    // Zeď = opravdu obsazeno
           VlevoVbok();     // Vrať   se na původní pole
```

```
            Krok();          // avšak skonči otočen
            VpravoVbok();    // o 90 stupňů vlevo
        }
        else
        {                    // Není zeď - zkouším dál
            Krok();          // Dojdi na testované pole
            if( Znacka() )
            {                // Pole jsme již navštívili
                  CelemVzad();     // Vrať se na původní pole
                  Krok();    // avšak skonči otočen
                  VpravoVbok();    // o 90 stupňů vlevo
                  Krok();
                  CelemVzad();
            }
            else             // Značka zde není =jsme hotovi
                   Hodnota( ANO );      // Na nalezeném poli zůstávám
        }/* Zed2 */
   }/* Zed1 */
   Predej();
}
/********** VolnoSikmoVlevo **********/
```

S využitím těchto dvou funkcí se nám výsledný program zjednoduší:

```
void /*****/ DalsiTah2 /*****/ ()
/* S použitím funkcí s vedlejším efektem, ale bez použití cyklů */
{
   Poloz();
   if( not VolnoPred() )
        if( not VolnoPred() )
            if( not VolnoPred() )
                  if( not VolnoPred() )
                        if( not VolnoSikmoVlevo() )
                              if( not VolnoSikmoVlevo() )
                                    if( not VolnoSikmoVlevo() )
                                          if( VolnoSikmoVlevo() )
   ;
   // Pokud se nám nepodařilo najít volné pole, máme smůlu
}
/********** DalsiTah2 **********/
```

Jak vidíme, veškerou potřebnou práci udělají funkce *VolnoPřed* a *VolnoŠikmoVlevo* v rámci svých vedlejších efektů, takže procedura *DalšíTah2* neobsahuje žádný výkonný příkaz. Je ale také naprosto jasné, že bez doprovodného výkladu je takovýto program naprosto nepřehledný a poznat z něj, co má program za úkol, je více než obtížné.

12.3 Funkce používané jako procedury

Vedlejší efekty funkcí nám připomínají přidruženou výrobu bývalých JZD. I

když jejich hlavní činností měla být zemědělská výroba, příjmy z přidružené výroby přesahovaly u mnohých z nich příjmy ze zemědělské výroby několikanásobně.

Obdobně je to i s funkcemi s vedlejším efektem. Mnohé z nich nejsou často volány kvůli tomu, co testují, ale kvůli svým vedlejším efektům. Oba jazyky proto umožňují volat i funkce jako procedury a jimi vracenou návratovou hodnotu ignorovat[23].

S využitím této možnosti bychom mohli náš problém naprogramovat takto:

```
void /*****/ DalsiTah3 /*****/ ()
/* S použitím funkcí s vedlejším efektem a s použitím cyklů */
{
  Poloz();
  Krat( 4, ZkousejPred() );
  if( Znacka() )
       Krat( 4, ZkousejSikmoVlevo() );
}
/********** DalsiTah3 **********/

void /*****/ ZkousejPred /*****/ ()
{
  if( Znacka )
       VolnoPred();
}
/********** ZkousejPred **********/

void /*****/ ZkousejSikmoVlevo /*****/ ()
{
  if( Znacka() )
       VolnoSikmoVlevo();
}
/********** ZkousejSikmoVlevo **********/
```

Toto řešení bychom mohli použít i v případě, kdy by byly podprogramy *VolnoPřed* a *VolnoŠikmoVlevo* definovány jako procedury. V jiných situacích se však často stává, že danou funkci je někdy vhodnější použít jako funkci a jindy jako proceduru. Pamatujte si proto, že je to nejen užitečné, ale především možné.

[23] POZOR! Turbo Pascal tuto možnost připouští až od verze 6.0, a to pouze v případě, že je v okně [Options | Compiler] nastavena možnost Extended syntax.

12.4 Zpřehlednění akcí

I když nám Karlovo prostředí slouží především jako vyučovací pomůcka, nabízí i prostředky pro zvýšení efektnosti některých řešení. Pokud bychom např. v předchozích příkladech s královskou procházkou opakovaně volali některou z procedur *DalšíTah* v situacích, kdy Karel najde dosud nenavštívené pole až po několika neúspěšných pokusech, zbytečně by se jeho skutečná cesta zatemňovala jeho předchozími neúspěšnými pokusy. Jistě uznáte, že by daleko lépe vypadalo, kdyby se Karlovy neúspěšné pokusy nezobrazovaly a zobrazovala by se pouze výsledná řešení.

Pro tyto účely nabízí modul Karel příkazy *Tajně*, *Zjevně*, *Dvorek*, *BezKarla* a *SKarlem* (jejich charakteristiku si můžete přečíst v příloze). První dva využijete v situacích, kdy se stav v dvorku během doby, kdy jej nebudete zobrazovat, nezmění, třetí pak v situacích, kdy přece jen k nějaké změně dojde. Použití posledních dvou příkazů je výhodné v situacích, kdy jedinou změnou, k níž došlo, je změna pozice Karla.

Zkuste se podívat na procedury a funkce, které jsme definovali v této kapitole, a pokuste se v nich o použití výše uvedených zpřehledňujících procedur.

Příklad

Chcete-li si procvičit probranou látku, pokuste se definovat proceduru **Spirála**, která způsobí, že Karel nakreslí spirálu podobnou jako na obrázku 12.1. Příkaz však napište tak, aby jej Karel dokázal splnit na dvorku jakékoli velikosti, počínaje rozměrem 4x4 pole. Kontrolní řešení najdete v souboru *SPIRALA.xxx*.

Obr. 12.1 Spirála na dvorku 10 × 10

13. Nestandardní pokračování

V této kapitole si povíme o programových konstrukcích, které teoretici programování zatracují, avšak v praxi se s nimi setkáte častěji, než je vhodné. Předem vás však nabádáme k tomu, abyste je používali s rozmyslem a pouze v odůvodněných případech. Jinak zbytečně zvyšujete pravděpodobnost, že se vám v důsledku drobných opomenutí objeví v programu závažné, těžko odhalitelné chyby.

Hlavním důvodem nepříznivého posuzování těchto programových konstrukcí je, že porušují zásady strukturovaného programování. Proto si v první podkapitole nejprve vysvětlíme, jaké vlastně zásady strukturovaného programování jsou a proč je vhodné je pokud možno úzkostlivě dodržovat.

Druhá podkapitola bude věnována předčasnému ukončení podprogramu. Tato konstrukce je z vysvětlovaných obratů nejméně nebezpečná a v praxi je také nejpoužívanější. Na druhou stranu lze její potřebu nejsnáze obejít pomocí prostředků, které se zásadami strukturovaného programování nejsou v rozporu.

V třetí podkapitole se soustředíme na nejpomlouvanější (a také nejnebezpečnější) příkaz – na příkaz skoku (**goto**). Za sebe vám můžeme sdělit, že jsme jej za posledních deset let téměř nepoužili, protože jsme jej prostě nepotřebovali (nepočítáme samozřejmě programy v asembleru, které se bez něj nemohou obejít již svou podstatou). V osmidílné učebnici programování, kterou v polovině osmdesátých let napsal Ing. Pecinovský spolu s MUDr. Kofránkem a Ing. Ryantem, se o něm autoři dokonce málem zapomněli zmínit, protože jej po celou dobu ani jednou nepotřebovali.

Původně jsme se domnívali, že se o něm i v této knize letmo zmíníme až dodatku, aby vás jeho znalost zbytečně nerozptylovala. Pak jsme si však uvědomili, že bez něj nelze v Pascalu 6.0 a starších verzích opsat některé konstrukce z jazyka C++, a proto jsme jej nakonec přece jen zařadili.

Ve čtvrté podkapitole se zaměříme na příkaz **break**. Tento příkaz nám umožní naprogramovat cyklus, který nebude mít podmínku, jež řídí jeho ukončení, ani na začátku, ani na konci, ale uprostřed.

V páté podkapitole si povíme o příkazu, který bychom mohli považovat za doplněk k příkazu **break**. Jedná se o příkaz **continue**, který je sice používán výrazně řidčeji než **break**, nicméně bez něj by nebyly vaše vědomosti úplné.

13.1 Zásady strukturovaného programování

Za otce strukturovaného programování je považován E. W. Dijkstra, který v polovině šedesátých let publikoval článek, v němž prohlásil příkaz **goto** za nebezpečný, a dokonce si dovolil tvrdit, že kvalita programu je nepřímo úměrná počtu použitých příkazů skoku. Tím se ovšem zle dotkl programátorů, pro něž byl nejlepším jazykem na světě starý dobrý FORTRAN, v němž se bez příkazu skoku není možno obejít (vyhnout se příkazu skoku umožňuje až FORTRAN 77).

Na stránkách odborných časopisů se rozpoutal ostrý boj. Na jedné straně stáli tzv. „opravdoví programátoři“[24], kteří na FORTRAN nedali dopustit a kteří prohlašovali, že zásady strukturovaného programování omezují jejich tvořivost. Jejich protivníci, pro něž se u nás vžilo podivuhodné označení „pojídači koláčů“, preferovali jazyk Pascal a jazyky jemu příbuzné a někteří z nich dokonce srovnávali význam strukturovaného programování s významem zavedení podprogramů.

Výsledek této bitvy byl dlouho nerozhodný. Zásadní zlom přinesla zpráva v časopisu *IBM Journal of Research and Development*, která popisovala výsledky důsledného uplatnění zásad strukturovaného programování při řešení dvou velkých projektů. Vykázaná produktivita programátorské práce byla na tehdejší dobu neuvěřitelná. Je samozřejmé, že všechny programátorské firmy chtěly hned této produktivity dosáhnout a začaly nutit své programátory, aby zásady strukturovaného programování dodržovali. Programátoři se sice bránili poukazováním na menší efektivitu výsledných programů, ale s těmito argumenty nemohli dlouho obstát, protože, jak jsme si již řekli, v současné době se většinou doba potřebná pro vývoj programu a náklady na tento vývoj vynaložené považují za daleko důležitější než případný pokles efektivity programu – a výkon aplikace lze daleko levněji zvýšit zakoupením výkonnějšího počítače než programátorskými prostředky. (Ovšem nic nelze přehánět...)

V současné době jsou již zásady strukturovaného programování všeobecně přijaty a o jejich užitečnosti profesionální programátoři nepochybují. Podívejme se nyní na tyto zásady podrobněji:

Vůdčí ideou strukturovaného programování je snaha o maximální přehlednost programu – ta totiž zásadním způsobem zvyšuje produktivitu pro-

[24] Termíny „opravdoví programátoři“ a „pojídači koláčů“ byly převzaty z překladu přednášky „Opravdoví programátoři nepoužívají Pascal“, jejíž text koloval před revolucí po republice a jejíž terminologie se ve vztahu k tomuto tématu často používá.

gramátorské práce. Kromě toho si žádná programátorské firma nemůže dovolit, aby s odchodem autora ztratila možnost dalšího vývoje programu, který uvedla na trh. Z toho tedy vyplývá, že strukturované programování má jedinou zásadu:

Program musí být napsán vždy tak, aby se v něm kdykoliv snadno vyznal nejen autor, ale jakýkoliv dostatečně schopný programátor.

Protože tato zásada je přece jen příliš obecná, byly různými autory vypracovány i různé systémy zásad konkrétnějších, které se snaží definovat jednoduchá a jasná pravidla, při jejichž dodržování bude hlavní zásady strukturovaného programování dosaženo. Většinou se doporučuje dodržovat při programování následující zásady:

1. Program není nic jiného než lineární posloupnost akcí. Každá akce má tedy vždy právě jeden vstup a právě jeden výstup.
2. Pokud je třeba nějakou část programu opakovat, lze pro její naprogramování použít cyklus s podmínkou na počátku (časem byl „vzat na milost“ i cyklus s podmínkou na konci). Tento cyklus pak vystupuje ve výše uvedené posloupnosti jako jedna z akcí – i on má pouze jeden vstup a jeden výstup.
3. Pokud je třeba v nějakém místě programu rozhodnout o směru dalšího postupu, je pro tento účel jedinou doporučovanou konstrukcí podmíněný příkaz typu **if – else**, případně jeho speciální a zobecněné podoby. I on vystupuje v posloupnosti v bodu 1 jako jedna z akcí – i on má pouze jeden vstup a jeden výstup.
4. Délka žádného podprogramu přesáhnout rozumnou velikost. Za tuto velikost bývá nejčastěji považována jedna, nejvýše dvě stránky výpisu. Všimněte si, že se v této zásadě vůbec nehovoří o tom, v jakém jazyce je program psán – platí jak pro programy v asembleru, tak pro programy ve vyšším programovacím jazyce.

V průběhu doby byly publikovány práce, v nichž jejich autoři dokazovali (a dokázali), že jakýkoliv algoritmus lze naprogramovat při dodržení výše uvedených zásad. Nicméně někdy je třeba v zájmu dodržení hlavní zásady strukturovaného programování porušit některou z uvedených zásad pomocných. Přestože jediným cílem těchto pomocných zásad je pomoci zpřehlednit vytvářené programy, v některých speciálních případech se programy

při striktním dodržování těchto pomocných zásad naopak zkomplikují.

Ne vždy je tím ovšem vinen problém. R. Pecinovský kdysi vedl v jednom nejmenovaném podniku kurs strukturovaného programování a jeho frekventanti za ním přišli s žádostí, zda by jim jeden jejich hotový program nepřepsal tak, aby v něm byly zásady strukturovaného programování dodrženy. V první chvíli se zdálo, že se právě jedná o jeden z těch případů, kdy se takovýmto přepisem program zesložití. Rozhodl se proto, že lepší než přepisovat předložený program bude napsat program zcela nový.

Požádal tedy o vysvětlení podstaty problému a vyvíjel s nimi program znovu. Nakonec se ukázalo, že nově vzniklý program, v němž byly všechny výše uvedené pomocné zásady dodrženy, je nejen jednodušší, a tím i přehlednější než program původní, ale že jim svou strukturou poskytl vlastně i zcela nový náhled na podstatu řešeného problému.

Často můžete slyšet laický názor, že hlavní zásadou strukturovaného programování je zákaz používání příkazu **goto** (příkaz skoku). Jak jste si mohli všimnout, ani v hlavní zásadě, ani v zásadách pomocných není o ničem takovém zmínka. Příkaz **goto** se ve strukturovaném programování nepoužívá ne proto, že by to bylo zakázáno, ale proto, že jeho použitím se programy většinou pouze znepřehledňují.

13.2 Předčasné opuštění podprogramu

Předčasné opuštění podprogramu je nejčastějším a většinou také nejméně nebezpečným porušením zásad strukturovaného programování. Realizuje se jednoduše: v okamžiku, kdy si myslíme, že jste již s úkolem hotovi, vložíme do pascalského programu příkaz

exit

Do procedury v C++ vložíme příkaz

return;

a do funkce v C++, napsané s použitím modulu *KAREL*, příkaz

Predej();

nebo

Vrat(*Výsledek* **);**

(Podrobnosti k poslednímu příkazu najdete v dodatku.) V obou jazycích ovšem musíme pamatovat na to, aby funkce již měla v době opouštění přiřazenu funkční hodnotu, jinak vrátíme volajícímu podprogramu místo funkční hodnoty nějaké blíže nedefinované smetí. (Překladač C++ to za vás ohlídá, avšak překladač Pascalu nikoliv.)

Řekli jsme si, že C++ nás ohlídá. Má to však jeden háček. Překladač občas tvrdí, že může nastat situace, kdy funkci nebude přiřazena hodnota, a to i v případě, kdy to evidentně možné není – např. pokud tělo funkce končí příkazem **if – else** a příkaz **Předej** ukončuje sice obě větve, nikoliv však tělo programu. Pomoc je jednoduchá: buď vyjmeme tento příkaz z větve **else** a umístíme jej až za ni, anebo jej na konci programu zopakujeme ještě jednou, i když víme, že na něj nikdy nemůže přijít řada.

Abychom si to procvičili, naprogramujme si funkci *Dvojznačka*, při jejímž plnění Karel zjistí, zda jsou na některém políčku před ním právě dvě značky. Vedlejším efektem této funkce bude, že tyto nalezené dvě značky Karel zvedne a na políčku, na němž našel dvě značky, také zůstane. Pokud cestou ke zdi dvě značky nenajde, zůstane stát u zdi.

