# C++ pod lupou

Nie som ortodoxným prívržencom nijakého dnes používaného jazyka, poznám ich už riadnu kôpku, ale najbližšie mám práve k C++. Prečo, o tom by sa dalo diskutovať donekonečna, nie je to však cieľom tohto seriálu. Nebudem vás presviedčať o výhodách a zamlčiavať nevýhody C++. Budem sa snažiť predostrieť vám tento jazyk v celej jeho kráse tak, aby ste po skončení boli schopní písať programy na relatívne slušnej (i keď určite nie profesionálnej) úrovni. Lenže pozor, nijaký kurz, seriál či kniha vám nemôžu dať to najdôležitejšie – prax. Vyžaduje si to napísať veľké množstvo vlastných programov a preštudovať ešte viac cudzích. Berte, prosím, teda tento seriál ako sprievodcu, ako akýsi návod, uvedomte si však, že podstatné množstvo skutočnej práce zostane v každom prípade na vás. Samozrejme, rátam s vašimi ohlasmi, pripomienkami či otázkami, ak niečomu nebudete rozumieť. Výhodu budú mať tí z vás, ktorí majú prístup k elektronickej pošte – moja adresa je na konci článku.

Pri výklade sa budem snažiť používať tie slovenské pojmy, ktoré sa mi budú zdať dostatočne bežné; ak to bude možné, uvediem v zátvorke príslušný anglický ekvivalent. Všetky programy by mali byť funkčné pre architektúru Intel a príslušné operačné systémy. Čo sa týka prekladačov potrebných na preklad programov, dlho postačí veľmi rozšírený Borland C++ 3.1, prípadne MS Visual C++ 1.0 (ktorý možno získať za zanedbateľnú sumu spolu s knihou *Programujeme v jazyce Visual C++*. Computer Press 1997). Podotýkam, že seriál sa nebude zaoberať opisom prostredia prekladačov a ich ovládania, na tieto účely odporúčam preštudovať si príručku alebo nápoveď k programu, prípadne obrátiť sa na niekoho, kto má viac skúseností.

A ešte posledná poznámka: Seriál čítajte pozorne a nepreskakujte odseky, pretože rozhodne nemám v úmysle vypĺňať text nepodstatnou „vatou“ a môže vám uniknúť podstatný detail. A teraz nám nezostáva nič iné ako začať.

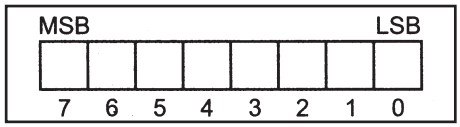
## Základné pojmy

Na úvod si dovolím objasniť niekoľko pojmov, ktoré budeme ďalej používať. Niekomu sa možno budú zdať triviálne, v takom prípade nech prejde na ďalší odsek.

**Bit** je základná jednotka informácie, premenná s veľkosťou 1 bit (logická premenná, boolovská premenná – boolean variable) môže nadobúdať dva stavy – log. 0 (false) a log. 1 (true).

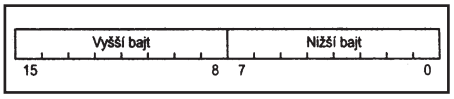
**Bajt** je postupnosť ôsmich bitov. V počítačovej branži je táto jednotka význačná vďaka tomu, že elementárnou adresovateľnou jednotkou pamäte je práve jeden bajt. Bity v bajte sa číslujú od 0 po 7, pričom bit č. 0 sa obyčajne kreslí najviac vpravo a nazývame ho najmenej významný bit (least significant bit – LSB), bit č. 7 je, naopak, najviac vľavo a nazýva sa najvýznamnejší bit (most significant bit – MSB). Na obr. 1 je bajt tak, ako sa zvykne kresliť – každý štvorček predstavuje jeden bit. Keďže bajt je zložený z ôsmich bitov a každý bit môže byť v jednom z dvoch stavov, podľa kombinatorického pravidla si ľahko spočítame, že premenná s dĺžkou jedného bajtu môže nadobúdať spolu $2^8 = 256$ rôznych hodnôt (podotýkam, ak to niekomu ešte nedošlo, že hodnota je v bajte uložená vo forme binárneho čísla). Najpoužívanejšie násobky bajtu sú

*kilobajt* (Kbyte, KB) = $2^{10}$ = 1 024 B,
*megabajt* (Mbyte, MB) = $2^{20}$ = 1 048 576 B
a *gigabajt* (Gbyte, GB) = $2^{30}$ = 1 073 741 824 B.



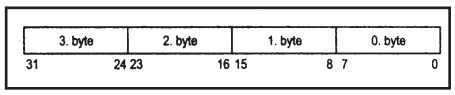

*Obrázok č. 1 Byte*

**Slovo** (word) je postupnosť dvoch bajtov, jeho dĺžka je teda 16 bitov (často sa nazýva aj *16-bitové slovo*). Funkciu najvýznamnejšieho bitu preberá bit č. 15 (čo je vlastne v poradí šestnásty bit sprava). Dva bajty, z ktorých sa slovo skladá, sa často nazývajú nižší (dolný, low) bajt (t. j. bity 0 až 7) a vyšší (horný, high) bajt (bity 8 až 15), pozri obr. 2. Premenná s veľkosťou jedného slova môže nadobúdať $2^{16} = 65\,536$ rôznych hodnôt.



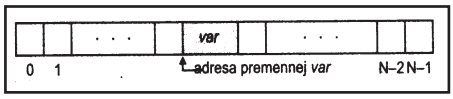

*Obrázok č. 2 Slovo*

**Dvojslovo** (double-word) je postupnosť 4 bajtov, resp. 32 bitov (často sa nazýva aj *32-bitové slovo*), pozri obr. 3. Počet rôznych hodnôt premennej tejto veľkosti je, samozrejme, $2^{32}$.



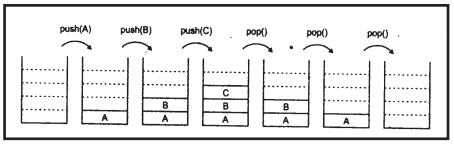

*Obrázok č. 3 Dvojslovo*

**Pamäť** (vo význame operačná pamäť počítača) je v zjednodušenom pohľade postupnosť (presnejšie vektor) bajtov, pozri obr. 4. Každý z týchto bajtov predstavuje jednu pamäťovú bunku a má svoju jedinečnú adresu (jedinečnú z hľadiska hardvéru, nie nevyhnutne z hľadiska programátora!). Program bežiaci na počítači je umiestnený niekde v pamäti (pre jednoduchosť teraz zabudnime na to, že existuje niečo ako virtuálna pamäť) – nejakú oblasť zaberá kód programu, nejakú zase dáta, s ktorými pracuje. Tu by som ešte objasnil pojem premenná v programátorskom zmysle – premenná je určitá oblasť pamäte, ktorá je programátorovi nejakým spôsobom prístupná, obyčajne svojím menom. Adresa premennej je adresou začiatku tejto oblasti, dĺžka oblasti závisí od typu premennej.




*Obrázok č. 4 Pamäť*

Nasleduje stručný a maximálne zjednodušený opis veľmi často používanej údajovej štruktúry – ide o **zásobník**. Zásobník je jedna z kľúčových vecí, na ktorých je založený celý beh programu v C++ (minimálne na platforme Intel), i keď programátor s ním prakticky vo väčšine prípadov vôbec neprichádza do styku. Jazyk C++ je však veľmi blízky programovaniu v jazyku symbolických inštrukcií (na vysvetlenie – assembler nie je jazyk, ale program na preklad z jazyka symbolických inštrukcií do strojového kódu daného procesora), v ktorom programátor zásobník na 99 % používať musí – už len preto, že ho používa procesor. Takže zásobník je údajová štruktúra, nazývaná aj LIFO, z anglického *Last-In-First-Out* (posledný dnu, prvý von). Jeho špecifickou vlastnosťou je, že údaje z neho nemožno vyberať ľubovoľne, ale len po jednom a vždy len ten údaj, ktorý bol do zásobníka vložený posledný (hovoríme, že je na vrchole zásobníka). Nad zásobníkom sú definované dve základné operácie – *push()*, ktorá vloží údaj do zásobníka, a *pop()*, ktorá údaj vyberie. Na objasnenie – po vykonaní série operácií *push(A); push(B); push(C)* ďalšou sériou *pop(); pop(); pop()* vyberieme údaje v poradí *C, B, A* (ak to nie je úplne jasné, pozri obr. 5). Konkrétnu realizáciu a používanie zásobníka vo vzťahu k C++ vysvetlím na príslušných miestach.




*Obrázok č. 5 Zásobník*

Väčšina operačných systémov používa pojmy **štandardný vstup a štandardný výstup**. Štandardný vstup (tiež *stdin*) je určité rozhranie, ktoré má program automaticky k dispozícii na vstup údajov vždy po svojom spustení. Implicitne ide o vstup z klávesnice. Štandardný výstup (tiež *stdout*) naproti tomu umožňuje programu jednoduchý výstup údajov, implicitne ide o výstup na obrazovku. K týmto dvom pojmom môžeme pridať ešte štandardný výstup pre chybové hlásenia (tiež *stderr*), ktorý smeruje implicitne takisto na obrazovku. Podotýkam, že program nemusí tieto rozhrania používať – ide o jedno z najzákladnejších spojení programu s operačným systémom (na úrovni výmeny údajov) a používa sa na jednoduchý vstup/výstup (spomeňte si napríklad na príkaz *time* v MS-DOS – čas zadaný z klávesnice si program prečíta zo štandardného vstupu a text, ktorý vidíte na obrazovke, posiela na štandardný výstup).

## Z histórie

Základné pojmy sme si objasnili, nasleduje okienko do histórie a príčin vzniku C++.

Jazyk C++ nevznikol „na zelenej lúke“. V sedemdesiatych rokoch (nášho storočia, samozrejme) bol vyvinutý pre potreby vývoja operačného systému UNIX jazyk nazvaný jednoducho C. Mnohým sú známe mená jeho autorov, alebo skôr duchovných otcov – boli to páni Brian Kernighan a Dennis Ritchie. Jazyk C je mimoriadne vhodný na písanie niečoho takého fundamentálneho, ako je operačný systém, pre jeho silnú väzbu na hardvér a mocné schopnosti pri zachovaní efektívnosti binárneho

kódu. Ak sa dobre pamätám, 90 % kódu Unixu bolo napísaných práve v jazyku C (zvyšok boli rôzne low-level procedúry, ako ovládače zariadení a pod., písané v jazyku symbolických inštrukcií). Dnes, keď svetom letí Java, sa môže zdať táto závislosť od technických prostriedkov skôr brzdou, ale existuje nemálo postupov a pravidiel, ako urobiť program portabilným (t. j. preložiteľným na viacerých platformách).

Jazyk C sa čoskoro natoľko rozšíril medzi profesionálnych i amatérskych programátorov, že jeho opis bol niekoľkokrát kodifikovaný normou ANSI (posledná kodifikácia – spoločná s normou ISO – je z roku 1990), v neposlednom rade kvôli zjednoteniu rôznych štandardov od rôznych firiem.

V priebehu osemdesiatych rokov v súvislosti s rozšírením princípov objektovoorientovaného programovania (OOP) Bjarne Stroustrup navrhol jazyk, ktorý bol založený na jazyku C, vylepšoval niektoré jeho črty a zaviedol mnohé nové, hlavne však umožňoval uplatniť spomínané princípy OOP. Stroustrup nazval svoj jazyk C++ a v súčasnosti by mal byť takisto kodifikovaný normou ANSI/ISO.

V našom seriáli sa budeme najprv zaoberať neobjektovými črtami jazyka C++ (tých je viac než dosť) a až po ich zvládnutí prejdeme k prostriedkom pre OOP.

### Prvý program

Ak ste sa prehrýzli obsiahlym teoretickým úvodom až sem, zrejme ste už netrpezliví a radi by ste čím skôr programovali. Možno vás trochu sklamem, ale mám pre vás len jeden veľmi jednoduchý program, ktorý je tak hlboko zakorenený v tradíciách výučby programovania (a v C/C++ obzvlášť), že si netrúfam ho vynechať. Jediným jeho cieľom je výpis pozdravu „Hello, world!“ na obrazovku (v skutočnosti sa zapisuje na štandardný výstup):

```
#include <stdio.h>

int main()
{
    printf(„Hello, world!\n“);
    return 0;
}
```

Tento program treba prepísať do samostatného súboru podľa zaužívaných konvencií s príponou .CPP a následne preložiť príslušným prekladačom a spustiť. To je možné buď priamo z integrovaného prostredia, alebo spustením prekladača z príkazového riadka, čo je však komplikovanejšie, už len z dôvodu pamätania si desiatok rôznych prepínačov. Po preklade dostaneme súbor s príponou .EXE, ktorý stačí spustiť a na obrazovke sa objaví pozdrav. Možno vás prekvapí veľkosť vytvoreného súboru (závisí od prekladača a jeho nastavenia, u mňa napríklad asi 9 KB), ale všetko má svoje opodstatnenie a časom sa dostaneme k vysvetleniu.

## Druhá časť: TYPY SÚBOROV

Uviedol som vám krátky program, predstavujúci zvyčajný úvod do programovania v jazyku C. Azda sa na mňa nikto nenahnevá, že som začal práve ním, hoci náš seriál sa venuje C++. Myslím si totiž, že je úplne zbytočné „vybafnúť“ na vás rovnaký program, ktorý by však naplno využíval črty jazyka C++. Musel by som vám aspoň jemne naznačiť, o čo v ňom ide, a to by som predbiehal výklad.

Mimochodom, pevne dúfam, že sa vám podarilo program úspešne preložiť do spustiteľnej podoby a následne aj spustiť. Žiaľ, nemôžem predvídať všetky problémy, ktoré by potenciálne u vás mohli nastať. V tejto veci vás musím opakovane odkázať na manuály, elektronické príručky a služobne starších programátorov (vo vlastnom záujme v uvedenom poradí, nie každý programátor bude ochotný na vaše – z jeho pohľadu možno otravné otázky – odpovedať).

Takže čo nás teraz čaká? Ako som naznačil minule, budeme sa zaoberať predovšetkým filozofiou vývoja programu v C++. Neľakajte sa, nepôjde o výťah z učebnice softvérového inžinierstva, mám na mysli špecifiká tvorby programu v C++. Opíšeme si, aké rôzne typy súborov sa počas celého procesu môžu (ale aj nemusia) vyskytnúť, a načrtneme celý proces, tak ako obyčajne prebieha.

### Typy súborov

Ako v každom inom „vyššom“ programovacom jazyku je aj program v C++ napísaný v človekom čitateľnej (nechcem napísať, že práve najzrozumiteľnejšej) forme, to znamená prakticky vždy v ASCII kóde. To, čo programátor vymyslí a prostredníctvom klávesnice vloží do počítača, sa nazýva *zdrojový text* (aj zdrojový kód – source code).

### Zdrojové súbory

Zdrojový text programu je uložený v obyčajnom textovom súbore (nijaké formátovacie znaky a podobné smeti), ktorý budeme kvôli jednoznačnosti ďalej nazývať *zdrojový súbor*. Jeho prípona je pre jazyk C (prekvapujúco) `.C`, pre jazyk C++ je to trošku zložitejšie. V operačných systémoch ako DOS (nie, to nemyslím vážne, DOS nie je operačný systém), Windows a iné (prevažne PC platforma) majú C++ súbory príponu `.CPP`. Naproti tomu v unixových systémoch sa takmer výhradne dáva prednosť prípone `.CC` (vlastne `.cc`, UNIX je case-sensitive). Predpokladám, že väčšina z vás má prístup k bežnému PC, a preto budem ďalej používať príponu `.CPP`.

Nikde nie je povedané, že program musí byť celý uložený v jedinom zdrojovom súbore. Naopak, z dôvodu intelektuálnej zvládnuteľnosti (a z úcty programátora k svojmu zdravému rozumu) sa väčšie programy rozkladajú do viacerých zdrojových súborov. Ďalším podstatným dôvodom je fakt, že v prípade zmien v programe je efektívnejší preklad menšieho súboru (zahŕňajúceho zmenu) ako neustále prekladanie jedného veľkého a neprehľadného programového súboru.

CPP súbory však nie sú jediné, v ktorých môže byť uložený zdrojový text programov. Ďalším – pre jazyky C a C++ typickým – typom súborov sú tzv. *hlavičkové súbory* (headerové súbory – header files) s príponou `.H` alebo v niektorých C++ prekladačoch aj `.HPP`. Tieto súbory môžu byť (a aj bývajú) pomocou zvláštnej direktívy `#include` *vložené* do zdrojových súborov, čo má pri preklade rovnaký dôsledok, akoby ste obsah hlavičkového súboru skopírovali miesto spomenutej direktívy. Výhoda je okamžite zrejmá v prípade, že máme jeden hlavičkový súbor vložený do viacerých zdrojových a opravíme v ňom napríklad nejakú chybu – zmena sa automaticky premietne do všetkých tých súborov, v ktorých je daný header vložený. V opačnom prípade by sme museli prácne rad za radom prechádzať jednotlivé zdrojové súbory a hľadať, kde všade sme napísali ten blud. V praxi sa do hlavičkových súborov vkladajú väčšinou informácie o tom, ktoré funkcie v programe existujú (tzv. prototypy funkcií – dostaneme sa k tomu neskôr), aké spoločné dáta tieto funkcie používajú a pod. Pre tých, čo poznajú Pascal – vloženie hlavičkového súboru v C++ má v hrubých črtoch rovnaký efekt ako použitie klauzuly `uses` v Pascale – umožňuje však oveľa viac.

Tretím typom súborov, ktorý môže obsahovať zdrojový text, ale ktorý sa nepoužíva tak často, sú súbory, obsahujúce kód písaný priamo v jazyku symbolických inštrukcií (asi pochopiteľnejší názov bude assemblerové súbory). Jazyk C++ je síce mocný, ale nie všemocný a v prípade, že potrebujete napísať superrýchlu funkciu trebárs na vykresľovanie tieňovaných polygónov, rýchlo zistíte, že najvhodnejším riešením bude asi použiť rovno strojový kód.

### Medzisúbory

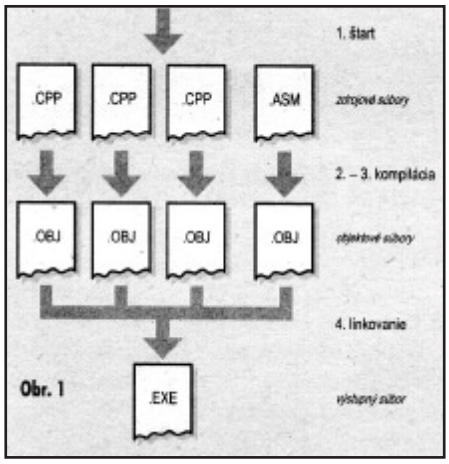
Vopred sa priznávam, že použitý názov *medzisúbory* (pokus o voľný výklad anglického termínu „intermediate files“) je mojím výmyslom, ale myslím, že dostatočne vystihuje postavenie tohto typu v celkovej hierarchii. Medzisúbory sú akýmsi medziproduktom celého procesu prekladu a pri vývoji programu jedným programátorom od začiatku až do konca si vôbec netreba trápiť hlavu ich existenciou.

Prvým typom medzisúboru je *objektový súbor* (object file). Na PC obyčajne s príponou `.OBJ`, v unixe s príponou `.o`. Tento súbor vznikne prekladom *jedného* zdrojového súboru (a ním zahrnutých hlavičkových) a obsahuje strojový kód prekladu toho, čo programátor do zdrojového súboru zapísal (a občas aj toho, čo tam nezapísal). Okrem toho obsahuje informácie o tom, čo sa v danom OBJ súbore nachádza (a môže byť použité inými časťami programu), ale aj to, čo sa v ňom nenachádza, no je potrebné na správnu funkciu kódu v tomto súbore (tzv. externé odkazy). OBJ súbor *nie je spustiteľný*, a to z jednoduchého dôvodu: formát OBJ súboru je iný ako formát EXE súbor. To je vážne najjednoduchší dôvod. Samozrejme, dá sa to vysvetliť aj komplikovanejšie – samostatný OBJ súbor nie je sebestačný, i keď celý program pozostáva z jediného zdrojového súboru. Treba k nemu totiž pribaliť ešte úvodný inicializačný kód, ale na túto tému je trochu priskoro, takže hádam niekedy v budúcnosti.

Druhým typom medzisúboru je tzv. *knižničný súbor* (knižnica – library file). Tento súbor (na PC s príponou `.LIB`, v unixe napríklad `.a`) nie je nič iné ako „zlepenec“ viacerých objektových súborov s definovaným formátom. Aby som bol úprimný, pri bežnom preklade programu sa tento typ vôbec nevyskytuje. No predstavme si situáciu, keď programátor napíše zbierku funkcií, odladí ju a chce ju použiť neskôr v inom projekte, prípadne ju chce poskytnúť svojmu okoliu (programátorskému, samozrejme – asi by nemalo význam ponúkať svojej priateľke či manželke zbierku obľúbených funkcií). Takže čo spraví: preloží potrebné zdrojové súbory do objektových a špeciálnym programom z týchto objektových súborov vytvorí jednu knižnicu (alebo aj viac). Tú následne dôkladne zdokumentuje, pribalí k nej hlavičkové súbory (bez nich je knižnica dosť ťažko použiteľná) a dá na svoju webovú stránku (napríklad).

### Výstupné súbory

Množné číslo v nadpise je trochu nadnesené, konečným cieľom celého úsilia sa býva obyčajne jeden súbor – výsledný program. Pravda, to „obyčajne“ platí pre platformu PC, kde existujúce prekladače produkujú takmer výhradne jediný výstupný súbor – klasický EXE súbor, rôzne hybridy typu „overlay“ alebo z prostredia Windows známu DLL knižnicu. V unixe (vďaka faktu, že prakticky všetky prekladače sú ovládané prepínačmi z príkazového riadka a rozumne sa dá pracovať len s pomocou súborov make) je možné jediným príkazom vyprodukovať aj viacero spustiteľných súborov (presnejšie, je možné vyprodukovať ľubovoľné množstvo ľubovoľných súborov, ale o tom sa tu baviť nebudeme). Pre naše ciele bude



Obr. 1

azda dostatočne vhodný model vývoja programu, ktorý vytvorí jediný spustiteľný súbor.

### Proces prekladu

Teraz, keď už vieme, s akými súbormi máme pri preklade do činenia, môžeme si načrtnúť, ako samotný preklad vyzerá. Tu sú jednotlivé kroky (pozri obr. 1):

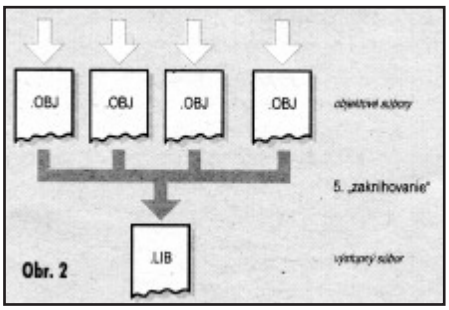
1. Máme vytvorené jednotlivé zdrojové súbory, príslušné hlavičkové súbory a rozhodneme sa spustiť preklad.

2. Prvým z nástrojov je tzv. preprocesor. Tento program má viacero funkcií, bližšie si o ňom povieme v príslušných častiach nášho seriálu. Zatiaľ vám poviem aspoň toľko, že práve on, má na starosti náhradu direktívy vloženia hlavičkového súboru jeho obsahom. Preprocesor môže existovať ako samostatný program, ale väčšinou býva súčasťou nasledujúceho nástroja.

3. Ďalším nástrojom je *kompilátor*. Často sa označuje aj názvom prekladač, to nám však trochu koliduje s pojmom „preklad“ ako proces, ktorý transformuje zdrojové súbory na výstupný súbor. Kompilátor zabezpečuje len jednu fázu tohto prekladu. Kompilátory pre jednotlivé jazyky sa obyčajne líšia, ich výstupom je však zhodne množina objektových súborov. Kompilátor môže byť volaný pre každý zdrojový súbor osobitne alebo dostane ako argument množinu zdrojových súborov, ktoré spracúva po jednom.

4. Dostali sme objektové súbory, teraz ich treba nejakým spôsobom spojiť. Na to slúži tretí nástroj, tzv. *linker* (pre jazykových puristov – spojovač). Proces linkovania spočíva jednak v zisťovaní, či každá použitá funkcia či premenná v niektorom objektovom súbore existuje, jednak vo vhodnom pospájaní jednotlivých modulov, pripojení inicializačného kódu a vytvorení daného výstupného súboru.

5. V prípade, že nechceme vyprodukovať výstupný súbor, ale knižnicu, miesto linkera príde na rad „knihovník“ (ospravedlňujem sa, nenapadá ma vhodnejší preklad pre „librarian“) (pozri obr. 2). Tento knihovník má v podstate podobnú úlohu ako linker, umožní však do knižnice uložiť aj také objektové súbory, ktorým ku šťastiu niečo chýba.



Obr. 2

Počas každej fázy prekladu (okrem tej prvej, samozrejme) môže dôjsť, a podľa rôznych programátorských zákonov celkom určite príde, k rôznym chybám. Stručne teraz načrtnem možné okruhy chýb. Medzi chyby preprocesora patrí najčastejšie nemožnosť nájsť príslušný hlavičkový súbor – treba si dať pozor na nastavenie ciest. Chyby kompilátora môžeme rozdeliť na syntaktické (to keď napíšete niečo, čomu kompilátor nerozumie) a sémantické (tie vzniknú vtedy, keď kompilátor síce rozumie tomu, čo ste napísali, rozhodne s tým však nesúhlasí). Linker vás môže tiež zavaliť kôpkou chybových hlásení, z ktorých najčastejším je použitie funkcie, ktorá nikde v programe nie je definovaná.

### Štruktúra programu

Prejdeme teraz konečne k tomu, ako taký program v C++ vyzerá a z čoho sa skladá. V prvom rade: jazyk C++ nepozná procedúry, ale iba funkcie. *Funkcia* je úsek programu s definovaným začiatkom, koncom, počtom a typmi argumentov a typom návratovej hodnoty. Funkciu možno z iného miesta programu zavolať, odovzdať jej prípadné argumenty, tým sa jej odovzdá riadenie a po úspešnom ukončení funkcia tomu, kto ju zavolal, môže vrátiť návratovú hodnotu.

Program v C++ v najjednoduchšom priblížení nie je vlastne nič iné ako množina funkcií, ktoré sa navzájom volajú. Pre tých, ktorým sú známe princípy objektovo-orientovaného programovania – aj striktne objektový program je množina funkcií, hoci tieto funkcie sú členmi príslušných tried. Vzhľadom na to, že začíname štúdiom neobjektových čŕt C++, postačí nám takýto pohľad.

Ak sa zamyslíme nad uvedeným modelom programu, rýchlo dospejeme k záveru, že treba určiť, ktorá funkcia dostane slovo ako prvá. V jazyku C++ má toto špecifické postavenie funkcia s názvom `main()`. Čo sa týka jej argumentov, možno jej odovzdať to, čo za názov programu napíšete na príkazovom riadku, ale k tomu sa ešte dostaneme. Jej návratovou hodnotou môže byť klasický výstupný kód (exit-code; testovateľný napr. v dávkových súboroch DOS-u pomocou premennej `ERRORLEVEL`). Keď program dospeje ku koncu funkcie `main()`, skončí sa (samozrejme, pomocou špeciálnych príkazov môže skončiť aj skôr).

Všeobecný (ale zatiaľ zjednodušený) tvar zápisu funkcie vyzerá takto:

```
fnc-typ fnc(arg-typ1 arg1, ... , arg-typn argn)
{
        ... kód funkcie ...
}
```

kde `fnc-typ` je typ návratovej hodnoty, `fnc` je meno funkcie, $\texttt{arg-typ}_i$ je typ argumentu$_i$ a $\texttt{arg}_i$ jeho meno.

Ľavá krútená zátvorka označuje začiatok kódu funkcie (ekvivalent pascalovského `begin`), pravá značí koniec kódu (ekvivalent `end`) a *nepíše* sa za ňou bodkočiarka. Medzi týmito dvoma zátvorkami je samotný kód, ktorý funkcia bude vykonávať. Ako si ukážeme v budúcich častiach, tu sa nachádzajú deklarácie premenných, volania iných funkcií, rôzne príkazy a pod.

Jednoduchý, jednosúborový program v C++ obyčajne vyzerá tak, že na začiatku zdrojového súboru sa uvedú hlavičkové súbory, ktoré treba vložiť (pomocou `#include`), nasledujú definície jednotlivých funkcií a jedna z týchto funkcií má názov `main()`. Tej sa pri štarte programu odovzdá riadenie a za normálnych okolností jej ukončením sa skončí celý program.

Máte teda predstavu o tom, ako vyzerá hrubá (ale vážne veľmi hrubá!) štruktúra programu v C++. Vzhľadom na to, že vaše doterajšie vedomosti (prosím znalých, aby sa neurazili) nestačia na vytvorenie vlastného programu, opäť mám pre vás na záver jednoduchý program, na základe ktorého by ste mali pochopiť spomínaný model behu programu. Uvedený program sa skladá z troch funkcií, ktoré sa navzájom volajú. Každá z týchto funkcií okrem toho vypíše nejaký text. Na vysvetlenie – `printf(„text“);` je volanie funkcie (definovanej však mimo nášho programu), ktoré spôsobí výstup daného textu na štandardný výstup. Aby sme túto funkciu mohli používať, treba na začiatku programu vložiť hlavičkový súbor `stdio.h`. Našu vlastnú funkciu môžeme zavolať tak, že uvedieme jej názov, do zátvorky prípadné argumenty (v našom prípade žiadne) a ukončíme bodkočiarkou.

```
#include <stdio.h>

void A()
{
        printf(„Hello from function A().“);
}

void B()
{
        printf(„Hello from function B().“);
        A();
}

void main()
{
        printf(„Hello form function main()“);
        B();
}
```

Ak si spomínate na predošlý program, tento obsahoval direktívu `#include`, definíciu funkcie `main()` a volanie funkcie `printf()` s argumentom „`Hello, world!`“. Na základe doterajšieho výkladu by vám nemalo robiť problém pochopenie jeho činnosti.

Dovolím si vám dať niečo ako domácu úlohu: vyskúšajte si dnešný program, pokúste sa ho pochopiť a doplniť doň ďalšie funkcie (vypisovať môžu hocičo). Pozor! Volať môžete (zatiaľ) len funkciu, ktorá bola definovaná *pred vaším* volaním (t. j. v našom príklade funkcia `B()` môže volať funkciu `A()`, ale funkcia `A()` nemôže volať funkciu `B()`).

## Tretia časť: LEXIKÁLNE JEDNOTKY

Doteraz sme si v skratke ukázali, ako taký jednoduchý program C++ vyzerá. Berte prosím tieto informácie ako prostriedok, o ktorý sa budeme neskôr opierať, aby ste vôbec boli schopní preberané konštrukcie v nejakom programe vyskúšať. V tejto a v nasledujúcich častiach preberieme (nie nevyhnutne v uvedenom poradí) lexikálne jednotky jazyka, typy premenných, konverzie medzi nimi, výrazy a operátory, príkazy, deklarácie a definície premenných a funkcií a iné. Potom si niečo povieme o štandardnej knižnici funkcií – čo je množina funkcií, ktoré už niekto za vás naprogramoval a dal vám k dispozícii spolu s prekladačom. Celá táto etapa seriálu bude pripomínať viac kurz jazyka C, ako som však už spomenul, budem tu vysvetľovať všetko okrem objektových čŕt C++. Po krátkej úvahe som sa rozhodol nevysvetľovať rozdiely medzi C a C++, myslím, že by vás to zbytočne plietlo, aj tak vás čaká informácií viac než dosť. V ukážkových programoch (skôr fragmentoch programov – maximálne zameraných na danú látku) budem používať funkcie štandardnej knižnice jazyka C (ak vôbec).

Napokon by som vás chcel ešte upozorniť na 32-bitový prekladač C/C++ s názvom DJGPP, na ktorý pre zmenu mňa upozornil jeden z čitateľov. Tento prekladač je šírený ako freeware a je dostupný na adrese http://www.delorie.com spolu s množstvom rôznych

utilít, integrovaným prostredím a rôznymi knižnicami. Rozhodne je to zaujímavá a úplne funkčná alternatíva ku komerčným prekladačom, a ak sa mi podarí nájsť si chvíľu času na bližšie oboznámenie sa s týmto produktom, sľubujem, že budem o jeho existencii uvažovať v ukážkových programoch.

## Lexikálne jednotky

Najprv by sme si mali hádam vysvetliť, čo vlastne je lexikálna jednotka. *Lexikálna jednotka* je základný stavebný prvok jazyka, a to nielen programovacieho, ale i bežného, ktorým hovoríme alebo píšeme (v tomto prípade je lexikálnou jednotkou slovo). Text v danom jazyku (pre programovací jazyk je textom program) je postupnosťou lexikálnych jednotiek (ktorá spĺňa určité pravidlá a pod., to nás teraz nezajíma).

Jazyk C++ pozná 5 typov lexikálnych jednotiek (tokens): *identifikátory, kľúčové slová, literály* (konštanty), *operátory* a ostatné *oddeľovače*. Lexikálne jednotky v programe nemusia nasledovať tesne jedna za druhou (to by sa asi dosť ťažko čítalo), ale môžu byť oddelené tzv. *bielymi znakmi* (white space – tento názov vznikol z tlačenej reprezentácie bieleho znaku – na papieri sa nevytlačí nič, zostane len biela medzera). Bielymi znakmi v C++ sú medzera (space, kód ASCII = 32), horizontálny tabulátor (HT, 9), vertikálny tabulátor (VT, 11), znak návratu vozíka (carriage return – CR, 13), znak prechodu na nový riadok (new line – LF, 10), znak prechodu na novú stránku (form feed – FF, 12) a nakoniec tzv. *komentár*.

Komentár je vysvetľujúci text, ktorý napíšeme do zdrojového súboru napríklad na objasnenie nejakého úseku kódu, na opis argumentov funkcií a pod. Prekladač komentáre ignoruje. Komentáre sú určené jednak pre druhých programátorov, ktorí budú napríklad upravovať náš program, aby pochopili, o čo v ňom ide a ako sa to realizuje, a jednak pre nás, aby sme sa v programe vyznali aj po pol roku. Vo vlastnom záujme odporúčam komentovať všetko, čo by mohlo byť zdrojom nejasností.

Jazyk C++ rozlišuje medzi dvoma typmi komentárov. Prvý sa začína dvojicou znakov /*, po ktorej môžeme napísať náš komentárový text a celý blok ukončíme dvojicou znakov */. Medzi oboma dvojicami môžu byť ľubovoľné znaky (aj prechod na nový riadok), ktoré sú kompletne ignorované.

Druhý typ komentára sa začína dvojicou znakov // a končí sa znakom prechodu na nový riadok (teda všetko od // po koniec riadka je komentár). Ignorujú sa všetky ďalšie znaky na tom istom riadku.

**Príklad:**

```
/*
 * funkcia main()
 */
int main()
{
    ...
}

alebo

int main()
{
    // telo funkcie main()
}
```

Kurzívou zvýraznené časti sú považované za komentáre (editor integrovaného prostredia každého lepšieho prekladača ich dokáže odlíšiť farebne).

Prvý typ komentárov nemožno vnárať do seba (ako napr. /* /* vnútorný */ vonkajší */), pretože po prvej dvojici /* sú nasledujúce znaky ignorované až po prvý výskyt */, po ktorom však v našom príklade nasleduje ešte dokončenie vonkajšieho komentára.

## Identifikátory

Prvým typom lexikálnej jednotky sú identifikátory. Slúžia na pomenovanie rôznych prvkov programu, ako premenné, funkcie, návestia, a pod. Pravidlo na vytváranie identifikátorov je jednoduché – identifikátorom môže byť ľubovoľne dlhá postupnosť písmen a číslic, v ktorej prvým znakom však musí byť písmeno. Za písmeno sa považuje aj znak _ (podčiarknutie – underscore). Platnými identifikátormi teda sú napríklad x, _open, r1998, ale nie už napr. 2pi. V názvoch identifikátorov sa rozlišujú veľké a malé písmená, teda napr. aa, aA, Aa, AA sú štyri rôzne identifikátory.

## Kľúčové slová

Kľúčové slová sú také reťazce znakov, ktoré sú určitým spôsobom vyhradené na použitie samotným jazykom a nesmú sa používať nijakým iným spôsobom. Ich zoznam nájdete v nápovednom systéme každého prekladača. Na ilustráciu slovo return, ktoré sme použili v prvom programe, je kľúčovým slovom a slúži na ukončenie funkcie a odovzdanie návratovej hodnoty. Jednotlivé kľúčové slová budeme preberať postupne pri príslušných témach.

## Operátory a oddeľovače

Operátory sú špeciálne znaky, pomocou ktorých sa vyjadrujú určité operácie nad premennými a/alebo konštantami. Klasickým príkladom je operátor +, ktorého funkciou je sčítanie svojich argumentov. Ostatné oddeľovače sa používajú v rôznych špecifických prípadoch. Jeden príklad za všetky: jednotlivé príkazy sa v C++ oddeľujú znakom ; (bodkočiarka), obyčajne písaným za príkazom na konci riadka. Jednotlivé operátory si preberieme v časti venovanej výrazom, ostatné oddeľovače a ich funkcia vyplynú z kontextu.

*Tabuľka 1*

| Escape sekvencia | ASCII znak | ASCII kód | Opis |
|---|---|---|---|
| \a | BEL | 7 | pípnutie |
| \b | BS | 8 | zmazanie posledného znaku (backspace) |
| \f | FF | 12 | nová stránka (form-feed) |
| \n | LF | 10 | nový riadok (line-feed) |
| \r | CR | 13 | návrat vozíka (carriage return) |
| \t | HT | 9 | horizontálny tabulátor |
| \v | VT | 11 | vertikálny tabulátor |
| \? | ? | 63 | otáznik |
| \‘ | , | 39 | apostrof |
| \“ | „ | 34 | úvodzovky |
| \\ | \ | 92 | opačné lomítko (backslash) |
| \ooo | #ooo | ooo | znak s kódom ooo |
| \xhhh | #hhh | hhh | znak s kódom hhh |

## Literály

Slovník cudzích slov definuje pojem literál ako „konštantu, definovanú priamo svojou hodnotou“. To v preklade znamená, že napr. znak ? nie je literálom, ale reťazec 3.14159 áno. V ďalšom budem používať zrozumiteľnejší termín konštanta namiesto literál.

V jazyku C++ máme štyri typy konštánt: celočíselné, s pohyblivou rádovou čiarkou, znakové a reťazcové.

**Celočíselná** (integer) konštanta je postupnosť číslic. Ak sa táto postupnosť začína inou číslicou ako 0 a ďalej obsahuje číslice 0 ÷ 9, ide o desiatkovú konštantu (v desiatkovej sústave). Ak sa začína číslicou 0 a ďalej obsahuje číslice 0 ÷ 7, ide o oktálovú konštantu (v osmičkovej sústave). A nakoniec, ak sa začína dvojicou 0x alebo 0X a ďalej obsahuje číslice 0 ÷ 9 a znaky A ÷ F, a ÷ f, ide o hexadecimálnu konštantu (v šestnástkovej sústave). Všimnite si, že súčasťou konštanty nie je prípadné znamienko - (mínus)! Typ celočíselnej konštanty závisí od jej veľkosti, ale keďže typy C++ sme ešte nepreberali, vrátime sa k tejto téme neskôr.

**Príklad:** Vyjadríme číslo 3894 vo všetkých troch sústavách a zapíšeme príslušné konštanty.

desiatková – 3894 = $(3894)_{10}$ → 3894
osmičková – 3894 = $(7466)_8$ → 07466
šestnástková – 3894 = $(F36)_{16}$ → 0xF36, 0XF36, 0xf36, 0Xf36

Konštanta s **pohyblivou rádovou čiarkou** (floating) sa skladá z celočíselnej časti, desatinnej bodky, desatinnej časti, znaku e alebo E a celočíselného exponentu s prípadným znamienkom. Ide, samozrejme, o zápis v tzv. vedeckej notácii, kde časť pred znakom E sa nazýva mantisa a časť za znakom E je exponent. Hodnota takto vyjadrenej konštanty je (mantisa × $10^{exponent}$). Celočíselná aj desatinná časť sú postupnosťami desiatkových číslic. Príkladom konštanty obsahujúcej všetky spomenuté polia je napr. 2.99E+8, čo je 2,99 × $10^8$ (a aj rýchlosť svetla vo vákuu). Buď celočíselnú, alebo desatinnú časť (ale nie obe!) môžeme vynechať, t. j. .34E-5 aj 6.E+7 sú platné konštanty s pohyblivou rádovou čiarkou, ale .E+4 už nie. Ďalej buď desatinnú bodku alebo písmeno e/E s exponentom (ale opäť nie oboje) môžeme vynechať, t. j. 5E-12 aj 3.14 sú platné konštanty s pohyblivou rádovou čiarkou, ale 765 už nie (to je totiž celočíselná konštanta). Znamienko pri exponente je povinné iba vtedy, ak je záporné (tj. miesto 2.99E+8 môžeme písať 2.99E8).

Znakovú (character) konštantu reprezentuje znak uzavretý v apostrofoch (‘) (v skutočnosti v apostrofoch môže byť aj viac znakov, vtedy ide o tzv. „širokoznakovú“ [wide-character] konštantu, ale tú teraz nebudeme brať do úvahy, azda snáď niekedy v budúcnosti). Príklady znakových konštánt sú ‘8’, ‘D’, ‘#’ a pod. Navyše sú v C++ definované tzv. escape sekvencie, ktoré slúžia na vyjadrenie inak nezapísateľných znakov. Tieto sekvencie sa začínajú znakom \ (backslash), ich zoznam je v tabuľke 1. Pomocou sekvencií \ooo a \x*hhh* môžeme zapísať ľubovoľný znak pomocou jeho hodnoty v kóde ASCII. Sekvencia \ooo opisuje znak pomocou *oktálovej* konštanty ooo (max. 3 číslice) nasledujúcej za znakom \. Sekvencia \x*hhh* opisuje znak pomocou hexadecimálnej konštanty *hhh*. Celá sekvencia reprezentuje jeden znak. Ako príklad si uvedieme zápis znaku S [kód ASCII = $(83)_{10}$ = $(123)_8$ = $(53)_{16}$] vo všetkých tvaroch:

```
‘S’ = ‘\123’ = ‘\x53’
```

V oboch typoch je sekvencia ukončená prvým neoktálovým, resp. nehexadecimálnym znakom.

**Reťazcová** konštanta je postupnosť znakov (resp. znakových konštánt bez apostrofov – možno používať aj escape sekvencie), uzavretá do úvodzoviek (“). Ak chceme v reťazci použiť úvodzovky ako znak, musíme pred ne napísať znak \. Ako si neskôr povieme, jazyk C++ nemá nijaký typ ‚reťazec znakov‘ a pre používanie reťazcových konštánt platia určité pravidlá – viac sa dozviete v časti venovanej typom C++. V pamäti je reťazcová konštanta uložená ako postupnosť znakov (lepšie povedané, bajtov s hodnotami reprezentujúcimi kódy ASCII jednotlivých znakov), ktorá je *ukončená* znakom ‘\0’ (NUL), t. j. napríklad konštanta “Hello” je v pamäti uložená ako postupnosť znakov ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’. Tento fakt je veľmi

dôležitý a potrebný na porozumenie práce s reťazcami. Pre úplnosť ešte dodám, že ak sa v programe vyskytujú za sebou dve reťazcové konštanty, oddelené iba bielymi znakmi, prekladač ich spojí do jedinej, nespája však možné escape sekvencie. Teda napríklad zápis `"\xA" "B"` predstavuje reťazec dvoch znakov `'\xA'`, `'B'`, a nie jeden znak `'\xAB'`.

## Štvrtá časť: TYPY JAZYKA C++

Toto pokračovanie seriálu začnem trochu retrospektívne, niekoľkými záležitosťami okolo druhej časti seriálu – výrobná lehota PC REVUE mi, žiaľ, nedovoľuje reagovať na predchádzajúce časti skôr ako o dva mesiace. Takže v prvom rade sa ospravedlňujem za menšie chyby v ukážkovom programe. Niežeby sa nedal preložiť, ale, ako ste hádam aj sami zistili, po spustení vypísal všetky texty do jedného riadka. Po prečítaní predošlej časti by ste mali byť schopní urobiť patričnú úpravu sami, takže len pre istotu dodávam, že na koniec každého vypisovaného textu patrí (tesne pred koncové úvodzovky) escape sekvencia \n, ktorá spôsobí chýbajúci prechod na nový riadok. Ďalšou „chybou" je preklep – vo funkcii main() má byť vo vypisovanom reťazci namiesto form slovo from (a na konci reťazca chýba bodka – ale to už len pre vyslovených punktičkárov, ako som napríklad ja…).

### Opakovanie

V predošlej časti sme prebrali lexikálne jednotky jazyka C++. Verím, že nešlo o práve najzrozumiteľnejšie čítanie, ale máme to aspoň za sebou. Povedali sme si, že program v C++ je postupnosťou lexikálnych jednotiek, oddelených bielymi znakmi. Dve lexikálne jednotky musia byť oddelené jedným alebo viacerými bielymi znakmi vtedy, ak by ich nebolo možné inak rozlíšiť. Uvediem príklad: majme postupnosť znakov `a--b`. O tejto postupnosti nedokážeme rozhodnúť, či ide o odčítanie zápornej premennej `b` od premennej `a` alebo o operáciu dekrementácie premennej `a` (`--` je operátor dekrementácie – zníženia hodnoty o 1) nasledovanú zápisom premennej `b` (hoci tento výraz nemá zmysel). Preto je potrebné do postupnosti vložiť na vhodné miesto nejaký biely znak, najčastejšie medzeru – v prvom prípade dostaneme výraz `a - -b`, v druhom `a-- b`. Toľko na ilustráciu.

Je možné, že ešte stále nemáte v tejto oblasti celkom jasno, preto si rozložíme teraz náš prvý program na postupnosť lexikálnych jednotiek. Najprv to však skúste sami a až potom si prečítajte správne riešenie. Tu je program:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

a tu je jeho dekompozícia:

1. `#include <stdio.h>` je direktíva preprocesora, *nie je* to lexikálna jednotka (nahradí sa obsahom súboru STDIO.H, ktorý je sám osebe postupnosťou lexikálnych jednotiek). O direktívach preprocesora si povieme v niektorej z budúcich častí seriálu.

2. Nasledujú dva prechody na nový riadok. Prechod na nový riadok je v zdrojovom súbore uložený ako kombinácia dvoch ASCII znakov – `CR` (ASCII kód 13) a `LF` (ASCII kód 10). Znak vyjadrený escape sekvenciou `\n` (čo je znak `LF`) sa pri výpise na obrazovku automaticky prevedie na dvojicu `CR/LF`. Tento prevod je do istej miery daný službami operačného systému a bližšie si o ňom povieme, keď budeme preberať štandardnú knižnicu jazyka C.

3. `int` je kľúčové slovo jazyka C++ a v našom programe určuje typ návratovej hodnoty funkcie `main()`. Za ním nasleduje jedna medzera – biely znak. `main` je identifikátor, ktorý predstavuje názov funkcie, dvojica znakov `()` by sa dala považovať za operátor, respektíve v tomto kontexte aj za „iný oddeľovač".

4. Prechod na nový riadok, na tomto riadku je jediný znak – oddeľovač `{`, začínajúci definíciu tela funkcie `main()`.

5. Opäť prechod na nový riadok a po ňom štyri medzery.

6. `printf` je identifikátor predstavujúci meno knižničnej funkcie. Po ňom nasleduje otváracia zátvorka, ktorá je súčasťou operátora `()`. Za zátvorkou je reťazcová konštanta – `"Hello, world!\n"`, ktorá je argumentom funkcie `printf()`. Riadok je ukončený zatváracou zátvorkou a oddeľovačom `;`, slúžiacim na oddeľovanie viacerých príkazov nasledujúcich za sebou.

7. Ďalší prechod na nový riadok a po ňom zase štyri medzery.