```
function (*****) Dvojznacka (*****) : boolean;
begin
  Dvojznacka := NE;              { Předpoklad – není tam }
  if( Zed )then                  { Ani tam být nemůže – není co řešit }
      exit;                      { Můžeš skončit }
  repeat                         { Vyhledávací cyklus }
      Krok;                      { Popojdi na další políčko }
      if( Znacka )then
      begin                      { Aspoň jedna značka tam je }
          Zvedni;
          if( Znacka )then       { Je pod ní další? }
          begin                  { Jsou tam aspoň dvě značky }
                Zvedni;
                if( not Znacka )then       { A ještě další? }
                begin      { Ne, jsou tam právě dvě značky }
                      Dvojznacka := ANO;    { Našli jsme je! }
                      exit;      { Můžeš skončit, protože všech }
                end;             { vedlejších efektů je již dosaženo }
                { Sem se dostaneme, pouze pokud bylo značek moc }
                Poloz;           { Vrať druhou zvednutou značku }
          end;             { Sem se dostaneme, i když jich bylo málo }
          Poloz;                 { Vrať první zvednutou značku }
      end;
  until( Zed );       { U zdi naše snahy končí }
end;(********** Dvojznacka **********)
```

```
bool /*****/ Dvojznacka /*****/ ()
{
```

```
  Podminka();
  Hodnota( NE );                // Předpoklad – není tam
  if( Zed() )                   // Ani tam být nemůže – není co řešit
       Predej();                // Můžeš skončit
  do
  {
       Krok();                  // Popojdi na další políčko
       if( Znacka() )
       {                        // Aspoň jedna značka tam je
            Zvedni();
            if( Znacka() )      // Je pod ní další?
            {                   // Jsou tam aspoň dvě značky
                 Zvedni();
                 if( !Znacka() ) // A ještě další?
                 {               // Ne, jsou tam právě dvě značky
                      Hodnota( ANO ); // Našli jsme je!
                      Predej();  // Můžeš skončit, protože všech
                 }               // vedlejších efektů je již dosaženo
                 // Sem se dostaneme, pouze pokud bylo značek moc
                 Poloz();        // Vrať druhou zvednutou značku
            }
                           // Sem se dostaneme, i když jich bylo málo
            Poloz();            // Vrať první zvednutou značku
       }
  } while( !Zed() );            // U zdi naše snahy končí
  Predej();                     // Nepovedlo se ji najít
}
/********** Dvojznacka **********/
```

13.3 Příkaz skoku goto

Příkaz skoku je obecně považován za velice nebezpečný; uvádíme jej zde především proto, abychom umožnili uživatelům Turbo Pascalu 6.0 a starších verzí alespoň náhradním způsobem naprogramovat konstrukce, o nichž budeme hovořit v následujících dvou podkapitolách věnovaných jazyku C++.

Abychom mohli počítač požádat, aby v průběhu vykonávání programu někam skočil a pokračoval odtamtud, musíme umět zařídit, aby trefil tam, kam jej posíláme – např. nějakým způsobem označit příkaz, kterým má plnění programu pokračovat. Taková značka se nazývá **návěští** (*label*) a vypadá v obou jazycích stejně: je to identifikátor následovaný dvojtečkou.

Návěští se smí v programu vyskytovat pouze na počátku příkazu. Proto má syntaktická definice příkazu tvar:

Příkaz:

Příkaz_bez_návěští

[*Návěští***:**] *Příkaz*

Z tohoto popisu vyplývá, že příkaz může mít i několik návěští.

Ve standardním Pascalu je identifikátor návěští celé kladné číslo. Turbo Pascal však povoluje i návěští ve tvaru identifikátoru, tj. posloupnosti alfanumerických znaků začínající písmenem.

Kromě toho nesmíte zapomenout, že v Pascalu se návěští musí deklarovat! Mezi hlavičku podprogramu a první **begin** vložíme deklaraci, která začíná klíčovým slovem **label**, za nímž následuje seznam deklarovaných návěští oddělených čárkami a ukončený středníkem. Syntaktický popis této deklarace má tvar:

Deklarace_návěští:
label *Identifikátor* [, *Identifikátor*] $_{\text{opak}}$ **;**

Příkaz skoku zadáme v programu v obou jazycích tak, že napíšeme klíčové slovo **goto** následované identifikátorem návěští. Když počítač dojde v plnění programu až k tomuto příkazu, nepokračuje dalším příkazem, ale příkazem označeným návěštím, na něž má skočit.

Jako ukázku použití tohoto příkazu si znovu naprogramujeme proceduru *Dvojznačka* z minulé kapitoly, avšak tentokrát místo předčasného návratu z funkce použijeme příkaz **goto**:

```
function (*****) Dvojznacka (*****) : boolean;
label Hotovo;                    { Deklarace návěští }
begin
  Dvojznacka := NE;              { Předpoklad - není tam }
  if( Zed )then                  { Ani tam být nemůže - není co řešit }
       goto Hotovo;              { Můžeš skončit }
  repeat                         { Vyhledávací cyklus }
       Krok;                     { Popojdi na další políčko }
       if( Znacka )then
       begin                     { Aspoň jedna značka tam je }
            Zvedni;
            if( Znacka )then     { Je pod ní další? }
            begin                { Jsou tam aspoň dvě značky }
                 Zvedni;
                 if( not Znacka )then  { A ještě další? }
                 begin           { Ne, jsou tam právě dvě značky }
                      Dvojznacka := ANO; { Našli jsme je! }
                      goto Hotovo;     { Můžeš skončit, protože všech }
                 end;            { vedlejších efektů je již dosaženo }
                 { Sem se dostaneme, pouze pokud bylo značek moc }
                 Poloz;      { Vrať druhou zvednutou značku }
            end;             { Sem se dostaneme, i když jich bylo málo }
            Poloz;           { Vrať první zvednutou značku }
       end;
  until( Zed );              { U zdi naše snahy končí }
  Hotovo:
```

```
end;
(********** Dvojznacka **********)
```

```
bool /*****/ Dvojznacka /*****/ ()
{
  Podminka();
  Hodnota( NE );           // Předpoklad – není tam
  if( Zed() )              // Ani tam být nemůže – není co řešit
       goto Hotovo;        // Můžeš skončit
  do {
       Krok();             // Popojdi na další políčko
       if( Znacka() )
       {                   // Aspoň jedna značka tam je
           Zvedni();
           if( Znacka() )  // Je pod ní další?
           {               // Jsou tam aspoň dvě značky
                Zvedni();
                if( !Znacka() )       // A ještě další?
                {                     // Ne, jsou tam právě dvě značky
                     Hodnota( ANO );  // Našli jsme je!
                     goto Hotovo;     // Můžeš skončit, protože všech
                }                     // vedlejších efektů je již dosaženo
                // Sem se dostaneme, pouze pokud bylo značek moc
                Poloz();              // Vrať druhou zvednutou značku
           }    // Sem se dostaneme, i když jich bylo málo
           Poloz();              // Vrať první zvednutou značku
       }
  } while( !Zed() );             // U zdi naše snahy končí
  Hotovo:
       Predej();
}
/********** Dvojznacka **********/
```

Na závěr si dovolíme ještě malou poznámku. I když příkaz **goto** v naprosté většině případů opravdu vede k nepřehlednému programu, existují situace (jsou velice vzácné, ale existují), kdy představuje nejmenší zlo.

13.4 Cyklus s podmínkou uprostřed: break

Ne vždy se nám v programu hodí, aby byla podmínka, jejíž hodnota ovlivňuje další průchod cyklem, umístěna na počátku nebo na konci cyklu. Pokud je např. vyhodnocení podmínky cyklu **while** spojeno s nějakým složitějším rozhodováním a pokud z nějakého důvodu není vhodné, aby byla celá akce spojená s vyhodnocením podmínky definována jako samostatná funkce, musíme při použití přísně strukturovaných prostředků řešit celý problém tak, že potřebnou hodnotu vypočteme nejprve před cyklem a potom ještě jednou na konci těla cyklu.

Toto řešení však porušuje základní filozofickou zásadu, která stojí v pozadí celého strukturovaného programování, a to zásadu maximální přehlednosti programu. Budeme- -li chtít výpočet hodnoty, podle níž se o opakování cyklu rozhoduje, nějak změnit, musíme si pamatovat, že stejnou změnu musíme udělat na dvou místech programu. Tím se však – jak jistě uznáte – prudce zvyšuje riziko chyb.

Pokud tedy nechceme (nebo nemůžeme) rozhodovací výpočet umístit do samostatné funkce, můžeme použít cyklus s podmínkou uprostřed. V obou jazycích to bude nekonečný cyklus, v němž na vhodném místě otestujeme řídicí podmínku a podle výsledku testu rozhodneme, zda v cyklu setrváme, nebo jej opustíme.

K takovémuto opuštění cyklu slouží v C++ příkaz **break**. V Turbo Pascalu 7.0 má stejný význam procedura *break*. Ve starších verzích Pascalu musíme místo toho použít příkaz skoku, kterým skočíme na návěští připojené k prvnímu příkazu za cyklem.

Jako ukázku si naprogramujeme příkaz *Rotace*, který způsobí, že Karel roztočí na všech polích dvorku větrníky. Toho lze dosáhnout např. tak, že jednotlivé počty značek budou zobrazeny znaky „|“, „\“, „-“ a „/“ a postupné přidávání a ubírání značek bude v konečném efektu působit dojmem roztočených větrníčků. Následující program uvedu pouze pro C++. Pascalští programátoři najdou vzorové řešení na doprovodné disketě v souboru *ROTACE.PAS*.

Abyste měli nastavování parametrů co nejjednodušší, je na doprovodné disketě připraven soubor *ROTACE.DVO*, v němž jsou uloženy potřebné parametry dvorku. Velikost dvorku nastavte tak, aby vás při běhu programu nerušilo „rolování“ způsobované postupnou změnou značek. Velikost dvorku, při níž již „rolování“ mizí a změna značek se na první pohled jeví synchronní, je vlastně i mírou rychlosti vašeho počítače a jeho videokarty.

```
void /*****/ Rotace /*****/ ()
{
  loop
  {                                          // Pořád kolem dokola
      loop
      {                                      // Jednotlivé řádky
          loop
          {                                  // Jednotlivé sloupce
               if( Znacka() )                // Větrníky roztáčíme
                    Zvedni();                // odebíráním značek
               else                          // Vybráno =
                    Krat( 3, Poloz );        // znovu zaplň
                    if( Zed() ) break;       // Konec řádku
```

```
                    Krok;
            }     /* cyklus řádku */
            if( !Prejdi() )                  // Přechod na další řádek se
                 break;                      // nezdařil = konec dvorku
       }   /* cyklus dvorku */
       Domu();
  }/* nekonečný cyklus */
}
/********** Rotace **********/
```

Příklad

Abyste si použití cyklu s podmínkou uprostřed (jehož výskyt je bohužel častější, než je programátorským teoretikům milé) procvičili, definujte si příkaz *OznačSloupce*, který způsobí, že Karel položí značku do nultého řádku všech sloupců, které obsahují značku. Pro zjednodušení předpokládejte, že v nultém řádku žádné značky nejsou. Vzorové řešení najdete na doprovodné disketě v souboru *OZNSLOUP.xxx*.

13.5 Příkaz continue

Příkaz **continue** je v jistém smyslu doplňkem k příkazu **break**. Na rozdíl od příkazu **break**, pomocí něhož cyklus opouštíme, příkaz **continue** zařídí, že se přeskočí zbytek těla cyklu a program bude pokračovat od nového testování řídicí podmínky.

Také příkaz **continue** najdeme až v Turbo Pascalu 7.0. Ve starších verzích musíme buď vystačit s prostředky strukturovaného programování, nebo použít příkaz skoku na návěští, kterým je označen příkaz, v němž testujeme řídicí podmínku cyklu. To znamená, že toto návěští klademe u cyklu **while** před **while** a u cyklu **repeat – until** před **until**.

Ukažme si vše opět na příkladě: Definujme příkaz *SmažJednotky*, který způsobí, že Karel vyčistí mezi sebou a zdí, k níž je natočen, všechna políčka, na nichž je právě jedna značka. Na políčka, na nichž není žádná značka, jednu značku položí. Je-li na políčku více značek, nezmění Karel nic. Procedura *SmažJednotky* by mohla v C++ vypadat takto:

```
void /*****/  SmazJednotky  /*****/ ()
{
  do                        // Dokud nedojdeme ke zdi
  {
       if( Znacka() )       // Je      zde značka?
       {
```

```
            Zvedni();        // Je více než jedna?
            if( Znacka() )
                 Poloz();    // ANO => Obnov původní stav
            else             // NE  => Nech ji zvednutou
                 continue;   // a jdi na další pole
       }
       else                  // Není zde žádná značka,
            Poloz();         // tak sem jednu polož
       if(Zed())             // Jsme u zdi?
            break;           // ANO - ukonči cyklus
       else
            Krok();          // NE - jdi na další pole
   } while (ANO); /* cyklus pres jednotlivá pole */
}
/**********  SmazJednotky  **********/
```

Řešení v Turbo Pascalu 7.0 je velice podobné:

```
procedure (*****)  SmazJednotky (*****);
begin
  repeat                          { Dokud nedojdeme ke zdi }
       if( Znacka ) then          { Je zde značka? }
       begin
            Zvedni;               { Je více než jedna? }
            if( Znacka ) then
                 Poloz            { ANO => Obnov původní stav }
            else                  { NE  => Nech ji zvednutou }
                 continue;        { a jdi na další pole }
       end
       else                       { Není zde žádná značka, }
            Poloz;                { tak sem jednu polož }
       if Zed then                { Jsme-li u zdi, konec, }
            break
       else
            Krok;                 { jinak jdi na další pole}
  until NE; (* cyklus pres jednotliva pole *)
end;
(********** SmazJednotky **********)
```

V Turbo Pascalu 6.0 a starších musíme použít příkaz skoku:

```
procedure (*****)  SmazJednotky  (*****);
label Test, Ven;
begin
  repeat                          { Dokud nedojdeme ke zdi }
       if( Znacka ) then          { Je zde značka? }
       begin
            Zvedni;               { Je více než jedna? }
            if( Znacka ) then
                 Poloz            { ANO => Obnov původní stav }
       else                       { NE  => Nech ji zvednutou }
            goto Test;            {       a jdi na další pole }
       end
       else                       { Není zde žádná značka, }
            Poloz;                { tak sem jednu polož }
       if Zed then
```

```
          goto Ven              { Jsme-li u zdi, konec }
      else
          Krok;                 { jinak jdi na další pole }
      Test:
  until NE; (* cyklus pres jednotliva pole *)
  Ven:
end;
(********** SmazJednotky **********)
```

Ještě k příkazu goto

Předchozí příklad ukazuje použití příkazu **goto**, které lze s klidným svědomím považovat za naprosto neškodné. Tutéž proceduru lze ovšem naprogramovat i jinak – a pak se z ní stane programátorský horor. Ukážeme si ho v Pascalu, i když něco podobného lze stejně snadno spáchat i v C++.

```
procedure (*****) SmazJednotky (*****);     { Odstrašující příklad }

label Test, Start;

begin
  goto Start;                   { Test pole, na němž stojíme }
  while( not Zed ) do           { Dokud nedojdeme ke zdi }
  begin
      Krok;                     { Testujeme další pole }
  Start:
      if( Znacka ) then         { Je zde značka? }
      begin
          Zvedni;               { Je více než jedna? }
      if( Znacka ) then
          Poloz                 { ANO => Obnov původní stav }
      else                      { NE  => Nech ji zvednutou }
          goto Test;            {       a jdi na další pole }
      end
      else                      { Není zde žádná značka, }
          Poloz;                { tak sem jednu polož }
  Test:
  end; (* cyklus pres jednotliva pole *)
end;
(********** SmazJednotky **********)
```

Problém není ve skoku, který nahrazuje příkaz **continue**, ale ve skoku dovnitř cyklu na návěští *Start* hned v prvním příkazu. Tato konstrukce ukazuje, že jsme se při návrhu procedury *SmažJednotky* nedokázali rozumně vyrovnat s odlišností prvního průchodu cyklem. Skutečnost, že začínáme vlastně zprostředka, ukazuje, že bude na místě nejspíš cyklus s podmínkou na konci.

Všimněte si, že tato konstrukce výrazně porušuje zásady strukturovaného programování, neboť v ní vstupujeme do těla cyklu jinudy než přes jeho hlavičku. (Cyklus, podobně jako jiné programové struktury, by měly mít

jediný vstupní a jediný výstupní bod. Vstupním bodem cyklu je jeho hlavička, v našem případě klíčové slovo **while**, které zde obcházíme.)

Takovýto „divoký“ skok do cyklu může mít – kromě toho, že snižuje přehlednost programu a zvyšuje riziko chyb při pozdějších úpravách – i další stinné stránky. Hlavička cyklu může totiž obsahovat operace, které je potřeba provést před začátkem cyklu a které zajistí jeho správný chod. Jestliže ji přeskočíme, tyto operace neproběhnou a výsledkem může být nesprávná funkce programu. (To záleží na programovacím jazyku, na druhu cyklu a na použitém překladači. Výše uvedený příklad se sice přeloží správně, ale to je jen šťastná náhoda, na kterou není radno spoléhat.)

Poznamenejme, že skok dovnitř cyklu zakazoval už i programovací jazyk FORTRAN IV, ve kterém se bez použití skoků programovat prostě nedalo, neboť neobsahoval ani složený příkaz ani úplné IF.

14. Rekurze

Princip **rekurze**, nebo jak se někdy také říká **rekurzivního volání**, spočívá v tom, že procedura nebo funkce volá buď přímo, nebo nepřímo sama sebe[25].

Rekurze patří mezi velice silné programátorské konstrukce. Zároveň však také patří mezi konstrukce velice málo používané. To je způsobeno částečně skutečností, že rekurzivní volání bývá většinou o něco méně efektivní než klasické řešení pomocí cyklů (většinou, ale ne vždy – někdy je tomu právě naopak). Hlavním důvodem však je, že řada programátorů princip rekurzivního volání a z něj plynoucí důsledky pro konstrukci programu nikdy pořádně nepochopila, a proto se bojí tuto konstrukci ve svých programech používat.

Nebudeme samozřejmě nikoho nutit, aby rekurzi používal, ale její znalost k výbavě profesionálního programátora bezpochyby patří. Naštěstí je prostředí robota Karla pro vstup do rekurzivního světa téměř ideální. Nepřítomnost datových struktur totiž vede ve svých důsledcích k tomu, že se řada příkladů bez využití rekurze ani řešit nedá.

S překrásným příkladem rekurze se jeden z nás svého času setkal na kursu programování pro děti, který vedla Dr. Pavla Polechová. Ta totiž vysvětlovala rekurzi na pohádce o slepičce a kohoutkovi. Tento nápad se nám tak zalíbil, že si ho vypůjčíme a pokusíme se na tomto příkladu vysvětlit princip rekurze také.

Začneme tím, že se pokusíme zapsat algoritmus, podle něhož slepička v této pohádce jednala, jako program v Pascalu[26]. Protože nejde o opravdový program, klidně necháme v identifikátorech písmena s diakritikou. Jednotlivé příkazy v komentářích očíslujeme, abychom se na ně později mohli odvolávat.

```
procedure (*****) SplňPřání (*****);
begin
  if UmíšPoskytnoutSlužbu then              {1}
      PoskytniSlužbu                        {2}
  else                                      {3}
  begin
```

[25] Nepřímé volání znamená, že podprogram volá jiný podprogram a ten pak volá podprogram původní. Řetězec volání, jímž podprogram rekurzivně zavolá sám sebe, může být samozřejmě delší, může obsahovat více mezičlánků. Nejobvyklejší je ale rekurze přímá, při které procedura nebo funkce volá přímo sama sebe.

[26] Chcete-li si nejprve přečíst verzi pohádky, kterou budeme programovat, otevřete si knížku „Pohádka o slepičce a kohoutkovi“ od Marie Majerové (Praha, Albatros 1985).

```
      VyhledejPoskytovateleSlužby;          {4}
      PožádejOSlužbu;                       {5}
      if PodmiňujeSlužbuSplněnímPřání       {6}
           SplňPřání; { Zde je REKURZE! }   {7}
      PřevezmiVýslužku;                     {8}
      PředejVýslužku;                       {9}
  end
end
(********** SplňPřání **********)
```

Začátek všichni znáte: kohoutek sní příliš velké sousto a začne se dusit. Slepička chce kohoutkovi donést mléko. Sama je poskytnout neumí (řádek 1), a proto tedy vyhledá dodavatele mléka – kravičku (řádek 4) a o mléko ji požádá (řádek 5). Kravička však podmiňuje dodávku mléka zprostředkováním subdodávky travičky (řádek 6). Slepička proto musí pokračovat dalším řádkem svého programu (řádek 7) a splnit nejprve přání kravičce.

Při plnění kraviččina přání začne slepička znovu provádět podprogram *SplňPřání*, který již plní. Tento podprogram se nám tu tedy objevuje ve druhé instanci – zavolal sám sebe a rozebíhá se podruhé, i když jeho první volání ještě neskončilo. Abychom rozlišili řádky jednotlivých instancí, budeme je od této chvíle psát jako zlomek s číslem instance v čitateli a číslem řádku ve jmenovateli.

Vraťme se tedy ke slepičce, která právě začíná provádět druhou instanci programu *SplňPřání*. Sama travičku poskytnout neumí (řádek 2/1). Vyhledá proto dodavatele travičky – louku (řádek 2/4) a požádá ji o travičku (řádek 2/5). Louka však podmiňuje dodávku travičky subdodávkou vodičky, čímž (řádek 2/6) donutí slepičku k vykonání příkazu na řádku 2/7.

Slepička tedy vstupuje do třetí instance programu *SplňPřání*. Sama vodičku poskytnout neumí (3/1), a proto vyhledá dodavatele vodičky – potůček (3/4) a požádá jej o vodičku (3/5). Potůček si stěžuje, že nemůže dodat vodičku, protože mu děti ucpaly pramen, a podmiňuje proto dodávku vodičky jeho uvolněním (3/6).

Slepička tak vstupuje do čtvrté instance podprogramu *SplňPřání*. Při testu podmínky na řádku (4/1) však zjistí, že toto přání dokáže splnit sama. Poskytne tedy požadovanou službu (řádek 4/2) a pramen uvolní. Tím je s plněním čtvrté instance podprogramu hotova a vrací se do třetí instance.

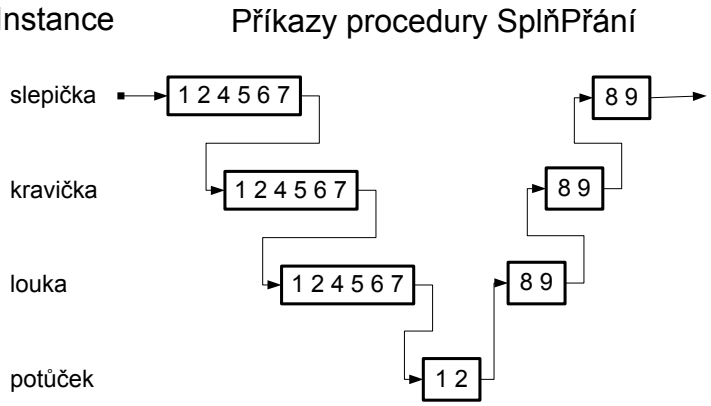
Ve třetí instanci pokračuje v plnění programu na řádku 3/8. Slepička převezme od potůčku vodičku (3/8) a předá ji louce (3/9). Tím je hotova s vykonáváním třetí instance a může se vrátit do druhé.

Ve druhé instanci je ve stejné situaci, v jaké byla před chvílí v instanci

třetí: čeká ji příkaz na řádce 2/8. Převezme tedy od louky travičku (řádek 2/8) a předá ji kravičce (řádek 2/9). Tím je s vykonáváním druhé instance hotova a může pokračovat v plnění instance prvé.

Také zde začne plněním řádku 1/8, tj. převzetím mléka od kravičky, a pokračuje řádkem 1/9 – předáním mléka kohoutkovi. Kohoutek radostně zakokrhá a vše šťastně končí.

Pro větší názornost jsme vám postup plnění jednotlivých příkazů znázornili na obrázku 14.1.



Obr. 14.1 Plnění příkazů v jednotlivých instancích při rekurzivním volání v proceduře SplňPřání

Stejnou metodou (tj. pomocí rekurze), i když s trochu jiným přístupem, bývá většinou programován problém hanojských věží. Pro čtenáře, kteří jej neznají, připomeneme legendu, která se k němu váže.

Kdesi poblíž Hanoje – nebo možná v Tibetu – je klášter a v něm bráhmani, kteří jsou pověřeni velice zvláštním úkolem: mají odměřovat čas trvání tohoto světa. Na vrcholku jedné z klášterních věží je komnata a v ní tři pruty: zlatý, stříbrný a železný. Na zlatém prutu je navlečeno 64 diamantových kroužků různé velikosti postupně od největšího do nejmenšího. Úkolem bráhmanů je postupně přeložit všechny kroužky ze zlatého prutu na stříbrný. Až je všechny přeloží, nastane konec světa.

Aby to neměli tak jednoduché a abychom si i my tohoto světa trochu užili, mají svůj úkol ztížen několika omezujícími podmínkami:

1. V každém okamžiku smějí uchopit a přenést pouze jeden kroužek.
2. Kroužky smějí na pruty navlékat vždy pouze menší na větší.
3. Aby byla úloha vůbec řešitelná, mohou při překládání využívat pro

přechodné odkládání kroužků i železný prut. Musejí však přitom dodržet předchozí dvě zásady.

Ať je tato tato legenda pravdivá nebo ne, konce světa se hned tak obávat nemusíme. Ti zvídavější z vás si totiž mohou matematickou indukcí odvodit, že k přemístění N kroužků je třeba 2^N-1krát přemístit nějaký kroužek. I kdybychom předpokládali, že bráhmani budou pracovat nevídanou rychlostí a přemístí každou desetinu sekundy jeden kroužek, budou se svým úkolem hotovi za 58 miliard let. (Pro srovnání: stáří vesmíru – od velkého třesku do dneška – se odhaduje na nějakých 10 – 15 miliard let.)

Vraťme se ale k řešení našeho problému. Úloha o hanojských věžích je velice známá a mnozí z vás jistě její řešení znají. Přesto se je zde pokusíme odvodit.

Abychom mohli správně přemístit pyramidu N kroužků, musíme na cílový prut nejprve navléci největší, tj. nejspodnější kroužek. Abychom to mohli provést, musíme mít všechny kroužky, které leží na něm, navlečeny na pomocném prutu.

I při navlékání kroužků na pomocný prut musejí být dodržena obě omezení, tj. musíme přemísťovat kroužky jeden po druhém a smíme pokládat pouze menší na větší. Když se nad problémem trochu zamyslíme, brzy přijdeme na to, že se jedná o tutéž úlohu; pouze se zmenšil počet přenášených kroužků o jeden a zaměnili jsme cílový a pomocný prut. (Největší kroužek, navlečený jako první na kterýkoli prut, neznamená žádné omezení, neboť na něj lze položit jakýkoliv jiný kroužek.)

To znamená, že přenesení N kroužků na cílový prut můžeme realizovat tak, že přeneseme N-1 kroužků na pomocný prut, poté přeneseme největší kroužek na cílový prut a nakonec přeneseme N-1 kroužků z pomocného prutu na cílový.

Podúlohu přenesení N-1 kroužků z výchozího sloupce na pomocný provedeme tak, že odložíme N-2 kroužků na původní cílový sloupec, pak přeneseme kroužek N-1 na pomocný sloupec a nakonec přeneseme N-1 odložených kroužků z cílového sloupce na pomocný. A tak dále, ještě hodně dlouho....

Doporučujeme vám vrátit se k tomuto příkladu, až budete umět pracovat s daty, tj. až se seznámíte s následujícími dvěma díly. Za pomoci nám známých prostředků ho sice již lze řešit, ale toto řešení je přece jenom trochu umělé.

O problému hanojských věží jsme hovořili především proto, abyste si na něm všimli metody řešení problému pomocí vyřešení posloupnosti stejných, pouze o trochu jednodušších problémů. (Připomeňme si znovu metodu rozděl

a panuj.)

Pokusme se nyní vyřešit pomocí rekurze několik jednoduchých úloh z Karlova prostředí. První z nich bude definice procedury *ZnačkuKeZdi*, která způsobí, že Karel dojde ke zdi, položí tam značku a pak se vrátí do svého původního postavení. Doplňkovou podmínkou k tomuto problému je požadavek, aby byl po skončení činnosti zachován počet značek na všech polích dvorku, s výjimkou pole u zdi, na něž Karel položil značku.

Řešením by mohl být např. následující program:

```
void /*****/ ZnackuKeZdi /*****/()
{
  if( Zed() )
       Poloz();             // Jsme u zdi - úloha je triviální
  else
  {
       Krok();              // Zkusíme zadání zjednodušit
       ZnackuKeZdi();       // A vyřešit úlohu z této jednodušší pozice
       CelemVzad();         // A nyní honem do výchozí pozice
       Krok()
       CelemVzad()
  }
}
/********** ZnackuKeZdi **********/
```

Řešení bychom mohli ještě zefektnit tím, že bychom jako první zadali v proceduře příkaz *BezKarla* a jako poslední pak příkaz *SKarlem*. Tím bychom Karlův přechod ke zdi a hlavně jeho neustálé otáčení cestou zpět utajili, takže by to navenek vypadalo, že Karel značku ke zdi hodí.

Vyřešme nyní trochu složitější příklad – naprogramujme příkaz *Minér*, který má simulovat následující situaci: Karel je ženista a dvorek je zaminované území, kde miny jsou představovány značkami. Karel právě dostal rozkaz odminovat políčko těsně u zdi a vrátit se na své stanoviště.

Aby Karel bez úhony k tomuto políčku došel, musí cestou všechny miny zneškodnit, tj. musí cestou všechny značky posbírat. Aby ale nepřítel jeho stanoviště nevypátral, musí cestou zpět uvést vše do původního stavu – tj. položit všechny miny zpět tam, odkud je vzal, samozřejmě s výjimkou miny u zdi.

```
void /*****/ Miner /*****/()
{
  if( Zed() )          // U zdi je všechno jasné
       Vyber();        // Karel má za úkol vybrat všechny značky
  else if( Znacka() ) // Nejsme u zdi
  {                    // a je pod námi značka
       Zvedni();       //              Zjednodušíme úlohu (zneškodníme
```

```
  minu)
       Miner();       // a pokusíme se ji vyřešit
       Poloz();       // Vracíme značku tam, kde jsme ji vzali
  else
  {                   // Nejsme u zdi a není pod námi značka
       Krok();        // Zjednodušíme – přiblížíme se ke zdi
       Miner();       // Vyřešíme zjednodušenou úlohu
       CelemVzad();   // Vracíme se do výchozího postavení
       Krok()
       CelemVzad()
  }
}
/********** Miner **********/
```

Poslední úlohou, jejíž řešení si naznačíme, bude naprogramovat proceduru *Vysbírej*, která způsobí, že Karel vysbírá značky tvořící na dvorku souvislou oblast. Při tom o dvou polích prohlásíme, že patří do téže souvislé oblasti, pokud spolu sousedí jednou stranou (roh nestačí).

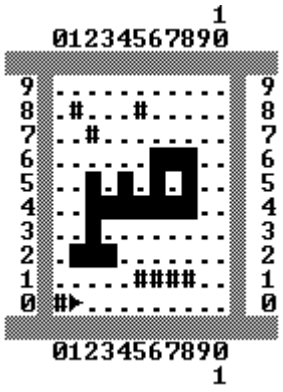
Jak Karlovi vysvětlíme, jak má tuto úlohu řešit? Není to tak složité. Předpokládáme- -li, že na počátku je Karel na některém z polí souvislé vyznačkované oblasti, necháme jej vybrat všechny značky z pole, na němž stojí, a pak jej pověříme, aby zkontroloval všechna čtyři sousední pole. Každé z těchto polí zkontroluje tak, že je bude považovat za výchozí pole souvislé oblasti. Ve srovnání s prvním testovaným polem však můžeme využít toho, že víme, že políčko, z něhož sem Karel přišel, je již vysbírané, a stačí tedy testovat pouze zbylé tři sousedy navštíveného pole.

Možná, že vám toto vysvětlení připadá moc složité. Doufáme, že program, který právě popsaný algoritmus implementuje, vám bude pochopitelný (ukážeme si ho pro změnu v Pascalu):

```
procedure (*****) Vysbirej (*****);
(* Vysbírá značky v souvislé vyznačkované oblasti,
  jejíž součástí je políčko, na němž Karel stojí *)
begin
  if( Znacka ) then
  begin
       Vyber;
       Krat( 4, Prover )
  end
end
(********** Vysbirej **********)

procedure (*****) Prover (*****);
(* Prověří, zda vyznačkovaná souvislá oblast pokračuje políčkem,
  které je před Karlem, a pokud ano, tak tuto větev vysbírá. Po skončené
  prověrce vrátí Karla na výchozí políčko, avšak předá jej otočeného o
  90° vpravo.
*)
begin
```

```
  if( Zed )then                  { Nepokračuje - je tam zed }
      VpravoVbok
  else                           { Mohla by tampokračovat }
  begin
      Krok if( Znacka )the
      begin                      { Pokračuje - jdeme ji vysbírat }
          Vyber;                 { Vyber napřed značky pod sebou }
          VlevoBok;              { a prověř sousední políčka }
          Krat( 3, Prover );     { s výjimkou toho,z kterého }
      end                        { právě přišel }
      else
          CelemVzad              {Proti výchozí pozici je otočen o 180°}
      Krok;                      { Zaujmi cílovou pozici }
      VlevoVbok
  end
end
(********** Prover **********)
```

Na následujícím obrázku vám nabízíme testovací podobu Karlova dvorku. Políčka označená plnými znaky patří do jedné souvislé oblasti. Na polích označených znaky # jsou sice také značky, ale ty již nepatří do souvislé oblasti tvořené znaky „■“. Umístěte Karla např. do pozice `4/4-S` a zkuste program spustit. Necháváme na vás, zda jej budete chtít vskutku krokovat, anebo zda se spokojíte s jeho spuštěním nějakou pomalejší rychlostí a budete pouze pozorovat, jak se s úkolem vypořádává.



Obr. 14.2 Testovací podoba Karlova dvorku

15. Metoda pokusů a oprav – backtracking

Jako chuťovku na závěr jsme si pro vás nechali jednu z rafinovaných metod řešení složitých úloh. Jak jste si mohli přečíst v názvu kapitoly, anglicky se tato metoda nazývá **backtracking**, což se v naší literatuře občas překládá doslovně, tj. jako **zpětné sledování**.

Abychom tuto metodu přístupu k řešení úloh odlišili od některých nestrukturovaných algoritmických obratů, které se shodou okolností nazývají stejně, zvolili jsme proto raději volnější překlad **metoda pokusů a oprav**.

Nyní se nelekněte – bude následovat trochu teorie. Pokud ji nemáte rádi, můžete ji klidně přeskočit. Pokusíme se pak ve výkladu pokračovat tak, abyste i vy tušili, o čem je řeč. Abychom těmto zapřisáhlým praktikům usnadnili orientaci, oddělíme teoretickou pasáž vodorovnými linkami.

Přesto vám doporučujeme, abyste se o její pročtení alespoň pokusili – budeme se snažit vše demonstrovat na praktických příkladech. (Neodpustíme si ale lehké rýpnutí: skutečný profesionál se bez teoretických znalostí neobejde. Kdo chce znovu vymýšlet věci, které již vymysleli jiní, nikdy se opravdovým profesionálem nestane.)

Metoda pokusů a oprav je aplikovatelná na úlohy, jejichž řešením je n-tice $(x_1, ..., x_n)$, kde každé x_i vybíráme z konečné množiny S_i. Typickým zadáním bývá úloha najít vektor, který maximalizuje nějakou kriteriální funkci.

Abychom vám to přiblížili na příkladu ze života, představte si vytěžovací stanici, kam přijíždějí nákladní auta, která nejsou plně naložena. Dispečer má za úkol vybrat ze zásilek určených do cílové stanice takovou podmnožinu, aby auto co nejlépe vytížil.

Pro určitost si představte, že přijede auto, které jede do Košic a může naložit ještě 1000 kg. Zásilky určené do Košic váží 110 kg, 70 kg a 60 kg. Dispečer zjistí, že když naloží 6 balíků po 110 kg, 4 balíky po 70 kg a 1 balík o hmotnosti 60 kg, využije nabízený prostor bezezbytku.

Jedním z možných řešení této úlohy je tedy trojice (6, 4, 1), přičemž každou položku jsme vybírali z konečné množiny {1, 2, ..., *PočetBalíkůDanéVáhy*}.

Kdybychom se rozhodli řešit úlohu hrubou silou, asi bychom vygenerovali všechny možné n-tice a vybrali z nich tu, která vyhovuje zadání úlohy. Sami

však asi cítíte, že to není nejlepší postup.

Podívejme se nyní, jak se takováto úloha řeší metodou pokusů a oprav. Algoritmus je poměrně jednoduchý: Zvolíme první prvek (tak, aby jeho hodnota neodporovala zjevným podmínkám úlohy) a pokusíme se najít zbylé *n*-1 prvky tak, abychom dostali řešení. Pokud se nám takových *n*-1 prvků najít nepodaří, změníme první prvek a hledáme znovu.

Jednoduché, že? Je třeba ovšem najít ještě oněch zbylých *n*-1 prvků. Řešení této dílčí úlohy můžeme hledat opět metodou pokusů a oprav: Zvolíme opět první prvek s vhodnou hodnotou a pokoušíme se najít hodnoty zbylých *n*-2 prvků tak, abychom dostali řešení původní úlohy.

Takto rekurzivně pokračujeme až do doby, kdy nám zbývá řešit úlohu pro jediný prvek. Zjistíme, zda je nebo není možné úlohu vyřešit (zda takový prvek existuje), a ohlásíme volajícímu programu splnění nebo nesplnění úlohy.

Obecný algoritmus řešení úloh metodou pokusů a oprav můžeme zapsat takto (zapisujeme jej v Pascalu):

```
function (*****) MPO (*****) : LogH;
{ Metoda pokusů a oprav }
begin
  Inicializace;                  { Připrav počáteční variantu }
  repeat
       Zaznamenej_nastavenou_variantu;
       if( Je_přípustná )then
       begin
            if( Řešení_je_úplné )then
            begin
                 MPO := ANO;      { Je úplné = úkol je vyřešen }
                 exit;            { Vracím zprávu o úspěchu }
            end
            else if( MPO )then
                 {Není úplné – dalo se rekurzivně dotáhnout? }
            begin
                 MPO := ANO;      { Dalo – vrátíme volajícímu }
                 exit;            { programu zprávu o úspěchu }
            end
       end
       Zruš_záznam; {Nevyšlo to – zkusíme to jinak }
  until( NeníDalšíMožnost );     { Připrav další variantu }
  MPO := NE;                     { Vrátíme zprávu o neúspěchu }
end;
```

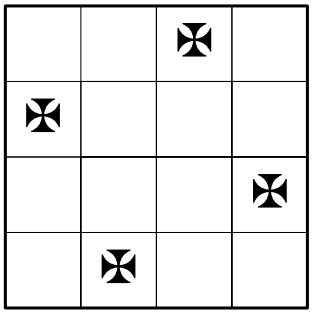
Tolik tedy o metodě obecně. Nyní si ji ukážeme na řešení úlohy, při němž vystačíte s prostředky, které máme k dispozici a které jsme se již naučili používat.

Použití metody pokusů a oprav si ukážeme na úloze osmi dam, kterou

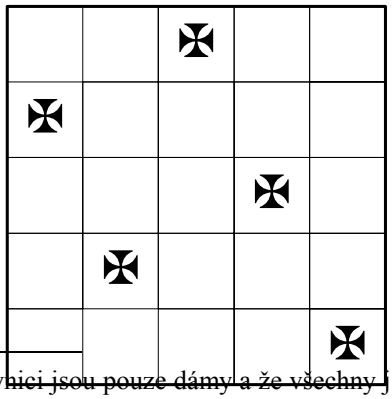
se již v roce 1850 snažil řešit vynikající matematik K. F. Gauss. Její úplné řešení se mu nepodařilo. Nelze se ostatně divit; podstata úlohy a její charakteristické vlastnosti jsou takové, že se analytickému řešení spíše brání. Jak však uvidíte za chvíli, řešení pomocí počítače nijak složité není.

Cílem této úlohy je rozmístit na šachovnici 8 dam tak, aby podle šachových pravidel neohrožovaly jedna druhou. Zkusíme úlohu vyřešit v poněkud obecnější podobě – rozmístit na šachovnici $n \times n$ polí n dam tak, aby se navzájem neohrožovaly[27].

Sami jistě rychle přijdete na to, že pro šachovnici o 2×2 a 3×3 polích není tato úloha řešitelná. Řešení pro šachovnice o velikosti 4×4, 5×5 a 8×8 polí můžete vidět na obrázcích 15.1, 15.2 a 15.3. Existují ovšem i další řešení – např. pro šachovnici 8×8 polí existuje 92 různých postavení, vyhovujících podmínkám naší úlohy.



Obr. 15.1 Jedno ze dvou možných řešení úlohy 4 dam

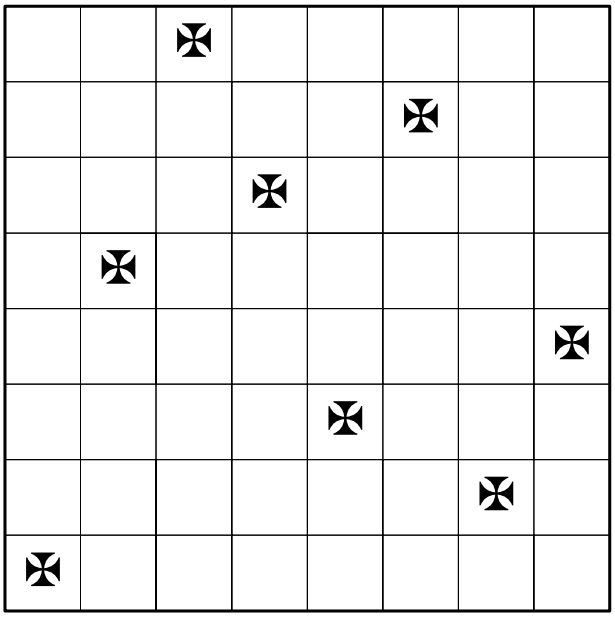


Obr. 15.2 Jedno z řešení úlohy 5 dam

[27] To znamená, že na šachovnici jsou pouze dámy a že všechny jsou navzájem nepřátelské. S obvyklým šachem to tedy nemá mnoho společného, spíše to připomíná Cimrmanův syrový pohádkový horor o 13 tchýních. Proto také nebudeme na následujících obrázcích rozlišovat barvu políček ani dam.

Aplikujeme-li nyní na tyto úlohy postup, o němž jsme hovořili při výkladu metody pokusů a oprav, umístíme první dámu do prvního řádku prvního sloupce, načež se pokusíme do zbývajících *n*-1 sloupců umístit zbývajících *n*-1 dam.

Jak již asi tušíte, zbývajících *n*-1 dam umístíme tak, že do druhého sloupce umístíme druhou dámu na pozici, kde nebude ohrožována první dámou, a pokusíme se rozmístit zbývajících *n*-2 dam. [28]



Obr. 15.3 Jedno z řešení úlohy 8 dam

Obecně zbývajících *n-k* dam umístíme tak, že do prvního neobsazeného sloupce (celkově *k*-tého v pořadí) umístíme první z nich (celkově *k*+1) na pozici, kde nebude ohrožována žádnou z předchozích *k* dam, které jsou umístěny ve sloupcích s čísly od 1 do *k*-1, a pak do zbývajících sloupců umístíme zbylých *n-k*-1 dam. Pokud se nám dámy nepodaří do zbývajících sloupců umístit, musíme pro tuto dámu (číslo *k*+1) najít náhradní postavení. Pokud jsme již všechna náhradní postavení v daném sloupci vyčerpali, ohlásíme nesplnění úlohy a program se musí postarat o změnu postavení dámy předchozí.

[28] Abychom zbytečně neplýtvali místem, nebudeme jednotlivé podprogramy uvádět přímo v textu, ale odkážeme zájemce na soubor *DAMY_P.xxx*, kde si mohou jejich zdrojový tvar vyhledat.

Algoritmus řešení této úlohy odvodíme z obecného algoritmu, avšak vzhledem k omezenosti našich programovacích prostředků v něm uděláme nějaké drobné změny (program budeme zapisovat v C++):

```
bool /*****/ Umisti /*****/ ()
{
  Podminka();
  // Přesuň se do výchozí pozice - počáteční varianta
  NaZapad(); KeZdi(); NaJih(); KeZdi(); VlevoVbok();
  loop
  {
       Poloz();                  // Zaznamenej navrhovanou pozici
       if( Bezpecna() )          // Prověř bezpečnost královny zde
       {                         // Na tomto poli je dáma bezpečná
           if( NaVychod(), Zed() )    // Skončili jsme?
           Vrat( ANO );          // ANO, již jsme u zdi = HOTOVO
           else
           {                     // NE - ještě musíme umístit zbylé dámy
                Krok();          // Přesuň se do dalšího sloupce
                if( Umisti() )   // A zkus rekurzivně umístit zbylé dámy
                     Vrat( ANO ); // Podařilo se - opouštíme funkci
           }
           // Nezdařilo se - musíme provést opravu výchozí pozice
           // Jsme v nejvyšším řádku následujícího sloupce
           NaZapad();            // Zpět do původního sloupce
           Krok();
           VlevoVbok();          // Na jih
           while( !Znacka() )    // Najdi dámu
           Krok();
       }/* Řešení se nepodařilo dotáhnout */
       /*** Nebo nevyhovovalo hned zpočátku */
       // Toto řešení tedy neprošlo - zkusíme to z další pozic
       // Karel stojí na poli s královnou
       Zvedni();                 // Zkusíme dámu přesunout výš
       if( NaSever(), !Zed() )   // Existuje ještě další řešení?
           Krok();               // ANO - přesuň se tam
       else                      // NE - vyčerpali jsme již všechny
           break;                // = opusť cyklus
  }/* cyklus jednotlivých řešení */
  Vrat( NE );                    // Nepovedlo se - dáváme o tom zprávu
}
/********** Umisti **********/
```

Než si začneme vyprávět o vlastním algoritmu, vysvětlíme si obrat, který jsme v předchozím programu dvakrát použili: Jde o provedení několika akcí při testování podmínky programové konstrukce **if**.

Logický výraz[29] může být v C++ tvořen několika výrazy oddělenými navzájem čárkami. Takovýto výraz se vyhodnocuje postupně zleva doprava a jeho hodnotou je hodnota naposledy vyhodnoceného podvýrazu.

[29] Zatím známe jen logické výrazy. Operátor čárka lze ale použít na všechny výrazy v C++.

Této vlastnosti jazyka se hojně využívá hlavně u cyklů. Zde nám totiž **operátor čárka** často pomáhá definovat vstupní či výstupní podmínku i u cyklů, které bychom jinak museli definovat buď s podmínkou uprostřed, nebo s podmínkou vyhodnocovanou na dvou rozdílných místech programu.

Je to užitečný operátor[30], ale doporučujeme nepřehánět jeho používání, protože jinak si program spíše znepřehledníte.

Vraťme se ale k našemu algoritmu. V kapitole o návrhu programu jsme si říkali, že hned poté, co definujeme nějaký podprogram, jej musíme vyzkoušet.

V naší funkci *Umísti* však používáme dosud nedefinovanou funkci **Bezpečná**. Jak si možná pamatujete, řekli jsme si, že místo nedefinovaných funkcí dáváme do programu jejich atrapy. Jak to však zařídit s funkcí, která může vracet dvě různé hodnoty, a my bychom přitom potřebovali vyzkoušet reakce našeho podprogramu na obě dvě?

Řešení, které vám nabídneme, vám možná bude připadat trochu krkolomné. Chceme vám však na něm ukázat, že i takovéto krkolomné řešení nám může v procesu ladění prokázat dobrou službu.

Nepohrdejte krkolomnými řešeními v situacích, kdy nedokážete vymyslet lepší.

Náš návrh spočívá v tom, že při provádění testované funkce Karla trochu zabrzdíme a necháme jej provádět nějakou činnost, která nebude mít vliv na vlastní řešení problému – např. jej necháme udělat sudý počet příkazů *ČelemVzad*. Během provádění těchto hluchých příkladů si rozmyslíme, zda výsledkem funkce má být hodnota *ANO* nebo hodnota *NE*, a podle toho buď stiskneme, nebo nestiskneme nějakou klávesu. Karel po „rozcvičce“ klávesu otestuje a hodnotu funkce nastaví podle toho, byla-li mezitím nějaká klávesa stisknuta či nikoliv.

Protože takovou atrapu můžeme potřebovat častěji, definujte si ji jako samostatnou funkci. Pojmenujeme ji např. *Dotaz* a přidáme ji do souboru *SPOLECNE.xxx*. Tato funkce by mohla vypadat např. následovně:

```
bool /*****/ Dotaz /*****/ ()
{
   Podminka();
   Rychlost( 8 );
```

[30] Pozor, operátor čárka má velmi nízkou prioritu – ve skutečnosti vůbec nejnižší ze všech operátorů v C++. Navíc časem poznáme i situace, kdy znak „čárka“ nebude znamenat operátor, ale oddělovač. Nejdřív se ale musíme seznámit s prací s daty.

```
  Krat( 10, CelemVzad );
  Hodnota( Stisknuto() );
  Rychlost( 16 );
  Predej();
}
 /********** Dotaz **********/
```

Atrapu funkce *Bezpečná* pak naprogramujeme takto:

```
bool /*****/ Bezpecna /*****/()
{
  Podminka();
  Vrat( Dotaz() );
}
/********** Bezpecna **********/
```

Předpokládejme tedy, že jsme již funkci *Umísti* odladili a že se chystáme definovat funkci *Bezpečná*, která má jako svoji funkční hodnotu vrátit informaci o tom, zda je testované políčko pro královnu bezpečné, tj. zda je neohrožuje žádná z doposud umístěných královen.

Královny mohou chodit čtyřmi směry: vodorovně, svisle a v obou uhlopříčných směrech. V žádném z těchto směrů nesmí ležet žádná z dosud umístěných královen.

Tím, že šachovnici zaplňujeme postupně zleva doprava, se nám však situace přece jenom trochu zjednodušuje. Za prvé nemusíme kontrolovat svislý směr, protože víme, že jsme zde ještě žádnou dámu neumístili a že jedinou dámou v daném sloupci bude ta, pro niž testujeme bezpečnost daného políčka. Zjednoduší se i testy vodorovného a obou šikmých směrů, protože nám stačí prozkoumat pouze tu část, která se nachází nalevo (na západ) od testovaného pole. Z toho tedy vyplývá, že pole budeme moci prohlásit za bezpečné, pokud nenajdeme žádnou z dříve umístěných dam v západním, jihozápadním ani severozápadním směru.

Definice funkce *Bezpečná* je jednoduchá – prostě otestujeme všechny tři směry, z nichž může nové dámě hrozit nebezpečí:

```
static bool /*****/  Bezpecna  /*****/ ()
{
  Podminka();
  Hodnota( ZDobry() && JZDobry() && SZDobry() );
  Predej();
}
/********** Bezpecna **********/
```

Vzhledem k jednoduchosti této funkce si dovolíme podlehnout přesvědčení, že jsme ji naprogramovali správně, a přejdeme k definicím testů v jednotlivých směrech. Ukážeme si pouze test v západním směru, ostatní jsou

podobné, i když nepatrně složitější. Náš první pokus může vypadat takto:

```
static bool /*****/  ZDobry  /*****/ ()
{                                   // Západ dobrý
  if( !Zed() )                      // Nemáš-li před sebou zeď, dojdi
  {
       NaZnNeboKeZdi();             // v daném směru na značku nebo ke zdi
       if( Znacka() )               // Najdeš-li značku, vrať se na výchozí
       {                            // pozici a předej výsledek NE
           JdiZpet();
           Vrat( NE );
       }
       else
       {
           JdiZpet();               // Dojdeš-li ke zdi, vrať se na výchozí
           Vrat( ANO );             // pozici a předej výsledek ANO
       }
  }
  else
       Vrat( ANO );                 // Jsi-li u zdi, vrať ANO
}
/**********  ZDobry  **********/
```

Funkce *JdiZpet* obstará návrat do výchozí pozice, na kterou jsme si prozíravě položili značku.