8. `return` je kľúčové slovo jazyka C++, po ňom nasleduje medzera a celočíselná konštanta `0`, ktorej hodnotu program vráti operačnému systému ako návratový kód. Na konci riadka je opäť oddeľovač `;`.

9. Posledný prechod na nový riadok a oddeľovač `}`, ktorý definíciu funkcie `main()` ukončuje.

Ak náhodou máte pocit, že to, čo tu teraz vysvetľujem, je zbytočné, nepochopiteľné a nepraktické, ubezpečujem vás, že všetko, čo ste sa o lexikálnych jednotkách dozvedeli, časom zužitkujete. Už viete, z akých prvkov sa program v C++ skladá, a pri pohľade na cudzí program by ste do určitej miery mali vedieť, čo je čo. Ide o podobnú situáciu, ako keď sa učíte cudzí jazyk – ak zhruba ovládate stavbu vety, viete, čo je podmet, čo prísudok a podobne, ľahšie sa zorientujete v nejakej vete, aj keď vaša slovná zásoba za veľa nestojí.

### Typy

Na začiatok niečo o tom, čo si predstavujeme pod pojmom *typy*. Keď píšeme nejaký program, väčšinou od neho očakávame, aby robil niečo užitočné. To vedie prakticky vždy k spracúvaniu nejakých údajov (ostatne, načo tie počítače vlastne máme, že?). Tieto údaje sú však rôznorodého charakteru, môžeme povedať, že majú rôzne *typy*. Program, ktorý s údajmi pracuje, musí vedieť, s akým typom údajov pracuje. Pod pojmom typ v jazyku C++ teda budeme rozumieť charakter údajov, ktoré sa v programe používajú. Azda bude lepšie namiesto o údajoch hovoriť o *objektoch* jazyka C++, hoci nejde o objekty známe z teórie objektovoorientovaného programovania, ale skôr o objekty fyzicky existujúce počas behu programu – v podstate sú to premenné a funkcie.

Typy jazyka C++ môžeme rozdeliť na dve skupiny – základné a odvodené typy.

### Základné typy

Prvým z množiny základných typov je `char`. Ako jeho názov napovedá, tento typ sa používa na reprezentáciu znakov znakovej sady daného počítača a jeho veľkosť je taká, aby bolo možné pomocou neho reprezentovať ľubovoľný znak tejto sady. Prakticky v každej implementácii C++ na PC je to 8 bitov – teda jeden bajt. Znak je v premennej typu `char` reprezentovaný svojou hodnotou v kóde ASCII, teda napríklad znak `'A'` je uložený ako hodnota `65`, a keďže kód ASCII obsahuje 256 znakov (mám na mysli 8-bitový kód ASCII), potrebujeme na ich vyjadrenie práve 8 bitov. Premennej typu `char` môžeme priradiť priamo číselnú konštantu v danom rozmedzí, dôsledok bude rovnaký ako po priradení znakovej konštanty. Nasledujúce dva zápisy sú teda ekvivalentné (predpokladáme, že c je premenná typu `char`):

```
c = 'A';
c = 65;
```

Čo je zaujímavé, premennú typu `char` vôbec nemusíme používať tak, akoby obsahovala znak, ale jednoducho ako celočíselnú premennú s rozsahom 256 hodnôt.

Typ `char` môže byť explicitne deklarovaný ako `signed char` alebo `unsigned char` (mimochodom, *signed* znamená „so znamienkom" a *unsigned* „bez znamienka"). Tieto dve modifikácie sa líšia rozsahom hodnôt, ktoré im možno priradiť (pozri tab. 1). Treba si však uvedomiť, že veľkosť oboch typov je rovnaká, t. j. 8 bitov, líši sa iba reprezentácia uložených údajov pri ich použití. Typ `unsigned char` ukladá znaky ako 8-bitové čísla bez znamienka (0 ÷ 255), typ `signed char` ich ukladá ako 8-bitové čísla so znamienkom v doplnkovom kóde (–128 ÷ 127). Znaky z dolnej polovice kódu ASCII (0 ÷ 127) sú teda prezentované pri oboch typoch rovnako, prezentácia znakov z hornej polovice sa líši – pri type `signed char` ide o rozsah –128 ÷ –1, pri type `unsigned char` o rozsah 128 ÷255.

Pre istotu uvediem ešte príklad: znak `'ô'` s kódom ASCII `147` (v kódovej stránke 852, t. j. Latin-2) je v prípade uloženia do premennej typu `unsigned char` prezentovaný (a prípadne použitý) ako číslo `147`, zatiaľ čo v prípade uloženia do premennej typu `signed char` dostaneme pri použití číslo `–109` ( =147 – 256). V pamäti je však v oboch prípadoch uložený ako binárne číslo `10010011`.

Typ `char` je vždy implementovaný ako jeden z typov `signed char` alebo `unsigned char`, ale pozor, z hľadiska prekladača sú všetky tri typy považované za navzájom *rôzne*!

Ďalším typom, ktorý v C++ existuje, je typ `int`. Jeho názov je odvodený z anglického slova *integer*, čo znamená celé číslo, logicky teda môžeme predpokladať, že tento typ sa používa na uchovanie celočíselných údajov. Jeho veľkosť je obyčajne taká, aby s ňou procesor, pre ktorý je daný program určený, pracoval čo najefektívnejšie. Napríklad v programoch určených pre reálny alebo 16-bitový chránený režim procesora x86 (programy pre MS-DOS, MS Windows 3.x) je veľkosť typu int 16 bitov, v programoch určených pre 32-bitový chránený režim (MS Windows 95, NT) je jeho veľkosť 32 bitov. Povolený rozsah hodnôt pre oba prípady je v tab. 1. Všimnite si, že typ int sa používa na uchovávanie hodnôt so znamienkom.

V prípade, že chceme explicitne použiť menší či väčší rozsah, máme k dispozícii dva modifikované typy – `short int` a `long int`. Ich veľkosť nie je nijako garantovaná, jediné, na čo sa môžeme spoliehať, je, že veľkosť typu `long int` nebude menšia ako veľkosť typu `int` a tá zase nebude menšia ako veľkosť typu `short int`. Vo väčšine implementácií jazyka C++ na PC má typ `short int` veľkosť 16 bitov a typ `long int` 32 bitov. Povolené rozsahy ich hodnôt sú opäť v tab. 1.

Takto definované typy sa všetky používajú na reprezentáciu čísel so znamienkom. Čo však v prípade, že potrebujeme uchovávať čísla bez znamienka? Na tieto účely máme v C++ k dispozícii k spomenutým trom celočíselným typom ich neznamienkové modifikácie – zoradené podľa veľkosti sú to `unsigned short int`, `unsigned int` a `unsigned long int`. Od svojich znamienkových ekvivalentov sa líšia rozsahom povolených hodnôt (pozri tab. 1), inak ich veľkosti a spôsob uloženia v pamäti sú zhodné, podobne ako pri type char.

*Tabuľka 1*

| Typ | Veľosť | Rozsah hodnôt | Rozsah hodnôt inak |
|---|---|---|---|
| char | 8 bitov | podľa implementácie | podľa implementácie |
| signed char | 8 bitov | –128 ÷ 127 | $-2^7 \div 2^7 - 1$ |
| unsigned char | 8 bitov | 0 ÷ 255 | $0 \div 2^8 - 1$ |
| short int | 16 bitov | –32768 ÷ 32767 | $-2^{15} \div 2^{15} - 1$ |
| int (16bit) | 16 bitov | –32768 ÷ 32767 | $-2^{15} \div 2^{15} - 1$ |
| int (32bit) | 32 bitov | –2147483648 ÷ 2147483647 | $-2^{31} \div 2^{31} - 1$ |
| long int | 32 bitov | –2147483648 ÷ 2147483647 | $-2^{31} \div 2^{31} - 1$ |
| unsigned short int | 16 bitov | 0 ÷ 65535 | $0 \div 2^{16} - 1$ |
| unsigned int (16bit) | 16 bitov | 0 ÷ 65535 | $0 \div 2^{16} - 1$ |
| unsigned int (32bit) | 32 bitov | 0 ÷ 4294967295 | $0 \div 2^{32} - 1$ |
| unsigned long int | 32 bitov | 0 ÷ 4294967295 | $0 \div 2^{32} - 1$ |
| float | 32 bitov | $\pm 3.4 \times 10^{-38} \div \pm 3.4 \times 10^{38}$ | $\pm 2^{-126} \div \pm 2^{128}$ |
| double | 64 bitov | $\pm 2.2 \times 10^{-308} \div \pm 1.8 \times 10^{308}$ | $\pm 2^{-1022} \div \pm 2^{1024}$ |
| long double | 80 bitov | $\pm 3.4 \times 10^{-4932} \div \pm 1.2 \times 10^{4932}$ | $\pm 2^{-16382} \div \pm 2^{16384}$ |

Tretím typom, alebo skôr množinou typov, sú typy s pohyblivou rádovou čiarkou. Sú tri: `float`, `double` a `long double`. Líšia sa svojou veľkosťou a presnosťou vyjadrovania reálnych čísel. Podobne ako pri celočíselných typoch nemáme túto presnosť nijako zaručenú, môžeme si byť iba istí, že presnosť typu `long double` nebude menšia ako presnosť typu `double` a tá zase nebude menšia ako presnosť typu `float`.

Jednotlivé implementácie C++ môžu mať rôzne veľkosti (na otázku, ako to je vo vašom prekladači, nájdete určite odpoveď v manuáli alebo elektronickej nápovedi), ako príklad sú v tab. 1 uvedené rozsahy týchto typov v Borland C++ 3.1. V tomto prekladači je typ `float` reprezentovaný 32-bitovým, typ `double` 64-bitovým a typ `long double` 80-bitovým číslom podľa štandardu IEEE 754.

Posledným zo základných typov (vlastne skoro posledným, ešte existuje tzv. vymenovaný typ, o ktorom si povieme neskôr) je typ `void`. *Void* znamená po anglicky prázdny a tento typ sa používa v prípade, že nepracujeme so žiadnym konkrétnym údajom. V programe nemôže existovať premenná typu `void`, lebo by vlastne nemala zmysel. Jednou z možností použitia je vyjadrenie, že určitá funkcia nemá argumenty alebo nevracia nijakú hodnotu – ak si spomeniete na program z druhej časti seriálu, naše funkcie boli definované s návratovým typom `void` – nevracali tomu, kto ich zavolal, nijaký údaj. Ďalej sa typ void používa pri práci s ukazovateľmi „niekam do pamäte“, ale o ukazovateľoch si budeme hovoriť až nabudúce.

## Typy konštánt

V predošlej časti sme sa zaoberali rôznymi konštantami, ale nepovedali sme si nič o tom, aké sú ich typy. Dnes už máme dostatočné znalosti, preto si tento výklad doplníme.

Znakové konštanty (s výnimkou „širokoznakových“) sú typu `char`.

Typ celočíselnej konštanty závisí od jej formy, hodnoty a prípadnej prípony. Najprv predpokladajme, že za konštantou nenasleduje nijaká prípona. Ak ide o desiatkovú konštantu, jej typ je prvým z nasledujúcich, do ktorého sa „zmestí“: `int`, `long int`, `unsigned long int`. Ak ide o osmičkovú alebo o šestnástkovú konštantu, jej typ je podobne jedným z nasledujúcich: `int`, `unsigned int`, `long int`, `unsigned long int`.

Ak za celočíselnou konštantou nasleduje prípona `u` alebo `U`, je jej typ podľa hodnoty `unsigned int` alebo `unsigned long int`. Ak za konštantou nasleduje prípona `l` alebo `L`, jej typ je jedným z nasledujúcich: `long int`, `unsigned long int`. A konečne, ak za celočíselnou konštantou nasledujú znaky `ul`, `lu`, `uL`, `Lu`, `Ul`, `lU`, `UL` alebo `LU` (všetky sú teda rovnocenné), je jej typ `unsigned long int`.

Typ konštanty s pohyblivou rádovou čiarkou je implicitne `double`. Ak chceme konštantu typu `float`, musíme za ňu pripísať príponu `f` alebo `F`. Ak, naopak, chceme konštantu `long double`, pripíšeme príponu `l` alebo `L`.

Posledné štyri odseky boli asi ťažko stráviteľné, ale čo sa dá robiť, presnosť je presnosť. Napravíme to aspoň zopár príkladmi – pozri tab. 2. Všimnite si druhý riadok, konštanta `45779` je typu `long int`, hoci rozsahovo by sa zmestila do typu `unsigned int`!

| Konštanta | Typ |
|---|---|
| 5263 | int |
| 45779 | long int (!) |
| 3000000000 | unsigned long int |
| 0x3A7B | int |
| 0xA3F9 | unsigned int |
| 0x4C001E77 | long int |
| 0xFFFFFFFF | unsigned long int |
| 92U | unsigned int |
| 512000U | unsigned long int |
| 1998L | long int |
| 3201376444L | unsigned long int |
| 362UL | unsigned long int |
| 8.854E–12 | double |
| 3.14159F | float |
| 0.766L | long double |

## Ukážkový program

Zatiaľ sme toho veľa nenaprogramovali. Ospravedlňujem sa, ale je to naozaj vo vašom záujme – nechcem vám dať k dispozícii dlhý program, ktorému takmer vôbec nebudete rozumieť. Na druhej strane som si vedomý toho, že najlepšie sa dá problém pochopiť na príkladoch. V tejto časti si preto uvedieme program, ktorý dokopy nerobí nič užitočné: deklaruje niekoľko premenných rôznych typov, priradí im hodnoty nejakých konštánt a nakoniec všetky premenné vypíše na obrazovku. O deklarácii premenných sme si ešte nehovorili, takže len v skratke – ak chceme deklarovať premennú nejakého základného typu, napíšeme najprv názov jej typu, jeden alebo viac bielych znakov a potom identifikátor premennej, pomocou ktorého sa budeme na ňu pri práci odkazovať. Celý riadok ukončíme bodkočiarkou. Priradenie hodnoty realizujeme uvedením identifikátora premennej, ďalej nasleduje operátor priradenia (klasické =, nie pascalovské := !), za ktorý zapíšeme priraďovanú hodnotu (nemusí to byť len konštanta, môže to byť aj iná premenná). Na výpis použijeme funkciu `printf()`, ktorej možnosti sú podstatne rozsiahlejšie ako výpis reťazca znakov (ako sme videli v predchádzajúcich programoch). Presný opis prvého argumentu tejto funkcie, pomocou ktorého celý výpis riadime, odporúčam vzhľadom na rozsiahlosť stručne prezrieť v manuáli prekladača (ale to nie je podmienkou – v kapitolách o štandardnej knižnici jazyka C si povieme viac). A tu je už sľubovaný program:

```
#include <stdio.h>

int main()
{
    char c;
    unsigned char uc;
    int i;
    long int li;
    unsigned int ui;
    unsigned long int uli;
    float f;
    double d;

    c = 'A';
    uc = '\x93';   // = 'ô'
    i = -1234;
    li = -654321;
    ui = 56789;
    uli = 3000000000;
    f = 2.99E+8f;
    d = -1.7E-50;

    printf("c   = %c\n", c);
    printf("uc  = %c\n", uc);
    printf("i   = %i\n", i);
    printf("li  = %li\n", li);
    printf("ui  = %u\n", ui);
    printf("uli = %lu\n", uli);
    printf("f   = %g\n", f);
    printf("d   = %lg\n", d);

    return 0;
}
```

Všetky celočíselné konštanty použité v programe majú typ daný svojou hodnotou, je teda zbytočné špecifikovať ho explicitne príponou. Program je viac-menej koncipovaný pre 16-bitové prostredie (MS-DOS), ak ho budete prekladať 32-bitovým prekladačom (ako napr. konzolovú aplikáciu pre Win32), dvojice premenných `i`, `li` a `ui`, `uli` budú mať rovnaký rozsah hodnôt.

Odporúčam vám vyskúšať si v uvedenom programe rôzne typy konštánt, priraďovanie obsahov premenných medzi sebou a ich vypisovanie. Vzor na výpis každého dôležitejšieho typu je v programe; pre tých, ktorí majú záujem, formátovací reťazec pre `short int` je `%hi`, pre `unsigned short int` je to `%hu` a konečne pre `long double` je ním `%Lg`.

## Piata časť: TYPY JAZYKA C++ (pokračovanie)

Na úvod obligátne doplnky k predošlej časti – vypadla mi poznámka k tabuľke č. 2, že údaje v nej sa vzťahujú, samozrejme, na 16-bitové prostredie, kde je veľkosť základného typu `int` 16 bitov. Okrem toho som si uvedomil, že v súvislosti s ukážkovým programom spomínam potrebu a spôsob deklarácie premenných, ale bez vysvetlenia, čo to vlastne deklarácia je. Keďže deklaráciám chcem venovať samostatný úsek seriálu, teraz len stručne: ak pracujeme v programe s nejakými údajmi, pravdepodobne ich máme uložené v premenných. V niektorých programovacích jazykoch, ako BASIC alebo PERL, stačí napísať napríklad `a = 8` a máme vytvorenú premennú `a`, poväčšine všeobecného alebo vopred dohodnutého typu. V C++ však musíme prekladaču explicitne povedať, akú premennú chceme používať a akého bude typu. Tento úkon sa nazýva *deklarácia* danej premennej. Prekladač na základe deklarácie okrem iného rezervuje pre premennú miesto v pamäti. Za povšimnutie stojí, že prvý model používajú zvyčajne interpretované jazyky, zatiaľ čo druhý prakticky bez výnimky kompilované jazyky.

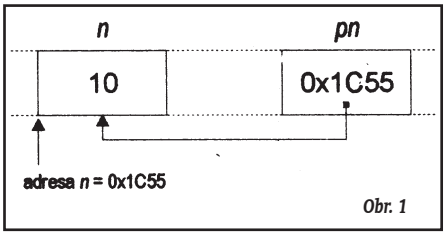
## Odvodené typy

V predchádzajúcej časti seriálu sme sa zaoberali základnými typmi jazyka C++. Na zopakovanie: ide o typy `char`, `signed char`, `unsigned char`, `int`, `short int`, `long int`, `unsigned int`, `unsigned short int`, `unsigned long int`, `float`, `double`, `long double` a `void`. Tieto typy sa obyčajne používajú (s výnimkou posledného) na uchovávanie jednoduchej neštruktúrovanej číselnej informácie. Okrem nich existujú takzvané odvodené typy, ktoré prichádzajú na rad v prípade, že potrebujeme pracovať s dátami nejakým spôsobom organizovanými. Tieto typy sú vždy založené na základných typoch alebo ich kombináciách.

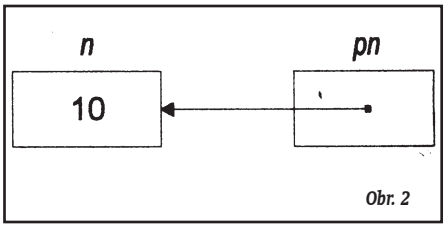
## Ukazovatele

Prvým a možno povedať, že najzložitejším, najodsudzovanejším, ale súčasne aj najmocnejším odvodeným typom je *ukazovateľ*, takisto *smerník* (pointer). Oba názvy sú ekvivalentné – ktorý z nich budete používať, je otázka vkusu. Na efektívne programovanie v C++ je veľmi dôležité dôkladne pochopiť princíp práce s ukazovateľmi, už aj z toho dôvodu, že spôsobujú najväčšie percento programových chýb.

Čo je to vlastne ukazovateľ? Typ ukazovateľ je vždy združený s nejakým iným typom, na ktorého inštanciu „ukazuje" (a nemusí to byť len jeden zo základných typov). *Premenná typu ukazovateľ obsahuje adresu miesta uloženia tejto inštancie v pamäti*. Uveďme si príklad: máme premennú n typu int, ktorej hodnota (obsah) je napríklad `10`. Nech je táto premenná umiestnená na adrese `0x1C55`. Potom obsah inej premennej `pn` typu „ukazovateľ na `int`", ukazujúcej na premennú `n` (ktorá je typu `int`), bude práve `0x1C55`. Celá situácia je znázornená na obrázkoch č. 1 a č. 2. Šípka ukazujúca



Obr. 1



Obr. 2

z premennej `pn` na premennú `n` vyjadruje, že `pn` ukazuje na `n` – takto sa to obyčajne zvykne kresliť. Obrázok č. 1 znázorňuje umiestnenie oboch premenných v pamäti a ich obsah, obrázok č. 2 sa používa častejšie a vyjadruje reláciu, ktorá existuje medzi ukazovateľom a premennou, na ktorú tento ukazovateľ ukazuje.

Deklarácia premennej typu ukazovateľ na nejaký základný typ je veľmi jednoduchá. Medzi názov typu a názov premennej vložíme znak * (hviezdička):

```
int n = 10;
int* pn;
```

(Je úplne jedno, či bude pred hviezdičkou alebo za ňou, alebo aj pred ňou, aj za ňou medzera, alebo tam dokonca medzera nebude vôbec, prekladač dokáže celú deklaráciu jednoznačne rozložiť na tri lexikálne jednotky – `int` [kľúčové slovo], * [operátor] a `pn` [identifikátor].)

Deklaráciu ukazovateľov na odvodené typy si ozrejmíme v časti venovanej deklaráciám.

Takto deklarovaný ukazovateľ má však jednu dôležitú vlastnosť: keďže sme dosiaľ explicitne nepovedali, kam má ukazovať, ukazuje jednoducho niekam do pamäte (kam, to závisí od predchádzajúceho obsahu pamäte, v ktorej je uložený) a túto oblasť považuje za inštanciu *svojho* typu!! Inak povedané, obsah premennej typu ukazovateľ sa interpretuje ako adresa, hoci to vôbec adresa nemusí byť (a v operačných systémoch s ochranou pamäte použitie takéhoto ukazovateľa môže skončiť známou „všeobecnou chybou ochrany"). Takýto ukazovateľ sa označuje často ako neinicializovaný a do určitej miery ho dokáže rozoznať prekladač už vo fáze prekladu. Na druhej strane treba poznamenať, že niekedy sa fakt, že úsek pamäte, na ktorý ukazovateľ ukazuje, prekladač interpretuje ako premennú daného typu, využíva na realizáciu vecí, ktoré by inak bolo možné urobiť len zložito a neefektívne. Rozhodne to však nepatrí ku každodennej praxi a oveľa častejšie to býva príčinou záhadných a neraz na prvý pohľad nevysvetliteľných chýb v programoch.

Aby sme mohli pracovať s ukazovateľmi, ktoré ukazujú na zmysluplné údaje, potrebujeme prekladaču oznámiť, že ukazovateľ má ukazovať na existujúci objekt daného typu, uložený v pamäti. To dosiahneme nasledujúcim zápisom:

```
pn = &n;
```

Znak `&` je operátor, ktorého aplikáciou na premennú `n` získame jej adresu a tú uložíme do nášho ukazovateľa. Ten teraz ukazuje na známu premennú a môžeme ho bez obáv používať. Inicializáciu ukazovateľa môžeme spojiť priamo s deklaráciou:

```
int* pn = &n;
```

Ako sa však dostaneme k premennej, na ktorú ukazovateľ ukazuje? Tento proces sa nazýva *dereferencia* ukazovateľa a vyjadrí sa uvedením operátora * pred jeho menom. Týmto spôsobom dostaneme obsah miesta, na ktoré ukazovateľ ukazuje, a *môžeme s ním ďalej narábať ako s premennou daného typu:*

```
*pn = 20;
printf("n = %i\ n", *pn);
```

Prvý príkaz predchádzajúceho príkladu priraďuje hodnotu 20 premennej typu `int`, na ktorú ukazuje ukazovateľ `pn`. Prekladač jazyka C++ je v tomto ohľade veľmi benevolentný a optimisticky predpokladá, že pn naozaj ukazuje na premennú typu `int`. V prípade, že by bol pn neinicializovaný, prepíše uvedený príkaz na náhodnú oblasť pamäte, čo, ako si viete predstaviť, môže mať vskutku kuriózne a ťažko predvídateľné následky. V druhom príkaze získavame obsah premennej `n` pomocou dereferencie ukazovateľa `pn` ukazujúceho na ňu.

## Referencie

Ďalším odvodeným typom, o ktorom si povieme, je *referencia* (reference). Občas sa možno stretnúť aj s názvom *referenčný typ*. Tento typ v jazyku C neexistuje a objavuje sa až v C++.

Referencia je veľmi podobná ukazovateľu a zjednodušene môžeme povedať, že je to ukazovateľ, ktorý sa automaticky dereferencuje. Obsahom premennej typu referencia je takisto adresa inej premennej, ale navonok sa referencia javí ako normálna premenná daného typu. Tak ako ukazovateľ aj referencia je združená s nejakým iným typom – hovoríme, že ide o referenciu na daný typ.

Aj deklarácia referencie je podobná deklarácii ukazovateľa, len namiesto znaku * vložíme medzi typ a meno premennej znak `&` (tzv. ampersand). Na rozdiel od neinicializovaného ukazovateľa sa vám však v tele funkcie nepodarí deklarovať neinicializovanú referenciu – prekladač ohlási chybu. Preto treba hneď pri deklarácii referenciu inicializovať uvedením mena združenej premennej (tentoraz bez operátora `&`):

```
int a = 94;
int& ra = a;
```

Všimnime si, že referenciu je potrebné inicializovať menom existujúcej premennej, nie je správny napríklad nasledujúci zápis:

```
int& ra = 94;
```

Prekladač síce pri preklade tohto riadka v mnohých prekladačoch neohlási chybu, ale (podľa nastavenia) vydá varovanie (warning), že na inicializáciu referencie bola použitá pomocná (dočasná – temporary) premenná, ktorej hodnota sa nastavila na 94. Dostaneme tak referenciu na premennú, ktorú sme vôbec nedeklarovali a ktorú pravdepodobne ani nechceme.

Po inicializácii už nie je nijakým spôsobom možné zmeniť referenciu tak, aby odkazovala na inú premennú. Referenčná premenná sa ďalej správa úplne rovnako ako premenná s ňou združená, to znamená, že každá operácia s referenciou mení v skutočnosti premennú, na ktorú referencia odkazuje:

```
int x = 1;
int& rx = x;
printf("x = %i\ n", x);
rx = 2;
printf("x = %i\ n", x);
```

Po prebehnutí tohto úseku programu sa na štandardnom výstupe objaví:

```
x = 1
x = 2
```

Z toho je zrejmé, že zmena premennej `rx` sa rovnako dotkla aj premennej `x`.

Referenčná premenná slúži teda ako nejaký „alias", pomocou ktorého sa môžeme odvolávať na inú premennú. Na tomto mieste sa sluší podotknúť, že deklarácia a používanie referencií priamo v tele funkcií sa prakticky obmedzuje na niekoľko málo situácií, ako napríklad práca s nejakou (z hľadiska zápisu) komplikovane prístupnou premennou – namiesto vypisovania siahodlhých reťazcov pri opakovanom prístupe k tejto premennej si jednoducho deklarujeme referenciu s krátkym a rozumným menom, ktorá na túto premennú odkazuje. Skutočná výhoda referencií sa však ukáže až pri ich používaní ako typu argumentov funkcie alebo návratovej hodnoty. Ale k tomuto sa dostaneme až pri rozprávaní o funkciách.

## Polia

Posledným odvodeným typom, ktorý si v tejto časti vysvetlíme, je pole. Pole nie je nič iné ako postupnosť prvkov jedného typu, zastrešená spoločným názvom. Jednotlivé prvky poľa sú prístupné pomocou ich indexov. Index prvého prvku poľa v jazyku C++ je vždy 0 a nedá sa zmeniť. Deklarácia poľa je podobná deklarácii jednoduchej premennej s tým rozdielom, že za meno deklarovanej premennej (typu pole) uvedieme navyše v hranatých zátvorkách počet prvkov poľa:

```
int a[24];
```

Týmto riadkom sme deklarovali premennú `a` typu „pole 24 prvkov typu `int`". Keďže prvky poľa sa indexujú od nuly, posledný z nich má index `23`. K jednotlivým prvkom pristupujeme uvedením názvu poľa a pridaním indexu príslušného prvku uzavretého v hranatých zátvorkách:

```
a[3] = 17;
printf("a[3] = %i\ n", a[3]);
```

Všimnite si, že prvky poľa sa správajú ako normálne premenné typu `int` (čo je vlastne úplne prirodzené).

Jazyk C++ nijako nekontroluje, či index, ktorý zapíšeme do zátvoriek, je v danom rozmedzí `0` až `N-1`, kde `N` je počet prvkov poľa. Tento fakt je dvojsečnou zbraňou – na jednej strane umožňuje efektívne vykonanie určitých operácií, na druhej strane však býva (podobne ako práca s ukazovateľmi) zdrojom obrovského počtu chýb. Je preto veľmi dôležité chápať princíp práce s poľom a dávať si veľký pozor na to, aký index použijeme.

V C++ nie je priamo implementovaný nijaký typ viacrozmerné pole. V prípade, že potrebujeme používať napríklad pole údajov, ktoré má dva rozmery, deklarujeme ho jednoducho ako pole jednorozmerných polí. Tento spôsob je mimoriadne flexibilný, ako ešte uvidíme, pretože umožňuje podobne deklarovať *n*-rozmerné pole ako jednorozmerné pole, ktorého prvkami sú polia (*n* – 1)-rozmerné a tak ďalej až po posledný rozmer. Príklad deklarácie trojrozmerného poľa:

```
int t[3][2][5];
```

V tomto príklade deklarujeme premennú `t` ako pole troch prvkov, z ktorých každý je dvojprvkovým poľom, pričom oba prvky tohto poľa sú opäť polia, tentoraz piatich prvkov typu `int` (uff!). Ľubovoľný prvok na najnižšej úrovni celej hierarchie (ktorý je typu `int`) je prístupný uvedením postupne všetkých troch indexov. Prvým prvkom (s najnižšími indexmi) celej trojrozmernej štruktúry je teda `t[0][0][0]` a posledným (s najvyššími indexmi) je prvok `t[2][1][4]`. Pozor, na rozdiel napríklad od Pascalu treba každý index uzavrieť do zátvoriek osobitne, výraz `t[1, 1, 1]` je teda *nesprávny*! (Presnejšie, syntakticky je správny, ale nie sémanticky, lebo znamená niečo úplne iné.)

Možno ste si všimli, že prvky dosiaľ deklarovaných polí neboli nijako neinicializované. To je v poriadku, ale ak chceme s poľom rozumne pracovať, potrebujeme nejakým spôsobom jednotlivým prvkom priradiť ich hodnoty. To je možné v zásade dvoma spôsobmi. Jedným je postupné priraďovanie hodnôt prvkom klasickým spôsobom. Keďže sa toto priraďovanie môže umiestniť do tzv. cyklu (o ktorom si povieme bližšie v časti venovanej príkazom jazyka C++), je tento spôsob vhodný, ak všetky prvky majú rovnakú hodnotu alebo medzi jednotlivými hodnotami existuje algoritmizovateľný vzťah, prípadne ak inicializačné hodnoty sú známe až za behu programu. Prvky poľa možno inicializovať aj priamo, vymenovaním jednotlivých hodnôt – tento spôsob si však vyžaduje poznať inicializačné hodnoty vopred. V takomto prípade do deklarácie doplníme za meno poľa operátor = a v krútených zátvorkách uzavretý zoznam hodnôt:

```
int a[6] = {  1, 2, 4, 8, 16, 32 } ;
```

Pri tomto spôsobe inicializácie je dovolené počet prvkov poľa vynechať – prekladač si ho určí automaticky z počtu inicializačných hodnôt:

```
int a[] = {  1, 2, 4, 8, 16, 32 } ;
```

V krútených zátvorkách môžeme uviesť aj menej hodnôt, ako je deklarovaná veľkosť poľa, vtedy sa zvyšné prvky inicializujú na nulu:

```
int a[6] = {  1, 2, 3, 4 } ;
```

(prvky `a[4]` a `a[5]` budú mať hodnotu `0`).
Polia prvkov typu `char` je možné inicializovať ešte jedným spôsobom – za operátorom = uvedieme reťazcovú konštantu:

```
char msg[] = "Hello";
```

Veľkosť tohto poľa je o jeden znak väčšia ako počet znakov v reťazci, lebo, ako si iste spomínate, každá reťazcová konštanta v C++ je ukončená znakom `\0`. Tu si treba všimnúť, že jazyk C++ nemá nijaký typ „reťazec znakov", namiesto toho sa pracuje s poľom znakov, ktoré by malo byť ukončené znakom `\0`, a to z toho dôvodu, že všetky funkcie štandardnej knižnice jazyka C++ tento formát očakávajú.

Znakové pole je, samozrejme, možné inicializovať aj klasickým spôsobom:

```
char msg[] = {  'H', 'e', 'l', 'l', 'o', '\ 0' } ;
```

sami však iste uznáte, že prvý spôsob je oveľa pohodlnejší.

Viacrozmerné polia (tento výraz používam iba pre pochopenie, v skutočnosti sú to všetko len *jednorozmerné* polia, ale spolu tvoria niečo, čo sa dá považovať za akýsi diskrétny viacrozmerný hyperpriestor) je možné inicializovať podobne, pričom jednotlivé inicializačné hodnoty sú opäť zoznamy hodnôt v krútených zátvorkách. Na rozdiel od jednoduchých polí však môžeme v deklarácii vynechať len veľkosť prvého rozmeru, ostatné treba uviesť (vyžaduje si to prekladač pre kontrolu typov a správny prístup k jednotlivým prvkom). Príklad inicializácie:

```
int b[][2][3] =
 {  {  {  111, 112, 113 } , {  121, 122, 123 }  } ,
    {  {  211, 212, 213 } , {  221, 222, 223 }  }  } ;
```

V tomto príklade deklarujeme pole b ako dvojprvkové (spočíta automaticky prekladač) pole, ktorého jednotlivé prvky sú opäť dvojprvkové (dvojka v hranatých zátvorkách) polia a každý prvok týchto polí je pole troch (trojka v hranatých zátvorkách) prvkov typu int. Inicializačné hodnoty vyjadrujú pre lepšie pochopenie súradnice jednotlivých prvkov v trojrozmernom priestore (s tým rozdielom, že polia indexujeme od nuly a súradnice sa začínajú od jednotky – to pre fakt, že nulou sa začína osmičková konštanta).

Na záver ešte jedna poznámka. Keďže prvkami viacrozmerného poľa sú opäť polia s rozmerom o jeden menším, je syntakticky úplne správne a často používané) pristupovať k jednotlivým podpoliam hlavného poľa pomocou príslušných indexov. Nech pole `t` je deklarované tak, ako je uvedené vyššie. Potom výraz `t[0]` je správny a reprezentuje prvý prvok poľa `t`, čo je dvojprvkové pole polí piatich prvkov typu int. Takisto výraz `t[0][0]` je správny a vyjadruje prvý prvok poľa `t[0]`, ktorý je sám osebe trojprvkovým poľom premenných typu int. A konečne `t[0][0][0]` je prvý prvok poľa `t[0][0][0]`, ktorého typ už je int. Z doterajšieho výkladu vyplýva, že „viacrozmerné" polia sú uložené v pamäti „po riadkoch", teda index, ktorý je najviac vpravo, sa mení najrýchlejšie.

### Nabudúce

Dnešné rozprávanie bolo dosť náročné na predstavivosť, je však dôležité vedieť, čo sa odohráva v pamäti pri behu programu, lebo len tak budete schopní úspešne a s minimálnym množstvom chýb pracovať so spomínanými typmi. V mnohých jazykoch sa programátor nemusí starať, ako sú údaje, s ktorými pracuje, reprezentované a uložené v pamäti. V C++ si tento luxus v prevažnej väčšine nemožno dovoliť, ak chcete písať programy, ktorým dokonale rozumiete. Na druhej strane máte úplnú kontrolu nad svojimi dátami a nestane sa vám, že preložený kód robí niečo, o čom neviete. Toto je tá silná väzba na hardvér, ktorú som spomínal v prvej časti nášho seriálu – neznamená to, že program v C++ je závislý od konkrétnej platformy, ale že programátor je schopný pracovať na úrovni veľmi blízkej strojovému kódu daného procesora a nie je obmedzovaný niekedy zbytočnými abstrakciami vyšších programovacích jazykov (hoci jazyk C++ patrí takisto medzi vyššie jazyky, povedal by som, že medzi nimi je umiestnený „najnižšie"). C++ je vhodnejším jazykom pre „ozajstných programátorov" ako pre „pojedačov koláčov". Pre tých, ktorí nerozumejú – skúste si v nejakej vyhľadávacej službe na Internete zadať heslá ako „real programmers" a „quiche eaters". Prípadne mi napíšte mail a ja vám pošlem zopár vtipných článkov (dúfam, že mi nepreplníte mailbox!).

Nabudúce si toto rozprávanie ešte doplníme vzájomným vzťahom polí a ukazovateľov, ktorý patrí k dosť zložitým témam v C++, a potom si začneme hovoriť podrobnejšie o deklaráciách alebo o výrazoch a operátoroch, podľa toho, aká postup mi bude zdať vhodnejší.

Na záver vám zaželám už len veľa úspechov pri študovaní dnešného textu a dôrazne vám odporúčam vyskúšať si všetky uvedené príklady a pokúsiť sa ich rôzne modifikovať, až kým nebudete mať pocit, že všetkému rozumiete. Zatiaľ, pravda, viete iba deklarovať premenné rôznych typov, priraďovať im hodnoty a vypisovať ich, ale na pochopenie príkladov to úplne postačí. Odporúčam vám ešte po preložení programu krokovať jednotlivé príkazy a sledovať hodnoty premenných pomocou nástrojov príslušného debuggera (veľmi silným nástrojom je napríklad príkaz Inspect v Borland C++, ktorý vám zobrazí premennú tak, ako je uložená v pamäti [vyžaduje si to trochu sa pohrabať v manuáloch a pohrať s jednotlivými možnosťami]).

## Šiesta časť: VÝRAZY A OPERÁTORY

Minule sme sa oboznámili s niekoľkými odvodenými typmi C++, konkrétne to boli polia, ukazovatele a referencie. Vo väčšine učebníc C++ sa téma ukazovateľov pokladá za príliš zložitú a odsúva sa na ďalšie strany. Ja som sa rozhodol opísať túto oblasť už teraz na začiatku, dúfam, že vás to priveľmi neodradilo. Každopádne sa ešte k ukazovateľom neraz vrátime, aby sme si objasnili všetky záludnosti, ktoré nám ich existencia pripravila. Pri výučbe C++ je základným problémom to, že takmer všetko so všetkým súvisí a je niekedy obťažné vysvetliť dôkladne jednu tému bez aspoň základnej znalosti inej témy. Na konci predošlej časti som spomínal, že dnes si ešte dopovieme niečo o vzájomnom vzťahu medzi poľami a ukazovateľmi. Táto téma je dosť náročná a myslím, že bude lepšie odložiť ju na neskôr a najprv sa venovať niečomu jednoduchšiemu, aby ste mohli začať písať aj drobné programy. Dnes sa preto budeme zaoberať výrazmi jazyka C++ a najpoužívanejšími operátormi, s pomocou ktorých budeme môcť v našich programoch realizovať aj výpočty.

### Výrazy a operátory

Najprv si vysvetlime, čo to vlastne *výraz* je. Stručne a jasne, výraz v C++ je niečo, čo sa dá vypočítať. Spôsob výpočtu je daný zápisom výrazu. Výraz má okrem iného tú vlastnosť, že ak sa pri jeho výpočte nevyvolajú nejaké vedľajšie účinky, môžeme ho akoby nahradiť svojou hodnotou. Iná definícia – výraz je postupnosť operátorov a ich operandov, ktorá definuje nejaký výpočet.

*Operátor* je obyčajne symbol (lexikálna jednotka), ktorý reprezentuje nejakú, poväčšine matematickú operáciu s údajmi. Každý operátor má jeden alebo niekoľko tzv. *operandov*, čo sú vlastne údaje, nad ktorými pracuje. Príklad: operácia sčítania sa bežne vyjadruje znamien-

kom +. V C++ takisto existuje operátor +, ktorý má dva operandy, a výsledkom aplikácie tohto operátora na jeho operandy je ich súčet. Operátory môžu byť *unárne* (majú jeden operand), *binárne* (majú dva operandy) alebo dokonca *ternárne* (majú tri operandy). Ďalej operátory delíme na *prefixové, infixové* a *postfixové*. Prefixový operátor je obyčajne unárny a zapisuje sa *pred* operand, postfixový operátor je takisto unárny a zapisuje sa za operand a infixový operand je binárny alebo ternárny a zapisuje sa medzi svoje operandy.

Výpočet hodnoty výrazu sa nazýva vyhodnocovanie výrazu a spočíva vlastne v aplikovaní operátorov na svoje operandy. Najzákladnejšími operandmi sú konštanty (literály) a identifikátory premenných. Hodnota konštánt pri ich vyhodnocovaní je zrejmá, hodnota identifikátorov premenných je daná obsahom týchto premenných. Okrem toho môžu byť operandmi operátora opäť výrazy, čím sa dostávame k rekurzívnej definícii výrazov, pri ktorej môžeme ísť teoreticky do ľubovoľnej hĺbky.

Predstavme si, že chceme spočítať dve čísla. Výraz, ktorý bude predstavovať toto sčítanie, bude teda pozostávať z operátora + a jeho dvoch operandov – dvoch sčítancov. Príklad zápisu takéhoto výrazu je `9 + 4`. Samozrejme, v jednom výraze môže byť viacej operátorov, každý operátor musí mať však dodržaný počet svojich operandov (počet operandov operátora sa ináč nazýva aj *arita* operátora). V prípade, že sa vo výraze nachádza viacero rôznych operátorov, je potrebné vedieť poradie, v ktorom sa jednotlivé operátory budú vyhodnocovať – hovoríme o rôznej *priorite* operátorov. Operátory s vyššou prioritou budú vyhodnotené skôr ako operátory s nižšou prioritou. Teda napríklad vo výraze `4 + 5 * 6` sa najprv vykoná násobenie (operátor * je operátorom násobenia) a až potom sčítanie, pričom druhým operandom operátora sčítania bude práve výsledok uvedeného násobenia. Ak sa vo výraze nachádza viacero rovnakých operátorov alebo viacero operátorov s rovnakou prioritou, poradie ich vyhodnocovania je dané tzv. *asociativitou* operátorov. Asociativita v C++ je dvojaká: buď zľava doprava, alebo sprava doľava. Čo to znamená, to si vysvetlíme na nasledujúcom príklade. Predstavme si, že chceme zapísať výraz, v ktorom sčítame tri čísla. Tento výraz bude zrejme vyzerať takto: `1 + 2 + 3`. Pri výpočte takéhoto výrazu môže procesor postupovať v zásade dvoma spôsobmi. Buď najprv sčíta prvé dva operandy a potom k tomuto súčtu pripočíta tretí operand, t. j. výraz sa vyhodnotí spôsobom `(1 + 2) + 3`, alebo najprv sčíta druhý a tretí operand a potom k výsledku pripočíta prvý operand, t. j. výraz sa vyhodnotí spôsobom `1 + (2 + 3)`. V prvom prípade by mal operátor + asociativitu zľava doprava, v druhom, naopak, asociativitu sprava doľava.

V prípade iných požiadaviek na vyhodnocovanie výrazov, než je implicitne dané ich prioritou a asociativitou, je možné podobne ako v matematike použiť explicitné zátvorky. Ich použitie sa však odporúča aj v prípade, že zo zápisu nie je celkom zrejmé poradie vyhodnocovania výrazu. Často sa tým predíde záhadným chybám v inak na pohľad správnom programe. Časť výrazu uzavretá v zátvorke predstavuje akoby samostatný výraz, ktorý sa kompletne vyhodnotí a jeho hodnota sa použije v ďalších výpočtoch. Ako príklad použitia zátvoriek modifikujeme jeden z predošlých – vo výraze `(4 + 5) * 6` sa teda najprv vykoná sčítanie, až potom násobenie (pri vyhodnocovaní výrazu sa najprv vyhodnotí obsah zátvorky, ktorý sa ďalej použije ako prvý operand operátora násobenia, teda celý výraz sa redukuje na `9 * 6`, potom sa oba operandy vynásobia, čím dostaneme výslednú hodnotu celého výrazu `54`.

Na tomto mieste sa sluší podotknúť, že okrem špeciálnych prípadov nie je nijako zaručené poradie vyhodnocovania viacerých operandov jedného operátora. To znamená, že v prípade zápisu výrazu `9 * 7 + 5 * 3` nemôžeme povedať, či sa najskôr vykoná prvé násobenie `9 * 7` alebo druhé `5 * 3`. V tomto prípade nám to veľmi nevadí, aj tak sa v konečnom dôsledku budú sčítavať čísla `63` a `15`, ale v prípade, že by vyhodnocovanie operandov malo vedľajšie účinky (napr. zmena nejakých premenných), môže dôjsť k tomu, že program sa bude správať úplne ináč, než sme očakávali.

## Základné operátory

V tejto časti seriálu si povieme len o najdôležitejších operátoroch, slúžiacich na realizáciu rôznych matematických a logických operácií. Niekoľko ďalších operátorov súvisí so zatiaľ nevysvetľovanými témami, preberieme si ich preto neskôr.

## Operátor priradenia

Prvým z operátorov, ktorý si uvedieme, je *operátor priradenia* =. S ním sme sa vlastne už niekoľkokrát stretli – v programe zo 4. časti a takisto pri deklarácii premenných spojenej s uvedením inicializačnej hodnoty. Vo všeobecnosti tento operátor slúži na priradenie novej hodnoty nejakej premennej. Operátor = je binárny, infixový a asociuje sa sprava doľava. Prvým (ľavým) operandom musí byť tzv. *l-hodnota* (l-value, z anglického left value). Tento pojem predstavuje taký objekt jazyka C++, ktorý existuje v pamäti a je možné mu priradiť hodnotu. Každý identifikátor premennej (s výnimkou konštantných premenných, o ktorých budeme hovoriť neskôr) je automaticky l-hodnotou, pretože reprezentuje premennú, uloženú v pamäti, ktorej hodnotu chceme zmeniť. Naproti tomu l-hodnotou nie je napr. konštanta (literál), pretože tá nemá vyhradené miesto v pamäti (bolo by to nanajvýš čudné, chcieť priraďovať nejakú [inú] hodnotu napríklad číslu `20`). Druhým (pravým) operandom je ľubovoľný výraz, ktorého typ sa zhoduje s typom ľavého operanda, prípadne medzi typmi oboch operandov musí existovať spôsob tzv. konverzie typov. Výraz, ktorý môže stáť na pravej strane priraďovacieho operátora, sa niekedy označuje ako *r-hodnota* (r-value). Výsledkom vyhodnotenia celého priraďovacieho výrazu je hodnota, ktorá bola priraďovaná, navyše tento výsledok je opäť l-hodnotou, ktorú predstavuje priraďovaná premenná. Uveďme si príklad:

```
a = 123
```

V tomto výraze priraďujeme premennej `a` (l-hodnota) hodnotu `123` (r-hodnota). Výsledná hodnota celého výrazu je `123` a navyše tento výraz (keďže je l-hodnotou) môže opäť stáť na ľavej strane iného priraďovacieho operátora. Prvý fakt môžeme využiť pri viacnásobnom priradení:

```
b = (a = 123)
```

Tento výraz priraďuje premennej `b` hodnotu predchádzajúceho výrazu, čo je, ako sme si povedali, `123`. Keďže sa operátor = asociuje sprava doľava, uvedené zátvorky sú zbytočné. C++ takto poskytuje efektívny spôsob inicializácie viacerých premenných rovnakou hodnotou:

```
i = j = k = 0
```

Druhý spomenutý fakt nám umožňuje zapísať výraz v tvare:

```
(a = 123) = 456
```

ktorý priradí premennej a hodnotu `123`, ktorú vzápätí prepíše hodnotou `456`. Táto konštrukcia však vo všeobecnosti nemá nijaký význam.Okrem tohto základného priraďovacieho operátora existuje niekoľko jeho variantov, ktoré majú všetky tvar *op*=, kde *op* je jeden z operátorov `*, /, %, +, -, <<, >>, &, ^, |` (pozri ďalej). Použitie týchto operátorov je jednoduché, vo všetkých prípadoch zápis

```
e op = v
```

je ekvivalentný zápisu

```
e = e op (v)
```

pričom výraz `e` sa vyhodnocuje len raz. Teda ak napríklad chceme zvýšiť hodnotu premennej `x` o hodnotu 5, máme dve možnosti:

```
x = x + 5
```

alebo `x += 5`

Druhá možžnosť je elegantnejšia a používanejšia.

## Aritmetické operátory

Jazyk C++ poskytuje najbežnejšie aritmetické operátory, známe z matematiky. Ide o *operátor sčítania* +, *odčítania* –, *násobenia* *, *delenia* / a *operátor operácie modulo* %. Všetky operátory sú binárne, infixové a asociujú sa zľava doprava. Prvé dva operátory sú tzv. *aditívne* a majú nižšiu prioritu ako druhé tri (tzv. *multiplikatívne*).

Výsledkom jednotlivých operácií sú po rade súčet, rozdiel, súčin, podiel a zvyšok po delení oboch operandov. Typ výsledku závisí od typu oboch operandov (bližšie v časti venovanej konverziám medzi typmi). Dôležité je uvedomiť si, že výsledkom podielu dvoch celočíselných operandov je *opäť celé číslo* (ktoré vznikne zo skutočného podielu jednoduchým odstránením desatinnej časti). To znamená, že operátor delenia realizuje jednak klasické delenie čísel s pohyblivou čiarkou, jednak celočíselné delenie, známe možno z Pascalu ako operátor div. Ak chceme vydeliť dve celé čísla a dostať podiel v tvare racionálneho čísla, musíme aspoň jeden z operandov pretypovať na niektorý z typov s pohyblivou čiarkou. O operátore pretypovania si povieme o chvíľu. Príklad:

```
        5 /  4 =  1     // !
       -5 /  4 = -1     // !
(double)5 / -4 = -1.25
     -5.0 / -4 =  1.25
```

V prvých dvoch prípadoch sú oba operandy celočíselné a taký je aj výsledok. V treťom prípade je prvý celočíselný operand explicitne pretypovaný na typ double, v štvrtom prípade je prvý operand typu double vzhľadom na svoj zápis.

Čo sa týka operátora modulo, jeho funkcia je výpočet zvyšku po delení dvoch celočíselných (`!`) operandov. Hodnota tohto zvyšku je vždy taká, aby platilo `(a / b) * b + a % b = a`, kde typ a aj b je niektorým z celočíselných typov. Aké znamienko však bude mať výsledok, nie je nijako určené a je to vecou implementácie príslušného prekladača. Obyčajne má zvyšok rovnaké znamienko ako delenec, i keď podľa striktnej matematickej definície zvyšok nemôže byť záporný. V tabuľke 1 sú uvedené výsledky operácií delenia a modulo pre prekladač Borland C++ 3.1.

Tabuľka č. 1

| a | b | a/b | a%b |
|---|---|---|---|
| 5 | 4 | 1 | 1 |
| -5 | 4 | -1 | -1 |
| 5 | -4 | -1 | 1 |
| -5 | -4 | 1 | -1 |

Okrem uvedených binárnych aritmetických operátorov má C++ ešte niekoľko unárnych operátorov. Medzi

prvé dva z nich patria *unárne plus* + a *unárne mínus* –. Tieto operátory sú prefixové a asociujú sa sprava doľava. Unárne plus nemá prakticky nijaký význam, unárne mínus mení znamienko svojho operanda (je ekvivalentné vynásobeniu operanda hodnotou –1). Dva operátory unárneho mínusu za sebou (musia však byť oddelené bielym znakom!!) sa navzájom „zrušia“. Teda výraz `a = - -b` je ekvivalentný výrazu `a = b`.

Ďalšie dva unárne aritmetické operátory sú *operátor inkrementácie* ++ a *dekrementácie* —. Operátor ++ zvýši hodnotu svojho operandu o 1, operátor – ju zase zníži o 1. Obidva operátory majú dva varianty – *prefixový* a *postfixový*. V prípade, že je daný operátor použitý ako prefixový (t. j. pred operandom), asociuje sa sprava doľava a hodnota celého výrazu je rovná hodnote operanda **po** inkrementácii, resp. dekrementácii. Pri použití operátora v *postfixovom* tvare (t. j. za operandom) sa asociuje zľava doprava a hodnota celého výrazu je rovná hodnote operanda **pred** inkrementáciou/dekrementáciou. Najlepšie bude ukázať si to na príklade:

```
int i, j, k;
i = 3;
j = ++i;    // 1
k = i++;    // 2
printf(„i = %i, j = %i, k = %i\n“, i, j, k);
```

Po vykonaní tohto krátkeho kódu bude mať premenná i hodnotu 5 a premenné j aj k hodnotu 4. Pri vyhodnocovaní výrazu na riadku označenom 1, kde je použitý prefixový variant operátora ++, sa najprv inkrementuje obsah premennej i z 3 na 4 a potom sa nová hodnota (t. j. 4) použije ako pravý operand operácie priradenia a priradí sa premennej j. Ďalej sa bude vyhodnocovať výraz na riadku označenom 2, keď sa inkrementuje obsah premennej i zo 4 na 5, ale ako pravý operand priradenia sa vezme hodnota premennej i pred inkrementáciou, teda 4, a tá sa priradí premennej k. Operátor – sa správa podobne.

Operandom prefixovej verzie oboch operátorov musí byť l-hodnota a výsledok celého výrazu je opäť l-hodnota, predstavovaná tým istým operandom. Je teda správny výraz `++a *= 3`, ktorý inkrementuje obsah premennej a a následne ho vynásobí číslom 3. Operandom postfixovej verzie operátorov musí byť takisto l-hodnota, ale v tomto prípade výsledok výrazu nie je l-hodnotou (výraz `a++ /= 5` teda nemá zmysel)! Typ výsledku aplikácie oboch operátorov je zhodný s typom ich operandov.

## Bitové operátory

Nazval som túto skupinu operátorov bitovými, lebo ich funkcia súvisí s ich bitovou reprezentáciou, hoci obyčajne sa takto nezvyknú nazývať.

Prvými dvoma bitovými operátormi sú operátory *bitového posunu* << a >>. Vieme, že každé celé číslo je reprezentované v pamäti ako postupnosť bitov. Oba operátory zmenia hodnotu svojich operandov tak, že túto postupnosť akoby posunú doľava (<<) alebo doprava (>>). Tieto operátory sú binárne, infixové a asociujú sa zľava doprava. Ich prvým operandom je posúvaná hodnota (môže to byť premenná i konštanta), druhým operandom je počet bitov, o ktoré sa prvý operand posunie. Oba operandy *musia byť celočíselné*! Výslednou hodnotou výrazu je posunutý prvý operand, typ výsledku je zhodný s typom tohto operanda. Pozor, pri aplikácii operátorov posunu na premennú sa jej obsah *nemení*!

V prípade, že druhý operand je záporný alebo väčší ako veľkosť prvého operanda v bitoch, výsledok je nedefinovaný. Pri posune doľava sa uvoľnené bity zapĺňajú nulami, pri posune doprava sa v prípade nezáporných čísel zapĺňajú tiež nulami, v prípade záporných čísel je výsledok implementačne závislý (väčšinou sa vzhľadom na reprezentáciu záporných čísel pomocou doplnkového kódu zľava dopĺňajú jednotky). Bity, ktoré na opačnej strane „vypadávajú“, sa strácajú.

Príklad použitia oboch operandov:

```
int n = 0x2AE3;
int p = n << 2;
int q = n >> 3;
printf(„p = 0x%X, q = 0x%X\n“, p, q);
```

Po prebehnutí tohto úseku programu budú mať premenné p a q hodnoty 0xAB8C a 0x055C. Je dobré si nakresliť bitovú reprezentáciu premennej n a potom ju posúvať doľava alebo doprava. Je zrejmé, že pri posune vpravo sa tri pravé bity „stratili“, pri posune vľavo sa sprava prisunuli nuly.

Posun vľavo o jeden bit je vo všeobecnosti ekvivalentný násobeniu dvoma, posun vpravo o jeden bit zase celočíselnému deleniu dvoma. Tento fakt sa často využíva a prekladače samy výrazy typu `a * 2^n` a `a / 2^n` transformujú vo výslednom kóde na `a << n` a `a >> n`.

Ďalšími bitovými operátormi sú *operátor bitového logického súčinu* &, *bitového logického súčtu* | a *bitovej logickej nonekvivalencie* ^. Všetky tri operátory sú binárne, infixové a asociujú sa zľava doprava. Ich operandy musia byť celočíselné a výsledkom ich aplikácie je hodnota, ktorej jednotlivé bity vznikli realizovaním logických funkcií AND, OR a XOR na zodpovedajúce bity oboch operandov (t. j. prvý bit výsledku dostaneme aplikáciou príslušnej funkcie na prvé bity oboch operandov, druhý na druhé bity atď.). Typ výsledku je zhodný s typom operandov. V tabuľke 2 sú na zopakovanie uvedené funkčné hodnoty všetkých troch logických funkcií ($a_i$ a $b_i$ sú zodpovedajúce *i*-te bity oboch operandov).

Tabuľka č. 2

| ai | bi | ai&bi | ai\|bi | ai^bi |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Posledným z množiny bitových operátorov je *operátor bitovej negácie* ~. Tento operátor je unárny, prefixový a asociuje sa sprava doľava. Jeho operand musí byť podobne ako pri predchádzajúcich operátoroch celočíselný a výsledkom aplikácie tohto operátora je hodnota, ktorej jednotlivé bity sú bitové doplnky bitov operanda. Bitovým doplnkom sa myslí doplnok do najväčšieho jednobitového binárneho čísla, t. j. 1. V princípe ide teda o zmenu 0 na 1 a naopak, inak povedané, bitovú negáciu. Typ výsledku je rovnaký ako typ operanda.

Uvedieme si ešte jednoduchý príklad na posledné štyri operátory:

```
int a = 0x1234;
int b = a & 0x0F0F;
int c = a | 0x0F0F;
int d = a ^ 0x0F0F;
int e = ~a;
printf(„b = 0x%X, c = 0x%X\n“, b, c);
printf(„d = 0x%X, e = 0x%X\n“, d, e);
```

Po vykonaní programu budú mať premenné b, c, d a e po rade hodnoty 0x0204, 0x1F3F, 0x1D3B a 0xEDCB. Z tohto príkladu vidieť možný spôsob použitia bitových operátorov. Operátor & môžeme použiť, ak chceme vynulovať niektoré bity prvého operanda (bity, ktoré budú v druhom operande nulové, spôsobia vynulovanie príslušných bitov prvého operanda), operátor | použijeme pri nastavovaní niektorých bitov prvého operanda (tie musia byť v druhom operande jednotkové) a nakoniec operátor ^ použijeme v prípade, že chceme určité bity prvého operanda invertovať (tie budú opäť v druhom operande jednotkové).

## Relačné a logické operátory

Relačné operátory sú operátory, pomocou ktorých môžeme zistiť niektoré relácie medzi operandmi. C++ umožňuje pomocou operátorov zistiť nasledujúce relácie: *menší* <, *menší alebo rovný* <=, *väčší* >, *väčší alebo rovný* >=, *rovný* == a *nerovný* !=. Všetky operátory sú binárne, infixové a asociujú sa zľava doprava. Ich výsledkom je 1, ak je relácia splnená, a 0, ak relácia nie je splnená, typ výsledku je int. Takto je to vo väčšine starších prekladačov. V novších prekladačoch, ktoré implementujú C++ podľa normy ANSI (kodifikovanej pomerne nedávno), je typom výsledku nový zavedený typ bool a výsledkom sú hodnoty true alebo false. O tomto type si však povieme až v budúcnosti. Operátory <, >, <= a >= majú vyššiu prioritu ako operátory == a !=.

Dozaista ste si všimli, že operátor priradenia je = a operátor porovnania je ==. V jazyku Pascal je, naopak, operátor porovnania = a operátor priradenia :=. Ak ste náhodou zvyknutí na Pascal, treba si dávať pozor a nezamieňať operátory medzi sebou, pretože to obyčajne vedie k chybám programu. Mimochodom, podľa niektorých zdrojov dôvodom odlišnosti operátorov C++ od operátorov Pascalu je fakt, že operácia priradenia sa v programe vyskytuje podstatne častejšie, a preto je vyjadrená jedným znakom na rozdiel od operácie testu na rovnosť, ktorá je vyjadrená dvoma znakmi.

Asociácia týchto operátorov nemá veľmi praktický význam, pretože výraz `a < b < c` sa vyhodnotí ako `(a < b) < c`, a nie ako `(a < b)` a *súčasne* `(b < c)`. To znamená, že s hodnotou c sa porovnáva výsledok operácie `a < b`, čo je 0 alebo 1.

Hlavnou oblasťou použitia výrazov s relačnými operátormi sú príkazy obsahujúce nejakú podmienku (príkazy vetvenia alebo cyklu). Keďže sme si zatiaľ o príkazoch nehovorili, v nasledujúcom príklade si vypíšeme výsledky výrazov obsahujúcich relačné operátory ako celé čísla:

```
int a = 10;
int b = 20;
printf(„a = %i, b = %i\n“, a, b);
printf(„a == b --> %i\n“, a == b);
printf(„a != b --> %i\n“, a != b);
printf(„a <  b --> %i\n“, a <  b);
printf(„a <= b --> %i\n“, a <= b);
printf(„a >  b --> %i\n“, a >  b);
printf(„a >= b --> %i\n“, a >= b);
```

Vyskúšajte si zmeniť počiatočné hodnoty premenných a a b, aby ste videli reakciu jednotlivých operátorov.

Logické operátory súvisia čiastočne s relačnými, pretože umožňujú logické spájanie podmienkových výrazov obsahujúcich relačné operátory. C++ má tri logické operátory, a to *operátor logického súčinu* &&, *operátor logického súčtu* || a *operátor logickej negácie* !. Je veľmi dôležité odlišovať tieto operátory od bitových operátorov, ktoré síce vyzerajú podobne, ale majú odlišnú sémantiku použitia!

Operátory && a || sú binárne, infixové a asociujú sa zľava doprava. Operátor && vracia hodnotu 1, ak sú oba jeho operandy nenulové, inak vracia hodnotu 0. Naproti tomu operátor || vracia hodnotu 1, ak aspoň jeden z operandov je nenulový. Ak sú oba nulové, vracia hodnotu 0. Tieto operátory majú istú špecifickú vlastnosť, a síce tú, že garantujú poradie vyhodnocovania svojich argumentov. V praxi to vyzerá tak, že najprv sa vyhodnotí ľavý operand, ak je tento operand nulový (pri &&), resp. nenulový (pri ||), vyhodnocovanie sa ukončí, lebo pravý operand už výsledok nemôže ovplyvniť (zamyslite sa, prečo). V opačnom prípade sa vyhodnotí aj pravý operand. Vzhľadom na asociativitu oboch operátorov sa podobne podvýrazy $e_i$ výrazu $e_1$ && $e_2$ && ... && $e_n$ (resp. $e_1$ || $e_2$ || ... || $e_n$) vyhodnocujú zľava doprava a vyhodnocovanie sa ukončí pri prvom nulovom (resp.

nenulovom) podvýraze. Tento spôsob vyhodnocovania sa niekedy nazýva tzv. *lazy evaluation* (doslova lenivé vyhodnocovanie). Sluší sa ešte podotknúť, že operátor `&&` má väčšiu prioritu ako operátor `||` (ako je to zaužívané v logike).

Operátor `!` je unárny, prefixový a asociuje sa sprava doľava. Jeho výsledkom je hodnota `0`, ak je jeho operand nenulový, a hodnota `1`, ak je nulový; typ výsledku je `int` (v novších prekladačoch opäť `bool`). Na rozdiel od operátora bitovej negácie `~`, pre ktorý platí, že `~~a == a`, to pre operátor `!` neplatí.

V nasledujúcom príklade si demonštrujeme spomínané neúplné vyhodnocovanie:

```
int a = 4;
int b = 1;
printf(„a = %i, b = %i\n“, a, b);
int r = (a > 3) && (--b) && (a++);
printf(„a = %i, b = %i, r = %i\n“,
       a, b, r);
```

(Operátory `&&` a `||` majú nižšiu prioritu ako relačné operátory, takže zátvorky sú vlastne zbytočné a prispievajú iba k väčšej prehľadnosti.) Pri vyhodnocovaní výrazu s operátormi `&&` sa najprv vyhodnotí podvýraz `a > 3`, ktorý je nenulový, takže sa pokračuje vyhodnocovaním výrazu `--b`. Jeho výsledok je `0`, teda vyhodnocovanie celého výrazu sa zastaví a do premennej `r` sa uloží výsledná hodnota `0`. Tretí podvýraz `a++` sa nevyhodnotí a hodnota premennej `a` zostáva `4`. Ak si skúsite zmeniť počiatočnú hodnotu premennej `b`, uvidíte, že po prebehnutí programu bude mať `a` hodnotu `5` a `r` hodnotu `1`. Na domácu úlohu si skúste program modifikovať tak, aby používal operátory `||`. Samozrejme, treba upraviť jednotlivé podvýrazy.

## Ostatné operátory

Do poslednej skupiny som zahrnul všetky ostatné operátory, ktoré sa už nedajú zatrieďovať do samostatných skupín.

Ako prvé dva si uvedieme operátory, o ktorých sme už hovorili v časti venovanej ukazovateľom. Ide o *operátor získania adresy* `&` a *operátor dereferencie* `*`. Oba sú unárne, prefixové a asociujú sa sprava doľava. Operátor `&` slúži na získanie adresy svojho operanda, ktorým musí byť l-hodnota (alebo názov funkcie, o tom si povieme neskôr). Výsledkom je ukazovateľ, ukazujúci na daný operand, typ výsledku je „ukazovateľ na typ operanda“. Operátor `*` slúži na dereferenciu existujúceho ukazovateľa a jeho operandom musí byť pointer na typ rôzny od `void`. Výsledkom je l-hodnota, uložená na adrese, ktorá je obsahom daného ukazovateľa. Príklad použitia sme uviedli v predošlej časti.

Ďalším operátorom, o ktorom sme si už nepriamo hovorili, je *operátor indexovania poľa* `[]`. Tento operátor je binárny, ale nedá sa povedať, či je infixový alebo postfixový (skôr niečo medzi tým). Prvým operandom je názov poľa, druhým operandom, ktorý musí byť celočíselný, je príslušný index. Výsledkom je l-hodnota, predstavujúca príslušný prvok poľa, typ výsledku je zhodný s bázovým typom poľa. Neskôr, keď si budeme hovoriť o vzťahu polí a ukazovateľov, uvidíme, že môžeme oba operandy vymeniť a výraz bude stále syntakticky aj sémanticky správny (t. j. `a[3]` aj `3[a]` sú platné odkazy na štvrtý prvok poľa `a`) – je to trochu kuriozita, ale je to tak.

Pri volaní funkcií sa používa *operátor volania funkcie* `()`. Bližšie si o tejto téme povieme v časti venovanej funkciám.

V prípade, že potrebujeme explicitne zmeniť typ nejakého výrazu na iný, použijeme *operátor pretypovania* (type-casting). Jeho zápis vyzerá takto: *(typ) výraz*. Spôsob zmeny typu závisí od oboch typov a opíšeme si ho v časti venovanej konverziám medzi typmi. Operátor pretypovania je unárny, prefixový a asociuje sa sprava doľava. Príklad použitia je v odseku o aritmetických operátoroch, kde explicitne pretypujeme číslo 5 na typ `double`, aby sme výsledok delenia dostali takisto ako hodnotu typu `double`. Tento (klasický) spôsob pretypovania je prevzatý z jazyka C. C++ navyše pridáva druhý spôsob, podobný funkčnému volaniu: *typ(výraz)*. Pri tomto spôsobe musí byť výsledný typ vyjadrený jednoslovným názvom. Teda zatiaľ čo pretypovanie napríklad hodnoty `5` na typ `double` môžeme zapísať buď ako `(double)5`, alebo ako `double(5)`, pretypovanie ukazovateľa `p` na ukazovateľ na typ `double` môžeme zapísať iba ako `(double*)p`. Predstavme si, že `p` je typu `void*`, ale sme si istí, že ukazuje na oblasť pamäte, v ktorej je uložené číslo typu `double`. Potom k tomuto číslu môžeme pristupovať nasledujúcim výrazom: `*(double*)p = 3.14`.

Ďalším operátorom je *operátor zistenia veľkosti* `sizeof`. Je unárny, prefixový a asociuje sa sprava doľava. Jeho argumentom môže byť typ, ktorý musí byť uzavretý v zátvorkách, alebo výraz, ktorý v zátvorkách nemusí byť. Operátor `sizeof` vracia veľkosť svojho argumentu v bajtoch, pričom je zaručené, že `sizeof(char) == 1`. Argumentom nesmie byť názov funkcie alebo typ `void`. Typom výsledku je zvláštny typ `size_t`, definovaný v štandardnom hlavičkovom súbore `stddef.h` pomocou mechanizmu používateľskej definície typov, o ktorom sme si ešte nehovorili. Tento typ je však ekvivalentný niektorému celočíselnému typu, obyčajne `unsigned int`. Aplikovaním operátora `sizeof` na referenciu dostávame veľkosť objektu, na ktorý referencia odkazuje, aplikovaním na pole dostávame celkovú veľkosť poľa v bajtoch. Takto môžeme zistiť počet prvkov poľa `a` pomocou výrazu `sizeof(a) / sizeof(a[0])`.

Predposledným operátorom je *operátor podmieneného vyhodnotenia* `?:`. Tento operátor je ternárny a v podstate infixový, asociuje sa sprava doľava. Aj tento operátor garantuje poradie vyhodnocovania svojich operandov. Postup vyhodnocovania je nasledujúci: najprv sa vyhodnotí prvý operand. Ak je *nenulový*, vyhodnotí sa druhý operand, ktorého hodnota bude výsledkom celého výrazu. Ak je však prvý operand *nulový*, vyhodnotí sa tretí operand a jeho hodnota bude výslednou hodnotou celého výrazu. Zvláštnosťou je, že sa vždy vyhodnotí práve jeden z oboch operandov (druhého a tretieho). Tieto dva operandy musia mať zhodný typ alebo musia byť konvertovateľné na nejaký spoločný typ, ktorý je aj typom výsledku. Ak sú oba operandy l-hodnoty, je aj výsledok l-hodnotou. Použitie tohto operátora bude jasnejšie z príkladu, v ktorom chceme vypočítať maximum z dvoch hodnôt:

```
int a = 123;
int b = 456;
int max = (a > b) ? a : b;
printf(„max(a, b) = %i\n“, max);
```

Operátor `?:` má veľmi nízku prioritu, takže netreba uvádzať zátvorky, ale ak ich napíšeme, zápis výrazu sa stane oveľa prehľadnejším. Uvedieme si ešte jeden príklad, z ktorého bude zrejmá asociativita operátora, a to spôsob výpočtu minima, tentoraz však z troch čísel:

```
int a = 123;
int b = 456;
int c = 789;
int min = (a < b) ? ((a < c) ? a : c)
                  : ((b < c) ? b : c);
printf(„min(a, b, c) = %i\n“, min);
```

Zátvorky vyjadrujú spôsob asociácie jednotlivých podvýrazov. A na záver ešte jeden príklad, z ktorého bude zrejmé, že sa vyhodnocuje len jeden z oboch operandov:

```
int x = 100;
int y = 200;
char c = ‚X‘;
c == ‚X‘ ? x++ : y++;
printf(„x = %i, y = %i\n“, x, y);
```

Pri vykonaní programu sa v závislosti od obsahu premennej c inkrementuje buď premenná `x`, alebo premenná `y`.

Zostáva nám už len posledný operátor, a tým je tzv. *operátor čiarka* `,`. Tento operátor je binárny, infixový a asociuje sa zľava doprava. Pri jeho vyhodnocovaní sa najprv vyhodnotí ľavý operand, ktorého hodnota sa vzápätí zabudne, ďalej sa vyhodnotí pravý operand a jeho hodnota sa stane hodnotou celého výrazu. Typ výsledku je zhodný s typom pravého operanda; ak je tento operand l-hodnotou, je ňou aj výsledok celého výrazu. Všetky vedľajšie účinky ľavého operanda sa vykonajú ešte pred vyhodnotením pravého operanda. Operátor `,` sa používa vtedy, ak na mieste, kde sa vyžaduje jediný výraz, potrebujeme zapísať niekoľko výrazov, obyčajne s nejakými vedľajšími účinkami. Príklad použitia, z ktorého je zrejmý aj spôsob vyhodnocovania (nehľadajte v ňom nejaký hlbší zmysel, nie je tam):

```
int i, j, k;
k = (j = 8, j++, i = j - 1, 10);
printf(„i = %i, j = %i, k = %i\n“,
       i, j, k);
```

Operátor `,` má úplne najnižšiu prioritu, pri priraďovaní premennej `k` sú preto nevyhnutné zátvorky okolo priraďovaného výrazu. Pri jeho vyhodnocovaní sa do premennej `j` uloží hodnota `8`, tá sa následne inkrementuje na `9`, potom sa do premennej `i` uloží hodnota `j - 1`, t. j. `8`, a výslednou hodnotou, priradenou premennej `k`, bude posledný podvýraz, konštanta `10`. Premenné `i`, `j`, `k` budú mať teda po rade hodnoty `8`, `9`, `10`.

## Záver

Na záver si pozrite tabuľku 3, v ktorej sú všetky dnes preberané operátory usporiadané podľa priority spolu s uvedeným smerom asociácie. V tejto tabuľke nie sú všetky operátory C++, tie zvyšné (je ich už len pár) si vysvetlíme v príslušných častiach seriálu. Jednotlivé úrovne priorít sú horizontálne oddelené a operátory na jednej úrovni majú rovnakú asociativitu. Priorita operátorov klesá zhora nadol.

Táto časť seriálu bola trochu rozsiahlejšia, ale nemalo by asi zmysel rozdeliť ju na dve kratšie. Ako sa poniektorí z vás ozvali, vyhovovalo by im rýchlejšie tempo výučby, takže toto je malý ústretový krok, lebo je pravda, že za pol roka sme toho zatiaľ veľa neprebrali. V budúcej časti sa budeme venovať príkazom jazyka C++, aby ste konečne mohli začať písať aj nejaké použiteľné programy. A propos, ak by náhodou niekomu ešte robilo problémy vyskúšať si dnešné úseky programov, podotýkam, že je potrebné ich vložiť do funkcie `main()` a na začiatok treba vložiť hlavičkový súbor `stdio.h` (direktívou `#include`).

Tab. č.3

| Operátor | Význam | Asociativita |
|---|---|---|
| ++ | inkrementácia (postfix) | L -> P |
| -- | dekrementácia (postfix) | |
| () | funkčné volanie | |
| [] | indexovanie poľa | |
| ++ | inkrementácia (prefix) | P -> L |
| -- | dekrementácia (prefix) | |
| ! | logická negácia | |
| ~ | bitová negácia | |
| - | unárne mínus | |
| + | unárne plus | |
| & | získanie adresy | |
| * | dereferencia | |
| sizeof | zistenie veľkosti | |
| (typ) | pretypovanie | |

| | | |
|---|---|---|
| * | násobenie | L -> P |
| / | delenie | |
| % | zvyšok po delení | |
| + | sčítanie | L -> P |
| - | odčítanie | |
| << | bitový posun vľavo | L -> P |
| >> | bitový posun vpravo | |
| < | menší | L -> P |
| <= | menší alebo rovný | |
| > | väčší | |
| >= | väčší alebo rovný | |
| == | rovný | L -> P |
| != | nerovný | |
| & | bitový AND | L -> P |
| ^ | bitový XOR | L -> P |
| \| | bitový OR | L -> P |
| && | logický AND | L -> P |
| \|\| | logický OR | L -> P |
| ?: | podmienené vyhodnocovanie | P -> L |
| = | priradenie | P -> L |
| op= | zložené priradenie | |
| , | čiarka | L -> P |

## Siedma časť: PRÍKAZY

Náš seriál má za sebou prvých šesť častí, teda prvého pol roka. Doposiaľ sme si povedali o tom, z akých stavebných jednotiek sa program v C++ skladá, aká je jeho základná štruktúra, opísali sme si typy údajov, s ktorými môžeme pracovať, a naposledy aj operácie, ktoré nad týmito údajmi môžeme vykonávať. Na to, aby sme mohli začať písať trochu zmysluplnejšie programy ako tých pár jednoduchých účelových príkladov, na ktorých sme si ukázali to, čo sme prebrali, nám chýba ešte niečo. To niečo, ako ostatne vyplýva z názvu dnešnej časti, sú príkazy jazyka C++.

Ako sme si povedali v jednej z predchádzajúcich častí, program je postupnosť lexikálnych jednotiek. Takto podaná definícia sa bude veľmi páčiť kompilátoru, programátor má však na svoj výtvor trochu iný pohľad. Dá sa povedať, že program je postupnosť príkazov. Do určitej miery je to pravda, ale o niečo presnejšia definícia hovorí, že program v C++ je postupnosťou deklarácií. Ak tomu zatiaľ veľmi nerozumiete, nič si z toho nerobte, časom sa vám to vyjasní. Na upokojenie si môžeme povedať, že v rámci jednej funkcie je program naozaj tvorený postupnosťou príkazov.

Príkaz je teda určitý spôsob, ktorým hovoríme počítaču, čo od neho chceme. V C++ existuje viacero druhov príkazov, napočudovanie ich však zase nie je až tak veľa, čo hádam poteší tých, ktorí po prečítaní predošlej časti nadobudli predstavu, že C++ obsahuje obrovské množstvo ťažko zapamätateľných operátorov a len zanietený nadšenec je schopný naučiť sa ich všetky s úspechom používať. Ak náhodou patríte medzi nich, povzbudím vás správou – usmievajte sa, operátorov v C++ je ešte viac…

Takže príkazy v C++ môžeme rozdeliť takto: *výrazový príkaz, deklaračný príkaz, zložený príkaz, príkazy výberu, príkazy cyklu* a *skokové príkazy*. Postupne si ich opíšeme v nasledujúcich odsekoch.

### Výrazový príkaz

S týmto príkazom ste sa už veľakrát stretli, aj keď o tom možno vôbec neviete. Jeho syntax je veľmi jednoduchá, je tvorený ľubovoľným výrazom, za ktorým nasleduje bodkočiarka `;`. Mimochodom, v C++ takmer každý príkaz treba ukončiť bodkočiarkou, inak bude kompilátor dôrazne protestovať. Dôsledkom vykonania výrazového príkazu je vyhodnotenie daného výrazu. Je pochopiteľné, že zmysel majú iba také výrazy, ktoré majú nejaký účinok na stav programu, teda priraďovacie výrazy, ako aj všetky výrazy s vedľajším účinkom (spomeňme si napr. operátory inkrementácie/dekrementácie, `++` a `--`) a ďalej volania funkcií. Podľa môjho skromného odhadu je väčšina príkazov v programe práve výrazových. Myslím, že pre tento druh príkazu netreba uvádzať nijaký príklad, stačí sa pozrieť na všetky doterajšie programy a „programčeky“, ktoré realizovali nejaký výpočet.

Je samozrejme možné vo výrazovom príkaze použiť výraz, ktorý nijako neovplyvní stav programu (= stav premenných programu), napr. `5 + 9;`, ale takýto príkaz teoreticky nemá zmysel (a šikovný prekladač ho v rámci optimalizácie vôbec nepreloží). Prakticky sa však používa (teda aspoň ja ho tak používam) v prípade, že deklarujete nejakú premennú, ktorej trebárs aj priradíte nejakú hodnotu, ale ďalej túto premennú v kóde nikde nepoužijete. Takáto situácia je úplne bežná počas vývoja programu, keď tá časť kódu, ktorá bude pracovať s danou premennou, ešte nie je napísaná. Prekladač v rámci zbežnej kontroly toho, čo prekladá (samozrejme, podľa nastavenia), vydá varovanie, že tá a tá premenná bola deklarovaná, ale nikde sa nepoužíva. Je to od neho isto milé, ale keď dookola prekladáte nejaký kus kódu a on vám dookola hlási to isté, po chvíli vás to prestane baviť. Toto konkrétne varovanie možno potlačiť – buď niekde v nastaveniach prekladaného projektu, alebo priamo v kóde – obyčajne pomocou krkolomného zápisu zhruba v tvare `#pragma warn -aus`, ale pri prvom spôsobe riskujete, že neskôr zabudnete nastavenie prepnúť späť (a je vhodné mať pri ladení programu zapnutých čo najviac varovaní – nikto nie je dokonalý a občas sa vám veru stane, že prehliadnete aj úplne drobnú a očividnú, ale z hľadiska správneho behu programu obyčajne tú najkritickejšiu chybu). A druhý spôsob, no uznajte sami, kto by si to pamätal! Pritom stačí použiť inkriminovanú premennú v nejakom triviálnom výraze bez vedľajších účinkov, teda napr. `a;`, a prekladač bude spokojný. Treba mať však vypnutú optimalizáciu (čo sa ináč pri ladení odporúča).

Zvláštnym typom výrazového príkazu je *prázdny príkaz*. Tento príkaz vznikne vtedy, ak z výrazového príkazu ten výraz jednoducho vynecháme. Zostane, pravda, tá bodkočiarka na konci. Prázdny príkaz sa používa v určitých situáciách, o ktorých si povieme o chvíľu. V podstate môže byť celá funkcia tvorená prázdnymi príkazmi, teda napr.

```
void dummy()
{
    ;
    ;
    ;
}
```

ale uznajte sami, veľké použitie by asi nemala (ak potrebujeme naozaj prázdnu funkciu – v niektorých prípadoch sa pre takúto funkciu používa názov stub, stačí ju zapísať takto: `void dummy(){}`).

### Deklaračný príkaz

Ani deklaračný príkaz vám nie je úplne neznámy. Tento príkaz je podobný výrazovému príkazu, len namiesto výrazu obsahuje deklaráciu. Deklarácie, to je tá hmlistá oblasť, ktorej sa stále dotýkam len tak okrajovo, lebo ide o mimoriadne zamotanú tému a budeme sa jej venovať v samostatnej časti seriálu. Takže zatiaľ berte deklaračný príkaz ako príkaz, pomocou ktorého zavádzame do programu nový identifikátor (teda napr. novú premennú, ale môže to byť aj nová funkcia alebo konštanta, dokonca ani nemusí byť úplne nová, deklarovať treba aj niektoré existujúce, ale v danom úseku implicitne neviditeľné objekty … a stačí, lebo úplne odbočíme od pôvodnej témy). Za povšimnutie stojí fakt, že sa deklaračný príkaz (ako ostatne každý iný) môže nachádzať na ľubovoľnom mieste vo funkcii. Ten, kto pozná jazyk C, si isto spomenie, že v ňom boli deklarácie chápané inak ako príkazy a museli byť všetky zhromaždené na začiatku funkcie. C++ našťastie toto nezmyselné obmedzenie zrušilo a umožnilo tak deklarovať premenné tam, kde sa naozaj používajú (resp. začínajú používať) – predstavte si, že máte dlhú funkciu, na jej začiatku deklaráciu nejakej premennej, ktorú používate až na konci funkcie. Ako málo chýba k tomu, aby ste po chvíli stratili prehľad, čo k čomu patrí a čo od čoho závisí. Takto si deklarujete danú premennú pekne na konci funkcie, máte ju „na očiach“ a viete, ku ktorej časti kódu patrí. Z tohto dôvodu dôrazne odporúčam všetkým, aj ortodoxným céčkarom, deklarujte premenné tam, kde ich začínate používať!

### Zložený príkaz

Tretím typom príkazu v C++ je tzv. zložený (compound) príkaz. Niekde sa možno stretnúť s pojmom príkazový blok alebo blokový príkaz, ale slovo „zložený“ lepšie vyjadruje jeho podstatu. Zložený príkaz vyzerá takto: ľavá krútená zátvorka `{`, ľubovoľný (aj nulový) počet iných príkazov (ľubovoľného typu) a nakoniec pravá krútená zátvorka `}`. Takýto príkaz navonok vystupuje ako jediný, hoci vnútri je vlastne tvorený niekoľkými príkazmi. Primárny zmysel je zrejmý – všade tam, kde syntax jazyka C++ povoľuje jediný príkaz (kde to je, to sa dozvieme v ďalších odsekoch), musíme v prípade, že potrebujeme vykonať viacero príkazov, použiť jeden zložený. Okrem toho pre deklarácie premenných vnútri zloženého príkazu platia určité pravidlá, ale o tom teraz hovoriť nebudeme. Príklad úseku programu so zloženým príkazom:

```
double s = 3800.0;
{
    double v, t = 12.58;
    v = s / t;
    printf(„v = %lg\n“, v);
}
s += 1218.0;
```

Z tohto príkladu nie je síce jasné, prečo sa zložený príkaz vôbec používa, ale vidno na ňom, ako sa takýto príkaz zapisuje, a čo je hádam najdôležitejšie, že za zloženým príkazom sa *nepíše* bodkočiarka. Ak ju tam dáme, vytvoríme dva príkazy – jeden zložený a hneď za ním jeden prázdny, čo v mieste, kde je povolený len jediný príkaz, spoľahlivo povedie k problémom.

Tu si dovolím malú kultúrnu vložku. Je nepísaným pravidlom, že v záujme čitateľnosti zdrojové texty dodržiavajú určité formátovanie. Samozrejme, je na každom, aby si zvykol na jeden štýl, ktorý mu vyhovuje, a ten používal. Patrí však k dobrým zvykom krútené zátvorky okolo zložených príkazov a/alebo funkcií zapisovať na samostatný riadok a vnútorné (vnorené) príkazy odsadzovať príslušným počtom medzier podľa úrovne vnorenia. Jednotkou odsadenia bývajú hádam najčastejšie štyri medzery (teraz sa na mňa zosype spŕška kritiky – dobre, niekedy aj dve medzery alebo osem, alebo jeden tabulátor, ale myslím, že štyri medzery je rozumný kompromis medzi prehľadnosťou programu a zbytočným plytvaním miestom). Niektorí programátori prvú krútenú zátvorku píšu na koniec predchádzajúceho riadka (teda nie pri takom zloženom príkaze, ako sme uviedli vyššie, tam to nemá zmysel, ale napr. pri definícii funkcie), tento spôsob je však podľa mňa o niečo menej prehľadný.

### Príkazy výberu

Priznám sa, veľmi sa mi nepáči pojem príkazy výberu, možno by bolo lepšie nazvať ich príkazmi vetvenia, ale asi najlepšie ich funkciu vystihuje anglický termín selection statements. A propos, pre tých, ktorí nevedia aspoň trochu po anglicky, mám zlú správu – rovno môžete prestať čítať. Prepáčte mi tú priamočiarosť, ale myslím, že programátor, ktorý nevie po anglicky,

je vopred odsúdený do roly bezvýznamného štatistu. Prakticky absolútna väčšina všetkých materiálov k vývojovým nástrojom, novým technológiám a podobne je v angličtine a na Internete sa so slovenčinou tiež ďaleko nedostanete (pokiaľ sa neobmedzujete na čítanie slovenských „sajtov“ a/alebo sťahovanie obrázkov slečien v intímnych polohách). Takže ak pociťujete medzery vo vzdelaní, nestrácajte čas.

Aby sme sa však vrátili k téme, príkazy výberu sú v C++ dva, a to príkaz `if` a príkaz `switch`. Slovko „výber“ sa vzťahuje na výber cesty, ktorou sa bude program uberať v závislosti od nejakých podmienok.

**Príkaz** `if`

Príkaz `if` umožňuje podmienené vykonávanie určitého úseku programu na základe splnenia alebo nesplnenia zadanej podmienky. Existuje v dvoch formách:

```
if (podmienka)
     príkaz
```

a

```
if (podmienka)
     príkaz1
else
     príkaz2
```

Prvá z oboch uvedených foriem spôsobí, že procesor vyhodnotí podmienku uzavretú v zátvorkách, a ak je jej hodnota *nenulová*, vykoná uvedený *výkonný príkaz*. Ak je podmienka nulová, pokračuje sa nasledujúcim príkazom. Tu vidíme prvý prípad, keď syntax jazyka C++ povoľuje len jeden príkaz. Ak chceme realizovať viacero príkazov, použijeme spomínaný zložený príkaz. Je lepšie, keď výkonný príkaz (jednoduchý či zložený) píšeme na samostatný riadok – pri krokovaní programu tak vieme, či bola podmienka splnená, alebo nie. Jednoduchý príkaz obyčajne odsadíme, pri zloženom odsadíme len vnorené príkazy, obe zátvorky ponecháme v jednej úrovni s doterajším textom. Prirodzene, ak sa vám takýto spôsob zápisu zdrojového textu nepáči, vytvorte si vlastný, ale pokiaľ možno konzistentný a nemenný.

V nasledujúcom príklade sa inkrementuje hodnota premennej `x` a v prípade, že dosiahne hodnotu `640`, vynuluje sa a inkrementuje sa hodnota inej premennej `y`. Obe premenné `x` a `y` môžeme brať napríklad ako súradnice nejakého objektu pohybujúceho sa po riadkoch štandardnej VGA obrazovky. Keď objekt prejde cez pravý okraj, objaví sa na ľavom okraji o riadok nižšie:

```
x++;
if (x == 640)
{
     x = 0;
     y++;
}
```

V prípade, že výkonným príkazom príkazu `if` je iný ako zložený príkaz, musí za ním nasledovať bodkočiarka. Za zloženým príkazom sa, naopak, bodkočiarka *nepíše*. Výkonným príkazom môže byť hocijaký príkaz, teda aj ďalší príkaz `if`, takáto konštrukcia je potom, samozrejme, ekvivalentná jedinému príkazu `if` s podmienkami spojenými operátorom logického súčinu. Na mieste podmienky môže stáť ľubovoľný výraz aritmetického typu alebo typu ukazovateľ, rozhodujúca je jeho hodnota (nulová/nenulová). Tento fakt môžeme s úspechom využiť a prepísať uvedený príklad takto:

```
if (++x == 640)
{
     ...
}
```

Pri vyhodnocovaní výrazu sa inkrementuje premenná `x`, potom sa jej hodnota porovná s konštantou `640`. V prípade rovnosti je celý výraz nenulový a vykoná sa príslušný zložený príkaz. Takýto spôsob zápisu je pre C++ typický a za cenu trošku ťažšej zrozumiteľnosti zdrojového textu napomáha jeho skrátenie a zefektívnenie jeho zápisu. Mimochodom, skúste sa zamyslieť, prečo v tomto príklade nemôžeme použiť postfixový variant operátora `++`.

Ak prvá forma príkazu `if` predstavovala akúsi jednocestnú výhybku, druhá forma umožňuje výber z dvoch alternatív, z ktorých sa vykoná práve jedna. Pri vykonávaní tejto druhej formy procesor opätovne vyhodnotí podmienku a v prípade, že jej hodnota bude nenulová, vykoná *príkaz*$_1$, po jeho skončení pokračuje prvým príkazom za celou konštrukciou `if`. V opačnom prípade, t. j. hodnota podmienkového výrazu je nulová, procesor vykoná *príkaz*$_2$. Pre oba príkazy platí to, čo sme už uviedli – ak sú jednoduché, píše sa za nimi bodkočiarka, ak sú zložené, bodkočiarka sa nepíše (a pre príkaz$_1$ sa ani nesmie!).

Niekedy potrebujeme vetviť program podľa viacerých alternatív. Jednou z možností je nasledujúca konštrukcia (nie je ničím výnimočná, ide v skutočnosti o sériu dvojcestných príkazov `if`):

```
if (podmienka1)
     príkaz1
else if (podmienka2)
     príkaz2
else if ...
     ...
else
     príkazn
```

Keby sme striktne dodržiavali odsadzovanie jednotlivých blokov podľa spomenutých pravidiel, dlhšie konštrukcie by boli príliš posunuté vpravo, preto sa takáto séria príkazov častejšie formátuje tak ako v našom príklade. Sémantika tejto konštrukcie je jednoduchá – postupne sa vyhodnocujú jednotlivé podmienkové výrazy, a keď sa narazí na prvý, ktorého hodnota je nenulová, vykoná sa jeho príslušný výkonný príkaz. V prípade, že ani jedna z podmienok nie je splnená, vykoná sa *príkaz*$_n$ (ak ho uvedieme; samozrejme, nemusí tam byť, v takom prípade sa jednoducho pokračuje prvým príkazom za celou konštrukciou).

Pri používaní druhej formy príkazu `if` môže dôjsť k nasledujúcej situácii. Majme úsek kódu:

```
if (podmienka1)
     if (podmienka2)
          príkaz1
     else
          príkaz2
```

Vidíme, že tento úsek obsahuje dva príkazy `if`, z ktorých len jeden má aj časť `else`. Otázkou tu je ktorý. Ako ste si zrejme domysleli z formátovania celého príkladu, uvedená časť `else` patrí k druhému z oboch `if`. Pri realizácii tohto kódu procesor najprv vyhodnotí *podmienku*$_1$. Ak je jej hodnota nenulová, začne vykonávať druhý príkaz `if` (štandardným spôsobom). Ak je však jej hodnota nulová, program pokračuje nasledujúcim príkladom za celým uvedeným úsekom. Ak chceme, aby sa časť `else` vzťahovala na prvý `if`, musíme ten druhý (vnorený) uzavrieť do krútených zátvoriek a vytvoriť z neho zložený príkaz (hoci `if` sám osebe je príkazom jednoduchým):

```
if (podmienka1)
{
     if (podmienka2)
          príkaz2
}
else
     príkaz1
```

Vyskúšajte si oba tvary (so zátvorkami aj bez), aby ste sa presvedčili o ich funkcii.

Na záver si uvedieme krátky príklad použitia druhej formy príkazu `if`. Predpokladajme, že v premennej `choice` typu `char` je písmeno predstavujúce voľbu používateľa z nejakého menu, ktoré umožňuje vytvorenie nového objektu a modifikáciu alebo zmazanie existujúceho objektu (napríklad položky v nejakej databáze):

```
char choice;

...  // výber z menu

if (choice == ‚C‘)
     Create();
else if (choice == ‚M‘)
     Modify();
else if (choice == ‚D‘)
     Delete();
else
{
     Beep();
     PrintError(„Unknown choice!“);
}
```

Funkcie `Create()`, `Modify()`, `Delete()` realizujú príslušné operácie s objektmi, funkcia `Beep()` vydá varovné pípnutie a `PrintError()` vypíše chybovú správu.

**Príkaz** `switch`

Druhým príkazom výberu je príkaz `switch`. Tento príkaz má podobnú funkciu ako zreťazenie viacerých dvojcestných príkazov `if`, na rozdiel od nich však umožňuje vetvenie na základe jedinej podmienky. Jeho syntax je v podstate veľmi jednoduchá:

```
switch (výraz)
     príkaz
```

Výkonný príkaz príkazu `switch` je prakticky vždy zložený (inak nemá príkaz `switch` vôbec význam). Ako sme si povedali, zložený príkaz predstavuje postupnosť iných príkazov. V rámci príkazu `switch` môžu byť jednotlivé vnorené príkazy označené pomocou špeciálnych návestí v tvare:

```
{
     ...
case konštantný-výraz:
     príkaz
     príkaz
     ...
}
```

Samozrejme, že nie každý príkaz musí mať pred sebou návestie. Každé návestie sa skladá z kľúčového slova `case`, za ktorým nasleduje *konštantný výraz* a dvojbodka. Rôzne návestia musia mať rôzne konštantné výrazy! Okrem toho sa môže v zloženom príkaze vyskytovať najviac jedno zvláštne návestie `default:`.

Pri realizácii príkazu `switch` sa vyhodnotí jeho podmienkový *výraz*. Tento výraz musí byť celočíselného typu. Potom sa vypočítaná hodnota porovnáva s konštantnými výrazmi v jednotlivých návestiach zloženého príkazu. Ak sa s nejakým konštantným výrazom zhoduje, program bude pokračovať príkazmi nasledujúcimi za daným návestím. Ak sa nezhoduje so žiadnym výrazom, budú sa vykonávať príkazy nasledujúce za návestím `default:` (ak tam je; ak nie, pokračuje sa nasledujúcim príkazom za príkazom `switch`). Pozor! Po skoku riadenia za príslušné návestie bude procesor vykonávať všetky nasledujúce príkazy nezávisle od ďalších prípadných návestí až po koniec celého zloženého príkazu. Ak chceme tento zložený príkaz ukončiť skôr, musíme použiť kľúčové slovo `break`, ktoré spôsobí okamžité vyskočenie z aktuálneho bloku, v našom prípade zo zloženého príkazu príkazu `switch`.