```
static void /*****/  JdiZpet  /*****/ ()
{
  CelemVzad();
  do
       Krok();
  while ( !Znacka() );
  CelemVzad();
}
/**********   JdiZpet   **********/
```

Mnohé z procedur můžeme naprogramovat dvěma způsoby: za použití rekurze a za použití cyklů. Abyste mohli obě možnosti porovnat, definovali jsme dvě množiny stejnojmenných funkcí realizujících požadované činnosti a umístili jsme je do samostatných modulů. Podle toho, který modul v projektu dovezete, se bude lišit i množina funkcí, které budou pro vaše programy vykonávat žádané služby.

Všechny definice rekurzivní verze naleznete v souborech *BACKTR_R.xxx*. Řešení pomocí cyklů najdete v souboru *BACKTR_C.xxx* , a proto je nebudeme podrobně rozebírat.

Na závěr bychom se chtěli ještě stručně zmínit o vztahu rekurze a cyklu. Rekurzivní volání je obecnější konstrukcí. Máme-li dost paměti, můžeme každý cyklus vyjádřit prostřednictvím rekurze; na druhé straně ne každou rekurzi lze jednoduše přepsat pomocí cyklu.

Rekurze je sice obecnější programovou konstrukcí, ale za tuto obecnost platíme zvýšenou režií – větší spotřebou paměti a delším časem provádění. V našem příkladě však tato propast není až tak veliká. Náš počítač by měl mít dostatek paměti pro běh rekurzivních definic na libovolné povolené velikosti dvorku a ani časy se neliší natolik, abychom mohli rozdíl rozumně změřit.

16. Dodatek

V této knize jsme se snažili seznámit vás se základy vytváření programů v jazycích Pascal a C++. Nyní přišel čas uspořádat získané vědomosti. V této kapitole najdete vedle celkového shrnutí probrané látky také přehled nejdůležitějších probraných konstrukcí jazyků C++ a Pascal a vše potřebné o systému Karel. Rádi bychom upozornili, že **některé syntaktické definice jsou zjednodušené** – odpovídají probrané látce, ale neukazují vše, co daný jazyk umožňuje. S přesnou podobou se seznámíte v dalších dílech, až se naučíte pracovat s daty a s objekty.

16.1 Celkové shrnutí

Zopakujme si, co jsme se všechno naučili:

1. Umíme pracovat s integrovaným vývojovým prostředím borlandských překladačů, tj. umíme otevírat, zavírat, zvětšovat a zmenšovat jejich okna a umíme nastavovat jejich jednotlivé volby.
2. Umíme pracovat s vestavěným editorem, tj. umíme otevírat, přejmenovávat a zavírat soubory, přesunovat bloky textu mezi soubory, vyhledávat párové závorky a mnoho dalších užitečných funkcí.
3. Známe základy práce se zabudovaným ladicím programem: umíme program přeložit a sestavit, umíme jej krokovat a nastavovat v něm zarážky, přepínat mezi ladicím prostředím a výstupní obrazovkou programu i využívat okna *Output* k průběžnému sledování výstupu testovaného programu.
4. Umíme definovat proceduru i funkci a víme, čím se od sebe liší. Víme, co jsou to vedlejší efekty funkcí, a umíme navrhnout funkce tak, aby nám využití jejich vedlejších efektů zjednodušilo, zpřehlednilo i zefektivnilo program.Víme, že oba překladače umožňují používání funkcí jako procedur.
5. Známe hlavní zásady technologie programování shora dolů i zdola nahoru a dokážeme posoudit, který přístup je za daných podmínek vhodnější. Víme, co je to dekompozice problému a známe důležitost včasné definice mezimodulového rozhraní.
6. Víme, jak má vypadat hlavní i řadový modul, co je mezimodulové rozhraní a jaký vliv má změna zdrojového textu těla modulu i změna textu me-

zimodulového rozhraní na celý proces překladu. Umíme rozdělit řešení problému do modulů.

7. Známe důležitost přiměřené grafické úpravy programů a známe nejpoužívanější způsoby této úpravy.
8. Známe nejdůležitější vlastnosti obou hlavních druhů podmíněných cyklů, tj. cyklu s podmínkou na počátku a cyklu s podmínkou na konci, a umíme se rozhodnout, který z nich je v dané situaci vhodnější, a použít jej v programu.
9. Známe různé varianty příkazu **if** a umíme je odpovídajícím způsobem použít ve svých programech.
10. Známe hlavní zásadu strukturovaného programování i nejdůležitější zásady pomocné a dokážeme podle nich programovat.
11. Známe programové konstrukce, které sice pomocným zásadám strukturovaného programování odporují, ale které nám někdy umožňují program zpřehlednit a tím vyhovět zásadě hlavní. Umíme tyto konstrukce použít ve svých programech.
12. Známe základní principy rekurzivního volání a umíme tuto mocnou konstrukci použít ve svých programech.
13. Známe základní myšlenku metody pokusů a oprav a umíme ji v konkrétních případech aplikovat.
14. Umíme v jazyku C++ používat pro zjednodušení některých konstrukcí operátoru čárka.

Tím jsme spolu ukončili první výlet do světa programování. Seznámili jsme se na něm se všemi programovými konstrukcemi, při jejichž používání nepotřebujete prakticky žádné znalosti o práci s daty. Dalším logickým krokem by tedy mělo být seznámení s použitím dat – nejlépe prostřednictvím dalších dílů naší řady knih o programování.

16.2 Probrané konstrukce jazyka C++

Lexikální konvence a zápis programu

Při překladu programu si musí počítač nejprve rozebrat zdrojový text na jed-

notlivé položky (anglicky označované *token*), jako jsou identifikátory, klíčová slova, operátory atd. Přitom platí, že z daných znaků vytvoří vždy největší možnou položku: K části, kterou již rozebral, bude přidávat další a další znaky, dokud to bude možné, dokud budou tvořit nějakou možnou položku.

To znamená, že pokud napíšeme

```
if ( Zed() ) Zvedni(); elsePoloz();
```

pochopí překladač zápis *elsePoloz* jako jeden celek, jako identifikátor, a nerozloží si ho na dvě položky, na klíčové slovo **else** a identifikátor *Poloz*.

Z toho plyne, že

- ✧ identifikátor, klíčové slovo ani víceznakový operátor (např. ||) nesmíme rozdělit žádným bílým znakem (mezerou, přechodem na nový řádek, tabulátorem, komentářem);
- ✧ pokud stojí bezprostředně vedle sebe dva identifikátory nebo identifikátor a klíčové slovo, musíme je oddělit alespoň jedním bílým znakem.

Navíc platí, že všude, kde smíme zapsat mezeru, smíme zapsat libovolný počet bílých znaků.

Komentář

Komentář obsahuje poznámky, kterými programátor vysvětluje smysl a důvod použitých obratů. Překladač komentář ignoruje, to znamená, že z hlediska sémantiky programu nemá komentář význam. V C++ používáme dva druhy komentářů:

1. Komentář začíná dvojicí znaků `/*` a končí dvojicí znaků `*/`. Tyto komentáře lze používat i v jazyku C.
2. Komentář začíná znaky `//` a končí na konci řádku.

Komentáře začínající `/*` a končící `*/`nelze vnořovat jeden do druhého.

Program a moduly

Každý program se skládá z jednoho nebo několika modulů. Jeden z nich je tzv. hlavní modul, ostatní označujeme jako řadové. Hlavní modul se v C++ liší od ostatních pouze tím, že obsahuje funkci[31] *main*. Vedle toho může obsahovat i definice dalších funkcí.

[31] Ve skutečnosti jde o proceduru, neboť v podobě, v níž jsme ji poznali, nevrací žádnou hodnotu. V terminologii jazyků C a C++ se však procedury a funkce nerozlišují.

Chceme-li v jednom modulu použít funkci definovanou v jiném modulu, musíme to překladači oznámit (modul musí tuto funkci importovat). K tomu stačí, když před použitím uvedeme prototyp dovážené funkce.

Abychom nemuseli prototypy stále dokola opisovat, ukládáme je do hlavičkových souborů. Hlavičkový soubor jakéhokoli modulu obsahuje deklarace prostředků, které tento modul vyváží. Do modulu, který chce tyto prostředky dovézt, pak vložíme hlavičkový soubor direktivou

```
#include <jméno_souboru>    // hlavičkový soubor pro knihovní funkce
```

nebo

```
#include "jméno_souboru"    // hlavičkový soubor pro naše moduly
```

Hlavičkové soubory mají obvykle příponu *.H* nebo *.HPP*.

Definice a deklarace procedury

Zjednodušený syntaktický popis definice procedury má tvar

Definice_procedury:
hlavička_procedury tělo_procedury

Hlavičku_procedury popíšeme takto:

Hlavička_procedury:
void *identifikátor_procedury* **()**

Tělo_procedury je složený příkaz, jehož syntaktickou definici najdete dále. Deklarace procedury (prototyp) je prostě hlavička, za kterou následuje středník:

Deklarace_procedury:
Hlavička_procedury **;**

Definice a deklarace funkce

Zatím známe pouze funkce vracející logické hodnoty. Zjednodušený syntaktický popis logické funkce má tvar

Definice_funkce:
hlavička_funkce tělo_funkce

Hlavička_funkce má tvar

Hlavička_funkce:

bool *identifikátor_funkce* **( )**

Tělo_funkce je složený příkaz, který musí začínat příkazem *Podminka()*. Další příkazy obsahují výpočet výsledku. Používáme-li Karla, můžeme vypočtenou hodnotu připravit ke vrácení příkazem

Hodnota(Výraz);

a do volajícího podprogramu se vrátit příkazem

Predej();

Následují-li tyto dva příkazy bezprostředně za sebou, můžeme je nahradit jediným příkazem

Vrat(Výraz);

Chceme-li vrátit hodnotu *výrazu*, můžeme také použít standardního příkazu jazyka C++ **return**:

Vrácení_vypočtené hodnoty:
return *výraz***;**

Tento příkaz ukončí provádění těla funkce a do volajícího podprogramu vrátí hodnotu *výrazu.*

Deklarace funkce (prototyp) je prostě hlavička, za kterou následuje středník:

Deklarace_funkce:
Hlavička_funkce **;**

Složený příkaz

Jestliže v programu potřebujeme zapsat několik příkazů na místě, kde nám syntaktická pravidla dovolují jediný příkaz, použijeme složený příkaz. Jeho syntaktický popis je

složený_příkaz:
{ [*Příkaz*]$_{\text{opak}}$**}**

Jde tedy o skupinu příkazů, uzavřenou ve složených závorkách. Za složeným příkazem neděláme středník; naopak před uzavírací složenou závorkou středník píšeme, pokud je součástí posledního z *příkazů* tvořících složený příkaz.

Výrazový příkaz, prázdný příkaz

V C++ můžeme libovolný výraz použít jako příkaz – stačí za něj připojit středník. Syntax tohoto tzv. příkazového výrazu je

Příkazový_výraz:
*výraz***;**

Zvláštním případem výrazového příkazu je prázdný příkaz, tedy příkaz, který nedělá nic. Takový příkaz můžeme zapsat buď jako samotný středník (prázdný výraz, tedy nic, ze kterého uděláme příkaz připojením středníku) nebo jako prázdný složený příkaz, tedy prázdné příkazové závorky.

Prázdný_příkaz:
;
{}

Volání procedury

Volání procedury způsobí, že se provedou příkazy obsažené v jejím těle. Syntaktický popis volání procedury je

Volání procedury:
*Identifikátor_procedury***();**

V C++ můžeme jako procedury volat i funkce. Poznamenejme, že všechny Karlovy příkazy – i vestavěné, se kterými jsme začínali – jsou volání procedur vyvážených modulem Karel. Poznamenejme, že také volání procedury je v C++ zvláštním případem výrazového příkazu.

Cyklus while

Syntaktický popis cyklu **while** je

Cyklus_while:
while (*podmínka* **)** *příkaz*

Podmínka je libovolný logický výraz. (V dalších dílech se dozvíme, že v C++ můžeme jako podmínku použít výraz s libovolnou číselnou hodnotou nebo dokonce ukazatel.) *Příkaz* tvoří tělo cyklu.

Při provádění cyklu **while** se nejprve vyhodnotí *podmínka*. Není-li splněna (má-li hodnotu *NE*), cyklus skončí a jeho tělo se neprovede ani jednou. Je-li *podmínka* splněna (má-li hodnotu *ANO*), provede se tělo cyklu a opět se vyhodnotí *podmínka*. Není-li splněna, provádění cyklu skončí, tělo se již nepro-

vede; je-li splněna, provede se tělo cyklu a opět se vyhodnotí *podmínka* atd.

Cyklus do – while

Syntaktický popis[32] cyklu **do – while** je

Cyklus_do-while:
do *příkaz* **while (** *podmínka* **);**

Podmínka je opět libovolný logický výraz a *příkaz* tvoří tělo cyklu. Zde se nejprve provede tělo cyklu. Pak se vyhodnotí *podmínka*; není-li splněna, provádění cyklu skončí, jinak se znovu provede tělo cyklu a vyhodnotí se podmínka. Má-li nyní hodnotu *NE*, cyklus skončí, jinak se opět provede tělo cyklu a opět se vyhodnotí podmínka atd.

Připomeňme si, že tělo tohoto cyklu se provede vždy alespoň jednou.

Podmíněný příkaz

Podmíněný příkaz slouží k rozvětvení algoritmu[33]. Jeho syntaktický popis je

Příkaz_if:
if (*podmínka* **)** *příkaz_1 [***else** *příkaz_2]*

Zde se nejprve vyhodnotí *podmínka*. Je-li splněna (má-li hodnotu *ANO*), provede se *příkaz_1*. Není-li *podmínka* splněna a chybí-li část s klíčovým slovem **else**, neprovede se nic, jinak se provede *příkaz_2*.

Pro čtenáře, kteří používají zároveň Pascal, může být poněkud matoucí, že před **else** může být středník. Píšeme jej za jednoduchým příkazem (přesněji je součástí tohoto příkazu), nikoli však za složeným příkazem:

```
if( Zed() ) Poloz(); else Zvedni();    // Zde je středník před else
if( Jih() )
{
  Krok();
  Poloz();
}                                      // Zde však být nesmí
else Zvedni();
```

[32] V příštím dílu se seznámíme ještě s cyklem **for**. Abychom jej mohli používat, musíme ovšem znát alespoň základy práce s daty.

[33] V příštím dílu se seznámíme s příkazem **switch**, který umožňuje rozvětvení algoritmu na více možností. Také použití tohoto příkazu je však podmíněno znalostí základů práce s daty.

Příkaz skoku

Příkaz skoku má tvar

Příkaz_skoku:
goto *návěští***;**

Návěští je identifikátor, připojený před příkaz a oddělený od něj dvojtečkou:

Příkaz_s_návěštím:
návěští **:** *příkaz*

Návěští musí ležet v téže funkci jako příkaz skoku. Na rozdíl od Pascalu ho nemusíme (a ani nemůžeme) deklarovat předem, je deklarováno svým výskytem u příkazu.

Příkaz break

Syntax tohoto příkazu je velice jednoduchá:

Příkaz_break:
break;

Tento příkaz smíme použít jen v těle některého z příkazů cyklu[34]; způsobí, že počítač ukončí provádění těla cyklu a přejde na první příkaz za ním.

Příkaz continue

Syntax příkazu **continue** je

Příkaz_continue:
continue;

Tento příkaz smíme použít jen v těle některého z příkazů cyklu; způsobí, že počítač ukončí právě prováděnou iteraci (průchod tělem cyklu) a znovu vyhodnotí podmínku opakování.

Návrat z podprogramu – příkaz return

Provádění procedury skončí tím, že řízení přejde přes uzavírací složenou **{** závorku jejího těla. Potřebujeme-li se z nějakých důvodů vrátit z procedury předčasně, použijeme příkaz **return**. Jeho syntax je

[34] V příštím dílu se dozvíme, že jej můžeme použít také v těle dosud neprobraného příkazu **switch**.

Příkaz_return:
return *[výraz]***;**

Při návratu z procedury jej použijeme v podobě

return;

Tento příkaz způsobí okamžitý návrat do volajícího podprogramu.

Ve funkci musíme před návratem vždy zařídit vrácení vypočtené hodnoty. To znamená, že v ní musíme použít buď některou z možností definovaných v modulu Karel, nebo příkaz

return *výraz*;

Podrobnosti najdete v oddílu věnovanému definici a deklaraci funkce.

16.3 Probrané konstrukce jazyka Pascal

Zápis programu

Pro zápis programu v Pascalu platí, že

- ✧ identifikátor, klíčové slovo ani víceznakový operátor (např. **:=**) nesmíme rozdělit žádným bílým znakem (mezerou, přechodem na nový řádek, tabulátorem, komentářem);
- ✧ pokud stojí bezprostředně vedle sebe dva identifikátory nebo identifikátor a klíčové slovo, musíme je oddělit alespoň jedním bílým znakem.

Navíc platí, že všude, kde smíme zapsat mezeru, smíme zapsat libovolný počet bílých znaků.

Komentář

Komentář obsahuje poznámky, kterými programátor vysvětluje smysl a důvod použitých obratů. Překladač komentář ignoruje, to znamená, že z hlediska sémantiky programu nemá komentář význam. V Pascalu používáme dva druhy komentářů:

1. Komentář začíná složenou závorkou `{` a končí složenou závorkou `}`.
2. Komentář začíná znaky `(*` a končí znaky `*)`.

Komentáře nelze vnořovat jeden do druhého. Začíná-li komentář složenou závorkou, musí touto závorkou také končit; začíná-li znaky `(*`, musí končit znaky `*)`.

Program a moduly

Každý program se skládá z jednoho nebo několika modulů. Jeden z nich je tzv. hlavní modul, ostatní označujeme jako řadové. Každý modul musí být v samostatném souboru. Zjednodušený syntaktický popis hlavního modulu vypadá takto:

Hlavní_modul:
*[***program** *identifikátor* **;***]* {*hlavička programu*}
\+ *[***uses** *identifikátor [***,***identifikátor]*$_{\text{opak}}$**;***]* {*deklarace importu*}
\+ *[deklarace_návěští]*
\+ *[deklarace_a_definice_procedur_a_funkcí]*
\+ **begin** *[příkaz*$_{\text{opak}}$**;***]* **end.**

Hlavička hlavního modulu obsahuje klíčové slovo **program** a *identifikátor* (jméno) programu. Je nepovinná. Za ní následuje úsek, ve kterém specifikujeme importy. Začíná klíčovým slovem **uses**, za nímž následuje seznam identifikátorů dovážených modulů. Pokud v programu používáme příkaz **goto**, musí následovat deklarace návěští (viz oddíl věnovaný příkazu skoku). Pak následují definice a deklarace funkcí a procedur; případné deklarace musejí obsahovat direktivu **forward**. Nakonec přijde vlastní tělo programu, které začíná **begin**. Obsahuje příkazy, které má náš program provést – zpravidla jde o volání procedur a funkcí, deklarovaných v hlavním programu nebo dovezených z různých modulů. Tělo programu končí klíčovým slovem **end**, následovaným tečkou. Vše za touto tečkou překladač ignoruje.

Řadový modul se v Pascalu nazývá jednotka (*unit*). Zjednodušený syntaktický popis jednotky je

Jednotka:
unit *identifikátor***;** {*hlavička jednotky*}
\+ **interface** {*následuje specifikace exportu*}
\+ *[***uses** *identifikátor [***,***identifikátor]*$_{\text{opak}}$**;***]* {*importy, které ovlivňují export*}
\+ *deklarace_vyvážených_procedur_a_funkcí* {*prototypy*}

+ **implementation** {*implementace export. prostředků*}
+ *[***uses** *identifikátor [,identifikátor]* $_{opak}$**;***]* {*importy, které neovlivňují export*}
+ *deklarace_návěští*
+ *deklarace_a definice_procedur_a_funkcí*
+ **begin** *[příkaz* $_{opak}$**;***]* **end.** {*inicializace modulu*}

Hlavička jednotky se skládá z klíčového slova **unit**, za nímž následuje *identifikátor* jednotky. **Tento identifikátor se musí shodovat se jménem souboru, ve kterém bude jednotka uložena**. To znamená, že např. jednotka *Alfa* musí být v souboru *Alfa.pas*. Pak přijde klíčové slovo **interface**, jež uvádí rozhraní jednotky – tedy specifikace exportovaných prostředků. Za ním může následovat specifikace dovážených jednotek. Pak následují specifikace vyvážených prostředků; my zde zatím nemůžeme zapsat nic jiného než prototypy vyvážených procedur a funkcí (bez direktivy **forward**).

Pak přijde klíčové slovo **implementation**. Za ním může opět následovat dovoz použitých modulů. Pokud používáme v inicializační části příkaz skoku, musí následovat deklarace použitých návěští. Pak přijdou na řadu definice vyvážených procedur a funkcí. V této části můžeme definovat (a také deklarovat) i pomocné, lokální procedury a funkce, které použijeme při implementaci vyvážených služeb, ale které nebudou mimo danou jednotku dostupné.

Nakonec přijde inicializační část modulu. Podobá se tělu hlavního programu: začíná klíčovým slovem **begin**, pak následuje posloupnost příkazů a končí klíčovým slovem **end** následovaným tečkou. Vše za touto tečkou překladač ignoruje. Pokud neobsahuje inicializační část ani jeden příkaz, můžeme vynechat i klíčové slovo **begin**.

Příkazy inicializační části se provedou před spuštěním hlavního programu.

Definice a deklarace procedury

Zjednodušený syntaktický popis definice procedury má v Pascalu tvar

Definice_procedury:
hlavička_procedury tělo_procedury

Hlavičku_procedury popíšeme takto:

Hlavička_procedury:

procedure *identifikátor_procedury***;**

Zbývá ještě doplnit popis *těla_procedury*:

Tělo_procedury:
[deklarace_návěští]
\+ *složený_příkaz*

Syntaktickou definici *složeného příkazu* najdete dále. Deklarace procedury (prototyp) je prostě hlavička, za kterou následuje středník, případně klíčové slovo (přesněji direktiva) **forward**:

Deklarace_procedury:
Hlavička_procedury **;** *[***forward ;***]*

Možnost s direktivou **forward** použijeme u prototypů v hlavním modulu a v sekci **implementation** jednotek. U prototypů v sekci **interface** jednotek se nepoužívá.

Definice a deklarace funkce

Zatím známe pouze funkce, vracející logické hodnoty. Zjednodušený syntaktický popis logické funkce má tvar

Definice_funkce:
hlavička_funkce tělo_funkce

Hlavička_funkce má tvar

Hlavička_funkce:
function *identifikátor_funkce* **: boolean;**

Popis *těla_funkce* je podobný jako popis *těla_procedury*:

Tělo_funkce:
[deklarace_návěští]
\+ *složený_příkaz*

Hodnotu, kterou funkce vrátí volajícímu podprogramu, definujeme příkazem

Vracená_hodnota:
identifikátor_funkce **:=** *výraz*

Běh funkce skončí tím, že řízení přejde přes závěrečné **end** těla funkce. Chceme-li ji opustit předčasně, můžeme zavolat proceduru *exit*.

Deklarace funkce (prototyp) je prostě hlavička, za kterou následuje středník:

Deklarace_funkce:

Hlavička_funkce **;** *[***forward ;***]*

Pro použití direktivy **forward** platí totéž, co v případě procedur.

Složený příkaz

Jestliže v programu potřebujeme zapsat několik příkazů na místě, kde nám syntaktická pravidla dovolují jediný příkaz, použijeme složený příkaz. Jeho syntaktický popis je

složený_příkaz:

begin *[Příkaz]*$_{\text{opak}}$ **end**

Jde tedy o skupinu příkazů, uzavřenou mezi „příkazovými závorkami“ **begin** a **end**. Protože **end** slouží jako oddělovač příkazů, nemusí být za posledním příkazem před ním středník.

Volání procedury

Volání procedury způsobí, že se provedou příkazy obsažené v jejím těle. Syntaktický popis volání procedury je

Volání procedury:

*Identifikátor_procedury***;**

Nejnovější verze Turbo Pascalu umožňují volat jako procedury také funkce. Musí ale být nastaven přepínač Extended syntax v okně Options|Compiler ve skupině Syntax options. Poznamenejme, že všechny Karlovy příkazy – i vestavěné, se kterými jsme začínali – jsou volání procedur vyvážených modulem Karel.

Prázdný příkaz

Prázdný příkaz je příkaz, který nedělá nic. Neobsahuje žádný znak, takže jeho syntaktická definice je opravdu triviální:

Prázdný příkaz:

{*definice neobsahuje žádný znak*}

Cyklus while

Syntaktický popis cyklu **while** je

Cyklus_while:
while *podmínka* **do** *příkaz*

Podmínka je libovolný logický výraz. *Příkaz* tvoří tělo cyklu.

Při provádění cyklu **while** se nejprve vyhodnotí *podmínka*. Není-li splněna (má-li hodnotu *NE*), cyklus skončí a jeho tělo se neprovede ani jednou. Je-li *podmínka* splněna (má-li hodnotu *ANO*), provede se tělo cyklu a opět se vyhodnotí *podmínka*. Není-li splněna, provádění cyklu skončí, tělo se již neprovede; je-li splněna, provede se tělo cyklu a opět se vyhodnotí *podmínka* atd.

Cyklus repeat – until

Syntaktický popis[35] cyklu **repeat – until** je

Cyklus_repeat_until:
repeat *příkaz* $_{\text{opak}}$ **until** *podmínka*

Podmínka je opět libovolný logický výraz a *příkaz* tvoří tělo cyklu. Zde se nejprve provede tělo cyklu. Pak se vyhodnotí *podmínka*; je-li splněna, provádění cyklu skončí, jinak se znovu provede tělo cyklu a vyhodnotí se podmínka. Má-li nyní hodnotu *ANO*, cyklus skončí, jinak se opět provede tělo cyklu a opět se vyhodnotí podmínka atd.

Připomeňme si, že tělo tohoto cyklu se provede vždy alespoň jednou.

Podmíněný příkaz

Podmíněný příkaz slouží k rozvětvení algoritmu[36]. Jeho syntaktický popis je

Příkaz_if:
if *podmínka* **then** *příkaz_1 [***else** *příkaz_2]*

Zde se nejprve vyhodnotí *podmínka*. Je-li splněna (má-li hodnotu *ANO*), provede se *příkaz_1*. Není-li *podmínka* splněna a chybí-li část s klíčovým slovem **else**, neprovede se nic, jinak se provede *příkaz_2*.

---

[35] V příštím dílu se seznámíme ještě s cyklem **for**. Abychom jej mohli používat, musíme ovšem znát alespoň základy práce s daty.

[36] V příštím dílu se seznámíme příkazem **case**, který umožňuje rozvětvení algoritmu na více možností. Také použití tohoto příkazu je však podmíněno znalostí základů práce s daty.

Příkaz skoku

Příkaz skoku má tvar

Příkaz_skoku:
goto *návěští*

Návěští je identifikátor připojený před příkaz a oddělený od něj dvojtečkou:

Příkaz_s_návěštím:
návěští **:** *příkaz*

Návěští musí ležet v téže funkci jako příkaz skoku. Musíme ho deklarovat předem, mezi hlavičkou a tělem funkce, v níž ho použijeme:

Deklarace_návěští:
label *návěští [***,** *návěští]*$_{\text{opak}}$**;**

Deklarace návěští je prostě seznam návěští (tedy jednotlivá návěští, oddělená čárkami) uvedený klíčovým slovem **label**. Jako *návěští* můžeme použít identifikátor nebo celé číslo bez znaménka.

Příkaz break

Turbo Pascal 7.0 obsahuje příkaz *break* (podle manuálu jde o proceduru, to však na věci nic nemění). Syntax tohoto příkazu je velice jednoduchá:

Příkaz_break:
break

Tento příkaz smíme použít jen v těle některého z příkazů cyklu; způsobí, že počítač ukončí provádění těla cyklu a přejde na první příkaz za ním.

Ve starších verzích Turbo Pascalu musíme tento příkaz nahradit skokem na návěští těsně za cyklem.

Příkaz continue

Turbo Pascal 7.0 obsahuje také příkaz *continue***.** (I to je podle manuálu vlastně procedura.) Jeho syntax je

Příkaz_continue:
continue

Tento příkaz smíme použít jen v těle některého z příkazů cyklu; způsobí, že počítač ukončí právě prováděnou iteraci (průchod tělem cyklu) a znovu vyhodnotí podmínku opakování.

Ve starších verzích Turbo Pascalu musíme tento příkaz nahradit skokem na návěští u prázdného příkazu, který zařadíme jako poslední příkaz těla cyklu.

16.4 KAREL – prostředí a příkazy

Zápis pozice

Karlovu pozici zapisujeme výrazem