Najlepšie hádam bude ozrejmiť si celú vec na príklade. Predpokladajme, že v premenných `a` a `b` sú dve čísla, s ktorými chceme vykonať jednu z aritmetických operácií na základe voľby používateľa. Tu je daný kód:

```
double a, b, res;
char op;

...  // voľba operácie

switch (op)
{
case ,+‘:
     res = a + b;
     break;
case ,-‘:
     res = a - b;
     break;
case ,*‘:
     res = a * b;
     break;
case ,/‘:
     if (b != 0.0)
     {
          res = a / b;
          break;
     }
default:
     error();
}
```

Premenná `op` by mala obsahovať jednu z hodnôt `,+‘`, `,-‘`, `,*‘` alebo `,/‘`. Podľa jej obsahu sa vyberie príslušné návestie a vykoná sa daná operácia. Výsledok sa uloží do premennej `res`. V prípade, že je v premennej `op` iný znak ako jeden zo štyroch uvedených, zavolá sa funkcia `error()`. Obsah `res` v takom prípade zostane nezmenený. Všimnite si v tomto príklade, že v prípade požiadavky na vydelenie oboch premenných sa testuje, či premenná `b` (t. j. deliteľ) nie je náhodou nulová. Ak nie je, delenie sa vykoná a príkaz sa ukončí kľúčovým slovom `break`. Čo sa však stane vtedy, keď `b` bude obsahovať hodnotu `0.0`? Zložený príkaz príkazu `if` sa nevykoná a bude sa pokračovať ďalším príkazom. Tentoraz ním však nie je príkaz `break`, ale (návestím `default:` označené) volanie funkcie `error()`. Inými slovami, funkcia `error()` sa vyvolá nielen pri chybnom kóde požadovanej operácie, ale aj pri požiadavke delenia nulovým deliteľom.

### Záver

Rozprávanie o príkazoch jazyka C++ si dokončíme nabudúce, keď si povieme o príkazoch cyklu a o skokových príkazoch. Potom si uvedieme nejaký väčší príklad, v ktorom použijeme maximum z toho, čo sme doteraz prebrali. Jemne sa dotkneme aj problematiky vstupu údajov z klávesnice, aby náš program bol aspoň trochu interaktívny. Potom nám z neobjektovej časti C++ zostáva prebrať deklarácie a všetky záludnosti s nimi spojené (okrem iného rozsah platnosti, viditeľnosti, ukladaciu triedu, linkovanie, používateľské typy), štandardné konverzie, dynamickú alokáciu pamäte a ešte niečo o funkciách. Keď sa tým všetkým úspešne prepracujeme, povieme si o štandardnej knižnici jazyka C (o základných funkciách, ktoré poskytuje) a potom už sa vrhneme na prostriedky, ktoré C++ poskytuje na podporu objektovoorientovaného programovania.

## Ôsma časť: PRÍKAZY (dokončenie)

V predošlej časti sme začali rozprávanie o príkazoch jazyka C++, opísali sme výrazové, deklaračné, zložené príkazy a príkazy výberu (vetvenia). Ako som sľúbil, dnes si opis doplníme ešte príkazmi cyklu a skokovými príkazmi.

### Príkazy cyklu

Príkazy cyklu (iteration statements) slúžia na vyjadrenie opakovania vykonávania určitej časti programu. Sú tri: príkaz `while`, príkaz `do` a príkaz `for`.

**Príkaz** `while`

Syntax tohto príkazu je nasledujúca:

```
while (podmienka)
     príkaz
```

*Podmienkou* je podobne ako v príkazoch výberu ľubovoľný výraz aritmetického typu alebo typu ukazovateľ. Tento výraz sa vyhodnotí; ak je jeho hodnota nenulová, vykoná sa *príkaz*. Opäť sa vyhodnotí výraz, ak je nenulový, opäť sa vykoná *príkaz* a celý cyklus sa opakuje, až kým sa nestane, že bude hodnota podmienkového výrazu nulová. Vtedy sa cyklus ukončí a pokračuje sa nasledujúcim príkazom za príkazom `while`. Výkonný príkaz príkazu `while` môže byť, samozrejme, zložený.

Tento druh cyklu predstavuje klasickú konštrukciu, známu z teórie štruktúrovaného programovania, s testom ukončovacej podmienky pred vykonaním výkonného príkazu. To znamená, že v prípade, ak je hodnota podmienkového výrazu rovná nule, príkaz sa vôbec nevykoná. Ako príklad si uvedieme jeden z možných spôsobov výpisu čísel od 1 do 10 spolu s ich druhými mocninami:

```
int i = 0;
while (++i <= 10)
     printf(„%2i - %3i\n“, i, i * i);
```

S formátovacím reťazcom uvedeným ako prvý argument funkcie `printf` si zatiaľ príliš hlavu nelámte, čísla, ktoré sú v ňom „navyše“, slúžia v podstate na zarovnanie vypisovaných čísel.

Všimnite si na tomto príklade, že na začiatku inicializujeme hodnotu premennej `i` nulou a v priebehu cyklu ju postupne inkrementujeme v rámci podmienkového výrazu. Po prvom vyhodnotení tohto výrazu bude `i` mať hodnotu `1`, t. j. počiatočnú hodnotu, ktorú potrebujeme, môžeme ju teda vypísať spolu s jej druhou mocninou. Po každom ďalšom teste bude `i` o jednotku väčšie. Po vypísaní príslušného riadka pre `i = 10` sa opäť vyhodnotí podmienkový výraz. Keďže je použitá prefixová verzia operátora `++`, premenná `i` sa najprv inkrementuje na hodnotu `11` a až potom sa porovná s hraničnou hodnotou nášho cyklu, čo je `10`. Podmienka, samozrejme, už nebude splnená a celý cyklus sa teda skončí. Náš príklad je možné napísať aj iným spôsobom (a nielen jedným), napríklad s použitím postfixovej verzie (nie príliš výhodné, na ďalšom riadku bude treba všade namiesto `i` písať `i - 1`) alebo sa premenná `i` inicializuje na jednotku, v teste sa zmení operátor `<=` na `<` (ale bude treba použiť postfixový `++`, a teda platí predchádzajúca poznámka) atď. Kto má chuť, môže vymýšľať ďalej (prípadne chvíľu počká, kým si uvedieme aj príkaz `do`; s ním sa možnosti ešte znásobia).

(Poznámka: Podľa súčasného návrhu normy ANSI C++ je možné namiesto podmienkového výrazu uviesť aj deklaračný príkaz s tým, že deklarovaná premenná má rozsah platnosti obmedzený na príkaz `while`. Keďže však mnoho dnes dostupných prekladačov túto a rôzne iné nové vlastnosti neimplementuje, nebudeme si o nich zatiaľ hovoriť, hádam niekedy na konci celého seriálu. Situácia okolo kodifikácie C++ normou ANSI/ISO je pomerne zložitá a ja sám v nej nemám momentálne úplne jasno.)

**Príkaz** `do`

Tento príkaz je veľmi podobný prechádzajúcemu, dokonca majú spoločné jedno kľúčové slovo. Tu je jeho syntax:

```
do
     príkaz
while (podmienka);
```

Sémantika príkazu `do` je prakticky zhodná so sémantikou príkazu `while`, s tým rozdielom, že teraz sa podmienkový výraz vyhodnocuje až po prvom vykonaní príkazu, môžeme si byť teda istí, že sa príkaz vykoná aspoň raz. Cyklus sa ukončí vtedy, keď bude hodnota podmienky nulová. Pozor na podobnosť so sémanticky takmer zhodnou konštrukciou – cyklom `repeat-until` z jazyka Pascal, ten sa ukončí v okamihu, keď podmienka bude splnená (čo v C++ znamená nenulová)!

Možno vás napadlo, prečo v opise syntaxe príkazu `do` je na konci bodkočiarka a pri príkaze `while` nie. V podstate sme si dôvod povedali minule, ale pre istotu to ešte raz objasním. Výkonným príkazom príkazu `while` je ľubovoľný príkaz. Ak je to zložený príkaz, začína a končí sa krútenými zátvorkami a nenasleduje za ním bodkočiarka. Za ostatnými typmi príkazov bodkočiarku píšeme (celá situácia je trochu komplikovanejšia, zapamätajte si jednoducho, že bodkočiarku treba písať všade okrem miesta za pravou krútenou zátvorkou; ak by sa vám pri striktnom dodržiavaní syntaxe stalo, že dostanete dve či viac bodkočiarok za sebou, stačí, prirodzene, len jedna). Zápis príkazu `while` sa končí zápisom jeho výkonného príkazu (bez bodkočiarky). Ak je zložený, bodkočiarka sa nenapíše ani za ním. Príklad:

```
while (n != 255)
{
     s += n;
     n++;
}
```

Ak je výkonný príkaz napríklad výrazovým príkladom, bodkočiarka sa zaň píše (pozri odsek o príkaze `while`). Naproti tomu zápis príkazu `do` obsahuje výkonný príkaz akoby „v sebe“ a končí sa riadkom s testovaným výrazom. Keďže tento riadok nie je samostatným príkazom, na jeho konci dáme vždy bodkočiarku, ktorou ukončíme celý príkaz `do`. Tak, a je to. Prejdite si to pokojne viackrát, je to dosť komplikované. Najlepšie je pozrieť si presnú syntax jednotlivých príkazov a potom ich skladať metódou Cut & Replace.

Uvedieme si ešte príklad použitia príkazu `do`. Vo všeobecnosti je tento príkaz vhodný tam, kde potrebujeme pred vyhodnotením podmienky vykonať aspoň jednu iteráciu, napríklad preto, že podmienkový výraz závisí od výsledku vykonania tela cyklu. Predstavme si situáciu, keď chceme postupne spracovať reťazec znakov, ktorý predstavuje jednotlivé riadky textu, oddelené znakom `\n`. Postupne prechádzame reťazcom a každý znak nejakým spôsobom spracujeme. Chceme, aby sa prechádzanie ukončilo pri nájdení prvého znaku `\n`, ale súčasne chceme, aby tento znak tiež podliehal spracovaniu. Takýto prípad je ako stvorený na použitie príkazu `do`:

```
char msg[] =
     „First line\n“
     „Second line\n“
     „Third line\n“;
int i = 0;
do {
     printf(„%c“, toupper(msg[i]));
}
while (msg[i++] != ,\n‘);
```

Ak si chcete tento príklad vyskúšať, musíte na začiatok programu pridať direktívu `#include <ctype.h>`, ktorá umožní používanie funkcie `toupper()` (v skutočnosti je to makro). Táto funkcia prevedie svoj argument typu `char` na príslušné veľké písmeno (samozrejme, len ak sa to dá, inak ho ponechá bezo zmeny).

V príklade máme pole znakov `msg[]`, ktoré predstavuje tri riadky textu, oddelené znakom nového riadka. V cykle sa posúvame po jednotlivých znakoch poľa pomocou indexovej premennej `i`. Každý znak poľa prevedieme na veľké písmeno a vypíšeme. Potom otestujeme, či sme už náhodou neprečítali znak `'\n'`, súčasne inkrementujeme premennú `i`. Cyklus sa zrejme končí načítaním a vypísaním znaku `'\n'`. Po prebehnutí dostaneme na konzole výstup – prvý riadok textu z poľa `msg[]`, navyše prevedený na veľké písmená. Na príklade tiež vidno jeden z možných spôsobov formátovania zdrojového textu obsahujúceho príkaz `do` – ľavá krútená zátvorka je tu na rovnakom riadku ako kľúčové slovo `do` (pre jeho malú dĺžku).

**Príkaz `for`**

Predchádzajúce dva príkazy umožňovali jednoduchý zápis opakovania bloku príkazov s definovanou ukončovacou podmienkou. Nič viac a nič menej. Jazyk C++ sa však radí medzi vyššie programovacie jazyky, a preto by mal poskytovať predsa len o niečo väčší komfort pri zápise cyklov. Preto máme k dispozícii ešte tretí príkaz cyklu, ktorým je príkaz `for`. Tento príkaz existuje v mnohých bežných jazykoch, no v C++ patrí jeho sémantika k jednej z najlepších a najmocnejších. Jeho syntax je o niečo zložitejšia:

```
for (init-príkaz výraz1; výraz2)
    príkaz
```

V zápise príkazu `for` nachádzame štyri „úseky“: *inicializačný príkaz*, *výraz*$_1$, *výraz*$_2$ a samotný *výkonný príkaz*. Realizácia celého cyklu vyzerá takto: Najprv sa vykoná inicializačný príkaz. Tento príkaz nemôže byť hocaký, povolený je len výrazový alebo deklaračný príkaz. Následne sa testuje *výraz*$_1$, ktorý predstavuje ukončovaciu podmienku celého cyklu. Požiadavky naň sú zhodné s požiadavkami na podmienkové výrazy v príkazoch `while` a `do`. V prípade, že hodnota *výrazu*$_1$ je nenulová, vykoná sa výkonný príkaz a hneď za ním sa vyhodnotí *výraz*$_2$ (jeho hodnota sa nikde nepoužije). Opätovne sa vyhodnotí *výraz*$_1$ a celý cyklus sa opakuje až do chvíle, keď hodnota *výrazu*$_1$ bude nulová. Celý príkaz `for` možno opísať pomocou doteraz prebraných príkazov takto:

```
init-príkaz
while (výraz1)
{
    príkaz
    výraz2;
}
```

Inicializačný príkaz slúži obyčajne na inicializáciu premenných, ktoré sa budú v cykle používať. V prípade, že ním je deklaračný príkaz, slúži *výhradne* na deklaráciu premenných (prípadne spojenú s ich inicializáciou). Tieto premenné však môžeme používať nielen v samotnom príkaze `for`, ale aj v nasledujúcich príkazoch. (Podľa normy ANSI je platnosť takto deklarovaných premenných obmedzená na príkaz `for`.) V prípade, že inicializačným príkazom je výrazový príkaz, ten slúži len na modifikáciu existujúcich premenných. Nie je možné kombinovať deklaráciu premenných s priraďovacím výrazom, teda príkaz

```
int i = 0, j = 0;
```

deklaruje dve nové celočíselné premenné `i` a `j` a inicializuje ich hodnotou `0`. V prípade, že premenná `j` už existuje (teda bola deklarovaná), prekladač ohlási chybu. Naproti tomu príkaz

```
x = 999, y = 1;
```

nastavuje hodnoty premenných `x` a `y`, ktoré museli byť už predtým niekde deklarované. V prvom prípade je čiarka oddeľovačom jednotlivých deklarovaných identifikátorov, v druhom príklade je operátorom (hľa, jedno z jeho použití). Samozrejme, je potrebné inicializačný príkaz ukončiť bodkočiarkou (vždy, lebo nikdy nejde o zložený príkaz).

*Výraz*$_2$ je výrazom, ktorý sa vyhodnotí po každej iterácii cyklu. Obyčajne preto slúži na nejaký „posun“ riadiacich premenných cyklu. Uveďme si príklad. Najobyčajnejší cyklus, v ktorom budeme meniť hodnotu premennej `a` od `1` do `100`, zapíšeme takto:

```
for (int a = 1; a <= 100; a++)
{
    ...
}
```

Pred spustením cyklu sa deklaruje premenná `a`, jej hodnota sa nastaví na `1` a následne sa začnú vykonávať jednotlivé iterácie. Po prebehnutí každej z nich sa vyhodnotí výraz `a++`, ktorý spôsobí inkrementáciu premennej `a`. Cyklus sa ukončí po 100. iterácii, keď sa premenná `a` inkrementuje na hodnotu `101` a následný test `a <= 100` vráti nulovú hodnotu.

Výraz$_2$ môže byť, samozrejme, aj zložitejší. Napríklad majme pole celých čísel, ktorým chceme súčasne prechádzať z oboch koncov smerom k stredu. Použijeme dva indexy `i` a `j`, z ktorých jeden budeme inkrementovať a druhý dekrementovať. Cyklus ukončíme vtedy, keď sa oba indexy „stretnú“. Predpokladajme, že pole čísel je už nejakým spôsobom naplnené:

```
int num[50];
for (int i = 0, j = 49; i <= j; i++, j--)
  {
    ...
  }
```

V prípade, že by bol počet prvkov poľa nepárny, budú v poslednej iterácii oba indexy rovnaké a podľa toho, akú operáciu nad poľom vykonávame, budeme musieť doplniť potrebný test (príkaz `if`). Je možné zmeniť aj priamo podmienkový výraz príkazu `for`. Lenže čo v prípade, ak rozmer poľa vopred nevieme (dostaneme ho napríklad ako argument funkcie)? Vtedy musíme použiť v tele cyklu príkaz `if`.

Oba výrazy zo zápisu príkazu `for` môžeme aj vynechať. Vynechaním *výrazu*$_2$ sa na prvý pohľad pripravujeme o možnosť zmeny stavu riadiacich premenných cyklu, ale uvedomte si, že táto zmena sa v určitých prípadoch dá úspešne realizovať v rámci *výrazu*$_1$ či priamo v rámci výkonného *príkazu*. Ak vynecháme *výraz*$_1$, dostaneme cyklus, ktorého ukončovacia podmienka bude vždy splnená, čiže nekonečný cyklus. Ako z takého cyklu vyskočiť, dozviete sa o chvíľu. Existujú dva spôsoby zápisu (takmer) nekonečného cyklu:

```
while (1)
{ ... }
```

a

```
for (;;)
{ ... }
```

Oba sú úplne ekvivalentné a je len na vás, ktorý si zvolíte.

Výhodou takto definovaného príkazu `for` je možnosť realizácie aj iných druhov cyklov ako bežných lineárnych, pri ktorých sa niektorá premenná (alebo premenné) inkrementuje či dekrementuje. Napríklad chceme v priebehu cyklu vygenerovať všetky čísla $2^i$, $0 \le i \le 7$ (ako nejaké binárne masky, s práve jedným nastaveným bitom). Vo väčšine jazykov si musíme pomôcť prídavnou premennou `i`, ktorá sa bude meniť od `0` po `7` a v rámci cyklu budeme vypočítavanú hodnotu vždy zdvojnásobovať. V C++ je možné ako riadiacu premennú cyklu použiť priamo počítanú masku:

```
for (int m = 1; m <= 128; m *= 2)
{
    ...
}
```

Počas vykonávania cyklu nadobudne premenná `m` postupne hodnoty `1`, `2`, `4`, `8`, `16`, `32`, `64` a `128`, čo sú práve požadované mocniny dvojky.

## Skokové príkazy

Poslednou skupinou príkazov jazyka C++ sú skokové príkazy. Ich názov napovedá, čo bude ich náplňou. Pod skokom rozumieme okamžitú nepodmienenú zmenu postupnosti vykonávaných príkazov programu. Aj príkazy výberu a príkazy cyklu zahŕňajú zmenu toku programu, ale viac-menej podmienenú a akosi skrytú, takže pod skokovými príkazmi budeme rozumieť len tie, ktorými explicitne meníme lineárny beh programu.

Skokové príkazy sú v C++ štyri: `break`, `continue`, `return` a `goto`. Postupne si teraz opíšeme každý z nich.

**Príkaz `break`**

O tomto príkaze sme si už hovorili – bolo to v súvislosti s príkazom `switch`, kde príkaz `break` slúžil na opustenie bloku, tvoriaceho výkonný príkaz príkazu `switch`. Okrem toho `break` môžeme použiť aj v tele ľubovoľného príkazu cyklu. V oboch prípadoch je jeho účinok rovnaký, teda ukončenie daného príkazu, inak povedané, vyskočenie z jeho výkonného bloku. Pravda, treba vziať do úvahy fakt, že toto vyskočenie sa týka najbližšieho bloku smerom zvnútra von (trochu krkolomný opis, najlepšie celú situáciu opisuje anglický termín *smallest enclosing block*). Lepší bude príklad:

```
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        if (j == 8)
            break;
    }
    if (i == 5)
        break;
}
```

V tomto jednoduchom ad-hoc príklade máme dva do seba vnorené cykly. Vnútorný cyklus vo svojom tele neustále testuje, či jeho riadiaca premenná `j` nenadobudla hodnotu `8`. Keď k takej situácii dôjde, vykoná sa príkaz `break`, vnútorný cyklus sa ukončí, ale pokračuje sa ďalším príkazom tela vonkajšieho cyklu, čo je príkaz `if`, ktorý testuje pre zmenu zhodu riadiacej premennej `i` tohto cyklu s hodnotou `5`. Opäť, keď zhoda nastane, aktivuje sa druhý príkaz `break` a ten ukončí aj vonkajší cyklus. Je dobré celý príklad odkrokovať alebo doplniť o výpis premenných `i` a `j`, aby ste sa uistili o okamihoch, v ktorých dochádza k ukončeniu každého z cyklov.

Príkaz `break`, ako ste sa hádam dovtípili, slúži aj na ukončenie spomínaného nekonečného cyklu. Obyčajne sa ukončenie viaže na nejakú podmienku a takáto konštrukcia je vhodná, ak potrebujeme cyklus, ktorého ukončovacia podmienka sa nevyhodnocuje ani na začiatku cyklu (ako pri `while`), ani na konci (ako pri `do`), ale niekde v prostriedku. Potom taký cyklus zapíšeme napríklad takto:

```
while (1)
{
    ...
    if (podmienka)
        break;
```

```
    ...
}
```

Spôsobov zápisu jedného cyklu je veľa a je len na vás, ktorý si vyberiete z hľadiska efektivity výsledného kódu.

**Príkaz** `continue`

Tento príkaz má podobnú oblasť použitia ako príkaz `break`, na rozdiel od neho však je možné príkaz `continue` použiť iba v príkazoch cyklu. Jeho funkcia je jednoduchá – spôsobí okamžité ukončenie práve prebiehajúcej iterácie a pokračovanie opäť od začiatku daného cyklu. Podobne ako `break` aj `continue` sa vzťahuje na najbližší obklopujúci blok cyklu. Pri použití v rámci príkazu `for` sa predtým, než sa riadenie vráti späť na začiatok cyklu, vyhodnotí výraz$_2$, rovnako ako po každej dokončenej iterácii. V príklade, ktorý nasleduje, postupne vypisujeme všetky čísla od 1 do 100, ktoré nie sú násobkom čísla 13. Tie, ktoré násobkom sú, jednoducho preskočíme.

```
for (int x = 1; x <= 100; x++)
{
    if (!(x % 13))
        continue;
    printf(„%4i", x);
}
```

Všimnite si, ako testujeme čísla, ktoré sú násobkom čísla `13`. Také čísla po vydelení trinástkou dajú zvyšok `0`. Použijeme preto operátor modulo (`%`). Keďže podmienkový výraz v príkaze `if` má byť nenulový, ak chceme dané číslo preskočiť (príkazom `continue`), použijeme operátor logickej negácie `!`. Zátvorky sú nevyhnutné, operátor `!` má vyššiu prioritu ako `%`.

**Príkaz** `return`

Tento príkaz sa používa na ukončenie behu práve vykonávanej funkcie a návrat do funkcie volajúcej, s prípadným odovzdaním návratovej hodnoty. Jeho syntax je nasledujúca:

```
return výraz;
return;
```

*Výraz* predstavuje návratovú hodnotu a je povolený iba vtedy, ak je funkcia deklarovaná s návratovým typom iným ako `void` (inými slovami, ak vôbec niečo vracia). Vo funkciách typu `void` je povolený iba druhý tvar, bez výrazu. Prirodzené ukončenie funkcie, keď sa dospeje k pravej krútenej zátvorke, ukončujúcej zápis funkcie, je ekvivalentné príkazu `return` bez výrazu. O funkciách sme si zatiaľ veľa nepovedali, takže len malý príklad. Funkcia, ktorá vracia výsledok výrazu $f(x) = x^2 + 3x - 6$:

```
double f(double x)
{
    return x*x + 3*x - 6;
}
```

a jej použitie:

```
printf(„f(4) = %lg\n", f(4));
```

**Príkaz** `goto`

Posledným zo skokových príkazov, často zatracovaným, ale predsa existujúcim a v podstate najviac oprávneným nosiť názov príkaz skoku, je príkaz `goto`. Jeho syntax je jednoduchá:

```
goto návestie;
```

kde návestie sa môže vyskytovať pred ľubovoľným príkazom vo funkcii a má tvar identifikátora ukončeného dvojbodkou:

```
    ...
návestie:
    príkaz;
    príkaz;
    ...
```

S trochu inými návestiami sme sa už stretli pri opise príkazu `switch`, ale tie nemôžeme použiť spolu s príkazom `goto`. A čo sa stane, keď program narazí na príkaz `goto`? Jednoducho skočí na prvý príkaz nasledujúci za daným návestím. V prípade, že chceme skočiť na koniec nejakého bloku a za návestím by už teda nebol nijaký príkaz, iba pravá krútená zátvorka, musíme tam nejaký príkaz pridať, a to najčastejšie prázdny príkaz, t. j. bodkočiarku:

```
while (...)
{
    ...
    goto end;
    ...
end:
    ;
}
```

Otázkou zostáva, či je použitie príkazu `goto` opodstatnené a či je vôbec z hľadiska programátorského štýlu vhodné ho používať. Moja odpoveď znie – áno, existujú prípady, keď by sa rovnaký efekt dosiahol len zavedením komplikovaných konštrukcií a značným znížením efektivity kódu. Obyčajne sa tento príkaz používa vtedy, ak chceme vyskočiť zvnútra viacerých vnorených cyklov, keď by normálne bolo treba na každej úrovni testovať, či náhodou nechceme celú hierarchiu cyklov ukončiť. Jazyk Java má pre tento príklad zavedené určité rozšírenie príkazu break, hádam preto ani v Jave príkaz `goto` nenájdeme. Takže smiete používať príkaz `goto`, ale s mierou, a pokiaľ možno vždy v smere dopredu, nikdy nie na skok naspäť!

### Záver

Prebrali sme všetky príkazy jazyka C++ a s využitím doterajších vedomostí by ste už mali byť schopní písať krátke a jednoduchšie programy sami. Odporúčam vám, pokúste sa o to, i keď to budú úplne zbytočné a účelové programy, ktoré vzápätí vymažete. Dobre programovať sa človek naučí okrem iného tak, že programuje veľa, popritom si všíma chyby, ktoré robí, snaží sa pochopiť, čo sa naozaj deje pri behu jeho program, čím sa neustále zdokonaľuje. Je výhodné, keď poznáte jazyk, v ktorom programujete, od A po Z, pretože ste potom pripravení na všetky prekvapenia, ktoré vám vývoj programov prichystá. Aj tak budete často prekvapení, keď objavíte nejakú „novinku".

## Deviata časť: INTERMEZZO

V dnešnom pokračovaní seriálu si ukážeme trochu rozsiahlejší program, demonštrujúci väčšinu toho, čo sme dosiaľ prebrali. Je jasné, že v ňom nebudú použité všetky konštrukcie, taký ad hoc program by veľmi užitočný nebol. Dlho som rozmýšľal nad tým, čo by mal program vlastne robiť, nakoniec som sa rozhodol pre jednoduchú ukážku z oblasti matematiky (keď už sa tá škatuľa volá počítač, nech nám niečo spočíta) – pôjde o riešenie sústavy lineárnych rovníc a o výpočet koreňov reálneho polynómu. Program nie je ani najefektívnejší, ani najdokonalejší, slúži len ako príklad, preto má niektoré obmedzenia – dokáže riešiť sústavy lineárnych rovníc maximálne s tromi neznámymi a neošetruje všetky možné chyby, ktoré by pri zadávaní údajov alebo výpočte mohli nastať.

### Trochu teórie

Prv než si niečo o programe povieme, dovolím si trošku teórie. Na riešenie sústav lineárnych rovníc existuje mnoho metód – analytických či numerických (iteračných). Väčšinu z nich nie je problém naprogramovať, ale ich algoritmus sa môže zdať príliš zložitý tomu, kto danú metódu nepozná. Na účely ukážkového príkladu som preto vybral jednu z najznámejších analytických metód, metódu determinantov. Ako je známe, sústavu lineárnych rovníc môžeme prehľadne zapísať maticovým zápisom

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$

kde **A** je (štvorcová) matica sústavy (rozmeru $n \times n$), **x** je hľadaný vektor riešení ($n \times 1$) a **b** je vektor pravých strán ($n \times 1$), kde $n$ je počet neznámych (t. j. počet rovníc). Pri tomto značení $i$-ta rovnica sústavy má tvar $a_{i1}x_1 + \ldots + a_{in}x_n = b_i$, kde $a_{ij}$ je prvok v $i$-tom riadku a $j$-tom stĺpci matice **A** a $x_i$, $b_i$ sú $i$-te prvky vektorov **x** a **b**. Jednotlivé neznáme $x_i$ môžeme vypočítať na základe jednoduchého vzťahu

$$x_i = D_i / D_S$$

kde $D_S$ je determinant sústavy – normálny determinant matice **A**, $D_i$ sú determinanty upravenej matice $\mathbf{A_i}$, ktorá vznikne náhradou $i$-teho stĺpca (kde $i$ je index počítanej neznámej) vektorom pravých strán **b**.

Naprogramovať túto metódu je na pohľad veľmi jednoduché, spočítame najprv determinant $D_S$ a potom v cykle budeme počítať jednotlivé determinanty $D_i$, deliť ich $D_S$ a vypisovať výsledky. Háčik je však v samotnom výpočte determinantu. Pre matice do rozmerov $3 \times 3$ existuje v podstate jednoduchý vzorec na priamy výpočet na základe prvkov matice, ale pre vyššie rozmery treba použiť rozvoj podľa niektorého riadka či stĺpca a počítať subdeterminanty, čo je už priveľmi komplikované pre náš príklad. Preto program dokáže počítať determinanty len pre matice rozmerov najvýš $3 \times 3$, teda dokáže riešiť sústavy maximálne s tromi neznámymi.

Druhá časť programu slúži na výpočet koreňov reálneho polynómu. Reálny polynóm je výraz v tvare

$$f(x) = a_n x_n + a_{n-1}x_{n-1} + \ldots + a_1 x + a_0$$

kde $a_i$ sú koeficienty polynómu a sú to reálne čísla. Číslo $n$ sa nazýva *stupeň* polynómu. Korene tohto polynómu sú také čísla $x$, pre ktoré platí

$$f(x) = 0$$

Polynóm $n$-tého stupňa má práve $n$ koreňov (ktoré však nemusia byť všetky reálne!). Malý príklad: polynóm $x^2 - x - 2$ je polynómom 2. stupňa a jeho koreňmi sú čísla –1 a 2, pretože $(-1)^2 - (-1) - 2 = 0$ aj $2^2 - 2 - 2 = 0$. Polynóm $x^2 + 1$ je takisto polynómom 2. stupňa, no nemá reálne korene (má dva komplexné: i a –i).

Na výpočet koreňov polynómov vo všeobecnosti neexistuje analytický vzorec, preto treba používať numerické (aj iteračné) metódy. Tieto metódy sú založené na postupnom približovaní sa k správnemu riešeniu pomocou nejakých výpočtov realizovaných v cykloch (iteráciách). Najvhodnejšou pre náš príklad je Newtonova interpolačná metóda. Jej princíp je veľmi jednoduchý: na začiatku si zvolíme nejaký odhad riešenia, $x_0$. V každom ďalšom kroku iterácie vypočítavame novú hodnotu $x_i$ na základe vzťahu

$$x_i = x_{i-1} - f(x_{i-1}) / f'(x_{i-1})$$

kde $f(x)$ je polynóm, ktorého korene hľadáme, a $f'(x)$ je jeho derivácia. Takto dostaneme postupnosť hodnôt $x_0, x_1, x_2, \ldots$, ktorá za určitých podmienok konverguje k niektorému z koreňov daného polynómu. Otázkou zostáva, kedy výpočet ukončíme. Tento problém je trošku zložitejší, ako by sa na prvý pohľad zdalo, preto si ho nebudeme komplikovať a dohodneme sa, že výpočet sa

skončí, keď sa dva po sebe idúce vypočítané členy postupnosti budú líšiť o číslo menšie ako nejaký vopred dohodnutý limit (značí sa gréckym písmenom epsilon). Nesmieme však zabudnúť na fakt, že polynóm nemusí mať žiadne reálne korene a v takom prípade by sme iterovali donekonečna (prečo, to si tu nebudeme vysvetľovať). Obmedzíme preto počet iterácií nejakou maximálnou hodnotou.

### Program

Toľko teoretická príprava. V prípade, že ste niečomu neporozumeli, vráťte sa k nej ešte raz po preštudovaní programu, a ak ani potom nebudete mať vo veci jasno, tak je mi to ľúto, skúste sa poobzerať po nejakej literatúre. Myslím, že na pochopenie výkladu stačia zhruba vedomosti na úrovni 3. až 4. ročníka strednej školy.

Nasleduje výpis programu. Program je rozdelený do dvoch súborov, `matemat.h` a `matemat.cpp`.

Výpis súboru `matemat.h`:

```
// matemat.h

// max. počet neznámych
#define NMAX    3
// max. stupeň polynómu
#define STMAX   10
// presnosť výpočtu koreňov
#define EPS     0.0000001
// max. počet iterácií
#define ITMAX   1000

int menu();
void confirm();

void linsys();
double det(int, double[NMAX][NMAX],
           double[NMAX], int = -1);

void newton();
double abs(double);
double poly(int, double[STMAX], double);
double dpoly(int, double[STMAX], double);
```

Výpis súboru `matemat.cpp`:

```
// matemat.cpp

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include „matemat.h“

//---------------------------------------
//  main()
//       Hlavná funkcia programu.
//---------------------------------------
int main()
{
    while (1)
    {
        int choice;
        if ((choice = menu()) == 3)
            break;

        switch (choice)
        {
        case 1:
            linsys();
            break;

        case 2:
            newton();
            break;
        }
    }

    return 0;
}

//---------------------------------------
//  menu()
//      Výpis ponuky činnosti programu.
//
//      Vstup:  žiaden
//      Výstup: 1 - sústava rovnic
//              2 - korene polynómu
//              3 - koniec
//---------------------------------------
int menu()
{
    clrscr();
    printf(
        „Matematika 1.0\n“
        „--------------\n“
        „\n“
        „1. riešenie sústavy lin. rovníc\n“
        „2. hľadanie koreňov polynómu\n“
        „3. koniec\n“
        „\n“
        „Vaša voľba: „);

    int choice;
    do {
        choice = getch();
        if (!choice)
            getch();
    }
    while (choice < ‚1‘ || choice > ‚3‘);

    printf(„%c\n\n“, choice);

    return choice - ‚0‘;
}

//---------------------------------------
//  confirm()
//       Požiada o stlačenie ľub. klávesu.
//
//       Vstup:  žiaden
//       Výstup: žiaden
//---------------------------------------
void confirm()
{
    printf(„\nStlačte kláves...\n“);
    if (!getch())
        getch();
}

//---------------------------------------
//  linsys()
//       Riešenie sústavy lin. rovníc.
//
//       Vstup:  žiaden
//       Výstup: žiaden
//---------------------------------------
void linsys()
{
    char buf[80];
    int n = 0;

    // vstup počtu neznámych
    while (n <= 0 || n > NMAX)
    {
        printf(„Počet neznámych: „);
        gets(buf);
        n = atoi(buf);
    }

    double A[NMAX][NMAX];
    double b[NMAX];
    double x[NMAX];

    // vstup prvkov matice A
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
        {
            printf(„ A[%i][%i] = „, i, j);
            gets(buf);
            A[i][j] = atof(buf);
        }

    // vstup prvkov vektora b
    for (i = 0; i < n; i++)
    {
        printf(„ b[%i]    = „, i);
        gets(buf);
        b[i] = atof(buf);
    }

    // výpočet vektora x
    double dets = det(n, A, b);
    if (dets == 0.0)
    {
        printf(„\n“
               „Matica A je singulárna, „
               „sústava nemá riešenie!\a\n“);
        confirm();
        return;
    }

    for (i = 0; i < n; i++)
        x[i] = det(n, A, b, i) / dets;

    // výpis výsledkov
    printf(„----------------------\n“);
```

```
    for (i = 0; i < n; i++)
        printf(„ x[%i]    = %lg\n", i, x[i]);

    confirm();
}

//----------------------------------------
//  det()
//      Výpočet determinantu matice.
//
//      Vstup:  n - skutočný rozmer matice
//              A - matica koeficientov
//              b - vektor pravých strán
//              s - nahradzovaný stĺpec
//      Výstup: determinant
//----------------------------------------
double det(int n, double A[NMAX][NMAX],
           double b[NMAX], int s)
{
    // vytvorenie (upravenej) kópie matice
    double A2[NMAX][NMAX];

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (s != -1 && s == j)
                A2[i][j] = b[i];
            else
                A2[i][j] = A[i][j];

    // výpočet determinantu
    double res;

    switch (n)
    {
    case 1:
        res = A2[0][0];
        break;

    case 2:
        res = A2[0][0] * A2[1][1]
            - A2[0][1] * A2[1][0];
        break;

    case 3:
        res = A2[0][0] * A2[1][1] * A2[2][2]
            + A2[1][0] * A2[2][1] * A2[0][2]
            + A2[2][0] * A2[0][1] * A2[1][2]
            - A2[0][2] * A2[1][1] * A2[2][0]
            - A2[1][2] * A2[2][1] * A2[0][0]
            - A2[2][2] * A2[0][1] * A2[1][0];
        break;
    }

    return res;
}

//----------------------------------------
//  newton()
//      Výpočet koreňov polynómu.
//
//      Vstup:  žiaden
//      Výstup: žiaden
//----------------------------------------
void newton()
{
    char buf[80];
    int st = 0;

    // vstup stupňa polynómu
    while (st <= 0 || st > STMAX)
    {
        printf(„Stupeň polynómu: „);
        gets(buf);
        st = atoi(buf);
    }

    // vstup koeficientov polynómu
    double a[STMAX + 1];

    for (int i = st; i >= 0; i--)
    {
        printf(„ a[%i] = „, i);
        gets(buf);
        a[i] = atof(buf);
    }

    // vstup počiatočného bodu
    printf(„Počiatočný odhad: „);
    gets(buf);
    double x0 = atof(buf);

    // výpočet koreňa
    double x1, dif;
    int it = 0;

    do {
        double fx = poly(st, a, x0);
        double dfx = dpoly(st, a, x0);

        if (dfx == 0.0)
        {
            it = ITMAX;
            break;
        }

        x1 = x0 - fx / dfx;
        dif = abs(x1 - x0);
        x0 = x1;
    }
    while (++it < ITMAX && dif > EPS);

    // výpis výsledku
    printf(„----------------------\n");
    if (it == ITMAX)
        printf(„ riešenie sa nenašlo\n");
    else
        printf(„ x    = %lg\n", x1);

    confirm();
}

//----------------------------------------
//  abs()
//      Výpočet absolútnej hodnoty.
//
//      Vstup:  x - číslo
//      Výstup: jeho absolútna hodnota
//----------------------------------------
double abs(double x)
{
    return x >= 0 ? x : -x;
}

//----------------------------------------
//  poly()
//      Výpočet hodnoty polynómu v bode.
//
//      Vstup:  st - stupeň polynómu
//              a  - koeficienty polynómu
//              x  - bod výpočtu
//      Výstup: hodnota polynómu v bode x
//----------------------------------------
double poly(int st, double a[STMAX],
            double x)
{
    double res = a[st];

    for (int i = st - 1; i >= 0; i--)
        (res *= x) += a[i];

    return res;
}

//----------------------------------------
//  dpoly()
//      Výpočet hodnoty derivácie polynómu
//      v bode.
//
//      Vstup:  st - stupeň polynómu
//              a  - koeficienty polynómu
//              x  - bod výpočtu
//      Výstup: hodnota derivácie polynómu
//              v bode x
//----------------------------------------
double dpoly(int st, double a[STMAX],
             double x)
{
    double res = 0.0;

    for (int i = st; i > 0; i--)
        (res *= x) += (i * a[i]);

    return res;
}
```

Prv než si preberieme program trochu podrobnejšie, povieme si niečo o tom, ako je v C++ realizovaný vstup údajov od používateľa. Nebudeme zachádzať do úplných podrobností, opíšeme si len niektoré možnosti, ktoré sú použité aj v našom programe. Bližšie sa vstupom (i výstupom) údajov budeme zaoberať v časti venovanej štandardnej knižnici.

### Vstup údajov z klávesnice

Údaje do programu možno dostať mnohými spôsobmi – z klávesnice, čítaním zo súboru, cez sieť. V tomto odseku sa obmedzíme na možnosti vstupu priamo z klávesnice.

Všetky uvedené funkcie sú súčasťou štandardnej knižnice jazyka C, a preto by mali byť k dispozícii v každom prekladači.

Prvou z funkcií, ktorú však v našom programe nemáme, je funkcia `scanf()`. Jej názov čiastočne napovedá, že je komplementárnou k funkcii `printf()` a má na starosti formátovaný vstup. Jej syntax je podobná syntaxi funkcie `printf()` – prvým argumentom je formátovací reťazec, po ňom nasleduje zoznam premenných, ktorých hodnoty chceme pomocou vstupu z klávesnice zmeniť. Ale pozor! Tu sa často robí chyba, pred každou premennou musí byť operátor `&`. V skutočnosti totiž funkcia `scanf()` ako svoje argumenty očakáva ukazovatele na príslušné premenné. Samozrejme, ak máme k dispozícii priamo takýto ukazovateľ, môžeme ho použiť. Príklad:

```
int a, *pa = &a;
scanf(„%i", &a);
scanf(„%i", pa);
```

Obe volania `scanf()` sú ekvivalentné.

Funkcia `scanf()` číta dáta zo štandardného vstupu, ktorým je implicitne klávesnica. Jej použitie je trochu ťažkopádne, hlavne z toho dôvodu, že operačný systém pri vstupe z klávesnice používa riadkovú vyrovnávaciu pamäť (line buffer), takže vstup sa deje po dávkach, ukončených klávesom Enter. Naproti tomu `scanf()` prijíma v podstate prúd znakov, ktorý spracúva podľa formátovacieho reťazca. Preto je často výhodnejšie použiť inú funkciu, `gets()`. Jej argumentom je pole znakov, do ktorého sa skopíruje znak po znaku celý riadok, zadaný používateľom. Takto načítaný riadok môžeme potom spracovať vo vlastnej réžii.

V našom programe zadávané údaje najprv načítame do poľa `buf[]`. Potom, keďže potrebujeme čísla, a nie reťazce znakov, pomocou knižničných funkcií `atoi()` a `atof()` prevedieme získané reťazce na čísla typu `int`, resp. `double`. Argumentom týchto funkcií je reťazec znakov, ktorý chceme previesť. Spôsob použitia je zrejmý z programu.

Tretí spôsob vstupu, o ktorom si dnes povieme, sa používa v prípade, že chceme okamžite reagovať na stlačenie nejakého klávesu (a nečakať, kým používateľ stlačí Enter). Na tento účel máme k dispozícii funkciu `getch()`. Táto funkcia čaká na stlačenie ľubovoľného klávesu, ktorého ASCII kód potom vráti. V prípade, že používateľ stlačil nejaký kláves, ktorý nemá ASCII kód (typicky funkčné klávesy, šípky a pod.), `getch()` vracia hodnotu `0` a pri ďalšom volaní tzv. scan-kód stlačeného klávesu. Funkcia `getch()` nevypisuje stlačený znak na obrazovku.

V programe je funkcia `getch()` použitá jednak pri výbere položky z menu, jednak po ukončení výpočtu a výzve na stlačenie nejakého klávesu. Vo funkcii `confirm()`, ktorá výzvu realizuje, je ukážka správneho spôsobu realizácie „čakania na stlačenie klávesu". V podmienke príkazu `if` je volanie funkcie `getch()`. Podmienka je splnená, keď `getch()` vráti nulu (operátor `!`) – v takom prípade sa `getch()` zavolá ešte raz. Keby sme netestovali návratovú hodnotu funkcie `getch()`, pri nasledujúcom volaní (a to nielen z nášho programu, ale hocikde v rámci operačného systému) by sme dostali scan-kód predchádzajúceho klávesu, ktorý by sme mylne interpretovali ako ASCII kód nového klávesu.

Funkcia `getch()`, ako sme už povedali, čaká na stlačenie klávesu. BIOS však stlačenia klávesov ukladá do vyrovnávacej pamäte pre prípad, že stihneme stlačiť viacero klávesov, ako program stačí spracovať. V takom prípade bude funkcia `getch()` (logicky) postupne vracať všetky stlačené klávesy uložené v tejto vyrovnávacej pamäti. Na zistenie, či používateľ stlačil nejaký kláves, slúži funkcia `kbhit()`, ktorá vráti nenulovú hodnotu, ak vo vyrovnávacej pamäti klávesnice je aspoň jeden znak. Ale pozor! Ak chceme túto funkciu použiť na prerušenie nejakej bežiacej slučky, nesmieme zabudnúť po jej skončení ten stlačený kláves prečítať a tak odstrániť z vyrovnávacej pamäte. Na ilustráciu toho, o čom hovorím, skúste si preložiť nasledujúci program:

```
#include <conio.h>

void main()
{
     while (!kbhit());
     getch();
}
```

Program čaká na stlačenie ľubovoľného klávesu (v slučke `while`), potom hodnotu klávesu prečíta (pokiaľ je slučka prázdna ako v tomto prípade, tak je vlastne zbytočná a postačí samotný príkaz `getch()`). Teraz si skúste zakomentovať riadok, na ktorom je volanie `getch()`. Keď program spustíte napríklad z Volkov Commandera (alebo iného, ale musí to byť dosovský program) stlačením klávesu Enter a ukončíte ho takisto stlačením Enteru, program sa spustí sám od seba znovu. Čo sa stalo? Keď sme stlačili Enter, program sa skončil, ale vo vyrovnávacej pamäti klávesnice toto stlačenie zostalo. Po skončení programu sa k slovu dostal Volkov Commander, ktorý tiež čaká na stlačenie klávesu. Ten si stlačený Enter „vytiahol" a interpretoval ho ako požiadavku na spustenie programu, na ktorom mal nastavený kurzor. Ale tým programom bol, samozrejme, opäť náš program. Pozor teda na takéto chyby!

Na záver rozprávania o vstupe údajov ešte upozornenie: ak chcete používať uvedené funkcie, musíte do vášho programu vložiť príslušné hlavičkové súbory. Pre `scanf()` a `gets()` je to `stdio.h`, pre `getch()` a `kbhit()` zase `conio.h`.

## Analýza programu

Program je rozdelený do dvoch súborov, `matemat.h` a `matemat.cpp`. Prvý z nich, `matemat.h`, je (vďaka svojej prípone) hlavičkovým súborom. Pozrime sa na jeho obsah. Nájdeme v ňom štyri riadky, začínajúce sa reťazcom `#define`. Ak tušíte, že ide o podobnú syntaktickú konštrukciu ako direktíva `#include`, tak ste na správnej stope – ide o direktívu preprocesora. Zatiaľ si povieme iba, že táto direktíva nám umožňuje definovať tzv. makrá. Jej syntax je `#define` meno reťazec, riadok nie je ukončený bodkočiarkou! Od miesta výskytu direktívy `#define` môžeme v kóde programu používať meno ako ekvivalent reťazca. Túto náhradu vykoná automaticky preprocesor ešte pred preložením programu kompilátorom (nalistujte si druhú časť seriálu). Reťazec môže byť prakticky ľubovoľná postupnosť znakov. Často sa používajú makrá, ktoré sa rozvinú na konštanty, podobne ako v našom príklade, kde napríklad makro `STMAX` ďalej v programe predstavuje maximálny dovolený stupeň polynómu. Túto hodnotu používame v kóde viackrát a v prípade, že by sme sa rozhodli ju zmeniť, museli by sme ručne nájsť všetky jej výskyty a prepísať ich novou hodnotou. Takto nám stačí zmeniť jeden riadok programu. Bližšie o direktíve `#define` v niektorej z budúcich častí.

Ďalej v hlavičkovom súbore nájdeme akoby definície funkcií, ale bez tela. Takéto definície, lepšie povedané deklarácie, nazývame prototypmi funkcií a pomocou nich hovoríme kompilátoru, že v našom programe existujú také a také funkcie s takými a takými argumentmi a návratovými hodnotami. Samozrejme, nie je nevyhnutné prototypy používať, ale v takom prípade môžeme v nejakom mieste programu zavolať iba funkciu, ktorá už bola niekedy predtým definovaná. To v našom programe neplatí, pozrite sa do `matemat.cpp`, kde napr. z funkcie `main()` voláme funkcie `linsys()` a `newton()`, ktorých definícia, teda telo sa nachádza o pár riadkov nižšie a teda až za ich volaním. Prototypy slúžia na akési predbežné oboznámenie kompilátora s tým, aké funkcie v programe mám, a súčasne mu umožňujú kontrolovať, či sú tieto funkcie volané so správnym počtom a typmi argumentov. Štandardné hlavičkové súbory, ako `stdio.h` a pod., sú plné prototypov príslušných funkcií, ako sa môžete i sami presvedčiť, a ich vložením (`#include`) do nášho programu si zabezpečíme možnosť použitia týchto funkcií. Kompilátor bude vedieť, že niekde tieto funkcie existujú, a keď ich nenájde v našom programe, pripojí ich z knižnicových súborov (`.lib`) v čase linkovania. Deklarácia prototypu funkcie je podobná definícii „plnej" funkcie, s tým rozdielom, že namiesto tela funkcie uzavretého v krútených zátvorkách napíšeme len bodkočiarku (povinná!). V deklarácii prototypu takisto nie je potrebné uvádzať názvy argumentov, stačí ich typ. Celú záležitosť okolo prototypov a všeobecne funkcií si opíšeme v ďalších častiach seriálu.

Možno vás napadla otázka, načo je nám vlastne náš hlavičkový súbor dobrý, veď celý jeho obsah by sme mohli napísať priamo do hlavného zdrojového súboru. Pravdu povediac, jeho použitie v našom konkrétnom príklade je zbytočné. No ak by sme v budúcnosti program rozširovali, pridávali ďalšie funkcie v samostatných zdrojových súboroch, potom v prípade nutnosti použiť niektorú z funkcií zo súboru `matemat.cpp` (napr. `confirm()`, tá je dosť všeobecná), bude nám stačiť vložiť v týchto ďalších súboroch hlavičkový súbor `matemat.h` a hneď budeme mať všetky ním „predstavené" funkcie k dispozícii.

Prejdime k hlavnému súboru `matemat.cpp`. Hneď na jeho začiatku nachádzame známe direktívy `#include`, ktoré jednak vkladajú niekoľko štandardných hlavičkových súborov, jednak náš vlastný `matemat.h`. Za povšimnutie stojí, že zatiaľ čo názvy štandardných súborov sú uzavreté v lomených zátvorkách (<>), názov nášho súboru je v úvodzovkách. Rozdiel medzi oboma spôsobmi zápisu je ten, že súbory uvedené v lomených zátvorkách prekladač bude pri preklade hľadať len v adresároch určených pre hlavičkové súbory (toto určenie sa realizuje napr. pomocou prepínačov príkazového riadka prekladača alebo v integrovaných prostrediach v niektorom z nastavovacích dialógov), kým súbory uvedené v úvodzovkách bude hľadať aj v adresároch určených pre zdrojové súbory. Mnoho integrovaných vývojových prostredí umožňuje zvoliť samostatný adresár pre zdrojové súbory a samostatný pre hlavičkové súbory (a takisto samostatný pre výstupné súbory, ako `.obj` a `.exe`). Nie že by ste museli striktne rozdeľovať súbory do adresárov, vždy to závisí od rozsahu a typu vytváraného projektu, ale typicky sa používa zápis s lomenými zátvorkami na vkladanie štandardných hlavičkových súborov a zápis s úvodzovkami na vkladanie súborov vlastných (na ich odlíšenie od tých štandardných), tak ako je to v našom príklade.

Program sa začína uvedením hlavnej funkcie programu `main()`. Všimnite si, že pred ňou, ako aj pred každou inou funkciou v programe, je krátky komentárový blok, v ktorom je uvedený názov funkcie, stručný opis, čo funkcia robí, ďalej názvy a opis jednotlivých argumentov a opis návratovej hodnoty. Takýto blok vám jednak pomôže aj po dlhšom čase zistiť, čo ktorá funkcia robí, čo očakáva na vstupe a čo dáva na výstupe, a jednak vizuálne oddeľuje definície jednotlivých funkcií, čo v dlhších súboroch výrazne pomáha pri orientácii. Ale späť k funkcii `main()`. Je veľmi jednoduchá, pomocou volania funkcie `menu()` zistí, ktorý z oboch výpočtov chce používateľ realizovať, a podľa toho zavolá jednu

z funkcií `linsys()` alebo `newton()`. Všimnite si podmienku v príkaze `if`:

```
if ((choice = menu()) == 3)
    break;
```

Takýto zápis je v C++ bežný a jeho význam by vám mal byť už jasný, ale pre istotu: najprv sa do premennej `choice` vloží návratová hodnota funkcie `menu()` a tá sa potom testuje na rovnosť s konštantou `3`, čo je dohodnutá konštanta pre voľbu „Koniec programu“. Zátvorky sú nevyhnutné, operátor == má väčšiu prioritu ako =. Ďalej v príkaze `switch` sa premenná `choice` testuje len na hodnoty `1` a `2`, pretože v tomto mieste programu už hodnotu `3` mať nemôže (a ani žiadnu inú, preto v bloku nenájdeme ani návestie `default:`). Celý výber a vetvenie sú uzavreté do nekonečnej slučky `while`, z ktorej je možné vyskočiť po výbere príslušnej položky menu – to je ten `break` za príkazom `if`.

Nasledujúcou funkciou v zdrojovom kóde programu je funkcia `menu()`. V nej za pozornosť stojí cyklus `do`, v tele ktorého sa čaká na stlačenie príslušného klávesu. Pozor, premenná `choice`, ktorá sa tu vyskytuje, nemá okrem názvu nič spoločné s podobnou premennou vo funkcii `main()`! Je tu takisto opisovaný test na stlačenie rozšírených klávesov, ktoré musíme čítať na dvakrát. Cyklus sa ukončí, keď stlačíme niektorý z požadovaných klávesov. Zaujímavé je, že hoci porovnávame premennú `choice` so znakovými konštantami (`'1'`, `'3'`), jej typ je `int`, a nie `char`. To je však úplne v poriadku, pretože oba typy sú celočíselné a vzhľadom na porovnanie kompatibilné. Uplatňujú sa tu určité štandardné konverzie, povieme si o nich neskôr. Na záver funkcie stlačený kláves (lepšie povedané príslušný znak) vypíšeme, pretože `getch()` to za nás nespraví. Ešte si všimnite, akým spôsobom prevedieme znak stlačeného klávesu na návratovú hodnotu – číslo `1` až `3`. Je to jednoduché, pretože ASCII kódy znakov `'0'` až `'9'` nasledujú za sebou, teda ak odčítame trebárs od konštanty `'3'` (ASCII kód `0x33`) konštantu `'0'` (ASCII kód `0x30`), dostaneme práve požadované číslo `3`. Dá sa to, pravda, aj inak, napríklad veľmi elegantne a profesionálne výrazom `choice & 0x0F`, pretože ASCII kódy znakov číslic sú v rozmedzí `0x30` až `0x39`, zrejme teda stačí ponechať dolné štyri bity (čo práve robí operátor `&` s maskou `0x0F`).

Ideme ďalej. Funkcia `confirm()` je natoľko triviálna, že o nej hádam netreba nič hovoriť. Nasleduje funkcia `linsys()`, ktorá má na starosti riešenie sústav lineárnych rovníc. Táto funkcia nemá žiadne vstupy, to znamená, že načítanie i výstup dát sa dejú až v jej vnútri. Ako prvé si vypýtame od používateľa počet neznámych. Ten musí byť v rozmedzí `1` až `NMAX`, čo je z uvedených dôvodov `1` až `3`. Vstup od používateľa sa bude opakovať dovtedy, kým nebude v požadovanom rozsahu. Podmienka cyklu `while` môže byť taká ako v našom príklade alebo aj `!(n > 0 && n <= NMAX)`, čo je klasická negácia tej našej použitím de Morganovho zákona.

Po zistení počtu neznámych si definujeme príslušné premenné. `A` ako dvojrozmerné pole, `b` a `x` ako polia jednorozmerné. Keďže zatiaľ nevieme vytvárať polia dynamicky, podľa potreby, definujeme naše premenné ako statické polia s maximálnym možným rozmerom, daným maximálnym počtom neznámych `NMAX`. Zaberie nám to trochu pamäte, ale to už dnes hádam až tak neprekáža. Pri výpočte použijeme len takú časť týchto polí, akú potrebujeme. Ďalej v cykle načítame hodnoty všetkých prvkov premenných `A` a `b`. Spočítame determinant sústavy a postupne v cykle budeme zisťovať determinanty pre jednotlivé neznáme, vypočítavať tieto neznáme a vypisovať ich na obrazovku. Na konci funkcie si vyžiadame stlačenie ľubovoľného klávesu (funkcia `confirm()`).

Funkcia `det()` slúži na výpočet determinantu štvorcovej matice. Tak ako je napísaná, funguje iba pre matice do rozmerov 3 × 3. Prvým jej argumentom je rozmer matice, druhým je samotná matica. Jej typ je uvedený ako `double A[NMAX][NMAX]`, i keď z nej možno použijeme menšiu časť. V skutočnosti sa však nekopíruje do funkcie celá matica, ale len ukazovateľ na ňu. O vzťahu polí a ukazovateľov sme si dosiaľ nehovorili, takže to zatiaľ berte ako fakt, že na to, aby program správne pracoval, musíme pri maticiach definovaných ako dvojrozmerné polia deklarovať ich typ v argumentoch funkcie takto (v skutočnosti je nevyhnutný len ten druhý `NMAX`, prvý by sme mohli vynechať – `double A[][NMAX]`). Podobne pri vektore `b`, i keď tu je to jednoduchšie a stačilo by napísať `double b[]`. Dokonca (môžete si to vyskúšať, ale bez nároku na vysvetľovanie) je možné napísať typ argumentu ako `double* b`. Posledným argumentom je číslo stĺpca, ktorý sa má nahradiť vektorom `b` (pri výpočte determinantov pre jednotlivé neznáme). Ak je tento argument rovný `-1`, chceme počítať normálny determinant matice `A`. Všimnite si, že pri volaní funkcie `det()` z funkcie `linsys()` pri výpočte determinantu sústavy (premenná `dets`) sme tento posledný argument vôbec neuviedli! Ako je to možné? Jazyk C++ povoľuje takzvané implicitné argumenty (bližšie o nich neskôr) – všimnite si v súbore `matemat.h` prototyp funkcie `det()` a deklaráciu posledného argumentu: `int = -1`. Tá hovorí, že ak tento argument neuvedieme, prekladač ho sám doplní implicitnou hodnotou `-1`. Výhodne tento fakt používame v našom príklade – je to prehľadnejšie ako písať tam tú mínus jednotku. Vnútri funkcie `det()` si deklarujeme premennú `A2` rovnakých rozmerov ako matica `A`, do ktorej túto maticu skopírujeme, prípadne nahradíme niektorý zo stĺpcov vektorom `b`. Potom už len spočítame determinant klasickým spôsobom a výsledok vrátime. Pochopiť spôsob náhrady stĺpcov vám dávam na domácu úlohu.

Výpočet koreňov polynómu má na starosti funkcia `newton()`. Podobne ako vo funkcii `linsys()` zistíme najprv stupeň polynómu (v rozsahu `1` až `STMAX` – polynómy 0. stupňa nemajú korene a nemá zmysel ich počítať), potom definujeme pole koeficientov `a[]`, ktorého dĺžka je o jeden väčšia ako stupeň polynómu, a následne toto pole naplníme vstupnými údajmi od používateľa. Ešte si vypýtame počiatočný odhad `x0` a môžeme začať iterovať. V rámci každej iterácie spočítame hodnotu polynómu v bode `x0`, hodnotu jeho derivácie v tomto bode a na základe oboch hodnôt spočítame nový „odhad“ `x1`. Zistíme rozdiel medzi hodnotami `x0` a `x1`, aby sme vedeli, či už máme skončiť (premenná `dif`), a potom vypočítaný odhad `x1` prekopírujeme do `x0`. V každom prechode inkrementujeme premennú `it`, ktorá nám počíta iterácie. Cyklus môže skončiť tromi rôznymi spôsobmi. Buď premenná `it` dosiahne hodnotu `ITMAX`, čo znamená, že sme vyčerpali maximálny počet iterácií a riešenie sme stále nenašli, alebo premenná `dif` klesne pod hodnotu `EPS`, čo je nastavená presnosť nájdenia koreňa, alebo pri výpočte zistíme, že hodnota derivácie nám vyšla nulová (čo znamená nesplnenie podmienok na použitie metódy), a vtedy umelo zvýšime hodnotu premennej `it` na `ITMAX` a z cyklu vyskočíme. Podmienka cyklu do musí byť zapísaná v takom poradí, ako je to uvedené v programe, pretože potrebujeme zaručiť inkrementáciu `it` v každom prechode a operátor `&&`, ako vieme, nevyhodnotí oba svoje operandy, pokiaľ prvý z nich je nulový. Na záver funkcie vypíšeme buď nájdený koreň, alebo oznam, že riešenie sa nenašlo. Rozhodujeme sa na základe obsahu premennej `it`, ktorá je v prípade nenájdenia riešenia rovná `ITMAX` (preto to umelé nastavovanie pri nulovej derivácii).

Nasleduje krátka a jednoduchá funkcia na výpočet absolútnej hodnoty, myslím, že ju opäť netreba opisovať podrobnejšie. Posledné dve funkcie – `poly()` a `dpoly()` – slúžia na výpočet hodnôt polynómu a jeho derivácie v danom bode. Argumentmi sú vždy stupeň polynómu, jeho koeficienty a bod, v ktorom hodnoty počítame. Na výpočet používame tzv. Hornerovu schému, čo nie je nič iné ako prepísanie polynómu

$$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$$

do tvaru

$$(\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$$

(v podstate postupné vynímanie $x$ „pred zátvorku“), čím ušetríme dosť veľa násobenia. Výpočet derivácie sa deje rovnakým spôsobom, s využitím faktu, že deriváciou uvedeného polynómu je polynóm

$$na_n x^{n-1} + (n-1)a_{n-1}x^{n-2} + \ldots + 2a_2 x + a_1$$

Cyklus, ktorým sa výpočet realizuje, je pri bližšom pohľade zrejmý, začíname najvyšším koeficientom $a_n$, ktorý uložíme do premennej `res`. Táto premenná bude predstavovať kumulovaný, dosiaľ vypočítaný výsledok. V každom kroku vynásobíme tento výsledok hodnotou $x$ (`res *= x`) a pripočítame ďalší koeficient (`res += a[i]`). Prípadne si program krokujte, aby ste cyklus lepšie pochopili.

Pri pozornom preskúmaní oboch cyklov (vo funkcii `poly()` a vo funkcii `dpoly()`) si isto všimnete jeden rozdiel – vo funkcii `poly()` začína premenná `res` s hodnotou najvyššieho koeficientu (`a[st]`) a v cykle premenná `i` začína s hodnotou `st - 1`. Naproti tomu vo funkcii `dpoly()` inicializujeme `res` na nulu a premenná `i` začína od hodnoty `st`. Napriek tomu oba cykly fungujú rovnako. Skúste sa zamyslieť prečo (je to inak veľmi jednoduché).

### Záver

Prebrali sme si celý program dosť podrobne, verím, že ste ho bez problémov pochopili. Majte na pamäti, že jeho možnosti sú obmedzené, pri jeho behu môže vzniknúť nejaká nepredvídaná chyba a podobne. Ak by ste mali chuť akokoľvek ho upraviť, vylepšiť, nech sa páči, fantázii sa medze nekladú.

## Desiata časť: VZŤAH POLÍ A UKAZOVATEĽOV

V predchádzajúcej časti sme si urobili krátku prestávku vo výklade a zaoberali sme sa ukážkou trochu väčšieho a užitočnejšieho programu v C++. Iste budete so mnou súhlasiť, že takto koncipovaných programov by bolo v záujme čo najväčšieho počtu príkladov podopierajúcich výklad vhodné uviesť čo najviac, ale to by sa nám seriál neúmerne predĺžil a nezostalo by miesto na mnoho ďalších, potrebnejších informácií. Aj preto by som vás rád znovu vyzval: vymýšľajte si zadania programov sami a pokúšajte sa ich aj úspešne realizovať. Pravdepodobne nie všetky problémy zvládnete hneď na prvýraz, preto sa k tým nevyriešeným vráťte neskôr, keď budete skúsenejší, vybavení väčšími znalosťami. Píšem to už síce po niekoľkýkrát, ale najväčšou devízou v programovaní (a teraz nejde len o C++, ale o programovanie v ľubovoľnom jazyku) je prax, získaná napísaním mnohých, často aj úplne zanedbateľných programov, na ktorých si trebárs objasníte nejakú spornú otázku. Čo sa týka tých spomínaných problémov, odporúčam vám jediné – študovať, študovať, študovať (ospravedlňujem sa za parafrázu) všetko, čo vám príde pod ruku, t. j. manuály, elektronické príručky, knihy, skriptá, články, diskusné skupiny a pod. (prípadne sa opýtať niekoho skúsenejšieho; tí, ktorí máte prístup k internetu, skúste napr. diskusnú skupinu Usenetu `comp.lang.c++`).

Ale poďme späť k jazyku C++. V tejto časti sa pozrieme na to, ako je to so vzťahom medzi poliami a ukazovateľmi. Táto téma je jednou z najdôležitejších a súčasne najzložitejších a jej dokonalé zvládnutie je nevyhnutné, ak nechcete zostať pri písaní svojich programov niekde „v plienkach“.

## Čo už vieme

V prvom rade si ozrejmíme základné fakty, týkajúce sa polí a ukazovateľov. Oboje sme síce už viac-menej podrobne preberali v predchádzajúcich častiach, ale pokiaľ máte tak ako ja staršie čísla PC REVUE beznádejne zavalené inými, novšími časopismi, iste vám príde vhod stručné zhrnutie. (Čo to tu rozprávam, veď vy už iste všetko ovládate z hlavy, alebo sa mýlim?)

Pole je údajový typ jazyka C++, ktorý predstavuje postupnosť po sebe idúcich premenných rovnakého typu. Pri deklarácii poľa uvádzame jeho veľkosť, t. j. počet jeho prvkov. Jednotlivé prvky sú identifikovateľné pomocou svojich indexov, na ich sprístupnenie používame operátor indexovania `[]`. Polia v C++ sa indexujú od nuly, to znamená, že pole `a`, deklarované takto:

```
int a[10];
```

obsahuje 10 premenných typu `int`, ktoré sú prístupné ako `a[0]` až `a[9]`. Jednotlivé premenné sa v pamäti nachádzajú tesne za sebou (obr. 1).

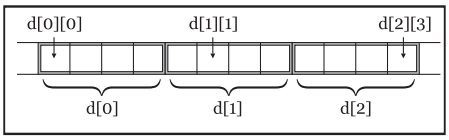

*Obr. 1*

Jazyk C++ priamo nepozná viacrozmerné polia, umožňuje však deklarovať polia, ktorých prvkami sú opäť polia. Pomocou tejto konštrukcie dokážeme vytvoriť údajový typ, ktorý sa správa prakticky rovnako ako viacrozmerné pole. Napríklad zápisom

```
double d[3][4];
```

deklarujeme objekt, ktorý je jednorozmerným poľom troch prvkov, `d[0]`, `d[1]` a `d[2]`. Každý z týchto prvkov je sám osebe poľom so štyrmi prvkami, premennými typu `double`. Tieto štyri prvky „druhej úrovne“ sú pri každom z uvedených troch polí prístupné pomocou ďalšieho indexu, čím dostávame spolu dvanásť rôznych premenných, prístupných ako `d[0][0]` až `d[2][3]`. Vo výrazoch môžeme použiť, samozrejme, aj polia `d[i]`, ktoré sa budú správať ako bežné jednorozmerné polia.

Ako vyplýva z gramatiky jazyka C++, pri prechádzaní „viacrozmerným“ poľom sa index uvedený najviac vpravo mení najrýchlejšie, čo znamená, že ak si predstavíme uvedené pole `d` ako maticu s tromi riadkami a štyrmi stĺpcami, bude táto matica uložená v pamäti po riadkoch, počnúc prvkom `d[0][0]`, po poslednom prvku prvého riadka, `d[0][2]` bude nasledovať prvý prvok druhého riadka, `d[1][0]` a tak ďalej až do konca. Prvky budú v pamäti uložené opäť súvisle, tesne za sebou (obr. 2).




*Obr. 2*

Typ ukazovateľ predstavuje premennú, v ktorej je uložená adresa inej premennej. Veľkosť ukazovateľa závisí od architektúry počítača a platformy, pre ktorú je program určený. S každým ukazovateľom je združený jeho tzv. doménový typ, čo je typ premennej, na ktorú tento ukazovateľ ukazuje. Ak ukazovateľ dereferencujeme (operátor `*`), dostaneme práve premennú doménového typu. Ako sme si hovorili, program nijakým spôsobom nekontroluje, či ukazovateľ vôbec obsahuje platnú adresu, a ak áno, či je na nej skutočne uložená premenná zodpovedajúceho doménového typu. Pri dereferencii ukazovateľa sa jednoducho oblasť pamäte, na ktorú ukazovateľ ukazuje a ktorej veľkosť je daná veľkosťou doménového typu, interpretuje ako premenná tohto typu. Ak máme smolu a obsah ukazovateľa je neplatný, v lepšom prípade sa nám program skončí výnimkou (platí výhradne pre operačné systémy s ochranou pamäte – Windows, UNIX), v tom horšom si v prípade zápisu do dereferencovanej oblasti prepíšeme napríklad program alebo rovno operačný systém (platí predovšetkým pre „operačný systém“ MS-DOS blahej pamäti).

## Tajomstvo polí

Polia v C++ sú ako každá iná premenná prístupné pomocou svojho názvu, t. j. pomocou bežného identifikátora. Ak máme ľubovoľnú premennú nejakého neštruktúrovaného typu (nie pole!), použitie jej identifikátora vo výraze spôsobí použitie jej obsahu, teda hodnoty, ktorá je v premennej uložená. Napríklad nech premenná `pi` typu `double` obsahuje (známu) hodnotu `3.14159`. Použitie tejto premennej vo výraze `2 * pi` bude mať za následok vynásobenie hodnoty v nej uloženej konštantou `2`.

V prípade polí je však celá situácia o niečo zložitejšia. Identifikátor poľa reprezentuje toto pole ako celok len v troch prípadoch:

- ak použijeme identifikátor poľa ako argument operátora `sizeof`,
- ak použijeme identifikátor poľa ako argument operátora `&`,
- ak použijeme identifikátor poľa ako inicializátor v deklarácii referencie.

Vo všetkých ostatných prípadoch použitia sa pole automaticky konvertuje na ukazovateľ na jeho prvý prvok!

Vysvetlime si teraz bližšie spomínané tri prípady, keď pole vystupuje ako celok. Ak aplikujeme na pole operátor `sizeof`, dostaneme jeho veľkosť v bajtoch. O tomto fakte sa môžete veľmi ľahko presvedčiť nasledujúcim programom:

```
float array[50];
printf(„sizeof(float) = %i B\n“,
sizeof(float));
printf(„sizeof array = %i B\n“,
sizeof array);
```

(Mimochodom, automaticky nazývam tento fragment kódu programom, pretože pokladám za samozrejmosť, že tú nevyhnutnú omáčku, ako direktívu(y) `#include`, obalenie kódu do funkcie `main()` a pod., zvládnete bez problémov sami. V budúcnosti už takéto pripomienky robiť nebudem.)

Takže po spustení programu dostaneme na PC nasledujúce výsledky: veľkosť typu `float` je 4 bajty, veľkosť 50-prvkového poľa `array` je 200 bajtov. A keďže 50 × 4 je ešte stále 200, je zrejmé, že operátor `sizeof` vrátil veľkosť celého poľa. A keď sme už pri zisťovaní veľkosti, vyskúšajte si ešte nasledujúci príklad, pomocou ktorého môžeme zistiť počet prvkov poľa (ak ho z nejakého dôvodu nepoznáme):

```
float array[50];
printf(„|array| = %i\n“,
      sizeof array / sizeof array[0]);
```

Po spustení program vypíše, že veľkosť poľa je očakávaných 50 prvkov.

Pole ako každý iný *objekt* C++ (mimochodom, pojem objekt v C++ predstavuje oblasť v pamäti [v angličtine „region of storage“]) má svoju adresu, ktorú môžeme zistiť aplikovaním operátora `&` na jeho identifikátor. Doménovým typom vráteného ukazovateľa je práve typ nášho poľa. Teda ak máme pole `a` obsahujúce desať prvkov typu `int`, hodnotou výrazu `&a` je ukazovateľ na toto pole s typom „ukazovateľ na pole desiatich prvkov typu `int`“. Ak chceme získanú hodnotu uložiť do premennej, musíme správne deklarovať jej typ. Bez nároku na vysvetlenie (zatiaľ), tu je príklad takej deklarácie:

```
int a[10];     // pole
int (*p)[10]; // ukazovateľ
p = &a;
```

Po vykonaní tohto úseku sú nasledujúce dva príkazy ekvivalentné:

```
a[3] = 8;
(*p)[3] = 8;
```

Tretím prípadom vystupovania poľa ako celku je jeho použitie v prípade, že inicializujeme referenciu. Referencie, ako si iste spomínate, sú premenné, ktoré vystupujú ako akési odkazy (aliasy) na iné premenné – každá operácia, ktorú vykonávame s referenciou, sa realizuje na príslušnej odkazovanej premennej. V podstate referencie sú automaticky dereferencovanými ukazovateľmi. Len jediný raz pracujeme priamo s obsahom referenčnej premennej, a to pri jej inicializácii, keď určujeme, na akú premennú sa bude referencia odkazovať.

Referenčná premenná sa môže odkazovať prakticky na ľubovoľný objekt C++ a niet dôvodu, prečo by sa nemohla odkazovať na pole. Jediné, na čo si treba dať pozor, je podobne ako v predchádzajúcom odseku správne deklarovanie typu referencie. S výnimkou deklarácie argumentov a návratovej hodnoty funkcie (a ešte zopár ďalších) musí byť povinne súčasťou deklarácie inicializácia. V ďalšom príklade, podobnom tomu predchádzajúcemu, ukážeme (opäť bez vysvetľovania), ako taká deklarácia spolu s inicializáciou vyzerá:

```
int a[10];          // pole
int (&r)[10] = a; // referencia
```

Opäť po vykonaní tohto úseku sú nasledujúce dva príkazy ekvivalentné:

```
a[3] = 8;
r[3] = 8;
```

Všetky zvyšné prípady použitia identifikátora poľa vedú k jeho automatickej konverzii na ukazovateľ na jeho prvý prvok. Tento ukazovateľ má doménový typ zhodný s typom prvku poľa, nie je však modifikovateľnou l-hodnotou, nemôžeme teda identifikátoru poľa nič priradiť priraďovacím operátorom ani ho inkrementovať či dekrementovať. Hodnotou ukazovateľa je adresa uloženia prvého prvku v pamäti a jeho dereferencovaním by sme tento prvý prvok mohli bez problémov sprístupniť (pretože sa jeho typ zhoduje s doménovým typom ukazovateľa).

Zaiste vás napadlo, ako je to so základnou operáciou realizovanou nad poľami – indexovaním. Aj pri aplikovaní operátora `[]` sa pole konvertuje na ukazovateľ, ale na pochopenie toho, čo sa následne deje, si musíme vysvetliť, čo je to tzv. adresová aritmetika.

## Adresová aritmetika

Pri rozprávaní o operátoroch + a – jazyka C++ sme si nepovedali, že ich operandmi môžu byť za určitých okolností aj ukazovatele. Na prvý pohľad sa môže zdať,

že nemá zmysel niečo k ukazovateľu pripočítavať či odpočítavať, no v skutočnosti to nielenže zmysel má, ale je to neoddeliteľná a nevyhnutná súčasť C++.

Zoberme si najprv operátor +. Jedným z jeho operandov (ale nie oboma!) môže byť ukazovateľ, potom musí byť druhý operand celočíselného typu. Sčítanie prebehne takto: vezme sa celočíselný operand, vynásobí sa veľkosťou doménového typu ukazovateľa a získaná hodnota sa pripočíta k adrese uloženej v ukazovateli. Teraz si predstavte, že nejaký ukazovateľ ukazuje na prvý prvok nejakého poľa. Ak k nemu pripočítame napríklad hodnotu 1, „posunie“ sa tento ukazovateľ o 1 × veľkosť prvku poľa a bude ukazovať na ďalší prvok poľa! Takto môžeme prechádzať po prvkoch poľa pomocou ukazovateľa a všetko, čo treba urobiť pre posun ukazovateľa, je jeho inkrementácia (pre posun v opačnom smere zase dekrementácia). Jediné, na čo treba dávať pozor, je kontrola, či náhodou nepristupujeme k prvkom, ktoré sú už mimo poľa (a ktoré tam vlastne vôbec nie sú). Ukazovateľ na prvý prvok poľa získame jednoducho – použijeme priamo identifikátor poľa.

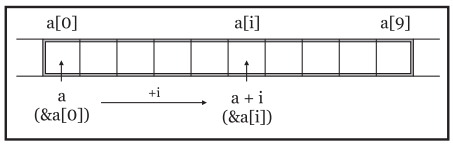
A teraz si vysvetlime, čo sa deje pri použití operátora []. Podľa definície, výraz a[i] znamená to isté (a tak sa aj vypočíta) ako výraz *(a + i). Opíšme si, ako prebehne vyhodnotenie výrazu. Identifikátor a reprezentuje pole. Keďže nenastal ani jeden zo spomínaných troch prípadov, konvertuje sa automaticky na ukazovateľ na prvý prvok celého poľa. K tomuto ukazovateľu sa teraz pripočíta hodnota premennej i. Ako už vieme, to má za následok, že ukazovateľ a + i bude ukazovať na prvok o i pozícií ďalej. Keďže prvý prvok poľa má index 0, prvok o i pozícií ďalej bude mať index i. No a teraz stačí získaný ukazovateľ dereferencovať a máme k dispozícii prvok a[i]. Situáciu osvetlí obr. 3. Krátky príklad:

```
int a[8];
int *p;

for (int i = 0; i < 8; i++)
    a[i] = i;

p = a;
p[2] = 222;
*(p + 5) = 555;

for (i = 0; p < &a[8]; p++, i++)
    printf(„a[%i] = %i\n“, i, *p);
```




*Obr. 3*

V príklade deklarujeme pole a ôsmich celých čísel a ukazovateľ p s doménovým typom int. V cykle for priradíme i-temu prvku poľa hodnotu i. Potom nasleduje priradenie p = a. Už vieme, že v tomto výraze sa a konvertuje na ukazovateľ na a[0]; tento ukazovateľ priradíme do premennej p. Ďalší riadok vyzerá, akoby na ňom bola chyba, indexujeme totiž nie pole, ale ukazovateľ. To je však v úplnom poriadku, operátor [] vďaka svojmu spôsobu vyhodnocovania môžeme použiť aj na ukazovatele (de facto len na ukazovatele – polia sa na ne konvertujú). Výraz p[2] teda prekladač vyhodnotí ako *(p + 2), čo nie je nič iné ako prvok a[2]. Na overenie si vyskúšame na nasledujúcom riadku aj rozpísaný tvar indexovacieho operátora. V druhom cykle for chceme vypísať hodnoty prvkov poľa a, aby sme videli, že sa hodnoty druhého a piateho z nich naozaj zmenili. Pre prístup k jednotlivým prvkom však tentoraz nepoužijeme operátor [], ale pomocný ukazovateľ, ktorý sa bude posúvať pozdĺž celého poľa. V premennej p, ktorú na tento účel použijeme, už máme adresu prvého prvku poľa a, premenná i nám poslúži na zistenie, na ktorý prvok momentálne ukazovateľ ukazuje (ale len pre výpis jeho indexu!). V inicializačnej časti cyklu for vynulujeme index i. Všimnite si ukončovaciu podmienku p < &a[8]. Za normálnych okolností prvok a[8] nie je súčasťou poľa, najvyšší prvok má index 7, ale na takéto účely je povolené použiť jeho adresu a môžeme si byť istí, že nenastane nijaká chyba (výnimka a pod.). Adresa prvku a[8] je vlastne adresou prvého bajtu za koncom poľa. My pri posúvaní ukazovateľa p chceme cyklus skončiť v okamihu, keď jeho hodnota bude s touto adresou zhodná – preto takáto podmienka. V rámci každej iterácie vypíšeme jednak index aktuálneho prvku (i) a jednak jeho hodnotu (*p). Nesmieme zabudnúť zakaždým premennú p inkrementovať a tak sa posunúť na ďalší prvok. Inkrementáciu môžeme vykonať buď vo vyhradenej časti príkazu for (ako v našom príklade), alebo priamo v rámci volania funkcie printf (teda miesto *p tam dáme *p++).

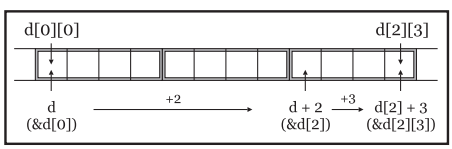
Tí šikovnejší z vás si možno všimli malú kuriozitu týkajúcu sa operátora []. Keďže pri jeho vyhodnocovaní dochádza k sčítavaniu a to je v C++ (ako aj bežne v matematike) komutatívne, môžeme namiesto *(a + 7) napísať *(7 + a), a teda následne namiesto a[7] napísať 7[a]. Je to zvláštne, ale správne a väčšina prekladačov nebude mať námietky.

Povedzme si teraz ešte, ako to vyzerá s indexovaním prvkov viacrozmerných polí. Takéto polia sú realizované pomocou jednorozmerných polí, celá situácia by teda mala byť jasná, no pre istotu si ju objasníme na príklade. Vezmime si už spomínané pole d, deklarované ako:

```
double d[3][4];
```

Indexovanie si opíšeme na sprístupňovaní prvku d[2][3]. Operátor [] sa asociuje zľava doprava, a teda ako prvý príde na rad index [2]. Identifikátor poľa d sa konvertuje na ukazovateľ na jeho prvý prvok. Tým je v tomto prípade štvorprvkové pole premenných typu double, ktorého veľkosť je 4 × sizeof(double), teda väčšinou 32 bajtov. Výraz d[2] sa, samozrejme, vyhodnotí ako *(d + 2). Dvojka sa vynásobí veľkosťou prvku poľa d, čo je spomínaných 32 bajtov. Tento offset sa pripočíta k začiatku poľa, čím dostaneme ukazovateľ na tretí jeho prvok, teda tretí (ergo posledný) riadok celej matice. Jeho dereferencovaním získame tento tretí prvok ako objekt typu „pole štyroch premenných typu double“. Dôležitý je tu fakt, že pole d je z hľadiska prekladača poľom troch prvkov, z ktorých každý má veľkosť 32 bajtov. To, že sú to opäť polia, nie je v danej chvíli podstatné.

Na výsledok výrazu d[2] aplikujeme opäť operátor [], tentoraz s indexom 3. d[2] je poľom štyroch prvkov s veľkosťou 8 bajtov. Index 3 sa teda vynásobí touto veľkosťou a pripočíta k ukazovateľu na začiatok poľa d[2]. Získame tak ukazovateľ na štvrtý prvok tohto poľa, ktorého dereferenciou získavame požadovaný štvrtý prvok zľava tretieho riadka celej matice. Vyhodnocovanie druhého operátora [] môžeme prepísať ako *(d[2] + 3), celý výraz d[2][3] sa teda vyhodnotí ako *(*(d + 2) + 3). Situácia je znázornená aj na obr. 4. Ešte malá poznámka: Vďaka komutatívnosti operátora + môžeme výraz d[2][3] zapísať aj ako 3[2[d]].




*Obr. 4*

To, čo sme si dosiaľ povedali o sčítavaní ukazovateľov s celými číslami, platí v rovnakej miere aj o odčítavaní, teda celočíselný operand sa vynásobí veľkosťou doménového typu ukazovateľa a výsledok sa odčíta od adresy uloženej v premennej typu ukazovateľ. Vďaka tomuto faktu je možné používať na indexovanie polí aj záporné indexy. Hoci na pohľad záporný index nemá zmysel, veď N-prvkové pole obsahuje len prvky s indexmi 0 až N−1, používa sa záporné indexovanie v prípadoch, keď k poľu pristupujeme pomocou ukazovateľa, ktorý ukazuje na iný ako prvý prvok. Teda ak napríklad máme úsek programu:

```
int a[100];
int *p = &a[50];
```

potom p[0] predstavuje 51. prvok poľa a, p[5] 56. prvok a p[−10] zase 41. prvok poľa a. V tomto príklade sme okrem iného aj určovali adresu 51. prvku poľa. Okrem uvedeného spôsobu je možné použiť aj trochu menej prehľadný, ale kratší spôsob:

```
p = a + 50;
```

Výraz je správny, pretože a po konverzii ukazuje na prvý prvok a pričítaním konštanty 50 dostaneme práve ukazovateľ na 51. prvok. Výrazy a + i a &a[i] sú teda ekvivalentné, ako aj výrazy *(a + i) a a[i] (ale to už vieme).

Dosiaľ sme si nepovedali ešte jeden dôležitý detail, že ukazovatele možno od seba odčítavať. Po tom, čo sme si doteraz povedali, vás už iste napadne, ako je to možné. Predpokladáme, že dva ukazovatele ukazujú na prvky toho istého poľa. Potom sa ich rozdiel vypočíta tak, že sa urobí normálne aritmetické odčítanie oboch adries, ktoré sa potom vydelí veľkosťou doménového typu týchto ukazovateľov (je jasné, že ho oba musia mať rovnaký). Ako výsledok dostaneme počet prvkov poľa medzi oboma ukazovateľmi. Teda ak napríklad p1 ukazuje na prvok a[2] a p2 na prvok a[7], hodnotou výrazu p2 − p1 bude číslo 5. V prípade, že nie je splnená podmienka, že oba ukazovatele ukazujú do toho istého poľa, výsledok je nedefinovaný (s jednou výnimkou a tou je odčítavanie od už spomínaného ukazovateľa za posledný prvok poľa). Typom výsledku odčítania dvoch ukazovateľov je špeciálny typ ptrdiff_t, definovaný v hlavičkovom súbore <stddef.h> (obyčajne ako niektorý z celočíselných typov).

Ukazovatele môžu tiež vystupovať ako operandy relačných operátorov. Treba však dodržať určité podmienky. V prvom rade možno porovnávať ukazovatele rovnakých doménových typov, „menší“ bude ten, ktorý ukazuje na objekt na nižšej adrese. Ďalej možno porovnávať ľubovoľný ukazovateľ s ukazovateľom typu void* (realizuje sa ako bežné porovnanie adries, uložených v oboch ukazovateľoch) a taktisto možno ľubovoľný ukazovateľ porovnávať s konštantným výrazom s hodnotou 0. Ukazovateľ s hodnotou 0 je často používaný na signalizáciu, že „neukazuje nikam“. V jazyku C bola na tento účel definovaná známa konštanta (skôr makro) NULL. V jazyku C++ sa odporúča používať priamo literál 0.

Konečne sme sa prelúskali celou problematikou známeho schizofrénneho vzťahu polí a ukazovateľov v C++. Teória okolo tejto záležitosti je na pohľad jednoduchá a exaktná, ale z praxe viem, že jej úplné pochopenie nie je otázkou jedného prečítania. Ak niečomu nerozumiete, prejdite si text ešte raz a ešte raz, vyskúšajte si všetko, aj najmenšie detaily na príkladoch, až kým nebudete mať pocit, že všetkému dokonale rozumiete. Trošku vás zatiaľ bude zrejme brzdiť fakt, že pravdepodobne neviete správne deklarovať rôzne komplikované ukazovatele na

ešte komplikovanejšie štruktúry (čím sú inak jazyky C a C++ neslávne známe). Vydržte, dostaneme sa k tomu.

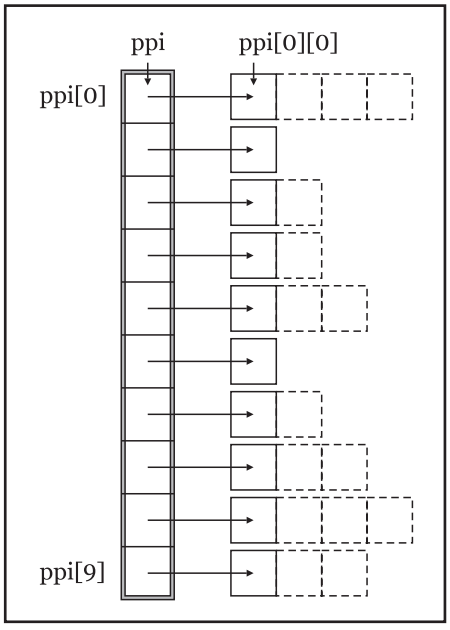
### Programujeme dynamicky

Na záver dnešného rozprávania si povieme niečo o dynamických premenných. Všetky premenné, ktoré sme dosiaľ deklarovali a používali, mali spoločnú vlastnosť: boli známe už v procese prekladu a bolo im pridelené miesto v dátovom segmente alebo na zásobníku (prípadne priamo v registroch procesora). Pre naše krátke a jednoduché programy to úplne stačilo. Predstavme si však, že potrebujeme mať v programe nejaké veľké dátové štruktúry a tieto štruktúry nechceme vytvárať ako lokálne premenné na zásobníku. Nehovorili sme si zatiaľ ešte nič o ukladacích triedach premenných, preto len stručne – všetky naše doterajšie premenné boli lokálne (automatické, deklarované vnútri nejakého bloku). Takéto premenné sa vytvárajú na zásobníku, ktorého veľkosť je obmedzená. Iným typom sú statické premenné, ktoré existujú v dátovom segmente programu, a pokiaľ sú inicializované, sú súčasťou binárneho obrazu programu, ktorý sa naťahuje z disku (zo súboru s programom). Niektoré prekladače, predovšetkým tie 16-bitové, však ukladajú do vykonateľného súboru aj neinicializované statické premenné a potom dostávame obrovské, hoci z hľadiska kódu jednoduché programy.

Dynamické premenné sa naproti tomu vytvárajú (t. j. prideľuje sa im miesto) až za behu programu v oblasti pamäte, nazývanej hromada (po česky halda, po anglicky heap – vyberte si). Vytvorenie dynamickej premennej a pridelenie určitej veľkosti pamäte sa obyčajne nazýva alokácia (opakom je dealokácia). V jazyku C na tieto účely slúžila dvojica knižničných funkcií `malloc()` a `free()`. Ich generálnou nevýhodou bola absencia akejkoľvek typovej informácie, jediné, čo bolo treba zadať, bola veľkosť prideľovanej pamäte. Funkcia `malloc()` vrátila ukazovateľ na začiatok pridelenej pamäte, ktorý bolo treba pretypovať na požadovaný typ. Funkcia `free()` alokovanú oblasť opäť uvoľnila. Celá operácia vyzerala asi takto:

```
char* buf = (char*)malloc(512);
...
free(buf);
```

Premenná `buf` po zavolaní funkcie `malloc()` obsahovala adresu polkilobajtovej oblasti pamäte, s ktorou bolo možné pracovať (vzhľadom na typ ukazovateľa sa táto oblasť primárne brala ako pole znakov). Po skončení práce bolo treba volaním funkcie `free()` túto oblasť opäť vrátiť, aby nedochádzalo k plytvaniu pamäťou.




*Obr. 5*

Jazyk C++ okrem iných vylepšení radikálne zmenil prácu s dynamickými premennými zavedením dvojice operátorov `new` a `delete`. Ich význam je čiastočne zhodný s dvojicou funkcií `malloc()` a `free()`, ich hlavnou výhodou je však fakt, že sú to operátory a ako také sú súčasťou jazyka (obe uvedené funkcie sú súčasťou štandardnej knižnice). Pri dynamickom alokovaní premenných je teda možné pracovať s typovou informáciou a vykonávať príslušné opatrenia súvisiace s vytváraním nových premenných. Tieto opatrenia sa však týkajú objektových premenných, a preto si o nich povieme až v druhej polovici nášho seriálu, keď si budeme hovoriť o objektových črtách jazyka C++.

#### Vytvorenie premennej

Najprv sa budeme zaoberať operátorom `new`. Syntax na jeho použitie (značne zjednodušená) je takéto:

```
new typ ( inicializátor )
```

Operátor `new` vracia ukazovateľ na uvedený typ, ktorý už netreba pretypovávať. Typom môže byť ľubovoľný z nám dosiaľ známych typov, ako aj ich rôzne povolené kombinácie. Otázka, ako správne uviesť takýto zložito kombinovaný typ, spadá do oblasti deklarácií, takže si ju zatiaľ odložíme. Inicializátor predstavuje počiatočnú hodnotu vytvorenej dynamickej premennej. Ukážeme si teraz niekoľko príkladov použitia operátora `new`:

```
int* pi = new int;
double* pd = new double(8.854E-12);
char* pc = new char[128];
long** ppi = new (long *)[10];
```

V prvom príklade sme vytvorili jednoduchú premennú typu `int`. Inicializátor sme neuviedli, preto jej hodnota bude nedefinovaná. Premenná bude prístupná pomocou ukazovateľa ako `*pi`. Druhý príklad sa líši od prvého iba prítomnosťou inicializačnej konštanty. V treťom príklade alokujeme pole 128 znakov. Všimnite si, že hoci alokovaným typom je pole, operátor `new` nevracia ukazovateľ na typ „pole“, ale ukazovateľ na typ prvku poľa. Aj alokácia jednoduchej premennej, aj alokácia poľa premenných vracajú ukazovateľ rovnakého typu. Je to trochu mätúce, ale treba si na to zvyknúť – jednoducho, keď alokujeme pole prvkov typu `T`, musíme použiť ukazovateľ na typ `T` (teda premennú typu `T*`). Prvky nášho poľa sú klasicky prístupné zápisom `pc[i]`.

Posledný príklad je trochu zložitejší. V ňom alokujeme pole desiatich ukazovateľov na typ `long` (t. j. pole desiatich prvkov typu `long*`). Podľa pred chvíľou uvedeného pravidla musíme použiť typ ukazovateľ na ukazovateľ na `long`, teda typ `long**`. Jednotlivé prvky tohto poľa sú neinicializované ukazovatele na typ `long` (obr. 5). Čo s nimi, to už je na nás. Môžeme im napríklad priradiť adresy nových, dynamicky vytvorených polí, z ktorých každé môže byť aj inej dĺžky. Dostaneme tak akúsi maticu s nerovnako dlhými riadkami, možností jej využitia je naozaj veľa.

#### Zrušenie premennej

Tak ako sme dynamickú premennú vytvorili, potrebujeme ju aj zrušiť. Na to slúži druhý z dvojice operátorov, operátor `delete`. Jeho použitie je veľmi jednoduché – jeho argumentom je ukazovateľ na dynamickú premennú, ktorej sa chceme zbaviť. V prípade rušenia iných premenných ako polia použijeme syntax:

```
delete ukazovateľ
```

v prípade rušenia polí zase syntax:

```
delete [] ukazovateľ
```

Dôvod odlišnej syntaxe pre dealokáciu polí spočíva v nutnosti volať pre každý objektový prvok poľa špeciálnu funkciu nazývanú deštruktor, ale o tom až v budúcnosti. Pre polia neobjektových premenných sú obe syntaxe ekvivalentné, ale je žiaduce odlišovať ich.

Ako príklad si uvedieme zrušenie dynamických premenných, vytvorených v predchádzajúcom odseku:

```
delete pi;
delete pd;
delete [] pc;
delete [] ppi;
```

Pred zrušením poľa `ppi` treba prípadne zrušiť dynamické premenné, na ktoré ukazujú jednotlivé prvky tohto poľa (výrazom `delete ppi[i]`, resp. `delete [] ppi[i]`).