```
r / s : z - Směr
```

kde

- ✧ *r* je číslo řádku, na němž se Karel nachází,
- ✧ *s* je číslo sloupce, v němž se Karel nachází,
- ✧ *z* je počet značek na políčku pod Karlem – pokud nebude důležitý, nebudeme jej ani uvádět,
- ✧ *Směr* je označení směru, do nějž je Karel natočen. Někdy budeme pro zkrácení místo plného označení směru (Východ, Západ, Sever, Jih) používat pouze jednopísmenné zkratky (V, Z, S, J).

Pozici, v níž Karel stojí v prvním řádku a ve druhém sloupci, má pod sebou tři značky a je obrácen na sever, budeme tedy při zkráceném zápisu charakterizovat výrazem `1/2:3-S`.

Pozici `0/0:0-V` nazýváme standardní pozice.

Základní příkazy

Příkazy, kterým Karel rozumí od samého počátku, jsou:

Krok

Není-li před Karlem zed', udělá krok ve směru, do něhož je natočen. V opačném případě ohlásí chybu.

VlevoVbok

Otočí se o 90 stupňů vlevo.

Polož

Není-li již na políčku, na němž Karel právě stojí, maximální povolený počet

značek (v našich úlohách většinou 9), položí na něj další značku. V opačném případě ohlásí chybu.

Zvedni
Je-li na políčku, na němž Karel právě stojí, alespoň jedna značka, zvedne jednu značku. V opačném případě ohlásí chybu.

Základní podmínky

Zeď
Podmínka je splněna, pokud je na políčku před Karlem zeď.

Značka
Podmínka je splněna, pokud je na políčku pod Karlem alespoň jedna značka.

Sever
Podmínka je splněna, pokud je Karel otočen směrem na sever, tj. nahoru.

Východ
Podmínka je splněna, pokud je Karel otočen směrem na východ, tj. vpravo.

Jih
Podmínka je splněna, pokud je Karel otočen směrem na jih, tj. dolů.

Západ
Podmínka je splněna, pokud je Karel otočen směrem na západ, tj. vlevo.

Stisknuto
Podmínka je splněna, pokud byl od posledního testu stisknut mezerník.

Příkazy pro definice funkcí v C++

Podmínka
Tento příkaz by měl být prvním příkazem ve všech definicích podmínek. Teoreticky by sice stačilo, aby předcházel příkazu *Hodnota*, ale pro zvýšení přehlednosti programů na to nehřešte a definici podmínek s ním zahajujte.

Hodnota
Druhý ze čtveřice příkazů umožňujících odsunout výklad práce s daty až za výklad funkcí. Slouží k přiřazení funkční hodnoty, která se zadá do závorek za něj. Jako funkční hodnotu můžeme přiřadit hodnotu logických konstant *ANO* a *NE*; můžeme také zavolat nějakou dříve definovanou funkci. Podrobnosti jsou vysvětleny v kapitole 9.

Předej
Třetí z příkazů využívaných při definování funkcí. Slouží k předání hodnoty funkce volajícímu programu. Je „funkční obdobou“ příkazu **return** používaného v procedurách. Měl by vždy být posledním provedeným příkazem funkce.

Vrať
Poslední z příkazů využívaných při definování funkcí. Slouží k přiřazení funkční hodnoty, která se zadá do závorek za něj, a k předání řízení volajícímu programu. Je tedy spojením příkazů *Hodnota* a *Předej*.

Pomocné příkazy

Krát
Tento příkaz vám může v řadě případů ušetřit mnoho psaní. Syntax příkazu je pro oba jazyky až na závěrečný středník shodná: do závorky za příkaz napíšeme počet opakování, čárku a jméno příkazu, který se má daný počet krát zopakovat. Formálně bychom ji mohli pro oba jazyky zapsat:

Příkaz_Krát:
    Krat (*Číslo*, *Identifikátor_příkazu*)

PárKrát
Tento příkaz není pouhým rozšířením možností příkazu *Krát*. Slouží k tomu, abychom mohli do našich programů zavést náhodu. S jeho pomocí můžeme např. náhodně zaplnit dvorek značkami nebo náhodně pohybovat Karlem po dvorku. V závorkách za tímto příkazem musíte uvést tři parametry: minimální počet opakování, maximální počet opakování a opakovanou proceduru.

Chceme-li např. na nějakém políčku položit náhodný počet značek nepřevyšující 5 s tím, že na něm nakonec v1astně nemusíme položit ani jednu značku, dosáhneme toho příkazem

```
ParKrat( 0, 5, Poloz );
```

Rychlost

Pomocí tohoto příkazu lze nastavit rychlost, s jakou bude Karel jednotlivé příkazy provádět. Požadovaná rychlost se může pohybovat od 0 do 17 a zadává se do závorek za příkazem.

Při rychlosti 0 počítač po každém zobrazeném kroku čeká na stisk klávesy SCROLLLOCK a teprve po něm vykoná další akci. Při rychlosti 1 provádí jednotlivé zobrazované akce s periodou přibližně 16 vteřin. Každá další rychlost zkracuje tuto periodu přibližně na polovinu, takže při rychlosti 15 se mezi jednotlivými akcemi čeká teoreticky 1 ms a při rychlosti 16 se již nečeká vůbec.

Požadujete-li úplné zobrazení stavu dvorku po každé zobrazované akci, můžete jako nejvyšší zadat rychlost 16. Zadáte-li rychlost 17, přestane se po dobu běhu programu zobrazovat pod dvorkem aktuální Karlův stav, čímž se rychlost provádění programu ještě o něco zvýší.

Stav dvorku se zobrazí až po ukončení celého programu nebo při pozastavení programu stiskem některého z přeřaďovačů, anebo v průběhu programu, poté co se vykoná příkaz *Rychlost* s hodnotou parametru menší než 17.

Dvorek

Soubor s informacemi o Karlově dvorku je obyčejný textový soubor (doporučujeme vám pro něj příponu *.DVO*), který si můžete sami připravit libovolným textovým editorem vytvářejícím soubor ve formátu ASCII (tj. samotné znaky). Musíte pouze dodržet základní formát.

Pokud budeme v dalších odstavcích hovořit o mezerách, budeme tím myslet libovolný počet mezer a tabulátorů.

Komentář

Komentář je libovolný text následující na řádku za jeho povinnou částí (podrobněji se o jejím formátu dozvíte v následujících odstavcích). Tento text slouží pouze k informaci programátora a systém jej při čtení ignoruje. Kromě toho se za komentářové považují i řádky následující za posledním významným řádkem.

1. řádek

První řádek obsahuje postupně informace o počtu řádků dvorku (*rd*), počtu sloupců dvorku (*sd*), počtu pozic na obrazovce, které zabere jeden slou-

sloupec (šířka sloupce *ss*) a maximální povolený počet značek na jednom poli (*zp*). Jednotlivá čísla se oddělují mezerou. Musí platit

```
rd <= (ro - 12) sd <= ((so - ss - 5) / ss)
ss <= 5 zp < 9
```

kde *ro* je počet řádků na obrazovce (25, 43 nebo 50) a *so* je počet sloupců na obrazovce (40 nebo 80).

To znamená, že při 25 řádcích obrazovky může mít dvorek maximálně 13 řádků, při 43 řádcích 31 řádků a při 50 řádcích 38 řádků. Obdobně platí, že „hustý“ dvorek (tj. dvorek s neprokládanými sloupci) může mít v 80sloupcovém režimu 74 sloupců a ve 40sloupcovém režimu 34 sloupců, „řídký“ dvorek (tj. dvorek s jednou mezerou mezi sloupci) 36 a 16 sloupců.

2. řádek

Ve druhém řádku jsou zapsány znaky představující podoby Karla otočeného postupně na východ, sever, západ a jih. Znaky je možno oddělit mezerou.

Pokud chcete Karla reprezentovat znakem, jehož zadání vám dělá těžkosti (např. znaky s kódy 0-31), můžete místo tohoto znaku napsat obrácené lomítko následované dekadickým kódem žádaného znaku. Tuto notaci musíte použít i pro zadání vlastního obráceného lomítka (obrácené lomítko má kód 92, takže zapíšete \92). Zápis kódu znaku v náhradní notaci je třeba od sousedních znaků oddělit mezerami. Pokud bychom chtěli Karla reprezentovat šipkami, mohl by mít druhý řádek tvar

```
\26 \24 \27 \25
```

Pokud bychom jej reprezentovali běžnými znaky, mohl by mít tvar

```
< ^ v >
```

3. řádek

Ve třetím řádku jsou zapsány mezerami oddělené znaky symbolizující přítomnost 0, 1, 2, ... , 9 značek na daném poli. Pro oddělování znaků mezerami a jejich náhradní reprezentaci platí totéž, co pro druhý řádek. Budeme-li tedy chtít, aby byl položený počet značek reprezentován odpovídající cifrou, napíšeme:

```
1 2 3 4 5 6 7 8 9
```

4. řádek

Čtvrtý řádek obsahuje Karlovu výchozí pozici. Nejprve je číslo řádku, pak mezerami oddělené číslo sloupce a nakonec mezerami oddělený znak

označující směr, do něhož bude Karel na počátku natočen (na velikosti písmene nezáleží). Standardní výchozí pozici, při níž je Karel v nultém řádku a nultém sloupci otočen na východ, bychom tedy zapsali

```
0 0 V
```

Další řádky

Další řádky definují počáteční stav Karlova dvorku. Každé políčko je reprezentováno jedním znakem. Mezi jednotlivými znaky mohou, ale nemusejí být mezery. Číslice symbolizuje počet značek umístěných na políčku, znak „Z“ nebo „z“ symbolizuje zeď.

Řádky jsou číslovány shora dolů, tj. obráceně, než jak budou číslovány na obrazovce. V důsledku toho zde můžete mít vždy znázorněn dvorek maximální možné velikosti a skutečný počet přečtených řádků a sloupců bude definován informacemi z prvého řádku.

Pro usnadnění práce s dvorkem nabízí systém následující příkazy.

NačtiDvorek

Po tomto příkazu počítač přečte parametry dvorku ze souboru, jehož jméno uvedete v závorkách za příkazem. Toto jméno uzavřou pascalisté mezi apostrofy a céčkaři mezi uvozovky, např. takto:

```
NactiDvorek( 'DVO\Zakladni.DVO' );                    {Pascal}
NactiDvorek( "DVO\Zakladni.DVO" );                    //C++
```

NovýDvorek

Po tomto příkazu vrátí systém dvorek do počátečního stavu. Pokud se od startu programu vykonal příkaz *NačtiDvorek*, je počátečním stavem dvorku naposledy načtený stav.

UložDvorek

Po tomto příkazu uloží systém charakteristiku okamžitého stavu dvorku do souboru, jehož název zadáváte stejně jako v příkazu *NačtiDvorek*. Tedy např.:

```
UlozDvorek( 'DVO\Nahradni.DV0' );                     {PASCAL}
UlozDvorek( "DVO\Nahradni.DV0" );                     //C++
```

TiskniDvorek

Po tomto příkazu vytiskne počítač okamžitou charakteristiku dvorku na tiskárnu. Tuto charakteristiku tiskne ve stejném formátu, v jakém by ji uklá-

ukládal do souboru.

Inicializace systému

Možnost načtení vámi definované podoby dvorku lze nejjednodušeji využít tak, že jméno souboru s charakteristikou požadovaného dvorku uvedete při volání svého programu jako parametr v příkazové řádce.

Pokud jste tedy vytvořili např. program *ROTACE* a chcete jej spustit na stejnojmenném dvorku, zadáte z příkazové řádky

```
ROTACE ROTACE.DVO
```

a Karel začne vykonávat váš program na požadovaném dvorku.

Pokud provozujete systém pod integrovaným vývojovým prostředím, zadáte jméno souboru s požadovaným dvorkem v Pascalu pomocí dialogového okna [Run | Parameters], v C++ pomocí resp. [Run | Arguments].

Pomocné příkazy pro ladění

Kromě výše uvedených základních příkazů a podmínek je do systému, který naleznete na doprovodné disketě, zabudováno i několik rozšiřujících příkazů, které vám mají usnadnit přípravu úloh a následné ladění programů a jejich vhodnou demonstraci.

Čekej

Počítač vykreslí na obrazovku aktuální stav Karlova dvorku, tj. pozici Karla a rozmístění stěn a značek, a poté zastaví provádění programu a počká na stisk libovolné klávesy. Pokud touto klávesou bude klávesa ESC, počítač provádění programu ukončí, v opačném případě v plnění programu dále pokračuje.

Ukaž

Počítač vykreslí na obrazovku stav Karlova dvorku, a pokud od posledního testu nikdo nestiskl klávesu ENTER, pokračuje ve vykonávání zadaného programu. Pokud byla tato klávesa stisknuta, počítač zastaví provádění programu a stejně jako u příkazu *Čekej* čeká na stisk další klávesy, aby pak program ukončil nebo dále pokračoval v jeho plnění.

BezKarla

Počítač nebude na dvorku Karla zakreslovat. Tím se na jednu stranu zrychlí provádění příkazů *Krok* a *VlevoVbok* a na druhou stranu se před divákem utají pomocné akce, které musí Karel pro splnění úkolu udělat.

SKarlem

Počítač bude opět Karla na dvorku zobrazovat. Tento režim je nastaven implicitně. Byl-li použit příkaz *BezKarla* vícekrát, bude „viditelný“ režim znovu nastaven až po stejném počtu příkazů *SKarlem*.

Tajně

Počítač bude program vykonávat, ale výsledek nebude zobrazovat. Chceme-li zobrazit aktuální stav dvorku, musíme o to explicitně požádat příkazem *Ukaž* nebo *Čekej*.

Zjevně

Počítač bude opět zobrazovat stav dvorku po každé akci. Tento režim je nastaven implicitně. Byl-li použit příkaz *Tajně* vícekrát, bude „zjevný“ režim nastaven až po stejném počtu příkazů *Zjevně*.

Ladím

Tento program nastavuje hladinu utajení, při níž se ještě zobrazuje stav dvorku po vykonání každého primitiva, na hodnotu uvedenou v následujících závorkách. Slouží k tomu, abychom mohli odladit podprogramy využívající utajení, aniž bychom je jakkoliv upravovali.

Chceme-li tedy ladit (a tedy zviditelnit) podprogram, který využívá utajení a je volán z jiného podprogramu, který taktéž využívá utajení, musíme dát v hlavním programu příkaz

```
Ladim( 2 );
```

Podrobnější vysvětlení utajovacích příkazů

Příkaz *Tajně* potlačuje zobrazování stavu dvorku po vykonání každého primitiva (*Krok*, *VlevoVbok*, *Polož*, *Zvedni*). Tím lze za prvé zrychlit chod programu a za druhé zneviditelnit akce, jejichž „veřejné provádění“ by působilo rušivě.

Vezměme si například funkci *ZnačkaVpředu*, která má vrátit informaci o tom, zda je na některém z políček před Karlem položena značka. Pokud bychom tuto funkci použili v programu, vypadalo by elegantněji, kdyby nebylo vidět, jak Karel vyrazí vpřed, aby zjistil, zda na některém políčku není značka, a pak se opět vrátil na své místo. Naprogramujeme proto funkci (C++)

```
bool /*****/ ZackaVpredu /*****/ ( )
Podminka ();
Tajne( ) ;
```

```
//Sem prijde vlastní tělo funkce
Zjevne ( ) ;
Predej ();
}
```

Stav dvorku se při testování takto naprogramované podmínky na obrazovce ani na chvilku nezmění. Systém si jednotlivá utajení kumuluje. Pokud bychom tedy v utajeném úseku použili podprogram, který své tělo také utajuje, příkaz *Zjevně* v tomto podprogramu nenastaví původní režim, ale pouze sníží stupeň utajení zpět na hodnotu nastavenou ve volajícím programu.

Rejstřík