#### Ešte malý príklad

Použitie oboch operátorov v jednom programe si demonštrujeme na krátkom príklade, v ktorom vytvoríme štruktúru na uloženie symetrickej matice. Pre takúto maticu platí, že $a_{ij} = a_{ji}$ (t. j. je symetrická podľa hlavnej diagonály). Uložiť teda postačí len jej polovicu. Použijeme spomínané pole ukazovateľov, ktorého každému prvku priradíme ukazovateľ na jeden riadok matice. Riadky budú mať dĺžku od 1 po N, kde N je rozmer matice.

```
const N = 5;
double** a = new (double*)[N];
for (int i = 0; i < N; i++)
    a[i] = new double[i + 1];
...
for (i = 0; i < N; i++)
    delete [] a[i];
delete a;
```

Najprv vytvoríme pole ukazovateľov `a`, potom v cykle jednotlivé subpolia (riadky matice) s rozmerom `i + 1`, kde `i` je aktuálny index do poľa `a`. Po vytvorení štruktúry môžeme pristupovať k jednotlivým prvkom matice pomocou bežného výrazu `a[i][j]`, len musíme dávať pozor, aby index riadka `i` bol väčší ako index stĺpca `j`. Čo sa udeje pri vyhodnocovaní tohto výrazu, ponechávam ako domácu úlohu (nemajte obavy, nabudúce bude aj riešenie). Po skončení práce s maticou zrušíme najprv jednotlivé riadky a potom aj samotné pole ukazovateľov.

## Jedenásta časť: DEKLARÁCIE I.

Vzhľadom na trochu väčšiu náročnosť vás vopred varujem, aby ste radšej čítali pozorne a nepreskakovali odseky, inak sa môže stať, že budete mať z celého výkladu iba guláš. Ako vždy budem podopierať teóriu príkladmi (a ako vždy v takýchto situáciách nie príliš zmysluplnými, skôr účelovými), takže to hádam spolu dokážeme… no dobre, dobre, nechám už politiku na pokoji.

### Základné pojmy

Výklad začneme objasnením určitých základných pojmov, ktoré sa problematiky deklarácií úzko týkajú. Pod menom budeme rozumieť objekt, funkciu, enumerátor (ešte nevieme, čo to je, ale dostaneme sa k tomu), typ, hodnotu alebo návestie. Nové meno zavádzame do programu práve deklaráciou. Každé meno má pridruženú oblasť programu, v ktorej môže byť použité. Túto oblasť budeme nazývať rozsah platnosti, čo je voľný preklad anglického termínu scope. Každé meno má svoj typ, na

základe ktorého môžeme určiť, čo sa s menom dá robiť a kde ho môžeme použiť. Jedno meno použité v rôznych miestach programu (rôznych blokoch, funkciách či súboroch) môže, ale nemusí reprezentovať tú istú entitu.

Objekt je oblasť, kde sú uložené údaje (ang. region of storage). Pojem objekt sa prakticky zhoduje s pojmom premenná, tak ako sme si ju dosiaľ prezentovali. Nie každý objekt však musí byť prístupný svojím menom (spomeňme si na dynamicky alokované premenné, ktoré sú prakticky prístupné len pomocou ukazovateľov na ne). Pomenované objekty majú definovanú tzv. ukladaciu triedu (storage class), ktorá určuje, kedy a ako sa objekt vytvorí, ako dlho bude existovať a kedy a ako sa zničí. Význam údajov uložených v objekte závisí od typu výrazu, ktorým k objektu pristupujeme.

## Deklarácie a definície

Je nevyhnutné, aby ste hneď od začiatku chápali rozdiel medzi deklaráciou a definíciou. Deklarácia, ako sme si už povedali, zavádza do programu jedno alebo viacero mien. Definíciou je taká deklarácia, ktorá súčasne spôsobí, že prekladač pri preklade daného úseku programu vyhradí pre deklarované meno pamäť. Deklarácie, ktoré nie sú definíciami, sa dajú zhrnúť do nasledujúcich kategórií: deklarácie funkcií, ktoré neobsahujú telo funkcie (pozri predposledný diel – ukážkový program a v ňom spomínané prototypy funkcií), deklarácie bez inicializácie obsahujúce špecifikátor `extern` (určuje, že meno má tzv. externé linkovanie – preberieme neskôr) a `typedef` deklarácie (vytvárajúce nové, programátorom definované typy); vynechali sme ešte niekoľko prípadov týkajúcich sa deklarácie tried a ich členov, túto problematiku však odložíme do druhej polovice seriálu. Všetky ostatné deklarácie sú súčasne definíciami, a teda spôsobia alokáciu pamäte prekladačom.

Uveďme si niekoľko príkladov definícií:

```
int a;
extern double pi = 3.14;
int f(int x) { return x*x; }
```

a niekoľko príkladov deklarácií, ktoré nie sú definíciami:

```
extern int a;
extern double pi;
int f (int x);
typedef int BOOL;
```

Pre každý objekt, funkciu a enumerátor musí byť v programe práve jedna definícia (ktorá má na starosti rezerváciu pamäte). Funkcia, ktorú nikde v programe nevoláme ani nezisťujeme jej adresu, nemusí byť definovaná, hoci sme uviedli jej prototyp.

## Rozsah platnosti

V C++ existujú štyri rôzne rozsahy platnosti: lokálny, funkčný, súborový a triedny. Ten posledný nám o niečo starším možno evokuje socialistickými ideológmi definovaný triedny boj, týka sa však rozsahu platnosti členov objektových tried a zatiaľ si ho nebudeme opisovať.

Lokálny rozsah platnosti (local scope) majú mená deklarované v bloku. Blokom sa myslí oblasť uzavretá v krútených zátvorkách, teda zložený príkaz, telo funkcie a pod. Meno s lokálnym rozsahom platnosti môžeme použiť len v bloku, v ktorom je definované, a prípadne v blokoch, ktoré sú v tomto bloku vnorené. V prípade lokálneho rozsahu platnosti obyčajne hovoríme o lokálnych menách.

Funkčný rozsah platnosti (function scope) je špecifickým rozšírením lokálneho rozsahu a mená takto deklarované možno použiť hocikde vo funkcii, v ktorej sú deklarované (to znamená, že aj skôr, ako vôbec boli deklarované, čo nie je možné pri predchádzajúcom type rozsahu, keď, ako už vieme, môžeme použiť meno až za bodom jeho deklarácie). Jedinými entitami, ktoré v C++ môžu mať funkčný rozsah platnosti, sú návestia. Isto si spomínate, ako deklarujeme návestie, a na to že v príkaze `goto` môžeme použiť aj návestie, ktoré sa nachádza o niekoľko riadkov programu dopredu. Rozsahom platnosti návestia je funkcia, v ktorej bolo deklarované; raz definované návestie nemožno deklarovať v tej istej funkcii znova. Návestia však nezdieľajú priestor mien s ostatnými identifikátormi, a preto môžeme mať v jednej funkcii napríklad návestie `end` aj premennú `end`.

Súborový rozsah platnosti (file scope) majú všetky mená deklarované mimo akéhokoľvek bloku alebo triedy. Použiť takéto meno môžeme hocikde v danom súbore (súbor berieme ako prekladovú jednotku [translation unit] – preklad programu sa deje po jednotlivých súboroch, spomínate si ešte na druhé pokračovanie seriálu?), ale takisto až za miestom deklarácie. Mená deklarované so súborovým rozsahom platnosti sa často nazývajú globálnymi a minimálne v danej prekladovej jednotke musia byť jedinečné. Funkcie môžeme deklarovať iba s týmto rozsahom platnosti – neexistujú lokálne funkcie ako v Pascale!

V podstate nám na začiatok stačí rozlišovať lokálne a globálne mená a vedieť, kedy ktoré môžeme používať. Na nasledujúcom príklade (tentoraz ide o kompletný program!) si ukážeme oba typy mien:

```
#include <stdio.h>

int a;

void foo()
{
     int b;
     printf(„Entering foo()...\n");
     a = 13;
     b = 25;
     printf(„a = %i, b = %i\n", a, b);
     printf(„Exiting foo()...\n");
}

void main()
{
     int b;
     printf(„Entering main()...\n");
     a = 5;
     b = 8;
     printf(„a = %i, b = %i\n", a, b);
     foo();
     printf(„a = %i, b = %i\n", a, b);
     printf(„Exiting main()...\n");
}
```

V príklade máme definovanú jednu globálnu premennú `a`. Okrem hlavnej funkcie `main()` sme definovali ešte jednu pomocnú funkciu `foo()`. V oboch funkciách sa nachádza deklarácia (pravda, aj definícia, ale nebudem to už ďalej zdôrazňovať) lokálnej premennej `b`. Volaná funkcia `foo()` napohľad modifikuje tie isté premenné ako funkcia `main()`, ale iba premenná `a` v oboch prípadoch predstavuje ten istý objekt, a to z toho dôvodu, že je deklarovaná ako globálna. Znamená to takisto, že všade v našom programe meno `a` reprezentuje tú istú premennú (pokiaľ ho neprekryjeme - pozri ďalší odsek) a nemôžeme nikde definovať inú globálnu premennú s rovnakým názvom. Meno `b` v každej z funkcií predstavuje rôzny objekt – miestnu, lokálnu premennú, o čom sa môžeme presvedčiť aj z kontrolných výpisov. (Poznámka: Výrok týkajúci sa jedinečnosti globálneho objektu s názvom `a` platí len pre jednosúborový program, ako napríklad ten náš. V časti o linkovaní sa dozviete, ako je to pri viacsúborových programoch.)

V prípade, že deklarujeme lokálnu premennú s rovnakým názvom, ako už existujúca (teda deklarovaná) globálna premenná (čo je, samozrejme, dovolené), bude pôvodná premenná pod svojím menom ďalej nedostupná. Ak chceme pracovať s oboma, v prvom rade nebudeme vymýšľať a lokálnu premennú nazveme ináč. Niekedy však nemáme iné východisko a vtedy nám na sprístupnenie globálnej premennej poslúži špeciálny operátor, o ktorom sme ešte nehovorili, a to operátor „rozlíšenia rozsahu platnosti" (ako vždy je lepší pôvodný anglický termín scope resolution operator), ktorého symbol je `::` (dve dvojbodky za sebou, inak aj „štvorbodka"). Tento operátor je unárny a prefixový, jeho asociativita nemá význam. Jeho operandom musí byť meno so súborovým rozsahom platnosti. Operátor nemá nijaký vplyv na obsah premennej ani na jeho interpretáciu (toho obsahu), používa sa v tomto prefixovom tvare len na sprístupnenie inak zakrytých globálnych mien. Príklad:

```
#include <stdio.h>

int i = 11;

void main()
{
     int i = 22;
     printf(„i = %i, ::i = %i\n",
i, ::i);
     {
          int i = 33;
          printf(„i = %i, ::i =
%i\n", i, ::i);
     }
}
```

V programe sa nachádzajú tri premenné `i`. Jedna globálna s hodnotou `11` a dve lokálne, jedna s hodnotou `22` na úrovni tela funkcie, druhá s hodnotou `33` vo vnorenom bloku. Druhá deklarácia zakrýva prvú a tretia druhú, globálna premenná je však stále prístupná prostredníctvom operátora `::`, ako vidieť z výpisov. Neexistuje však spôsob, akým by sme sprístupnili zakrytú lokálnu premennú (v našom príklade premennú `i` s hodnotou `22`).

Na tomto mieste spomeniem ešte malú zaujímavosť – bod deklarácie nejakého mena sa nachádza hneď za jeho deklarátorom a pred prípadným inicializátorom. V praxi z toho vyplýva možnosť inicializácie premennej samej sebou (ale neviem, na čo by to bolo dobré) a, naopak, nemožnosť inicializovať lokálnu premennú globálnou premennou rovnakého mena:

```
double t = 1.0;
{
     double t = t;
}
```

Lokálna premenná `t` sa neinicializuje obsahom globálnej premennej `t`, ale svojím vlastným obsahom (ktorý navyše nie je definovaný).

## Linkovanie

Všetky mená so súborovým rozsahom platnosti (teda globálne mená) v programe majú pridruženú vlastnosť zvanú linkovanie (linkage). Upozorňujem, že vlastnosť „linkovanie" nemá nič spoločné s činnosťou „linkovanie", ktorá sa deje počas prekladu programu. Takýto preklad som zvolil z toho dôvodu, že termín „spájanie" viac inklinuje k opisu činnosti ako k názvu vlastnosti.

Linkovanie v C++ môže byť dvojakého charakteru: interné a externé. Mená s interným linkovaním sú lokálne pre daný súbor (prekladovú jednotku) a z častí programu nachádzajúcich sa v iných súboroch nie sú viditeľné ani inak prístupné. Ak máme v jednom súbore deklarované nejaké meno s interným linkovaním, môžeme to isté meno použiť v inom súbore bez toho, aby nám prekladač vynadal. Samozrejme, v inom súbore toto meno bude predstavovať úplne inú entitu (premennú, funkciu, typ a pod.).

Druhým typom linkovania je externé linkovanie. Ako už isto tušíte, meno, ktoré má externé linkovanie, bude predstavovať v celom programe (t. j. v rámci všetkých

jeho súborov!) jedinečnú entitu. Takéto meno musí byť v rámci všetkých súborov definované práve raz, deklarované môže byť aj viackrát, ale všetky jeho deklarácie sa musia zhodnúť na použitom type/typoch (pri premenných je to jeden typ – typ premennej, pri funkciách je to viacero typov – typy argumentov, návratovej hodnoty atď.). Ak vám nie je jasné, na čo je dobré deklarovať jedno meno viackrát, odpoveď je jednoduchá – meno s externým linkovaním je prístupné zo všetkých súborov práve vďaka deklarácii v každom zo súborov, v ktorých ho chceme používať. Pritom práve v jednom z nich bude aj naozaj definované, t. j. bude mu pri preklade pridelená pamäť.

O tom, ako rozlišujeme interné a externé linkovanie, si povieme o niečo neskôr, keď budeme opisovať štruktúru deklarácie mena.

## Ukladacia trieda

Posledným z opisovaných atribútov deklarovaných mien je ukladacia trieda. Táto vlastnosť súvisí s uložením objektov v pamäti a na jej základe rozlišujeme dva typy objektov: automatické a statické.

Automatické objekty sa môžu nachádzať len v rámci nejakého bloku, vznikajú (fyzicky v pamäti) pri každom prechode programu miestom svojej deklarácie a zanikajú pri opustení bloku. Je zrejmé, že automatické objekty môžu mať iba lokálny rozsah platnosti. Prekladač prakticky vždy umiestňuje automatické objekty na zásobník (t. j. do zásobníkového segmentu programu), čo v praxi vyzerá asi tak, že obyčajne pri vstupe do funkcie sa na zásobníku vyhradí miesto (posunom ukazovateľa zásobníka, pre 386+ je to známy register ESP, resp. SP) pre všetky lokálne premenné funkcie a na konci funkcie sa miesto zase zruší. Pokiaľ automatické objekty sami explicitne neinicializujeme, ich implicitným obsahom budú tie smeti, ktoré na danom mieste zásobníka práve boli. Prípadná inicializácia sa vykonáva pri každom prechode deklaráciou.

Všetky lokálne premenné funkcií a všetky premenné deklarované v rámci zložených príkazov sú automatické. Pozor však, automatickými sú aj premenné deklarované v rámci výkonného príkazu príkazov `for`, `if` a pod. Teda v nasledujúcom príklade:

```
int x = -1;
while (++x < 100)
     for (int i = 0; i < 5; i++)
     { ... }
```

je premenná `i` automatická a vzniká a zaniká pri každej iterácii vonkajšieho cyklu `while` (čiže stokrát). Opäť tento fakt nadobudne význam až pri premenných objektového typu. Súčasne je premenná `i` nedostupná mimo príkazu `while`, pretože jeho výkonný príkaz sa považuje z hľadiska rozsahu platnosti za blok. Pri použití niektorého zo skokových príkazov môžeme skočiť aj dovnútra bloku, nesmieme však preskočiť deklaráciu automatickej premennej, spojenú s inicializáciou. Ak takýto skok potrebujeme, musíme uzavrieť automatickú premennú do samostatného bloku. Príklad nesprávneho skoku:

```
void foo()
{
     goto here;  // chyba!
     ...
     int x = 123;
     ...
here:
     ...
}
```

a správneho:

```
void bar()
{
     goto here;  // ok
     ...
     {
          int x = 123;
          ...
     }
     ...
here:
     ...
}
```

Naproti tomu statické objekty existujú a zachovávajú si svoju hodnotu (samozrejme, pokiaľ ju nezmeníme) počas celého behu programu. Vznikajú pri štarte programu obyčajne ešte pred volaním funkcie `main()` (termín „vznikajú“ nadobudne význam až v súvislosti s objektovými typmi), implicitne sa inicializujú na nulu pretypovanú na svoj typ (polia sa vynulujú po prvkoch, objektové premenné sa inicializujú špeciálne) a zanikajú po skončení programu (návratom z `main()` alebo volaním `exit()`). Všetky globálne objekty sú statické, lokálnym takúto ukladaciu triedu musíme v prípade potreby explicitne predpísať (kľúčovým slovom `static`, ale k tomu sa o chvíľu dostaneme).

## Tajomstvo deklarácií

Tak sme si opísali, aké rôzne vlastnosti majú deklarované mená. Tieto vlastnosti menám priraďujeme práve prostredníctvom deklarácií. Ale určite už netrpezlivo čakáte, kedy sa konečne k opisu tých deklarácií dostaneme. Takže pozor… teraz! Hm, ale raz neviem, z ktorej strany mám začať. Celá táto problematika je jeden veľký prepletenec faktov a je prakticky nemožné podať ho lineárne so zachovaním nejakej rozumnej príčinnej súvislosti, čím som chcel povedať, že sa nedá povedať najprv A, potom B a nakoniec C, lebo opis A si vyžaduje znalosť C… no myslím, že vyvolávam zbytočný zmätok, takže radšej začnime.

Celé tajomstvo je na prvý pohľad veľmi jednoduché. Deklarácia jedného či viacerých mien vyzerá takto:

*spec-list decl-list ;*

kde „spec-list“ je zoznam špecifikátorov a „decl-list“ zoznam deklarátorov. Deklarátory opisujú, čo vlastne ideme deklarovať a aký názov to bude mať, špecifikátory zase vlastnosti novo deklarovaného mena/mien. Za určitých okolností možno jedno alebo druhé vynechať, povieme si o tom pri konkrétnych prípadoch. Najjednoduchší príklad deklarácie:

```
double val;
```

V tejto deklarácii je deklarátorom identifikátor novej premennej `val`, špecifikátorom kľúčové slovo `double`, ktoré určuje typ tejto premennej.

Okrem uvedeného vzoru deklarácie existujú ešte špeciálne prípady, ako napríklad `asm` deklarácia, ktorá je však implementačne závislá a nebudeme si ju tu zatiaľ opisovať, potom deklarácia šablón alebo deklarácia umožňujúca pripojiť k programu C++ skompilované funkcie napísané v jazyku C. K posledným dvom sa vrátime v budúcnosti.

Teraz si výklad rozdelíme na dve časti. Najprv si povieme o špecifikátoroch, pod pojmom deklarátor si zatiaľ predstavujte ten najjednoduchší – identifikátor premennej, potom sa budeme venovať rôznym deklarátorom. Obe témy sa dosť prekrývajú a najlepšie by bolo preberať ich paralelne, ale to sa dosť ťažko realizuje aj v bežnom živote, nieto ešte v časopise. Dovolím si vás poprosiť, aby ste sa po prebratí oboch častí vrátili k tej prvej a ešte raz si všetko prešli (prípadne môžete takto iterovať aj viackrát).

## Špecifikátory

Špecifikátory, ktoré môžeme použiť v deklarácii, sa dajú rozdeliť takto: špecifikátory ukladacej triedy, funkčné špecifikátory, špecifikátory typu a špecifikátor `typedef`. V zozname v deklarácii sa môže nachádzať aj viac ako jeden špecifikátor, ale zase nie je možné kombinovať všetko so všetkým. Na určenie, ktoré špecifikátory môžeme spolu skombinovať, je asi najlepšie použiť zdravý rozum (a trochu nasledujúcej teórie).

**Špecifikátory ukladacej triedy**

Sú štyri: `auto`, `register`, `static` a `extern` a sú to všetko kľúčové slová C++. Netýkajú sa však len ukladacej triedy, ale ako uvidíme o chvíľu, aj linkovania. Prvý z nich, špecifikátor `auto` určuje, že deklarované meno bude automatickým objektom. Použiť ho môžeme iba pri deklarácii lokálnych objektov (v rámci nejakého bloku) a pri deklarácii formálnych argumentov. V praxi som ho však ešte v živote nevidel. Je totiž úplne zbytočný, pretože lokálne objekty sú automaticky automatické (to je slovná hračka, čo?).

Druhý špecifikátor `register` je v podstate tým, čo `auto`, s rovnakými pravidlami použitia, navyše však znamená pre prekladač pomôcku alebo náznak (hint), že objekt takto deklarovaný sa bude používať dosť často a bolo by dobré, keby bol umiestnený priamo v registri procesora. Prekladač sa však aj tak zariadi po svojom – ak objekt nemôže uložiť do registra (napríklad niekde získavame jeho adresu operátorom `&` alebo sa do registra nevojde), tak špecifikátor `register` ignoruje, inak ho často v rámci optimalizácie doplní aj sám. Príklad – asi najkratšia (nie najrýchlejšia!) implementácia funkcie `strcpy()`, ktorá kopíruje jeden reťazec do druhého:

```
void strcpy(register char *dst,
            register char *src)
{
     while (*dst++ = *src++);
}
```

Reťazec, ako vieme, je v C++ ukončený znakom `\0`. Argument `src` predstavuje zdrojový reťazec (ukazovateľ a súčasne pole znakov – to už ovládame, pozri minulú časť), argument `dst` je ukazovateľom na miesto, kam sa má reťazec skopírovať (predpokladáme, že je to miesto dostatočne veľké). V každej iterácii sa skopíruje jeden znak zo `*src` na `*dst` a oba ukazovatele sa posunú. Všimnite si, že priraďovací výraz používame priamo ako podmienkový výraz príkazu `while`. Kopírovanie sa zastaví po zápise znaku `\0`, keď bude výsledkom priraďovacieho výrazu nulová hodnota a cyklus `while` sa ukončí. Oba ukazovatele sa zrejme používajú dostatočne intenzívne, aby malo zmysel ich uložiť do registrov.

Samozrejme, špecifikátor `register` môžeme použiť aj pri deklarácii lokálnych premenných:

```
char msg[] = „hello“;
register char *ptr = msg;
while (*ptr = toupper(*ptr))
     ptr++;
```

V príklade máme znakové pole `msg` a ukazovateľ `ptr`, ktorý ukazuje na začiatok poľa. V cykle prechádzame jednotlivými znakmi reťazca a meníme ich na veľké písmená. Ukazovateľ slúži na pamätanie polohy v poli a bolo by dobré, keby sa nachádzal v registri. Opäť používame úsporný zápis cyklu, ktorý sa skončí v okamihu, keď budeme chcieť previesť na veľké písmeno záverečný znak `\0`. Pre použitie makra `toupper()` musíme vložiť do programu hlavičkový súbor `<ctype.h>`.

Posledné dva špecifikátory `static` a `extern` sa týkajú linkovania. Kľúčové slovo `static` explicitne vyjadruje, že globálne meno bude mať interné linkovanie, čo, samozrejme, znamená, že takto deklarované (a súčasne definované!) meno bude viditeľné len v rámci súboru, ktorý deklaráciu obsahuje. Zdôrazňujem, že toto sa týka globálnych mien, teda mien so súborovým rozsahom platnosti! (Deklarácia globálnych `static` premenných sa používala predovšetkým v jazyku C

pri snahe o modulárne programovanie – čo súbor, to samostatný modul a jeho privátne premenné nemali byť zvonka viditeľné.) Deklarácia s kľúčovým slovom `extern` znamená presný opak, teda prirađuje globálnemu menu externé linkovanie, ale len ak toto meno nebolo už raz deklarované ako `static`. Pokiaľ neuvedieme špecifikátor `extern`, meno má externé linkovanie automaticky, ale (vráťte sa o niečo späť) je tu jeden obrovský rozdiel – deklarácia so špecifikátorom `extern` a bez inicializácie nie je definíciou! V praxi sa používa asi nasledujúci postup – v tom súbore, v ktorom má byť premenná definovaná, sa `extern` neuvedie, bude tam iba klasická definičná deklarácia. Vo všetkých ostatných sa v deklarácii špecifikátor `extern` použije, a to obyčajne tak, že táto deklarácia sa zapíše do hlavičkového súboru, ktorý sa potom vloží (`#include`) do potrebných zdrojových súborov. Je to asi mierne zamotané, skúsme si vec objasniť na nasledujúcom príklade:

Toto je súbor `subor1.cpp`:

```
// subor1.cpp
int a = 8;
void foo() { ... }
```

K nemu pridružíme hlavičkový súbor `subor1.h`:

```
// subor1.h
extern int a;
void foo();
```

Potom máme ešte dva zdrojové súbory, v ktorých chceme používať premennú `a` a funkciu `foo()`:

```
// subor2.cpp
#include „subor1.h“
void bar() { a = 10; }

// subor3.cpp
#include „subor1.h“
void main()
{
     foo();
     ...
}
```

Nehľadajte v príkladoch hlbší zmysel, nie je tam… Z výpisov vidíme, že vďaka hlavičkovému súboru `subor1.h` máme sprístupnenú premennú `a` ako aj funkciu `foo()` aj v ostatných súboroch. Obe tieto mená majú implicitne externé linkovanie (lebo sme ich nedeklarovali ako `static`). Premennej `a` bola pridelená pamäť pri preklade po nájdení jej deklarácie v súbore `subor1.cpp`, v ostatných súboroch si iba prekladač poznačil, že meno `a` je definované (a jeho použitie neznamená chybu), ale že pamäť mu bola pridelená niekde inde a všetky odkazy na jeho skutočné fyzické umiestnenie v pamäti (to je treba vedieť, keď chceme s premennou pracovať) sa vyriešia až počas fázy linkovania. Pokiaľ však deklarujeme nejakú premennú ako `extern` a nikde v programe ju nedefinujeme, dostaneme od linkera chybové hlásenie typu „Symbol xxx not defined“. Čo sa týka funkcie `foo()`, tam je to jasné, tým, že prekladač videl jej prototyp (ten je takisto v hlavičkovom súbore), považuje ju za deklarovanú. Ak ju nikde nedefinujeme alebo definujeme v inom súbore ako statickú, linker nám takisto vynadá.

Priznávam, že toto je jedna z najzložitejších oblastí C++ (a to sme sa ešte nedostali k virtuálnym funkciám, preťažovaniu operátorov či šablónam!). Treba si skrátka celú vec nechať uležať v hlave, vrátiť sa k nej neskôr, prípadne skúšať na príkladoch. Ešte si uvedieme, aké kombinácie použitia oboch špecifikátorov sú povolené a aké nie:

```
static int a;
int a;
```

Prvá deklarácia prirađuje premennej `a` interné linkovanie, druhá je však chybou, prekladač ju bude pokladať za redeklaráciu už raz definovanej premennej.

```
static int b;
extern int b;
```

Prvá deklarácia prirađuje premennej `b` interné linkovanie, druhá toto linkovanie nijako nemení. Konštrukcia je korektná, prekladač nebude namietať.

```
int c;
static int c;
```

V prvej deklarácii premenná `c` získa automaticky externé linkovanie. Druhá deklarácia je chybná (redeklarácia existujúcej premennej, navyše ani linkovanie nesedí).

```
extern int d;
static int d;
```

V prvej deklarácii priradíme premennej `d` explicitne externé linkovanie, druhá deklarácia by mala byť chybná (nesedí linkovanie), ale obyčajne ju prekladače povolia (maximálne vydajú varovanie). Zvyšné kombinácie (`extern` a nič, resp. nič a `extern`) sú, samozrejme, v poriadku, lebo ide o dvojicu deklarácia-definícia, ktorá je povolená.

Zatiaľ čo špecifikátor `extern` môžeme použiť aj pri deklarácii mien v rámci nejakého bloku (t. j. mien s lokálnym rozsahom platnosti) a jeho význam zostane taký istý (ledaže je možno výhodnejšie to nerobiť na úrovni bloku, ale na úrovni súboru, aby príslušné meno bolo prístupné všetkým blokom – funkciám v celom súbore), špecifikátor `static` použitý na lokálne meno má význam úplne odlišný. Predovšetkým sa nedá povedať, že `static` lokálne meno má interné linkovanie – to má vždy, nikdy ho nebudeme vidieť ani z iných funkcií, nieto ešte z iných súborov. Takže tým, že deklarujeme lokálne meno ako `static`, hovoríme prekladaču, že toto meno bude statické (v zmysle ukladacej triedy). Implicitne je totiž každé lokálne meno, ktoré nie je `extern`, automatické (akoby bolo deklarované s `auto`). A teraz je to už úplne zauzlené… alebo nie? Dúfam, že nie. Malý príklad na statickú lokálnu premennú:

```
int check_in()
{
     static int count = 0;
     return ++count;
}
```

Funkcia `check_in()` slúži na počítanie napríklad výskytu nejakých udalostí. Vždy, keď udalosť nastane, zavoláme túto funkciu a ona nám vráti jej akoby poradové číslo. Zrejme je na implementáciu tejto činnosti potrebná nejaká vnútorná pamäť. Tú predstavuje práve statická premenná `count`. V príklade ju inicializujeme na nulu (čo je vlastne zbytočné, lebo tak sa inicializuje implicitne) – dôležité však je, že inicializácia statickej premennej sa vykoná iba raz, a to pri štarte programu (resp. pokiaľ ide o takúto jednoduchú premennú, tak ešte pri preklade, keď sa jej pridelí miesto v pamäti; inicializačná hodnota bude zapísaná priamo v binárnom obraze programu, hovorili sme si o tom v časti venovanej dynamickým premenným). Každý ďalší prechod programu miestom deklarácie premennej `count` už prakticky nerobí nič. Vďaka svojej statickosti si `count` drží svoju hodnotu aj medzi volaniami funkcie `check_in()`, takže každé jej volanie vráti poradové číslo o jednotku vyššie.

## Funkčné špecifikátory

Sú dva, jeden z nich, `virtual`, sa týka virtuálnych členských funkcií objektových typov a povieme si o ňom až v druhej polovici seriálu. Druhým špecifikátorom je kľúčové slovo `inline`. Použiť ho môžeme len pri deklarácii alebo definícii funkcií, ale v prípade, že máme sekvenciu deklarácia (prototyp) – volanie funkcie – definícia (telo), musíme `inline` uviesť už pri deklarácii. Dôvodom je fakt, že tento špecifikátor prirađuje funkciám implicitne interné linkovanie. Jeho význam je jednoduchý, slúži opäť ako pomôcka, náznak pre prekladač, že funkcia, ktorú deklarujeme, je natoľko jednoduchá a často používaná, že sa neoplatí volať ju klasickou metódou, t. j. obyčajne inštrukcia CALL s adresou funkcie a telo funkcie sa nachádza niekde samostatne, ale namiesto toho sa každé volanie tejto funkcie nahradí priamo jej telom. To znamená, že zatiaľ čo pri klasickom spôsobe sa v programe telo funkcie nachádza raz a každé volanie sa realizuje jednou inštrukciou, pri deklarácii `inline` sa telo funkcie v programe nachádza toľkokrát, koľkokrát funkciu voláme. Prekladač môže opäť tento špecifikátor ignorovať, obyčajne v prípade, keď usúdi, že funkcia je pridlhá alebo obsahuje cykly.

Ako sme už povedali, `inline` funkcie majú interné linkovanie; ak teda chceme takúto funkciu použiť vo viacerých súboroch programu, musíme nielen jej deklaráciu, ale aj jej definíciu uviesť do hlavičkového súboru. Toto pri klasických funkciách nemôžeme – linker by ohlásil niečo ako „Multiple definition of xxx“ a skončil s chybou. Uveďme si jednoduchý príklad:

```
// max.h
inline int max(int a, int b)
{
     return a > b ? a : b;
}

// main.cpp
#include „max.h“
int main()
{
     int x, y;
     ...
     int m = max(x, y);
     ...
     return 0;
}
```

Ide o dva súbory – `max.h` a `main.cpp`. V prvom z nich máme `inline` deklaráciu funkcie `max(`, ktorá vráti väčšie z dvoch celých čísel. O jej jednoduchosti niet pochýb. V druhom súbore funkciu používame a v rámci tohto súboru má `max()` interné linkovanie. V každom ďalšom súbore, v ktorom by sme ju potrebovali, nám stačí vložiť súbor `max.h` a funkcia je k dispozícii (ale opäť s interným linkovaním; v konečnom dôsledku teda bude v programe existovať toľko funkcií `max()`, v koľkých súboroch ich deklarujeme; práve fakt, že funkcie sú `inline`, však zabráni existencii viacerých zbytočných rovnakých kódov v binárnom obraze programu – v skutočnosti tam tých kódov bude viac rovnakých, ale priamo v miestach ich príslušných volaní). No, to som tomu zase dal…

## Ešte nekončíme

Rozsah článku ma, bohužiaľ, núti prerušiť výklad, hoci sme si ešte nepovedali o špecifikátoroch typu a o `typedef` špecifikátore. Necháme to nabudúce, keď začneme hovoriť aj o deklarátoroch. Ako tak pozerám späť, zrejme deklarácie dokončíme až v trinástom pokračovaní, potom venujeme ešte jednu časť štandardným konverziám, jednu alebo dve štandardnej knižnici jazyka C a potom sa už môžeme pustiť do objektovej polovice seriálu.

## Dvanásta časť: DEKLARÁCIE II.

Tak vás vítam v novom roku 1999. Dúfajme, že bude lepší ako tie doposiaľ (a horší ako tie, čo ešte len prídu). V každom prípade je posledným rokom, ktorý sa začína dvojčíslím 19. Zámerne nepíšem, že je posledným rokom tohto storočia, pretože podľa mňa (a podľa mnohých racionálne zmýšľajúcich ľudí) 21. storočie a súčasne tretie tisícročie sa začína až 1. januára 2001 o 0.00 hod. miestneho času. A to z toho dôvodu, že kresťanský letopočet, odvíjajúci sa od roku narodenia Krista, sa začal rokom 1, jeho prvé desaťročie sa skončilo rokom 10 (po desiatich rokoch od začiatku), prvé storočie rokom 100 a tak ďalej. Teraz máme 20. storočie, ktoré sa skončí v okamihu, keď sa skončí rok 2000 (= 20 × 100). Samozrejme, náš letopočet sa nezačal tak, že niekto vyhlásil: „Teraz je rok 1, začíname!“ Bol zavedený (ak sa nemýlim) o niečo neskôr, po spočítaní času, ktorý uplynul od cirkvou stanoveného počiatku, teda od roku narodenia Krista (nepýtajte sa ma, ktorý pápež bol vtedy hlavou cirkvi a kto výpočty realizoval, to vám veru nepoviem). Ale v každom prípade okamih, v ktorom sa rok 1999 zmení na rok 2000, bude asi veľmi vzrušujúci a môžeme ďakovať osudu, že sme sa narodili v správny čas a tento prechod zažijeme. S oslavami nového milénia však bude treba počkať ešte 365 dní.

Ak máme veriť všetkým katastrofickým scenárom, ktoré opisujú stav sveta 1. januára 2000 ako takmer apokalyptický, mali by sme sa už pomaly začali baliť, kopať si zemľanku niekde v záhrade, nakúpiť si zásoby jedla aspoň na dva roky a podobne. Alebo nie? Problém roku 2000 (Y2K, Year 2k problem) sa čoraz neodbytnejšie hlási k slovu a vďaka „zaručeným“ správam rôznych novinárov, ktorí citujú nemenej zaručené vyhlásenia anonymných analytikov, softvérových inžinierov, prípadne počítačových poradcov, je až neprimerane medializovaný, bohužiaľ, s presne opačným účinkom – priemerne (t. j. dosť málo) inteligentný Američan je schopný spomenuté prípravy realizovať a na tie dva roky sa naozaj niekde „zakopať“. A čo my? Zožením si lopatu alebo to riskneme a budeme všetky varovania ignorovať? Ako obyčajne riešenie je niekde uprostred. Rozhodne Y2K neovplyvní priemerného či občasného používateľa PC – keby mu náhodou jeho dosové účtovníctvo prestalo fungovať (čo by mal otestovať ešte pred Silvestrom), jednoducho si zaobstará nový program. Jeho firmu to pravdepodobne nepoloží a jeho samého takisto nie. Horšie je to s podnikmi, kde informačné systémy slúžia na zabezpečenie alebo realizáciu kritických operácií, od ktorých závisia ľudské životy, dodávka energie, plynulý chod spoločnosti – to sú napr. nemocnice, elektrárne, banky, dopravné spoločnosti a iné. Takéto podniky (za predpokladu, že ich IS sú ovplyvnené prechodom do nového tisícročia) budú musieť rozhodne celú záležitosť brať vážne a dovolím si tvrdiť, že väčšina (dúfam, že podstatná) z nich dnes nesedí so založenými rukami, ale realizuje príslušné opravy.

### Priestupné roky a príkazový riadok

Verím, že mi prepáčite, že som takpovediac zneužil úvod svojho seriálu na krátky pohľad na problémy súvisiace s prechodom do roku 2000. Aby som nadviazal na hlavnú tému seriálu, uvedieme si krátky program, pomocou ktorého sa dá otestovať, či je zadaný rok priestupný, alebo nie. Tento test totiž niektoré programy realizujú nesprávne a rok 2000 napríklad označia za nepriestupný. Od zavedenia gregoriánskeho kalendára sa za priestupné roky považujú tie, ktoré spĺňajú nasledujúce kritériá:

- ak je rok deliteľný číslom 4 a súčasne nie je deliteľný číslom 100, potom je priestupný;
- ak je rok deliteľný číslom 100 a súčasne nie je deliteľný číslom 400, potom nie je priestupný;
- ak je rok deliteľný číslom 400, potom je priestupný;
- vo všetkých ostatných prípadoch rok nie je priestupný.

Podľa týchto pravidiel teda rok 2000 priestupný je, pretože hoci je deliteľný číslom 100, je súčasne deliteľný aj číslom 400. Ak si pravidlá zhrnieme, rok je priestupný len vtedy, ak je deliteľný číslom 400 alebo ak je deliteľný číslom 4 a súčasne nie číslom 100. Prv než budete pokračovať, skúste si príslušný test napísať pomocou operátorov jazyka C++. Je to jednoduché, použijeme operátor `%` (modulo) a logické operátory `&&` a `||`. Príklad je koncipovaný ako samostatný program:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (argc != 2)
        exit(1);

    int r = atoi(argv[1]);
    printf(„Rok %i je „, r);
    int p = ( (r % 4 == 0)
           && (r % 100 != 0) )
           || (r % 400 == 0);
    printf(p ? „priestupný.\n“
             : „nepriestupný.\n“);
    return 0;
}
```

Operátor `%` vráti nulovú hodnotu, ak jeho prvý argument je deliteľný druhým. Premennú `r` predstavujúcu rok testujeme trikrát – na deliteľnosť číslami `4`, `100` a `400`. Podmienka priestupnosti (uložená do premennej `p`) je splnená, ak `r` je deliteľné `4` (`r % 4 == 0`) a súčasne (`&&`) `r` nie je deliteľné `100` (`r % 100 != 0`) alebo (`||`) `r` je deliteľné `400` (`r % 400 == 0`). Namiesto `r % 4 == 0` môžeme písať aj `!(r % 4)` a namiesto `r % 100 != 0` iba `r % 100`, čo sa v praxi obyčajne aj robí, ale na ilustračné účely je náš zápis vhodnejší. Po vyhodnotení testu bude v `p` nenulová hodnota, ak rok `r` je priestupný, a nulová, ak je nepriestupný.

Rok, ktorý chceme otestovať, sa programu zadá ako voliteľný argument na príkazovom riadku (bohužiaľ, opäť musím nariekať nad nepružnosťou slovenčiny a uviesť výstižnejší anglický názov command-line argument). Čo je to argument, definuje prekladač a/alebo prostredie, v ktorom program beží, obyčajne je to časť príkazového riadka, ohraničená z oboch strán medzerami. Pokiaľ chceme medzeru (a prípadne rôzne „zakázané“ znaky) zahrnúť do argumentu, ohraničujeme ho úvodzovkami. Dosiaľ sme si nepovedali, akým spôsobom je príkazový riadok prístupný programátorovi, takže to teraz napravíme. Funkcia `main()`, ktorú sme si uvádzali vždy bez argumentov, v skutočnosti má minimálne dva argumenty (je tu menšia slovná kolízia – argumenty funkcie vs. argumenty príkazového riadka; význam by mal byť jasný z kontextu). Ak tieto argumenty nepoužívame, nemusíme ich deklarovať, ale ak s nimi chceme pracovať, musí byť prvý z nich typu `int` a druhý typu `char**`. Hoci ich názvy nie sú určené normou, podľa zaužívaných konvencií ich deklarujeme ako `argc` a `argv`. Prvý z nich, `argc`, vyjadruje počet argumentov (alebo parametrov?) zadaných na príkazovom riadku. Tento počet je však o jeden vyšší ako skutočný, pretože programu sa ako prvý z argumentov dodá názov jeho vykonateľného súboru (s cestou či bez nej, to opäť závisí od prekladača či prostredia – dokonca napríklad prekladač GCC pre DOS/Windows znaky `,\‘` v ceste zmení podľa konvencií Unixu na znaky `,/‘`). Druhý z argumentov predstavuje pole reťazcov reprezentujúcich jednotlivé zadané argumenty príkazového riadka. Vieme, že reťazce sú vlastne polia znakov, preto ide v skutočnosti o pole ukazovateľov na tieto reťazce. Ďalej vieme, že pole sa v prípade, ako je tento (teda odovzdávanie do funkcie), prevádza na ukazovateľ na jeho prvý prvok. Preto typ argumentu `argv` je „ukazovateľ na ukazovateľ na `char`“. Takýto ukazovateľ môžeme pri zápise formálneho argumentu funkcie deklarovať buď ako `char** argv`, alebo tak ako v našom príklade `char* argv[]`. Druhý spôsob je, myslím, zrozumiteľnejší a zreteľne vyjadruje, že ide o pole ukazovateľov na `char`.

Prvý z argumentov programu, ktorý je prístupný ako `argv[0]`, obsahuje spomínaný názov súboru, v ktorom je program uložený. Nasledujú zvyšné argumenty v tom poradí, v akom boli zadané na príkazovom riadku. Navyše máme zaručené, že celé pole `argv[]` bude ukončené nulovým ukazovateľom, t. j. platí, že `argv[argc] == 0`. K jednotlivým argumentom môžeme pristupovať nielen pomocou indexov v rozsahu `0` až `argc-1`, ale aj prostredníctvom ukazovateľa na typ `char*`, ktorý inicializujeme hodnotou `argv`. Jeho postupnou inkrementáciou a následnou dereferenciou dostávame príslušné ukazovatele na argumenty. Na tento účel môžeme použiť aj priamo premennú `argv`, ktorá vďaka tomu, že je formálnym argumentom funkcie, správa sa ako bežná lokálna premenná, inicializovaná príslušným skutočným parametrom. Pri prípadných nejasnostiach sa vráťte o dve časti naspäť k obrázku, ktorý znázorňuje pole ukazovateľov.

Nasledujú tri verzie programu, ktorý vypíše všetky svoje argumenty, ale každý na samostatný riadok:

```
// echo1.cpp
int main(int argc, char* argv[])
{
    for (int i = 1; i < argc; i++)
        printf(„%s\n“, argv[i]);
    return 0;
}

// echo2.cpp
int main(int argc, char* argv[])
{
    while (--argc)
        printf(„%s\n“, *++argv);
    return 0;
}

// echo3.cpp
int main(int argc, char* argv[])
{
    while (*++argv)
        printf(„%s\n“, *argv);
    return 0;
}
```

Všetky tri programy, samozrejme, treba doplniť direktívou `#include <stdio.h>`. Prvý z programov (`echo1.cpp`) je dostatočne jasný a netreba ho hlbšie analyzovať. Druhý z nich (`echo2.cpp`) na počítanie zostávajúcich argumentov využíva premennú `argc`. Prefixová verzia dekrementačného operátora `--` je použitá preto, aby sme preskočili prvý argument s menom súboru. Pri vypisovaní jednotlivých argumentov používame premennú `argv`, ktorá ukazuje na práve aktuálny argument (resp. na ukazovateľ na tento argument). Inkrementáciou `argv` sprístupňujeme ďalšie a ďalšie argumenty. Opäť z rovnakých dôvodov ako predtým použijeme prefixový operátor `++`. Konečne tretí program (`echo3.cpp`) je podobný druhému, cyklus však ukončíme nie vtedy, keď premennú `argc` znížime na nulu, ale vtedy, keď obsahom `*argv` bude prázdny ukazovateľ. To je, ako vieme, signál konca poľa. Inkrementovať `argv` musíme pri jeho prvom použití, teda už v podmienke príkazu `while`.

### Pokračujeme v deklaráciách

Konečne by sme sa však mali vrátiť k tomu, čo sme minule nedokončili. Priznám sa, že predchádzajúci výklad

bol úplne neplánovaný, pôvodne som si chcel len trošku zafilozofovať o blížiacom sa konci storočia – a hľa, ako sa mi to vymklo spod kontroly. Ale to nič, ide o užitočnú tému, ktorá vám možno uľahčí písanie niektorých programov.

Naposledy sme skončili rozprávaním o špecifikátoroch, používaných pri deklaráciách. Prebrali sme špecifikátory ukladacej triedy a funkčné špecifikátory, okrem nich C++ definuje ešte špecifikátory typu a tzv. špecifikátor `typedef`.

## Špecifikátory typu

Základnými špecifikátormi typu sú nám už dávno známe kľúčové slová `char`, `short`, `int`, `long`, `signed`, `unsigned`, `float`, `double` a `void`. S ich pomocou vyjadrujeme výsledný typ deklarovaného mena. Pre ich použitie, resp. ich vzájomnú kombináciu však platia určité obmedzenia.

Špecifikátor `char` môžeme použiť spolu so špecifikátormi `signed` alebo `unsigned` (alebo aj bez nich, ale nie s oboma naraz), deklarujeme tak mená s typmi `char`, `signed char` a `unsigned char` (vieme, že to sú z hľadiska prekladača tri rôzne typy!).

Špecifikátor `int` môžeme skombinovať so špecifikátormi `signed` alebo `unsigned` a navyše so špecifikátormi `short` alebo `long` (opäť nie s oboma naraz). Povolené sú teda nasledujúce kombinácie:

```
int
signed int
unsigned int
short int
signed short int
unsigned short int
long int
signed long int
unsigned long int
```

Typ `int` sa však používa natoľko intenzívne, že ho možno zo všetkých uvedených kombinácií vynechať, s výnimkou tej prvej, keď ho zrejme vynechať nemôžeme, lebo by sme tým stratili akúkoľvek informáciu o type. Okrem toho je špecifikátor `signed` prakticky zbytočný, lebo všetky celočíselné typy (okrem typu `char`) sa implicitne berú ako znamienkové. Množina možných kombinácií sa nám takto zredukuje na nasledujúci zoznam:

```
int
unsigned
short
unsigned short
long
unsigned long
```

Je úplne zbytočné používať inak zapísané modifikácie – tieto pokrývajú všetky možnosti a navyše majú minimálnu dĺžku.

Špecifikátor `float` nemožno kombinovať so žiadnym iným, špecifikátor `double` môžeme doplniť špecifikátorom `long`, čím deklarujeme meno s typom `long double`. Niektoré prekladače (ako napr. Borland C++ 3.1) povolia aj kombináciu `long float`, ktorú interpretujú ako typ `double`, ale to je ich implementačné špecifikum, ktoré norma oficiálne nepovoľuje. Špecifikátor `void` nepripúšťa nijaké modifikátory.

## Špecifikátory `const` a `volatile`

Ľubovoľný špecifikátor typu môže byť doplnený jedným zo špecifikátorov `const` a `volatile` (aj oboma). Pomocou kľúčového slova `const` deklarujeme tzv. konštantné objekty. Takéto objekty majú hodnotu určenú pri inicializácii a ďalej v programe ich nemožno meniť. Na rozdiel od „konštánt“ jazyka C, definovaných direktívou `#define`, ktoré boli vlastne makrami a nahrádzali sa skutočnými hodnotami počas spracovania zdrojového textu preprocesorom, objekty deklarované ako `const` sa správajú ako bežné, hoci nemodifikovateľné objekty plne pod správou kompilátora. Ten ich môže v rámci optimalizácie nahradiť vo výslednom binárnom kóde priamo príslušnými literálmi, ale pri preklade doplní ich mená do tabuľky symbolov pre debugger, a teda k nim máme počas ladenia prístup (čo pri #definovaných konštantách neplatilo).

Konštantné objekty môžeme použiť vo výrazoch, ktorých výsledok musí byť známy už vo fáze prekladu – tzv. konštantných výrazoch, ako jednotlivé návestia príkazu `switch`, rozmery polí a pod. Konštantnými môžu byť aj polia, ktoré sa takto stávajú poľami konštantných prvkov. Samostatnou kapitolou sú ukazovatele. Pri použití špecifikátora `const` pri deklarácii ukazovateľa musíme rozlišovať dve diametrálne odlišné polohy. Môžeme deklarovať ukazovateľ na konštantný objekt, vtedy je `const` pred špecifikátorom doménového typu ukazovateľa:

```
const int ci = 123;
const int * pci = &ci;
```

Ukazovateľ `pci` ukazuje na konštantnú premennú `ci`, ktorej hodnotu nemôžeme meniť (ani s pomocou dereferencie tohto ukazovateľa), ale hodnotu samotného ukazovateľa meniť môžeme! `Pci` tak môže ukazovať na inú konštantnú premennú, dokonca mu môžeme priradiť aj adresu nekonštantnej premennej, ktorá sa však bude javiť pri prístupe cez dereferencovaný ukazovateľ ako konštantná. Je teda možný nasledujúci zápis:

```
int i = 456;
const int * pci = &i;
```

Okrem toho môžeme deklarovať ako konštantný samotný ukazovateľ. O takýchto ukazovateľoch si však povieme pri opise deklarátorov.

Druhým podobným špecifikátorom je kľúčové slovo `volatile`. Pomocou neho napovedáme kompilátoru, že objekt takto deklarovaný môže byť menený asynchrónne voči behu nášho programu, napríklad hardvérom, obslužnou rutinou prerušenia, iným procesom alebo iným threadom. Kompilátor by teda nemal vo vzťahu k objektu `volatile` vykonávať nijaké optimalizácie a pri každom prístupe k takémuto objektu by mal čítať jeho hodnotu vždy nanovo z pamäte.

Používanie špecifikátora `volatile` sa dosť podobá používaniu `const`, t. j. môžeme definovať ukazovatele na `volatile` objekty alebo `volatile` ukazovatele na (ľubovoľné) objekty (opíšeme si neskôr spolu s konštantnými ukazovateľmi). Samozrejme, nemusíme sa starať, či objekty, ktoré meníme, sú alebo nie sú `volatile`, jediným obmedzením je určitá kompatibilita objektov a/alebo ukazovateľov vzhľadom na priradenie. Podobné pravidlá platia aj pre špecifikátor `const` a dajú sa zhrnúť takto: konštantný objekt možno priradiť nekonštantnému, ale nie naopak, pre `volatile` objekty takéto obmedzenie nie je; ukazovateľu na konštantný/`volatile` objekt môžeme bez problémov priradiť ukazovateľ na nekonštantný/nevolatile objekt, naopak je to síce povolené, ale neodporúča sa (môže to spôsobiť výnimku pri prístupe k objektom, lebo konštantné objekty môže prekladač umiestniť v read-only segmente či stránke pamäte). Tam, kde sa požaduje hodnota typu `const T` alebo `volatile T`, môžeme použiť objekt typu `T`. Konštantné/`volatile` môžu byť aj referencie, ktoré sa inak používajú ako bežné premenné príslušného typu (`const T&` ako `const T`, `volatile T&` ako `volatile T`, atď.)

## Deklarácia enumerácií

Enumerácia je zvláštnym druhom typu, ktorý môžeme opísať aj ako vymenovaný typ. Vo všeobecnosti je typ údajov nielen v C++ definovaný jednak rozsahom možných hodnôt, jednak množinou operácií, ktoré sú nad daným typom definované. Typ enumerácie je špecifický tým, že jeho rozsah hodnôt explicitne určujeme my, a to vymenovaním všetkých hodnôt, ktoré premenná takéhoto typu môže nadobudnúť. Jednotlivé hodnoty musia byť celočíselné (aj enumeračný typ sa považuje za celočíselný) a deklaráciou enumerácie súčasne deklarujeme tieto hodnoty ako pomenované konštanty. Pozrime sa, ako taká deklarácia vyzerá:

```
enum identifikátor { zoznam }
```

Identifikátor v deklarácii predstavuje meno novo definovaného enumeračného typu a platia preň rovnaké pravidlá ako pre identifikátory bežných premenných. Nezdieľa však s nimi priestor mien. Zoznam je zoznamom vymenovaných konštánt. Tieto konštanty sú zapisované buď v tvare „identifikátor“, alebo v tvare „identifikátor = hodnota“. Pokiaľ neuvedieme pri identifikátore konštanty hodnotu, použijú sa implicitné hodnoty – prvá konštanta bude mať hodnotu `0` (nula), každá ďalšia hodnotu o jednotku vyššiu. Uvedenie hodnoty túto postupnosť narušuje tým, že danej konštante priraďuje explicitne zadanú hodnotu. Jednotlivé konštanty sú oddelené čiarkou.

Pozrime sa na príklad, ktorý situáciu ozrejmí:

```
enum rgb { red, green, blue };
```

Táto deklarácia opisuje nový typ `rgb`, ktorého hodnotami sú konštanty `red` (s hodnotou `0`), `green` (s hodnotou `1`) a `blue` (s hodnotou `2`).

```
enum flag { r, h, s = 5, a };
```

Flag je typ, ktorý môže nadobúdať štyri rôzne hodnoty: `r`, `h`, `s`, `a`. Ich celočíselné reprezentácie sú v poradí `0`, `1`, `5` a `6`.

Hodnoty definovaných konštánt nemusia byť nijako usporiadané, dokonca sa môžu aj zhodovať (hodnoty, nie názvy konštánt!). Premennej typu enumerácia môžu byť priradené iba hodnoty rovnakého typu, teda nemôžeme premennej uvedeného typu `rgb` priradiť celočíselnú hodnotu `1`, hoci v obore jeho možných hodnôt je konštanta `green`, ktorej celočíselná reprezentácia je `1`. Možné to je len explicitným pretypovaním, ktoré sa však neodporúča vzhľadom na skutočnosť, že priradením napr. výrazu `(rgb)5` premennej typu `rgb` bude mať takáto premenná obsah, ktorý síce formálne je typu `rgb`, ale nie je zhodný so žiadnou z povolených a vymenovaných konštánt. Opačné priradenie, teda priradenie hodnoty typu `rgb` premennej typu `int` je možné a má zmysel, do premennej sa uloží celočíselná reprezentácia hodnoty. Teda napr. výraz `i = blue` priraďuje premennej `i` hodnotu `2`.

V prípade, že vynecháme v deklarácii enumerácie jej identifikátor, dostávame tzv. anonymnú enumeráciu. Takýto typ nemožno ďalej v programe používať a jeho jediným zmyslom je deklarácia pomenovaných konštánt. Teda

```
enum { RUNNING, READY, BLOCKED };
```

deklaruje tri pomenované konštanty: `RUNNING`, `READY` a `BLOCKED`. Ich hodnoty obyčajne nie sú zaujímavé, dôležitý je fakt, že existujú a možno ich používať. Hoci typom týchto konštánt je enumerácia, každé ich použitie vedie k automatickej konverzii na `int`.

Deklaráciu enumeračného typu môžeme použiť ako špecifikátor pri deklarácii premenných tohto typu. V princípe sú možné dva spôsoby použitia. Pri prvom súčasne s deklaráciou premennej deklarujeme aj nový enumeračný typ. Použijeme bežnú syntax – najprv uvedieme špecifikátor (v tomto prípade deklaráciu `enum`) a za ním deklarátor (t. j. napr. identifikátor premennej či premenných):

```
enum foo { A, B } x, y;
```

V príklade deklarujeme nový typ `foo` s dvoma povolenými hodnotami `A` a `B` a dve premenné `x` a `y` tohto typu.

Pri druhom spôsobe najprv typ `foo` deklarujeme samostatne a až potom (v inom deklaračnom príkaze) deklarujeme aj premenné `x` a `y`:

```
enum foo { A, B };
foo x;
enum foo y;
```

Zaujímavý je dvojaký spôsob deklarácie oboch premenných – bez kľúčového slova `enum` a s ním. Oba spôsoby sú ekvivalentné, druhý použijeme vtedy, ak identifikátor `foo` je zakrytý nejakou inou deklaráciou, napríklad lokálnej premennej s názvom `foo`. Vtedy je enumeračný typ dostupný svojím kvalifikovaným menom `enum foo`.

## Špecifikátor `typedef`

Posledný špecifikátor, o ktorom si povieme, je trochu zvláštny. Hoci syntakticky je jeho použitie podobné použitiu iných špecifikátorov, význam jeho použitia je úplne iný – pomocou kľúčového slova `typedef` deklarujeme nové meno, ktoré môžeme neskôr použiť ako pomenovanie typu. Inak povedané, `typedef` predstavuje mechanizmus používateľskej definície typov. Jeden spôsob explicitnej deklarácie typov sme tu už mali – ide o enumeračný typ. Pomocou `typedef` však deklarujeme jednoslovné pomenovanie inak vo všeobecnosti komplexného typu. Klasický príklad: chceme deklarovať pole desiatich ukazovateľov na typ `char`. Pokiaľ vieme, ako na to priamo, napíšeme:

```
char* array1[10];
```

Na tomto príklade to vyzerá veľmi jednoducho a priamočiaro, ale predstavte si pole desiatich ukazovateľov na funkcie s návratovým typom „ukazovateľ na `int`“ a s jedným argumentom, ktorý je ukazovateľom na funkciu bez argumentov, a s návratovým typom `void` (áno, aj také veci sa nájdu v C++ a dokonca sa aj používajú). Pokiaľ nie ste v deklaráciách príliš zbehlí, asi neprídete na to, že správna forma zápisu deklarácie takéhoto poľa je:

```
int* (*array2[10])(void (*)());
```

Toto už vyzerá zložitejšie, však? Veľmi jednoduchým a pre začiatočníkov odporúčaným spôsobom, ako s takýmito komplikovanými typmi „vybabrať“, je deklarácia pomocných typov práve pomocou mechanizmu `typedef`. Pole `array1` potom deklarujeme asi takto:

```
typedef char* pchar;
pchar array1[10];
```

V tomto úseku kódu deklarujeme nový typ `pchar`, ktorý je okrem názvu úplne ekvivalentný s typom `char*`. Potom jednoducho deklarujeme pole `array1` ako pole desiatich prvkov typu `pchar`, čo sú, ako vieme, ukazovatele na typ `char`. Z príkladu vidíme aj spôsob použitia kľúčového slova `typedef`. Deklarácia vyzerá úplne rovnako, ako keby sme deklarovali bežnú premennú nejakého typu, navyše však pred celú deklaráciu pripojíme `typedef`. Novo deklarované meno nebude potom predstavovať premennú nejakého typu, ale akési náhradné meno (alias) pre tento typ. Je jasné, že z hľadiska miesta použitia je `typedef` takým istým špecifikátorom ako napr. `const`, len význam má úplne iný.

Uveďme si ešte príklad zjednodušenej deklarácie uvedeného poľa `array2`. Opäť si najprv deklarujeme pomocné typy – ukazovateľ na funkciu typu `void` bez argumentov:

```
typedef void (*pvf)();
```

a na funkciu vracajúcu `int*` s argumentom čerstvo deklarovaného typu `pvf`:

```
typedef int* (*pipfpvf)(pvf);
```

Teraz už môžeme bez problémov zapísať deklaráciu poľa `array2`:

```
pipfpvf array2[10];
```

Názvy oboch typov vyjadrujú nie príliš zreteľnú snahu o zakódovanie „zloženia“ typu do jeho mena (`pvf` = pointer to void function, `pipfpvf` ani nebudem rozoberať).

V uvedenom príklade sa používajú trochu obskúrne veci, ako ukazovatele na funkcie, polia ukazovateľov, o ktorých sme si zatiaľ nehovorili (prídu na rad v časti venovanej deklarátorom, teda onedlho). Na nich však možno najlepšie demonštrovať význam špecifikátora `typedef`. Okrem toho používame vlastné typy napr. vtedy, ak vopred nevieme, aký typ bude použitý pre určité premenné vo výslednom programe. Zadefinujeme si teda pomocou kľúčového slova `typedef` nový typ, ekvivalentný nejakému prvému odhadu (alebo prvému pokusu) a všetky relevantné premenné a objekty budeme deklarovať pomocou tohto nového typu. Ak sa v budúcnosti rozhodneme pre prechod k inej reprezentácii (napríklad namiesto `float` použijeme `double` alebo namiesto `int long`), stačí prepísať jediný riadok, ten, na ktorom `typedef` typ deklarujeme. Alebo iný príklad: chceme deklarovať typ `WORD` ako šestnásťbitové číslo bez znamienka. Používame nejaký starší 16-bitový prekladač, a preto deklarácia bude vyzerať takto:

```
typedef unsigned int WORD;
```

(špecifikátor `int` nie je povinný, je tam len pre názornosť). Čo však v prípade, že zmeníme prekladač a prejdeme na 32-bitovú platformu, kde je typ `int` dvakrát dlhší? Potom nám stačí zmeniť túto jedinú deklaráciu, a to asi takto:

```
typedef unsigned short WORD;
```

(keby sme boli predvídaví, deklarovali by sme typ `WORD` už vopred ako `unsigned short` – na väčšine platforiem je typ `short` 16-bitový, aspoň do rozšírenia sa 64-bitových procesorov).

Nakoniec ešte zopár drobností. Pomocou `typedef` mechanizmu môžeme raz definované meno typu predefinovať tak, aby predstavovalo rovnaký typ:

```
typedef int I;
typedef int I;
typedef I I;
```

Okrem toho nemôžeme redefinovať meno použité v inej ako `typedef` deklarácii:

```
enum rgb { red, green, blue };
typedef int rgb;  // chyba!
```

## Deklarátory

Máme za sebou približne polovicu preberanej témy. Pokračovať budeme opisom deklarátorov, ktoré spolu so špecifikátormi robia deklarácie mien kompletnými. Na rozdiel od špecifikátorov, ktoré opisujú typ, ukladaciu triedu a iné vlastnosti deklarovaných mien (objektov či funkcií), pomocou deklarátorov určujeme názvy týchto mien [je to trochu krkolomný opis, ale pre objasnenie – ak deklarujeme napr. meno `buf`, tento jeho identifikátor (`buf`) je súčasťou deklarátora], ďalej môžeme modifikovať typ deklarovaných mien a prípadne môžeme uviesť aj inicializačnú hodnotu.

Za zoznamom špecifikátorov v deklarácii nasleduje zoznam deklarátorov, oddelených čiarkou. Každý deklarátor môže obsahovať inicializáciu – o tých si však povieme až na záver. Špecifikátory uvedené v deklarácii sa vzťahujú na každý deklarátor. Výsledný typ deklarovaného mena je potom daný jednak spoločným zoznamom špecifikátorov, jednak vlastnými modifikáciami, ktoré sú privátne pre každý deklarátor. Celú deklaráciu môžeme rozložiť na postupnosť čiastkových deklarácií. Každá z nich sa dá zapísať v tvare

```
T D
```

kde `T` je typ (daný špecifikátormi) a `D` je deklarátor (pre jednoduchosť vynechávame možnú inicializáciu). V prípade, že `D` je obyčajný, ničím „neprikrášlený“ identifikátor, bude jeho deklarovaným typom typ `T`. Prípadné uzavretie deklarátora `D` do zátvoriek nijako nezmení význam deklarácie, môže však pomôcť pri určovaní výsledného typu.

## Deklarácia ukazovateľov

Na záver tejto časti seriálu si ešte povieme, ako je to s deklaráciou ukazovateľov a referencií. Hoci sme si kedysi pri rozprávaní o nich povedali okrem iného aj to, ako ich deklarujeme, bol to vzhľadom na nedostatok potrebných vedomostí (vašich, nie mojich) len základný a neúplný opis. Teraz si uvedieme všetky možnosti, ktoré nám C++ poskytuje.

Deklarácia ukazovateľa má vo všeobecnosti nasledujúci tvar:

```
T * cv-kvalifikátory D1
```

Zápis je tak trochu rekurzívny, `D1` je opäť deklarátor (ľubovoľný). Ak je na mieste `D1` obyčajný identifikátor, deklarácia mu priradí typ „cv-kvalifikovaný ukazovateľ na `T`“. V prípade zložitejšieho deklarátora je výsledným typom nejaký komplexnejší typ, ktorého základným kameňom je „cv-kvalifikovaný ukazovateľ na `T`“. Už teda tušíme, ako deklarovať napríklad pole ukazovateľov na typ `int` – deklarátor `D1` bude nejakým spôsobom deklarovať pole (uvidíme ďalej ako). Zostáva ešte ozrejmiť, čo sú to cv-kvalifikátory. Ide o (medzerami oddeľovaný) zoznam kľúčových slov `const` a `volatile`. Zoznam nie je povinný a prakticky nemá význam použiť v ňom každé zo slov viac ako raz. Význam týchto kvalifikátorov je vám pravdepodobne jasný. Okrem spomínaného ukazovateľa na konštantný objekt môžeme deklarovať aj konštantný ukazovateľ na (ľubovoľný) objekt:

```
int i = 789;
int * const cpi = &i;
```

Premennú, na ktorú konštantný ukazovateľ ukazuje, môžeme pomocou neho meniť, nemôžeme však takýto ukazovateľ „prinútiť“, aby ukazoval na inú premennú. Nasledujúci kód je teda chybou:

```
int i = 222;
int j = 333;
int * const cpi = &i;
cpi = &j;
```

Konštantné ukazovatele môžeme bez problémov priradiť nekonštantným ukazovateľom, naopak to, samozrejme, nejde. Ďalej môžeme deklarovať aj konštantný ukazovateľ na konštantný objekt:

```
const int ci = 1000;
const int * const cpci = &ci;
```

Ukazovateľ `cpci` je v tomto prípade úplne obmedzený – nemôžeme meniť ani jeho hodnotu, ani prostredníctvom neho hodnotu premennej, na ktorú ukazuje. Nič nám, pravda, nebráni pretypovať si ho na taký ukazovateľ, aký nám vyhovuje, a hodnotu potom veselo meniť. Takýto prístup však svedčí o nie príliš čistom programátorskom štýle a zvyčajne o chybách v návrhu softvéru.

Pre kvalifikátor `volatile` platia podobné pravidlá. Oba kvalifikátory môžeme rôzne kombinovať:

```
const int * volatile vpci;
volatile char * const volatile cvpcc;
```

V príklade deklarujeme `vpci` ako `volatile` ukazovateľ na konštantný objekt typu `int`, `cvpcc` ako konštantný a `volatile` (na pohľad paradox, ale konštantný je len vzhľadom na náš program) ukazovateľ na `volatile` objekt typu `char`.

### Deklarácia referencií

Referencie sú v mnohom podobné ukazovateľom. Ich deklarácia vyzerá takto:

`T &` cv-kvalifikátory `D1`

O častiach tejto deklarácie platí to, čo je uvedené v predchádzajúcom odseku, s výnimkou toho, že nemôžeme deklarovať typ `void&` a ďalej neexistujú referencie na referencie, polia referencií ani ukazovatele na referencie. To vyplýva z faktu, že referencia ako taká nepredstavuje plnohodnotný typ – de facto neexistuje premenná, ku ktorej by sme mohli pristupovať ako k referencii, vzhľadom na to, že po inicializácii sa referenčná premenná správa ako tá premenná, na ktorú sa odkazuje, s rovnakým typom aj vlastnosťami (s jedinou výnimkou – pozri ďalej).

Aj pri referenciách môžeme použiť kvalifikátory `const` a `volatile`, ale ich praktický význam je nulový – prekladače obyčajne tieto kvalifikátory (za znakom `&`) ignorujú. Iné je referencia na konštantný objekt, tá má veľký význam spolu s ukazovateľom na konštantný objekt predovšetkým ako formálny argument funkcie. Predstavme si, že odovzdávame nejakej funkcii ukazovateľ na reťazec, ktorý by táto funkcia nemala meniť (reťazec, nie ukazovateľ). Normálne je ukazovateľ jediným prostriedkom, ako funkcii odovzdať argument odkazom (C++ pozná iba odovzdávanie hodnotou). Ak však chceme prípadnej zmene zabrániť, stačí deklarovať argument danej funkcie ako konštantný:

```
void fnc(const char* str)
{ ... }
```

Podobná situácia je pri referenciách. Referenciou odovzdávame jednak argument, ktorý chceme meniť (teda použitie ekvivalentné ukazovateľu), a jednak argument, ktorý je priveľký na to, aby sa odovzdával hodnotou, čo by znamenalo kopírovanie veľkého množstva údajov na zásobník. V tomto druhom prípade môžeme deklarovať referenciu na konštantný objekt, čím zabránime prípadnej (i nechcenej) modifikácii pôvodného objektu. Pri formálnom parametri typu referencia sa do funkcie naozaj odovzdáva obsah tejto referencie, t. j. adresa odkazovanej premennej – to je tá výnimka, o ktorej som už hovoril.

Referencie na `volatile` objekty majú podobný význam ako ukazovatele na tieto objekty. V praxi sa vôbec `volatile` premenné používajú dosť zriedka, pokiaľ totiž nejaký objekt môže byť menený asynchrónne, na pozadí, obyčajne sa prístup k nemu nejakým spôsobom synchronizuje (semafory, zámky a pod.)

## Trinásta časť: DEKLARÁCIE III

Dnešná časť je svojím spôsobom mimoriadna. Predovšetkým, verte či neverte, už uplynul rok odvtedy, čo vyšla prvá časť seriálu C++ pod lupou. Nemal som vtedy ani najmenšie tušenie, či a ako bude moja práca akceptovaná, nevedel som s istotou, ako dlho bude seriál vychádzať. Dnes vidím, že moje rozhodnutie pustiť sa do náročnej a tak trochu i nevďačnej úlohy bolo správne – svedčí o tom množstvo e-mailov, ktoré som od vás za ten rok dostal. Príjemne ma prekvapilo, že z tohto množstva boli možno dva či tri výrazne kritické; napriek tomu si však nemyslím, že je moje dielo dokonalé a (prevažne vďaka mojej perfekcionistickej povahe) práve pri retrospektívnom pohľade zisťujem, že veľa vecí by som dnes spravil ináč. Žiaľ, jednotlivé časti seriálu sú už „in statu quo“ a nedá sa spraviť nijaké „undo“. (Mimochodom, aj vás občas štve, že v živote neexistuje táto možnosť, taká bežná vo väčšine dnešného softvéru?) Chcem sa poďakovať všetkým tým, ktorí si našli chvíľku času a vyjadrili svoj názor na obsah i formu seriálu. Verím, že vás rovnako uspokoja časti, ktoré len vyjdú, a že vydržíte až do konca.

Druhá vec, pre mňa oveľa významnejšia ako pre vás, je fakt, že vo chvíli, keď čítate tieto riadky, mal by som mať za sebou štátnu skúšku na Fakulte elektrotechniky a informatiky STU. Vďaka zvyčajnému časovému posunu medzi uzávierkou a vyjdením PC REVUE vám nemôžem povedať, s akým výsledkom, ale pevne verím, že všetko dopadne dobre. Našťastie sa mi podarilo bez výrazných stresových situácií zvládnuť prácu na diplomovke aj písanie seriálu. Dúfam, že ste nepostrehli nijaké zníženie kvality v posledných dvoch či troch častiach. Ak áno, nebojte sa, už to bude zase v poriadku.

Nezachádzajme však do filozofických úvah. Vráťme sa po tretíkrát k problematike deklarácií. Dnes výklad na túto tému na veľkú radosť (moju a zrejme i vašu) dokončíme. Už nám chýba len veľmi málo, aby sme sa prehupli do trochu atraktívnejšej objektovej polovice seriálu.

### Pokračujeme v deklarátoroch

Minule sme si začali hovoriť o druhej zložke deklarácií – o deklarátoroch. Vieme, že pomocou deklarátorov určujeme názov novo deklarovaných objektov a môžeme modifikovať typ, určený pomocou špecifikátorov. Dve z týchto modifikácií sme si aj podrobne opísali: deklaráciu ukazovateľov a deklaráciu referencií. Modifikácie typu, ktoré sú súčasťou deklarátorov, možno za dodržania určitých pravidiel rôzne reťaziť a získavať tak prakticky neobmedzenú množinu nových, často relatívne zložitých typov (ako polia ukazovateľov na funkcie a pod.). Teraz si objasníme problematiku deklarácie polí, deklarácie a definície funkcií a inicializácie deklarovaných objektov.

### Deklarácia polí

Deklarácia poľa prvkov ľubovoľného typu má nasledujúci tvar:

`T D1` [*konštantný-výraz*]

Meno, ktoré deklarujeme (a ktorého identifikátor je súčasťou deklarátora `D1`, prípadne je s ním totožný), bude mať typ „pole … prvkov typu `T`“. To, akého typu budú prvky poľa, určuje tvar deklarátora `D1`. Konštantný výraz, ktorý je súčasťou deklarátora, udáva počet prvkov deklarovaného poľa. V určitých prípadoch ho môžeme vynechať – ak ide o deklaráciu, a nie definíciu (t. j. poľu je prideľovaná pamäť niekde inde), ďalej pri deklarácii formálnych argumentov funkcií a konečne vtedy, keď je súčasťou deklarácie zoznam počiatočných hodnôt prvkov poľa (vtedy si prekladač rozmer poľa dokáže zistiť sám).

Typ prvkov poľa môže byť prakticky ľubovoľný: od primitívnych typov cez ukazovatele, štruktúry/triedy, enumerácie až po iné polia. Vieme už, že pole, ktorého prvkami sú opäť polia, svojím správaním navodzuje dojem viacrozmerného poľa. Nebudeme sa už vracať k jednotlivým špecifikám, ak máte chuť osviežiť si vedomosti, pohľadajte si príslušné časti seriálu v starších číslach. Upozorním len na jeden drobný detail – pri deklarácii viacrozmerného poľa môžeme vynechať (samozrejme, len v uvedených prípadoch) najviac jeden rozmer, a to ten, ktorý je najbližšie k identifikátoru (teda prvý), napríklad:

```
int m[][2][3];
```

Tento fakt vyplýva z nutnosti poznania rozmeru vnorených polí pri výpočte umiestnenia ich prvkov v pamäti (a navyše použitie týchto vnorených polí nespadá ani pod jeden uvedený prípad!).

Ukážme si príklad zložitejších deklarácií polí. Ak chceme deklarovať napríklad pole konštantných, nevolatile ukazovateľov (ktorých deklarátor, ako už vieme, vyzerá takto: `* const D`), použijeme zápis: `T * const D` [*konštantný-výraz*]. Medzery medzi jednotlivými zložkami nie sú povinné. Tu je deklarácia poľa `cal` desiatich konštantných ukazovateľov na typ `long`:

```
long * const cal[10];
```

Ak chceme deklarovať pole neznámej veľkosti, ktorého prvkami budú konštantné reťazce (teda ukazovatele na konštantný `char`), použijeme takýto zápis:

```
const char * msgs[];
```

V predchádzajúcej časti sme si ukazovali deklaráciu poľa ukazovateľov na funkcie. Hoci ešte nevieme deklarovať ukazovateľ na funkciu, dokážeme zapísať deklaráciu poľa takýchto ukazovateľov. Ak vám prezradím, že ukazovateľ na najjednoduchší typ funkcie – bez argumentov, bez návratovej hodnoty, teda typu `void f()`, deklarujeme ako:

```
void (*f)();
```

mali by ste byť schopní bez problémov zapísať deklaráciu poľa napr. dvanástich takýchto ukazovateľov:

```
void (*f[12])();
```

Prakticky teda deklarácia poľa spočíva v pripojení hranatých zátvoriek s uvedeným rozmerom poľa za príslušný deklarátor. Umenie však spočíva v nájdení správneho miesta, t. j. v rozpoznaní správneho deklarátora. Na tomto mieste by som rád ukázal odlišný spôsob deklarácie poľa ukazovateľov a deklarácie ukazovateľa na pole. V oboch prípadoch použijeme ako základ typ `int`. Najprv si ukážeme výsledok prvého príkladu. Pole ukazovateľov na `int` deklarujeme takto:

```
int * api[];
```

Tu je dôležité si uvedomiť, že deklarujeme api ako pole „nejakých“ prvkov, ktorých typ je zhodou okolností ukazovateľ na `int`. Dôraz sa tu kladie na hierarchiu typov v rámci deklarácie. Najprv napíšeme deklarátor pre pole:

```
api[]
```

Sme na začiatku hierarchie, použijeme preto priamo identifikátor výsledného objektu `api`. Ďalej potrebujeme deklarátor, ktorý bude vyjadrovať typ ukazovateľ. Vieme, že musíme použiť zápis:

```
* D1
```

Deklarátor `D1` predstavuje niečo, čo sa bude v budúcnosti používať ako ukazovateľ. Deklarujeme pole ukazovateľov, za `D1` teda dosadíme vyššie zapísaný deklarátor poľa. Keď totiž sprístupníme nejaký prvok tohto poľa, dostaneme objekt, ktorý má zmysel dereferencovať (predradením operátora `*`). Všimnite si úzku spojitosť deklarácie s následným spôsobom použitia. Keď je niečo deklarované ako „`int` *xxx*“, potom všade tam, kde prekladač bude očakávať typ `int`, môžeme použiť deklarované „*xxx*“. Takisto ak deklarátor ukazovateľa má tvar „`* D1`“, môže byť `D1` hocičo, ale musí sa to vyhodnotiť na typ ukazovateľ. Spojenie našich dvoch deklarátorov vykonáme jednoducho použitím substitúcie a výsledok doplníme príslušným špecifikátorom základného typu, čím dostaneme žiadaný tvar:

```
int *api[];
```

To, či a kam budeme dávať medzery, závisí iba od nás, ibaže nesmieme vynechať medzeru tam, kde by mohlo dôjsť k dvojznačnosti pri interpretácii prekladačom.

V druhom prípade deklarujeme objekt `pai` ako ukazovateľ na pole prvkov typu `int`:

```
int (*pai)[];
```

Tentoraz deklarujeme `pai` ako ukazovateľ na „nejaký“ typ. Týmto typom je čírou náhodou pole celých čísel. Deklaráciu začneme zápisom deklarátora pre ukazovateľ:

```
* pai
```

Okrem neho potrebujeme ešte deklarátor, ktorý bude vyjadrovať pole prvkov. Nič jednoduchšie – použijeme zápis:

```
D1 []
```

Tentoraz ako `D1` musíme uviesť niečo, čo bude v budúcnosti predstavovať pole. Keďže deklarujeme ukazovateľ na pole, zrejme týmto poľom bude náš dereferencovaný ukazovateľ. Zápis takejto dereferencie je (náhodou?) zhodný s uvedeným deklarátorom ukazovateľa. Nie, to nie je náhoda, to je úmysel. Za `D1` naozaj musíme dosadiť deklarátor ukazovateľa. Lenže pozor, teraz oba deklarátory nemôžeme spojiť jednoducho pomocou substitúcie! Dôvodom je skutočnosť, že operátor `[]` má vyššiu prioritu ako `*` a my potrebujeme najprv ukazovateľ dereferencovať a až potom výsledok indexovať. Pomôžeme si rovnako ako vo výrazoch použitím okrúhlych zátvoriek. Do nich uzavrieme deklarátor ukazovateľa `*pai` a až teraz ho substituujeme za `D1`, čím dostaneme to, čo potrebujeme:

```
int (*pai)[];
```

Ak máte momentálne v hlave tak trochu „mišung“, tak vás upozorňujem, že dokonalé porozumenie predchádzajúceho príkladu je absolútne nevyhnutné na neskoršie úspešné ovládnutie umenia deklarácie objektu akéhokoľvek, ľubovoľne zložitého typu. Je veľmi dôležité uvedomiť si hierarchiu typov a podľa nej pri deklarácii postupovať. Nie je problém pomýliť sa, musíte byť preto maximálne pozorní – odmenou vám obyčajne bude nulový výskyt výnimiek typu GPF pri behu programu (vo Windows; v DOS-e to znamená, že program nebude prepisovať pamäť tam, kde nemá čo robiť).

Prejdeme si spolu ešte jeden príklad, ktorý bude dosť náročný. Ukážem vám však, že pri dodržaní správneho postupu deklarácie dôjdeme aj k správnemu výsledku. Deklarujeme si (a teraz dávajte pozor) ukazovateľ na pole desiatich ukazovateľov, z ktorých každý bude ukazovať na dvadsaťprvkové pole ukazovateľov na typ `const char`. Mohli by ste namietať, že také niečo sa v praxi hádam ani nedá použiť, ale to by ste sa mýlili – predstavte si, že máme program, ktorý používa pri rôznych operáciách dvadsať textových reťazcov. Ukazovatele na tieto reťazce máme uložené v nejakej tabuľke a reťazce sprístupňujeme z tabuľky použitím výrazu typu `*tab[i]`. Mohli by sme ich mať síce natvrdo zapísané v kóde programu všade tam, kde ten – ktorý reťazec používame, ale to by sme nemali možnosť realizácie nasledujúceho kroku: zmeny všetkých reťazcov napr. do iného jazyka. Veľmi jednoduchým a efektívnym spôsobom môžeme túto zmenu zabezpečiť výmenou používanej tabuľky. Znamená to, že budeme mať v programe viacero (minimálne teda dve) tabuliek s ukazovateľmi na príslušné reťazce. Samozrejme, najpriamočiarejším spôsobom výmeny tabuliek je zmena hodnoty nejakého všeobecne prístupného ukazovateľa, ktorý bude v každom okamihu ukazovať na práve používanú tabuľku. No ak máme viac tabuliek (dohodli sme sa, že ich máme desať), treba si pamätať adresu každej z nich – najjednoduchšie opäť vo forme nejakej tabuľky, indexovanej napr. pomocou kódu aktuálneho „jazyka“. Potom budeme mať vo všeobecne prístupnej premennej uložený nie ukazovateľ na aktuálnu tabuľku, ale tento kód a ukazovateľ získame vyhľadaním v tabuľke. Ukazovateľ na takúto tabuľku bude mať práve ten spomínaný komplikovaný typ a použijeme ho napríklad pri odovzdávaní adresy tabuľky do nejakej funkcie.

Na začiatok si uvedieme, k čomu chceme dospieť. Uvedený ukazovateľ s názvom p deklarujeme ako:

```
const char *(*(*p)[10])[20];
```

Na pohľad to nevyzerá až tak zložito, čo poviete? Poďme si teda ukázať jednotlivé kroky. Budeme postupovať podľa hierarchie (sledujte súčasne opis typu). V prvom rade deklarujeme ukazovateľ (na niečo). Teda napíšeme:

```
*p
```

Tento ukazovateľ ukazuje na typ „pole desiatich prvkov“:

```
D1[10]
```

Každý prvok poľa je opäť ukazovateľom:

```
*D2
```

A každý z týchto ukazovateľov ukazuje na pole dvadsiatich prvkov:

```
D3[20]
```

Nakoniec každý z týchto dvadsiatich prvkov je ukazovateľom (na typ `const char`):

```
*D4
```

Teraz len jednoducho postupujeme zhora nadol a každý riadok dosadíme do nasledujúceho riadka za príslušný deklarátor (pozor na prioritu, tam, kde treba, doplníme zátvorky!). Uvedieme si tu jednotlivé fázy vytvárania deklarácie, v ktorých bude substituovaná časť vždy obalená zátvorkami a zvýraznená kurzívou:

```
*p
(*p)[10]
*((*p)[10])
(*((*p)[10]))[20]
*((*((*p)[10]))[20])
```

Nakoniec odstránime nadbytočné páry zátvoriek, doplníme špecifikátor základného typu (`const char`) a dostávame želaný výsledok:

```
const char *(*(*p)[10])[20];
```

Verím, že ste celý postup bez problémov zvládli. Ukážem vám ešte jeden zo spôsobov kontroly, či to, čo ste napísali, je správne. Celý výraz predstavuje deklaráciu objektu nejakého typu. Keď sa naň pozrieme ako na deklaráciu v tvare „`const char` *xxx*“, znamená to, že časť *xxx* musí byť typu `const char`. Špecifikátor `const char` odstránime a ďalej postupujeme viac-menej rekurzívne. Časť *xxx* vyzerá takto: `*(*(*p)[10])[20]`. Vieme, že `[]` má vyššiu prioritu ako `*`, preto najprv oddelíme operátor `*` (postupujeme v opačnom poradí, ako určuje priorita!). Časť *xxx* sa nám transformuje do podoby „`*` *yyy*“, kde *yyy* je logicky typu „ukazovateľ na (už odstránený) `const char`“. V ďalšom kroku z časti *yyy* (ktorá teraz vyzerá ako `(*(*p)[10])[20]`) oddelíme `[20]`, čím dostaneme výraz „*zzz* `[20]`“, kde *zzz* je typu „pole dvadsiatich ukazovateľov na `const char`“. Časť *zzz* je teraz ekvivalentná `(*(*p)[10])`. Zátvorky nepotrebujeme, preto ich jednoducho odstránime a ďalej pokračujeme úplne rovnako. Na konci sa dopracujeme k vnorenému identifikátoru p, ktorého typ by nám mal vyjsť zhodný s naším zamýšľaným typom. Ak nám vyšlo niečo iné, v deklarácii je chyba.

(Poznámka: Vzhľadom na to, že sme sa dosiaľ nevenovali deklarácii funkcií, je deklarácia v uvedenom príklade prakticky len striedavou kombináciou polí a ukazovateľov. V praxi sa používajú aj oveľa komplikovanejšie typy, v ktorých sa vyskytujú ukazovatele na funkcie, na triedy, polia týchto ukazovateľov a podobne. Každá deklarácia sa však dá postupne rozvinúť na sériu jednoduchých deklarátorov, na základe ktorých dokážeme deklarovaný typ identifikovať.)

## Deklarácia a definícia funkcie

Aj pre deklaráciu funkcie existuje podobný recept ako pre deklaráciu predchádzajúcich objektov (pozor, ide o deklaráciu čiže prototyp funkcie, nie o jej definíciu – chýba telo funkcie):

`T D1` ( *zoznam-deklarácií-argumentov* )

Meno, ktoré takto deklarujeme (určené identifikátorom, vnoreným v deklarátore `D1`), bude mať typ „… funkcia s argumentmi podľa *zoznamu-deklarácií-argumentov*, vracajúca typ `T`“. Návratovým typom `T` môže byť ľubovoľný typ (vrátane `void`) s výnimkou polí a funkcií (keď chceme vrátiť pole, definujeme ako návratový typ ukazovateľ na typ prvku poľa; vracanie funkcií sa realizuje aj pomocou ukazovateľov na funkcie, ktoré ako návratový typ použiť môžeme).

Zoznam deklarácií argumentov je zoznamom bežných deklarácií, oddelených čiarkou. Tieto deklarácie však neurčujú nové premenné, ale tzv. formálne argumenty funkcie. Formálne z toho dôvodu, že skutočné hodnoty argumentov funkcie sú známe až v okamihu jej volania. V tele funkcie však potrebujeme na jednotlivé argumenty odkazovať, formálne argumenty sú teda formálnymi zástupcami skutočných argumentov. V rámci definície funkcie (t. j. jej tela) sa formálne argumenty správajú ako bežné, automatické (lokálne) premenné, ktorých rozsah platnosti sa začína okamihom deklarácie a končí sa v okamihu ukončenia tela funkcie. Tieto lokálne premenné sa vytvárajú na zásobníku a inicializujú sa hodnotami skutočných argumentov v okamihu volania funkcie. Formálne argumenty môžeme v tele funkcie ľubovoľne meniť, tieto zmeny sa nijako neprejavia na

hodnotách premenných, ktoré sme ako skutočné argumenty funkcii poskytli. Aby to bolo jasnejšie, ukážeme si krátky príklad:

```
#include <stdio.h>

void f(int a)
{
    a = 5;
}

void main()
{
    int b = 3;
    printf(„b = %i\n“, b);
    f(b);
    printf(„b = %i\n“, b);
}
```

V príklade máme definovanú funkciu `f()`, ktorá má jediný argument `a` typu `int`. V tele funkcie tomuto argumentu priraďujeme hodnotu `5`. Z funkcie `main()`, kde sme deklarovali lokálnu premennú `b`, zavoláme funkciu `f()` a odovzdáme jej ako skutočný argument túto premennú `b`. V C++ sa argumenty vždy odovzdávajú hodnotou (*by value*), preto funkcia `f()` dostane iba obsah premennej `b`, ktorým inicializuje svoj formálny argument (a teda svoju lokálnu premennú) `a`. Ako uvidíme po spustení programu, hodnota premennej `b` sa volaním funkcie `f()` nijako nezmení. Funkcia `f()` po inicializácii argumentu `a` stratila akékoľvek spojenie s objektom (v našom prípade premennou `b`), ktorého hodnota bola použitá ako skutočný argument.

Hovorili sme si už o tom, ako vyriešiť požiadavku odovzdávania argumentov odkazom (*by reference*). V zásade sú možné dva spôsoby. Pri prvom funkcii odovzdáme ako argument ukazovateľ na premennú, ktorú chceme meniť. Nasledujúci príklad ukazuje notoricky známy typ funkcie na výmenu obsahov dvoch premenných (typu `int`):

```
void swap(int* pa, int* pb)
{
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

Funkcia je natoľko triviálna, že ju nejdem ani vysvetľovať. Pri jej volaní musíme ako argumenty uviesť ukazovatele na premenné, ktorých obsahy vymieňame (čo je prakticky jej jediná nevýhoda).

Pri druhom spôsobe riešenia problému odovzdávania argumentov odkazom deklarujeme formálny argument ako referenciu na typ tejto premennej. Pri volaní sa (dosiaľ neinicializovaná!) referencia inicializuje odkazom na odovzdávanú premennú, ktorú tak v rámci funkcie môžeme meniť. Pozmeňme trochu predposledný príklad tak, aby funkcia `f()` dostala svoj argument ako odkaz:

```
#include <stdio.h>

void f(int& a)
{
    a = 5;
}

void main()
{
    int b = 3;
    printf(„b = %i\n“, b);
    f(b);
    printf(„b = %i\n“, b);
}
```

Pridali sme jediný znak `(,&')` do programu a jeho výstup je úplne iný – tentoraz funkcia `f()` dostane ako argument referenciu na objekt, ktorý bol použitý ako skutočný argument, takže každá zmena, ktorú táto funkcia nad svojím argumentom vykoná, sa prejaví aj na tomto skutočnom argumente.

Za domácu úlohu zmeňte funkciu `swap()` tak, aby jej argumentmi boli referencie. Je jasné, že sa zmení aj spôsob jej volania. Pre zaujímavosť tu uvádzam aj alternatívne spôsoby výmeny dvoch (celočíselných!) premenných `a` a `b`:

```
a += b;
b = a - b;
a -= b;
```

alebo aj:

```
a ^= b;
b ^= a;
a ^= b;
```

Porozmýšľajte nad tým, ako fungujú a ako je možné, že pri prvom z nich neprekáža, keď pri sčítaní/odčítaní dôjde k pretečeniu.

Aby sme sa však vrátili k funkciám – zoznam argumentov funkcie môže byť aj prázdny, to vtedy, keď funkciu nepotrebujeme volať s argumentmi. Ekvivalentom prázdneho zoznamu argumentov (ktorý zapisujeme ako `()`) je zoznam s jediným kľúčovým slovom `void`. Inak toto kľúčové slovo ako typ argumentu nemôžeme použiť (samozrejme, s výnimkou ukazovateľov na void, ale to už vieme). Ďalej má C++ jednu špecifickú vlastnosť: umožňuje deklarovať funkciu s neznámym, resp. s premenným počtom argumentov. Zoznam deklarácií argumentov takejto funkcie musí vyzerať takto:

( *zoznam-deklarácií-argumentov* , ... )

alebo:

( *zoznam-deklarácií-argumentov* ... )

Za znakmi výpustky (elipsy) " . . . " už nesmie nasledovať nijaký ďalší argument. Zoznam deklarácií argumentov nachádzajúci sa pred výpustkou môže byť aj prázdny. Príkladom takejto funkcie je vám dobre známa funkcia `printf()`, ktorej prototyp si môžete vyhľadať v hlavičkovom súbore `stdio.h`. Nájdete tam niečo príbuzné tomuto:

```
int printf(const char*, ...);
```

Tento prototyp udáva, že prvým argumentom funkcie `printf()` je vždy reťazec. Čo uvedieme za ním, to sa nedá povedať dopredu, a preto prekladač ani nebude kontrolovať zhodu typu ďalších argumentov (ani nemá s čím). Funkcia sama, samozrejme, musí nejako vedieť, aké dostala tie zvyšné argumenty – konkrétne `printf()` si to zistí na základe obsahu prvého argumentu (formátovacieho reťazca). Iným spôsobom je použitie konvencie, že posledný argument, ktorý uvedieme, bude mať nejakú známu hodnotu (napr. `0`). Spôsob, akým je možné pristupovať k nedeklarovaným argumentom, je implementačne závislý, a preto na tieto účely máme k dispozícii súbor makier definovaných v hlavičkovom súbore `stdarg.h`. Povieme si o nich však až pri rozprávaní o štandardnej knižnici.

Vráťme sa ešte na chvíľu k zoznamu argumentov. Jeho význam spočíva predovšetkým v tom, že prekladač, vidiac prototyp deklarovanej funkcie, môže pri každom jej volaní kontrolovať, či bola dodržaná (vzhľadom na štandardné konverzie) zhoda typov formálnych a skutočných argumentov. Inak povedané, ak deklarujeme funkciu s argumentom typu `char*` a voláme s argumentom typu `double`, prekladač nám pri preklade dôrazne vynadá. V tejto súvislosti treba spomenúť dôležitý fakt, že jazyk C++ povoľuje, aby dve funkcie s rôznymi typmi argumentov mali rovnaký názov. Tento zdanlivý detail je v skutočnosti obrovskou výhodou, ktorú oceníte pri návrhu rozsiahlejších systémov. Ukážeme si tu len jednoduchý (a značne otrepaný) príklad, z ktorého však bude zrejmé, o čo ide. Predstavme si, že potrebujeme funkciu na výpočet maxima z dvoch hodnôt. Nič ľahšie, definujeme funkciu `max()`:

```
int max(int a, int b)
{
    return a > b ? a : b;
}
```

Lenže čo v prípade, ak potrebujeme počítať maximum z dvoch hodnôt, ktoré sú iného typu ako `int`? V jazyku C bolo treba definovať sériu funkcií s obskúrnymi menami, ako `max_int()`, `max_long()`, `max_double()` a pod. V C++ môžeme bez problémov definovať funkciu `max()` znova s iným typom argumentov:

```
double max(double a, double b)
{
    return a > b ? a : b;
}
```

(Tento príklad je zvláštny okrem iného aj tým, že telo funkcie je rovnaké pre oba prípady. V takej situácii sa obyčajne použije šablóna funkcie. Ale o tom až niekedy v budúcnosti.)

Funkcie, ktoré majú rovnaký názov, sa musia líšiť v počte a/alebo type svojich argumentov, nestačí len odlišnosť v návratovom type! Inak by sa totiž prekladač nevedel rozhodnúť, ktorú z oboch funkcií použiť (v prípade, že návratovú hodnotu funkcie po jej zavolaní ignorujeme). Rozhodovanie prekladača, ktorú z viacerých funkcií použiť, je téma na samostatný článok, takže zatiaľ ju nenápadne preskočíme. Čo sa týka rozlišovania argumentov, typy definované pomocou `typedef` sa pokladajú za zhodné so svojimi ekvivalentmi (ehm, to asi nie je veľmi jasné – jednoducho ak máme v programe riadok `typedef int bool;`, je typ `bool` zhodný s typom `int` a funkcie `f(int a)` a `f(bool a)` *nie sú* odlišné). Počet a typ argumentov funkcie spolu s jej identifikátorom a typom návratovej hodnoty tvoria tzv. signatúru funkcie (ktorú v mierne prispôsobenej podobe v podstate používa linker) a v programe musí mať každá funkcia túto signatúru jedinečnú. Celý fenomén existencie viacerých funkcií s rovnakým názvom sa v angličtine nazýva *function overloading*. Toto slovné spojenie si preložte, ako sami uznáte za vhodné, ja sa prikláňam najviac k prekladu „prekrývanie funkcií“, oproti možno formálne správnemu, ale akosi násilnému a príliš doslovnému „preťažovanie funkcií“. *Overload* síce v angličtine znamená aj „preťažiť“, ale (to len tak na zamyslenie) slovo load sa používa v počítačovom slangu na opis úplne inej činnosti, ako je nakladanie či nabíjanie, čo vy na to?

Pri argumentoch funkcie ešte chvíľu zostaneme. C++ poskytuje pre začínajúceho programátora ešte jedno lákadlo, a tým sú implicitné argumenty funkcií. Pod implicitným argumentom si predstavte taký formálny argument, ktorý si pamätá svoju implicitnú hodnotu. Tou sa inicializuje, ak pri volaní funkcie neuvedieme jeho hodnotu explicitne. Zoberme si veľmi jednoduchý príklad – chceme vypočítať absolútnu hodnotu komplexného čísla pomocou funkcie `cabs()`. Prvým argumentom tejto funkcie nech je reálna zložka komplexného čísla, druhým imaginárna zložka. Aby sme funkciu využili aj na výpočet absolútnej hodnoty reálnych čísel (to sú vlastne všetky komplexné čísla s nulovou imaginárnou zložkou), povieme deklaráciou funkcii, že vždy, keď jej nedodáme druhý argument, má ho implicitne brať ako nulový. Prototyp takej funkcie vyzerá takto:

```
double cabs(double re, double im = 0.0);
```

Vidíme, že deklarácia implicitného argumentu vyzerá úplne rovnako ako deklarácia bežnej premennej spojená s inicializáciou. Musíme však mať na pamäti dôležitý fakt. Len čo vyhlásime niektorý z argumentov za implicitný, musíme ako implicitný deklarovať aj každý ďalší. To vyplýva aj zo spôsobu volania funkcie – keď raz nejaký argument vynecháme, musíme vynechať aj všetky „za ním“. Tým som vlastne naznačil, ako zabezpečíme použitie implicitnej hodnoty argumentu pri volaní funkcie: jednoducho príslušný argument vynecháme. Našu funkciu `cabs()` môžeme teda volať dvoma spôsobmi:

```
double a1 = cabs(1.23, -4.56);
double a2 = cabs(7.89);
```

V prvom prípade do premennej `a1` ukladáme absolútnu hodnotu čísla 1,23 – 4,56i, v druhom do premennej `a2` absolútnu hodnotu čísla 7,89. Z príkladu je zrejmé aj to, že môžeme implicitnú hodnotu argumentu zmeniť explicitným uvedením inej hodnoty (v našom prípade –4,56 namiesto nuly). Počet implicitných argumentov nie je okrem uvedeného pravidla nijako obmedzený, funkcia môže mať všetky argumenty implicitné alebo nemusí mať ani jeden. V prípade viacerých prototypov (resp. prototypu a skutočnej definície) funkcie nesmú byť implicitné argumenty predefinované (ani rovnakou hodnotou!). V praxi teda obyčajne uvediete implicitnú hodnotu do prototypov, ktoré máte uložené v hlavičkovom súbore.

Zhrňme si teraz stručne, čo sa deje, keď program dospeje do bodu, v ktorom má zavolať nejakú funkciu. Všetky skutočné argumenty, ktoré majú byť funkcii odovzdané, sú (resp. ich hodnoty) skopírované na zásobník. Poradie, v ktorom sa toto kopírovanie deje, je špecifické pre jazyk C++ – argumenty sa kopírujú sprava doľava (ako prvý sa uloží do zásobníka posledný argument funkcie a ako posledný sa uloží prvý argument) na rozdiel napríklad od Pascalu, kde sa argumenty kopírujú presne naopak, zľava doprava. Preto napríklad C++ podporuje premenný počet argumentov funkcie a Pascal nie. Obyčajne je totiž počet argumentov zakódovaný v prvom z nich a tento prvý argument je po zavolaní funkcie na vrchole zásobníka. Vieme, že zásobník je LIFO štruktúra, pristupovať teda (bežným spôsobom) môžeme iba k údajom na jeho vrchole. Po skopírovaní skutočných argumentov sa do zásobníka uloží ešte návratová adresa (aby procesor vedel, kam sa má vrátiť, keď sa funkcia skončí) a riadenie sa prenesie na začiatok volanej funkcie. Funkcia vykoná to, čo má, a eventuálne vráti volajúcemu kódu návratovú hodnotu. Tá je uložená v niektorom z registrov, prípadne aj na zásobníku. Po návrate z funkcie *volajúca funkcia* vyčistí zásobník od argumentov, ktoré tam vložila. Aj toto je špecifikom C++, v iných jazykoch (zase ma napadá len ten Pascal) zásobník čistí *volaná funkcia*. Kedysi dávno som mal v úmysle aj nakresliť, ako to vyzerá na zásobníku po zavolaní nejakej funkcie, ale predbehol ma jeden z kolegov v seriáli o programovaní v assembleri. Takže ak sa chcete poučiť, viete, kam sa máte obrátiť.

Ešte si povedzme niečo o definícii funkcií. To je, ako vieme, deklarácia, ktorá obsahuje aj telo funkcie (čiže samotný opis toho, čo funkcia robí). Ako vyzerá definícia, to hádam netreba osobitne ukazovať, je to jednoducho spojenie špecifikátorov funkcie, deklarátora opísaného v predchádzajúcich riadkoch (neukončujeme bodkočiarkou – nejde o deklaračný príkaz!!) a zloženého príkazu, predstavujúceho telo funkcie.

Na záver rozprávania o funkciách si ukážeme (podobne ako pri deklarácii polí) rozdiel medzi deklaráciou funkcie vracajúcej ukazovateľ a deklaráciou ukazovateľa na funkciu. Majme najprv funkciu `f()`, ktorá má dva celočíselné argumenty a ktorá vracia ukazovateľ na typ `int`. Pri zápise jej deklarátora postupujeme podobne, ako sme si to hovorili pri deklarácii polí, ale treba dávať pozor na to, že tu nie je až taká jasná hierarchia toho, čo deklarujeme. Chceme deklarovať „funkciu, ktorá vracia ukazovateľ“, začneme preto deklarátorom funkcie:

```
f(int a, int b)
```

Funkcia `f()` má vracať ukazovateľ, napíšeme teda ďalej deklarátor ukazovateľa:

```
* D1
```

Spojením oboch deklarátorov a pridaním typového špecifikátora dostaneme výsledok:

```
int *f(int a, int b);
```

Toto je naozaj deklarácia funkcie vracajúcej ukazovateľ na typ `int`. Druhým prípadom bude ukazovateľ `pf` na funkciu, ktorá má opäť dva celočíselné argumenty a vracia samotný typ `int`. Deklarujeme „ukazovateľ na funkciu“, tentoraz teda začneme deklarátorom ukazovateľa:

```
*pf
```

Ďalej napíšeme deklarátor požadovanej funkcie:

```
D1 (int a, int b)
```

Oba deklarátory spojíme. Aj tu však musíme dať pozor, lebo operátor volania funkcie `()` má vyššiu prioritu ako operátor `*`. Použijeme preto zátvorky a dostaneme takýto výsledok:

```
int (*pf)(int a, int b);
```

Toto je deklarácia ukazovateľa na funkciu s danými argumentmi a návratovým typom.

Zaujímavým príkladom je deklarácia funkcie, ktorá má ako argument aj ako návratovú hodnotu ukazovateľ na nejakú inú funkciu. Takouto funkciou je napríklad funkcia `signal()` zo štandardnej knižnice, ktorej prvým argumentom je typ `int` a druhým argumentom i návratovou hodnotou je ukazovateľ na funkciu typu `void f(int n)`. Takýto ukazovateľ by sme vedeli deklarovať bez problémov, čo však so spomenutou funkciou `signal()`? Riešenie je dosť náročné, postup, ktorý sme si povedali, nie je taký názorný. Problém je totiž v tom, že funkcia `signal()` vracia ukazovateľ na inú funkciu. Normálne je návratový typ funkcie vyjadrený typovým špecifikátorom, tu však typovým špecifikátorom vyjadrujeme návratový typ funkcie, na ktorú ukazuje ukazovateľ, ktorý je návratovým typom našej funkcie `signal()`. Asi to bude jasnejšie z ukážky postupu. Začíname deklarátorom funkcie s požadovanými argumentmi:

```
signal(int i, void (*pf)(int s))
```

Nasleduje deklarátor ukazovateľa (všeobecného):

```
* D1
```

Ukazovateľ ukazuje na funkciu s jediným argumentom typu `int`:

```
D2 (int s)
```

Teraz deklarátory spojíme a pridáme k nim ešte spomínaný typový špecifikátor, v našom prípade `void`:

```
void (*signal(int i, void (*pf)(int s)))(int s)
```

Ak máte pocit, že ste tak trochu mimo, nič si z toho nerobte, toto patrí takpovediac medzi „vyššiu matematiku“ C++ a časom iste všetkému porozumiete. Na domácu úlohu si vyskúšajte napríklad deklaráciu poľa ukazovateľov na funkcie, deklaráciu referencie na funkciu a podobne – fantázii sa medze nekladú.

## Abstraktné deklarátory

K úplnosti výkladu o deklarátoroch nám ešte čosi chýba. Tým čímsi sú tzv. abstraktné deklarátory. Abstraktný deklarátor vznikne z klasického odstránením identifikátora objektu, ktorý deklarujeme. Nasledujú príklady klasických deklarátorov objektov typu `int`, ukazovateľ na `int`, pole prvkov typu `int`, pole ukazovateľov na `int`, ukazovateľ na pole prvkov typu `int`, funkcia, vracajúca ukazovateľ na `int` a ukazovateľ na funkciu vracajúcu `int`:

```
int i
int *pi
int ai[]
int *api[]
int (*pai)[]
int *f()
int (*pf)()
```

a tu sú abstraktné deklarátory týchto objektov:

```
int
int *
int []
int *[]
int (*)[]
int *()
int (*)()
```

Abstraktné deklarátory sa v praxi aj naozaj používajú, a to pri deklarácii funkcií (pozor, nie pri definícii!). Vtedy totiž prekladač nepotrebuje vedieť názvy formálnych argumentov, zaujíma ho len ich typ. Môžeme teda v zozname argumentov funkcie vynechať ich identifikátory. Napríklad spomínanú funkciu `signal()` môžeme deklarovať aj takto (a v hlavičkovom súbore tak aj bude):

```
void (*signal(int, void (*)(int)))(int)
```

Okrem toho sa abstraktné deklarátory používajú na vyjadrenie typu pri explicitnom pretypovaní operátorom (*typ*).

## Inicializácia premenných

O tom, ako sa inicializujú jednotlivé typy premenných, sme už písali. Zhrnieme si preto všetky vedomosti (a doplníme novými) o inicializácii, ktorá je zo svojej podstaty súčasťou deklarácie.

Inicializáciu v zásade môžeme zapísať dvoma spôsobmi. Prvý z nich je tento:

*špecifikátory deklarátor = init-hodnota*

Druhý z nich (používaný prevažne pri objektových premenných):

*špecifikátory deklarátor ( init-hodnota )*

Premenné môžeme inicializovať ľubovoľnými výrazmi (s ohľadom na kompatibilitu typov), s použitím už deklarovaných premenných či funkcií, prípadne konštánt. Objekty typu `T` môžeme inicializovať inými objektmi typu `T` bez ohľadu na `const` či `volatile` modifikátory. Ukazovateľ typu `const T*` môžeme inicializovať ukazovateľom typu `T*`, ale nie naopak! Statické premenné (z hľadiska ukladacej triedy) sú implicitne inicializované na nulu, konvertovanú na príslušný typ, automatické a registrové premenné majú implicitne nedefinovanú hodnotu.

Polia inicializujeme pomocou zoznamu hodnôt, uzavretého v krútených zátvorkách. Počet prvkov v tomto zozname nesmie presiahnuť deklarovanú dĺžku poľa, ak dĺžku neuvedieme, prekladač si ju dosadí automaticky podľa dĺžky inicializačného zoznamu (ktorý je v takom

prípade povinný). Znakové polia (a rovnako aj ukazovatele na typ `char`) môžeme inicializovať pomocou reťazcových konštánt. Bližšie informácie nájdete v časti venovanej poliam.

Referencie typu `T&` musia byť inicializované povinne objektom typu `T` (alebo kompatibilným). Referencia na `volatile T` môže byť inicializovaná objektom typu `volatile T` alebo `T`, ale nie `const T`. Naopak, referencia na `const T` môže byť inicializovaná objektom typu `const T` alebo `T`, ale nie `volatile T`. Referencia na typ `T` môže byť inicializovaná iba objektom typu `T`. Ďalšie detaily opäť nájdete v časti venovanej referenciám.

Priraďovanie hodnoty formálnym argumentom funkcie a vytváranie návratovej hodnoty sa považuje za inicializáciu (preto referencie ako argumenty nemusia byť inicializované pri deklarácii funkcie).

### A máme to za sebou

Tak sme sa úspešne dostali na koniec dlhočizného výkladu o deklaráciách. Je to naozaj náročná téma, ktorej dokonalé zvládnutie je podmienené hlavne praxou. Nezúfajte, ak nemáte vo všetkom jasno. A ako pravidelne pripomínam, vracajte sa k starším častiam seriálu a predovšetkým si všetko skúšajte na príkladoch. Odmenou vám budú funkčné programy.

## Štrnásta časť: KONVERZIE

### Štandardné konverzie

Témou tejto kapitoly sú konverzie (implicitné, teda vykonávané automaticky za rôznych okolností) medzi údajovými typmi v C++. K týmto konverziám najčastejšie dochádza pri vyhodnocovaní výrazov, v ktorých majú jednotlivé operandy rôzne typy. Takisto sa štandardné konverzie uplatňujú pri inicializácii premenných, pri odovzdávaní argumentov funkciám, skrátka všade tam, kde sa očakáva jeden typ a použije sa iný typ. Popri štandardných konverziách jazyk C++ pozná aj konverzie definované používateľom (teraz nemyslím pretypovanie operátorom *(typ)*!) – tieto konverzie sa však týkajú objektových typov (tried), preto o nich zatiaľ nebudeme hovoriť.

### Celočíselné rozšírenia

Prvým typom automatickej konverzie typov je *celočíselné rozšírenie*. Ide o voľný (t. j. môj) preklad pôvodného anglického termínu „integral promotion“. Tento typ konverzie sa vyskytuje tak často, že si to vlastne ani neuvedomíme. Totiž všade tam, kde sa očakáva typ `int` (v znamienkovom či bezznamienkovom variante), môžeme vždy použiť aj ľubovoľný z typov `char`, `short int`, enumeračný (`enum`) typ alebo bitové pole (budeme si o ňom hovoriť neskôr) – opäť v znamienkovej i bezznamienkovej modifikácii. Každý z týchto typov sa v takom prípade pred použitím konvertuje na typ `int`, prípadne `unsigned int`. Táto konverzia je priamočiara, pretože konvertujeme typ s menším rozsahom hodnôt na typ s väčším rozsahom hodnôt, takže v konečnom dôsledku sa nanajvýš ku konvertovanej hodnote zľava doplnia nuly.

Na tomto mieste je vhodné spomenúť, že pri odovzdávaní argumentov funkcii sa obyčajne typ `char` pomocou celočíselného rozšírenia konvertuje na typ `int`, čo vyplýva z toho, že argumenty sa odovzdávajú cez zásobník a inštrukcie na prácu so zásobníkom poväčšine nevedia ukladať len jeden bajt (čo je bežná veľkosť typu `char`). Čo sa týka typu `short int`, ten býva väčšinou 16-bitový a to ešte dnešné procesory zvládnu. V prípade 32-bitových programov sa však môže stať, že vzhľadom na efektivitu programu bude prekladač aj typ `short int` konvertovať na (v tomto prípade 32-bitový) `int`.

Ak odovzdávame funkcii také argumenty, ku ktorým neexistujú ich formálne ekvivalenty (t. j. funkcia bola deklarovaná s výpustkou `[...]`), dochádza k celočíselnému rozšíreniu menovaných typov vždy, nezávisle od prostredia a/alebo typu programu. Takisto sa v takom prípade konvertuje typ `float` na `double`.

### Celočíselné konverzie

Celočíselné rozšírenia sa vzťahujú na uvedené typy. Často však potrebujeme (resp. nie my, ale prekladač) vykonať konverziu medzi dvoma všeobecnými celočíselnými typmi (napr. `unsigned int` a `long`). Pravidlá sú pomerne jednoduché. Ak prevádzame typ s väčším rozsahom na typ s menším rozsahom, musíme rátať s tým, že ak sa výsledok nezmestí do cieľového typu, dostaneme nedefinovanú hodnotu (implementačne závislú). Ak, naopak, prevádzame typ s menším rozsahom na typ s rovnakým či väčším rozsahom, konverzia sa podarí, ale výsledok bude závisieť od znamienkovosti či beznamienkovosti oboch typov.

V prípade, že konvertujeme `signed` typ na `unsigned` typ, výsledkom bude najmenšie také neznamienkové číslo, ktoré je s prevádzaným znamienkovým kongruentné modulo $2^n$. (Na vysvetlenie: čísla $a$, $b$ sú kongruentné modulo $m$, ak $a = b + k \cdot m$, kde $k$ je celé číslo). V praxi táto na pohľad komplikovaná požiadavka vedie k tomu, že bitová reprezentácia konvertovaného čísla sa obyčajne vôbec nezmení (nanajvýš dôjde k doplneniu nulami či jednotkami zľava), pretože záporné čísla sa kódujú pomocou doplnkového kódu. Pri jeho použití napríklad 16-bitové čísla 65 534 a –2 (ktoré sú kongruentné modulo $2^{16}$), vyzerajú v pamäti rovnako, totiž `0xFFFE`. Ak teda napríklad funkcii, ktorá očakáva argument typu `unsigned int`, odovzdáme hodnotu –2, funkcia dostane hodnotu 65 534 (predpokladáme 16-bitový typ `int`!). Podobne ak funkcia očakáva typ `unsigned long`, hodnota –2 sa skonvertuje na hodnotu 4 294 967 294, čo je `0xFFFFFFFE` hexadecimálne (došlo k tzv. znamienkovému rozšíreniu, t. j. k doplneniu jednotkami zľava).

Pri konverzii `unsigned` typu na `signed` typ je to o niečo jednoduchšie – ak konvertovaná hodnota môže byť zobrazená vo výslednom type, k nijakej zmene nedochádza. V opačnom prípade je výsledok nedefinovaný, ale to je situácia, keď prevádzame typ s väčším rozsahom na typ s menším rozsahom (rozsahom sa myslí interval zobraziteľných čísel, nie bitová šírka typu).

### Konverzie medzi `float` a `double`

Konverzie medzi typmi s pohyblivou desatinnou čiarkou sú v mnohom podobné celočíselným. Odpadajú nám síce starosti so znamienkami, ale stále sa môže stať, že výsledok nebude zobraziteľný v požadovanom cieľovom type. V takom prípade je, samozrejme, výsledná hodnota nedefinovaná. Ďalej môže nastať situácia, že výsledok síce spadá do rozsahu cieľového typu, ale tento typ má menšiu presnosť. Vtedy je výsledkom najbližšie vyššie či nižšie číslo, zobraziteľné v cieľovom type. Nakoniec, ak prevádzame typ s menšou presnosťou na typ s väčšou presnosťou, hodnota sa nijako nezmení.

### Konverzie medzi celými a reálnymi číslami

V nadpise tohto odseku som použil výraz reálne čísla. Samozrejme, to je nezmysel, v počítači nedokážeme reprezentovať reálne čísla, iba racionálne (to sú tie s pohyblivou desatinnou čiarkou). Ale musím brať ohľad na šírku nadpisov, preto mi odpusťte tento malý kompromis. V ďalšom texte prevažne z priestorových dôvodov budem pod reálnymi číslami myslieť tie racionálne, ktoré sú reprezentované typmi `float`, `double`, `long double`.

Konverzia reálnej hodnoty na celočíselnú sa deje odseknutím desatinnej časti. Takáto konverzia je strojovo závislá (obyčajne ju vykonáva matematický koprocesor, resp. FPU časť procesora), nie je napríklad presne definované, ako konvertovať záporné čísla, resp. ktorým smerom ich „odseknúť“ – či smerom k nule alebo od nej. Okrem toho môže nastať situácia, keď sa (po odseknutí desatinnej časti) výsledok nezmestí do rozsahu celočíselného typu a v takom prípade je výsledok ako vždy nedefinovaný.

Opačná konverzia, t. j. prevod celého čísla na reálne, sa vykonáva tak presne, ako to výsledný reálny typ umožňuje. Vieme, že reálne čísla sa uchovávajú v tvare $mantisa \times 2^{exponent}$ a vzhľadom na konečnú dĺžku mantisy sa so zvyšujúcim exponentom zväčšujú vzdialenosti medzi dvoma číslami, ktorých mantisa sa líši v najmenej významnom bite. Pre ilustráciu: predstavme si, že máme zariadenie, ktoré dokáže uchovávať reálne čísla na tri platné číslice v tvare $m \times 10^e$. Je zrejmé, že rozdiel medzi číslami $1{,}23 \times 10^2$ (čo je číslo 123) a $1{,}24 \times 10^2$ (čo je zase 124) je 1. Ak však zvýšime exponent na 3, rozdiel medzi $1{,}23 \times 10^3$ (čo je 1230) a $1{,}24 \times 10^3$ (čo je 1240) je desaťkrát väčší a teda všetky čísla medzi 1230 a 1239 vrátane budú (napríklad) reprezentované ako $1{,}23 \times 10^3$. Dochádza tu k očividnej strate presnosti, danej, ako som už spomínal, konečnou dĺžkou mantisy. V počítači je situácia veľmi podobná, až na to, že základ exponentu je iba 2 a mantisa je podstatne dlhšia. Stále sa však môže stať, že niektoré celé číslo nebude možné zobraziť úplne presne pomocou reálneho čísla.

### Aritmetické konverzie

Tento typ konverzií sa uplatňuje pri vyhodnocovaní výrazov, keď dva operandy jedného operátora (obyčajne aritmetického) majú rôzny typ. Vtedy sa postupuje podľa nasledujúceho vzoru:

- ak jeden z operandov je typu `long double`, aj druhý sa prevedie na `long double`
- inak, ak jeden z operandov je typu `double`, aj druhý sa prevedie na `double`
- inak, ak jeden z operandov je typu `float`, aj druhý sa prevedie na `float`
- inak sa na oboch operandoch vykoná prípadné celočíselné rozšírenie
- potom, ak jeden z operandov je typu `unsigned long`, aj druhý sa prevedie na `unsigned long`
- inak, ak jeden z operandov je typu `long` a druhý typu `unsigned`, potom ak typ `long` je schopný zobraziť všetky hodnoty typu `unsigned`, prevedie sa operand typu `unsigned` na `long`, inak sa oba operandy prevedú na `unsigned long`
- inak, ak jeden z operandov je typu `long`, aj druhý sa prevedie na `long`
- inak, ak jeden z operandov je typu `unsigned`, aj druhý sa prevedie na `unsigned`
- inak sú oba operandy typu `int` a k ďalšej konverzii nedochádza

Toto je presný a exaktný postup, pomocou ktorého sa oba operandy prevádzajú na najbližší spoločný typ (smerom nahor). Pri hlbšej analýze postupu zistíme, že takýmto spoločným typom nikdy nie je nižší typ ako `int`, resp. `unsigned int`.

Aritmetické konverzie sa týkajú väčšiny binárnych aritmetických operátorov a dokonca aj niektorých unárnych operátorov. Pri aplikácii unárneho operátora na celočíselný operand sa na ňom obyčajne najprv vykoná

celočíselné rozšírenie. Ak máme teda napríklad premennú `c` typu `char`, potom výraz `+c`, ktorý na pohľad nemá nijaký efekt, je typu `int`! Rovnako sa mení typ `short`. Mimochodom, bolo by teraz rozumné vrátiť sa k časti venovanej operátorom (to bola 6. časť seriálu) a prejsť si vtedajší výklad ešte raz – uvedomíte si tak, kedy a pri ktorých operátoroch má zmysel uvažovať o aritmetických konverziách. Podotýkam, že aj pri operátoroch, ktorých operandy musia byť celočíselné (ako `<<`, `>>` a pod.), dochádza pred vyhodnotením ku konverzii, a to minimálne k celočíselným rozšíreniam.

## Konverzie ukazovateľov

Aj pri práci s ukazovateľmi (inicializácia, priradenie, porovnanie) dochádza k niekoľkým konverziám. Konštantný výraz, ktorého hodnota je nulová, možno konvertovať na špeciálny ukazovateľ, nazývaný nulový ukazovateľ. Je zaručené, že hodnota takéhoto ukazovateľa bude odlišná od akejkoľvek platnej hodnoty. Obyčajne je bitová reprezentácia nulového ukazovateľa postupnosťou núl. Nulový ukazovateľ sa používa veľmi často, hlavne na vyjadrenie, že daný ukazovateľ neukazuje nikam (resp. že jeho hodnota je neplatná). Operátor `delete` aplikovaný na nulový ukazovateľ nemá nijaký nežiaduci efekt (čo je zaručené prekladačom!).

Ukazovateľ na ľubovoľný nekonštantný a ne-`volatile` objekt môžeme konvertovať na typ `void*`. Dokonca aj ukazovateľ na funkciu môžeme konvertovať na `void*`, ak typ `void*` je dostatočne široký (lebo `void*` je podstate ukazovateľom do dátového segmentu a ukazovateľ na funkciu je ukazovateľom do kódového segmentu; oba typy ukazovateľov nemusia byť rovnako široké).

Výraz typu „pole prvkov typu `T`“ môžeme konvertovať na ukazovateľ na prvý prvok poľa (čo sa aj s určitými výnimkami, o ktorých, dúfam, už veľmi dobre viete, deje automaticky). Výraz typu „funkcia vracajúca typ `T`“ sa pri použití automaticky konvertuje na typ „ukazovateľ na funkciu vracajúcu `T`“ s dvoma výnimkami. Jedna z nich je zrejmá – je to klasické funkčné volanie, operátor `()`. Druhou výnimkou je aplikácia operátora `&`, čo je vlastne explicitné vyjadrenie spomínanej konverzie.

## Explicitné konverzie

Okrem štandardných konverzií, ktoré sa vykonávajú viacmenej automaticky, máme možnosť explicitne predpísať prekladaču, že chceme konvertovať hodnotu jedného typu na iný typ. Takáto konverzia, ako vieme, sa zapisuje v tvare `(`*typ*`)`výraz alebo typ`(`*výraz*`)`. Platia pre ňu určité pravidlá, o ktorých si teraz povieme.

Ak môže byť nejaký typ skonvertovaný na iný pomocou štandardných konverzií, môže byť skonvertovaný aj pomocou explicitných konverzií; význam bude rovnaký.

Ľubovoľný ukazovateľ možno explicitne konvertovať na celočíselný typ s dostatočnou veľkosťou. Výsledok bude mať obyčajne rovnakú bitovú reprezentáciu, takže dostaneme príslušnú adresu, ktorá je obsahom premennej typu ukazovateľ. Naopak môžeme explicitne konvertovať celočíselnú hodnotu na ukazovateľ. Je zrejmé, že výsledok nemusí vôbec ukazovať na platné dáta, ale to už je starosť programátora. Tento trik sa často používal v DOS-e pri prístupe k videopamäti – jednoducho sa zapísala takáto deklarácia:

```
char far * vga = (char far *) 0xA0000000;
```

Pre tých, ktorí vedia, že začiatok framebuffera klasickej VGA karty leží na adrese `A000:0000`, je všetko jasné. Kľúčové slovo `far` je špecialitou dosovských prekladačov a vyjadruje, že ide o tzv. vzdialený ukazovateľ, ktorý sa skladá z dvoch častí – segmentovej a offsetovej, ktoré sú v pamäti uložené tak, že pri konverzii `far` ukazovateľa na celé číslo dostaneme hodnotu *segment* * $2^{16}$ + *offset*. V prípade uvedenej adresy teda dostaneme číslo, uvedené ako inicializátor v našej deklarácii.

Ukazovateľ na jeden typ môžeme explicitne konvertovať na ukazovateľ na iný typ; výsledok, ktorý dostaneme, však môže pri použití spôsobiť výnimku vzhľadom na prípadné požiadavky na zarovnanie objektov v pamäti (niektoré procesory vyžadujú napríklad umiestnenie objektov typu `int` na adresách deliteľných štyrmi).

Ľubovoľný objekt môžeme explicitne konvertovať na referenčný typ `X&`, ak ukazovateľ na tento objekt môžeme explicitne konvertovať na typ `X*`. Výsledok pretypovania na referenciu je ako jediný spomedzi všetkých pretypovaní l-hodnotou.

Je povolené pretypovať medzi sebou ukazovateľ na funkciu a ukazovateľ na dáta, pokiaľ majú oba typy ukazovateľov dostatočne veľkú bitovú šírku. Samozrejme, opäť sa môže stať, že pri použití výsledného ukazovateľa dôjde k chybe ochrany pamäte. Ďalej je povolené pretypovať medzi sebou ukazovatele na rôzne funkcie (t. j. funkcie s rôznym počtom a typmi argumentov a návratovej hodnoty). Ako dopadne volanie funkcie pomocou takto pretypovaného ukazovateľa, ťažko predvídať (hlavne keď funkcia očakáva nejaké argumenty, ktoré nedostane).

Ukazovateľ na konštantný (`const`) objekt možno pretypovať na ukazovateľ na nekonštantný objekt. Takisto možno pretypovať konštantný objekt alebo referenciu na takýto objekt na nekonštantný objekt, resp. referenciu naň. Pri pokuse o modifikáciu konštantného objektu pomocou pretypovaného ukazovateľa či referencie môžu nastať dva prípady: buď dôjde k chybe ochrany pamäte (ak je napríklad konštantný objekt uložený v read-only segmente), alebo sa nestane nič, t. j. výsledok bude rovnaký ako pri modifikácii nekonštantného objektu. Rovnaké pravidlá platia pre `volatile` objekty a ukazovatele/referencie na ne.

## Čo robí preprocesor

V druhej polovici tejto časti si povieme pár slov o tom, čo sa deje vo fáze spracovania C++ programu preprocesorom a ako toto spracovanie môžeme ovplyvňovať. Vieme, že preprocesor je prvý, kto vidí zdrojový text programu počas jeho prekladu. Výstup preprocesora sa následne odovzdá kompilátoru. Činnosť preprocesora môžeme zhrnúť do troch základných bodov: rozvoj makier, podmienená kompilácia a vkladanie súborov.

Príkazy, ktorými ovplyvňujeme fázu preprocesingu, sa nazývajú obyčajne direktívy preprocesora a začínajú sa znakom `#`. Je dôležité vedieť, že tento znak musí byť prvým znakom na riadku iným ako biela medzera (t. j. medzera alebo tabulátor), inak preprocesor príslušnú direktívu nerozpozná. Direktívy preprocesora majú svoju vlastnú gramatiku, nezávislú od gramatiky C++ – de facto sa kompilátor C++ nikdy nedozvie, že to, čo dostal na vstupe, prešlo nejakým preprocesorom. Efekt direktívy preprocesora sa končí na konci spracúvanej prekladovej jednotky (t. j. súboru), pokiaľ ho explicitne nezmení iná direktíva.

V prípade, že potrebujeme zapísať dlhšiu direktívu (a pri používaní makier to nie je nič nezvyčajné), môžeme pokračovať na ďalšom riadku, musíme však predchádzajúci riadok ukončiť znakom `\` (back-slash). Preprocesor dva (či viaceré) takto rozdelené riadky pred spracovaním jednoducho spojí. Ako príklad (trochu netypický) môže slúžiť nasledujúca direktíva (ktorá je vám už veľmi dobre známa):

```
#include \
    <stdio.h>
```

Znak `\` však nesmie byť posledným znakom súboru.

Preprocesing môžeme rozložiť do niekoľkých fáz. Skutočná implementácia môže byť, samozrejme, ľubovoľná, ale výsledný efekt musí byť zhodný. Tu sú jednotlivé fázy:

- vykonajú sa prípadné systémovo-závislé preklady znakov (napr. `CR/LF` namiesto `LF` a pod.), trigrafy (pozri ďalej) sa nahradia svojimi jednoznakovými ekvivalentmi
- spoja sa riadky rozdelené pomocou znaku `\` (jednoducho sa zo zdrojového textu vymaže každá dvojica znakov `\` a prechod na nový riadok)
- zdrojový text sa rozloží na postupnosť lexikálnych jednotiek (nielen jazyka C++, ale aj jazyka preprocesora!), každý komentár sa nahradí jednou medzerou
- vykonajú sa jednotlivé direktívy a definované makrá sa nahradia svojimi rozvojmi
- „escape“ sekvencie v znakových a reťazcových konštantách (ako `\n` a pod.) sa nahradia svojimi ekvivalentmi
- susediace reťazcové konštanty sa spoja (ako napr. „`ab`“ „`cd`“ sa spojí do „`abcd`“)

Výsledkom preprocesingu je upravený zdrojový text programu, ktorý sa predloží na ďalšie spracovanie kompilátoru.

Spomínané trigrafy (trigrafové sekvencie) sú dnes už prakticky zbytočnou a dávno zastaranou súčasťou C++. Ich cieľom malo byť umožnenie zápisu niektorých „interpunkčných“ znakov, nevyhnutných na zápis programu, aj v takých znakových sadách, ktoré prekypovali diakritikou na úkor práve týchto znakov. V tabuľke č. 1 je uvedený vždy znak a jeho trigrafový ekvivalent. Nie všetky prekladače podporujú trigrafy, to už však dnes zrejme nikoho trápiť nebude.

**Tabuľka č. 1**

| znak | trigraf | znak | trigraf | znak | trigraf |
|---|---|---|---|---|---|
| # | ??= | [ | ??( | { | ??< |
| \ | ??/ | ] | ??) | } | ??> |
| ^ | ??' | \| | ??! | ~ | ??- |

## Definícia a rozvoj makier

Jednou z najdôležitejších úloh preprocesora je rozvoj vložených makier. Pod makrom budeme v ďalšom texte rozumieť textové makro, čo nie je nič iné ako vopred definovaná postupnosť znakov, ktorá sa počas preprocesingu nahradí inou, takisto vopred definovanou postupnosťou znakov. Poviete si, na čo je to dobré, to nemôžem do programu priamo napísať tú výslednú postupnosť? Čo však v prípade, že máte nejaký úsek programu (hocijaký, od jedinej číslice, vyjadrujúcej počet kanálov vašej zvukovej karty, cez krátku textovú správu, pomocou ktorej žiadate od používateľa, aby stlačil nejaký kláves, až po jednu celú funkciu) a tento úsek sa vyskytuje na viacerých miestach. Ak by nebolo makier, každá zmena takéhoto opakovaného úseku by znamenala komplikované prechádzanie celým zdrojovým textom, hľadanie všetkých výskytov a ich úpornú modifikáciu, nehovoriac o zúfalých pokusoch udržať zhodné všetky kópie úseku. Makrá celú námahu zjednodušia obmedzením hľadania miesta modifikácie na jediný bod – miesto definície makra. V celom programe sa bude používať len názov tohto makra, ktorý sa počas preprocesingu nahradí jeho skutočným obsahom. Je pravda, že so zavedením konštantných objektov v C++ (teda objektov deklarovaných ako `const`) význam makier mierne ustúpil do pozadia, no vzhľadom na možnosť definície parametrizovaných makier stále ešte existujú situácie, keď je vhodné použiť namiesto konštanty makro preprocesora (i keď, aby som bol objektívny, aj parametrizované makrá majú svoju náhradu – funkcie `inline`). Základnou nevýhodou

použitia makier je skutočnosť, že ich reprezentáciu pozná iba preprocesor, a preto nie sú dostupné počas ladenia v debuggeri.

Prvý typ makier (bez parametrov) sa definuje pomocou direktívy `#define`:

`#define` *identifikátor reťazec*

Počnúc prvým riadkom za touto direktívou nahradí preprocesor všetky výskyty *identifikátora* uvedeným *reťazcom*. Reťazec môže obsahovať aj medzery, prípadné biele znaky medzi ním a identifikátorom a za ním (v definícii, samozrejme) sa ignorujú. Raz definovaný identifikátor môžeme pomocou `#define` predefinovať iba na rovnaký substitučný reťazec. Príklad:

```
#define N 256

double matrix[N][N];
```

Po prechode preprocesorom sa uvedená deklarácia zmení na:

```
double matrix[256][256];
```

Druhým typom makier sú makrá s parametrami. Definujú sa podobne:

`#define` *identifikátor* ( *param* , ... , *param* ) *reťazec*

Medzi identifikátorom a ľavou okrúhlou zátvorkou **nesmie** byť medzera! Parametre makra sú takisto bežné identifikátory. Každý výskyt identifikátora, nasledovaného zátvorkou (, zoznamom lexikálnych jednotiek a pravou zátvorkou ) sa nahradí uvedeným reťazcom, v ktorom budú výskyty jednotlivých (formálnych) parametrov nahradené skutočnými parametrami (podobne ako pri volaní funkcií). Skutočné argumenty (lexikálne jednotky) musia byť oddelené čiarkami. Počet formálnych a skutočných argumentov musí byť zhodný. Ukážeme si príklad:

```
#define INDEX_MASK 0xFF00
#define EXTRACT(word,mask) word & mask
```

S takto definovanými makrami sa výraz

```
index = EXTRACT(data, INDEX_MASK);
```

rozvinie na

```
index = data & 0xFF00;
```

Je veľmi dôležité uvedomiť si, že pri rozvoji makier dochádza k čisto textovej substitúcii, nezávislej od konštrukcií a gramatiky C++. Z toho dôvodu sa riadok s direktívou `#define` nekončí bodkočiarkou (pokiaľ to tak explicitne nechceme). A ďalej si treba dobre premyslieť, čo všetko môže byť skutočným argumentom nášho makra. Klasický príklad – majme makro, ktoré vypočíta súčin svojich argumentov (no, nevypočíta, ale vytvorí taký výraz):

```
#define KRAT(a,b) a * b
```

Na pohľad je všetko v poriadku. Ak však zavoláme toto makro s nasledujúcimi parametrami:

```
x = KRAT(5 + 6, 7 + 8);
```

dostaneme po substitúcii výsledok:

```
x = 5 + 6 * 7 + 8;
```

čo zrejme nie je to, čo sme chceli. Univerzálnym riešením je preto používať pri definícii takýchto parametrických makier zátvorky všade tam, kde by mohlo dôjsť k nesprávnej interpretácii. Správne definované makro `KRAT` potom vyzerá takto:

```
#define KRAT(a,b) ((a) * (b))
```

Ak vám nie je jasné, prečo je celý súčin navyše obalený ďalším párom zátvoriek, zamyslite sa nad volaním `KRAT(1+2, 3+4) + KRAT(5+6, 7+8)`.

Pri definícii makier máme k dispozícii dva špeciálne operátory, # a ##. Každý formálny parameter v definícii makra, ktorému tesne predchádza operátor #, bude pri rozvoji makra nahradený nie priamo skutočným parametrom, ale reťazcovým literálom C++, obsahujúcim tento skutočný parameter. Myslím, že príklad to ozrejmí:

```
#define LABEL(text) „label_“ #text
```

Ak toto makro použijeme napríklad v tvare:

```
LABEL(x001)
```

dostaneme po nahradení takýto výsledok:

```
„label_“ „x001“
```

z čoho po spojení reťazcov vznikne literál:

```
„label_x001“
```

Argumentom makra `LABEL` môže byť prakticky ľubovoľný text, ktorý sa pri substitúcii de facto „obalí“ úvodzovkami.

Druhý z operátorov, ##, slúži na spájanie lexikálnych jednotiek. Ak sa vyskytne v definícii makra, po nahradení formálnych parametrov skutočnými sa jednoducho odstráni spolu so všetkými bielymi medzerami, ktoré ho obklopujú. Príklad:

```
#define PTRDEF(T) typedef T* ptr_ ## T
```

Argumentom tohto makra by malo byť meno existujúceho typu. Makro slúži na deklaráciu nového typu, ekvivalentného s ukazovateľom na pôvodný typ. Jeho použitie napríklad v tvare:

```
PTRDEF(double);
```

vedie k substitúcii:

```
typedef double* ptr_double;
```

ktorej dôsledkom je deklarácia nového typu `ptr_double`. Vidíme, že postupnosť znakov `ptr_ ## T` z definície makra sa po nahradení `T` reťazcom `double` a následnom spojení zmenila na `ptr_double`.

Po nahradení všetkých parametrov makra skutočnými argumentmi je výsledok opätovne podrobený analýze a prípadným ďalším substitúciám. Ak sa však v rozvoji makra vyskytne jeho názov, rekurzívna substitúcia sa nevykoná (pretože by sa nikdy neskončila). Rozvoj makra sa skončí, keď sa v ňom nenájde nijaký ďalší reťazec podliehajúci substitúcii. Ak je prípadne výsledkom rozvoja niečo, čo na pohľad vyzerá ako direktíva preprocesora, už sa to ďalej nespracúva.

V prípade, že potrebujeme predefinovať existujúce makro, musíme ho najprv zrušiť direktívou `#undef`. Jej syntax je jednoduchá:

`#undef` *identifikátor*

Ak náhodou *identifikátor* nie je platným názvom makra, direktíva sa ignoruje.

## Vkladanie súborov

Na vkladanie iných (obyčajne hlavičkových) súborov do zdrojových súborov programu slúži direktíva `#include`. Opisoval som ju už v jednej z predchádzajúcich častí, takže sa nebudem opakovať. Stručne len spomeniem, že sú povolené dva tvary:

`#include <`*meno_súboru*`>`

a

`#include „`*meno_súboru*`“`

(rozdiely pozri v predchádzajúcom výklade). Ak reťazec za direktívou `#include` nemá ani jeden z dvoch uvedených tvarov, podlieha normálnemu spracovaniu preprocesorom, ktorého výsledok musí mať jeden z týchto tvarov.

## Podmienená kompilácia

Preprocesor jazyka C++ umožňuje riadiť, ktoré časti zdrojového kódu sa budú prekladať a ktoré nie. Všeobecná schéma na vyjadrenie podmienenej kompilácie je nasledujúca:

```
#if konšt-výraz
...
#elif konšt-výraz
...
#else
...
#endif
```

Prvá z direktív, `#if`, začína úsek podliehajúci podmienenej kompilácii. Ak je *konštantný výraz*, ktorý je argumentom tejto direktívy, nenulový, všetok nasledujúci text až po prvú z direktív `#elif`, `#else` a `#endif` sa bude prekladať. V opačnom prípade (t. j. výraz je nulový) sa text ignoruje a bude sa hľadať nasledujúca direktíva. Ak je ňou `#elif`, celý proces s vyhodnocovaním výrazu sa opakuje, ak je ňou `#else`, prekladať sa bude text nasledujúci za `#else`. Celý blok podmienenej kompilácie ukončuje direktíva `#endif`. V rámci jedného bloku môže byť aj viacero direktív `#elif`. Je zrejmé, že prekladu bude podliehať najviac jeden úsek kódu, a to ten, ktorý nasleduje za direktívou s nenulovou hodnotou výrazu, resp. za direktívou `#else`. Tento úsek kódu podlieha, samozrejme, aj ďalšiemu spracovaniu preprocesorom (ako prípadná ďalšia podmienená kompilácia, rozvoj makier atď.).

Konštantný výraz musí spĺňať požiadavky na správnosť z hľadiska gramatiky C++, musí byť celočíselný, nesmie obsahovať pretypovanie, operátor `sizeof` a enumeračné konštanty. Typy `int` a `unsigned int` sa berú ako `long` a `unsigned long`. Navyše je dovolené používať špeciálny unárny operátor `defined`, ktorý sa vyhodnocuje na jednotku, ak je jeho argument definovaným (a dosiaľ nezrušeným) makrom, a na nulu, ak nie je. Možno ho použiť v dvoch tvaroch:

`defined` *identifikátor*

alebo

`defined` (*identifikátor*)

Direktívu v tvare:

`#if defined` *identifikátor*

môžeme skrátene zapísať aj takto:

`#ifdef` *identifikátor*

a direktívu v tvare:

`#if !defined` *identifikátor*

zase takto:

`#ifndef` *identifikátor*

Ešte si ukážeme krátky príklad. V nasledujúcom úseku kódu definujeme typ NUMBER podľa zvolenej presnosti ako `double` alebo `float`. Voľba sa realizuje na základe existencie či neexistencie makra `HIGH_PRECISION`:

```
#define HIGH_PRECISION
```

```
#ifdef HIGH_PRECISION
     typedef double NUMBER;
#else
     typedef float NUMBER;
#endif

NUMBER vector[1000];
```

Samozrejme, pokiaľ chceme, aby program reagoval na túto našu voľbu, musíme ďalej dôsledne používať typ NUMBER.

### Ďalšie direktívy

V C++ máme k dispozícii okrem spomínaných direktív ešte niekoľko ďalších. Prvá z nich, direktíva `#line`, slúži na zmenu aktuálneho čísla riadka či mena súboru na účely symbolického ladenia. Samozrejme, nemení tieto údaje z hľadiska kompilátora, ale len na účely prípadného výpisu, zmenou špeciálnych automaticky definovaných makier `__LINE__` a `__FILE__` (pozri ďalej). Jej syntax je nasledujúca:

`#line` *konštanta „meno_súboru“*

Meno súboru nie je povinné. Riadok s touto direktívou pred spracovaním podlieha textovej substitúcii.

Direktíva `#error` slúži na zastavenie prekladu:

`#error` *správa*

Správu uvedenú v direktíve vypíše kompilátor po zastavení prekladu. Táto direktíva sa používa v prípade, že nechceme pokračovať v preklade, lebo nie je napríklad splnená určitá podmienka. Jednoduchý príklad – ak sa omylom budeme pokúšať C++ program kompilátorom jazyka C, môže takáto direktíva zastaviť preklad bez toho, aby sme od kompilátora dostali dlhočizný zoznam čudesných chýb:

```
#ifndef _cplusplus
#error „C++ compiler must be used!“
#endif
```

Makro `_cplusplus` je automaticky definované, pokiaľ používame kompilátor jazyka C++.

Direktíva `#pragma` umožňuje výrobcom kompilátorov doplniť existujúcu funkčnosť preprocesora a/alebo kompilátora vlastnými vylepšeniami. Jej rôzne formy sú vždy implementačne závislé, typicky sa používa na potlačenie rôznych druhov varovaní, na optimalizáciu prekladu, definíciu zarovnávania dátových štruktúr a pod. Všeobecná syntax je:

`#pragma` *reťazec*

Pre bližšie informácie treba nahliadnuť do dokumentácie k prekladaču.

Poslednou direktívou je tzv. prázdna direktíva, ktorá nemá nijaký efekt. Zapisuje sa veľmi jednoducho:

```
#
```

Počas kompilácie prekladač automaticky definuje niekoľko makier. Dve z nich, `__LINE__` a `__FILE__`, sme už spomínali. Prvé z nich obsahuje vždy číslo práve prekladaného riadka, druhé zase meno práve prekladaného súboru. Makro `__DATE__` obsahuje dátum prekladu (vo forme reťazcového literálu, t. j. v úvodzovkách) a makro `__TIME__` logicky čas prekladu (opäť vo forme reťazca). Tieto makrá nemožno predefinovať ani zrušiť. Navyše by mal každý prekladač C++ definovať makro `_cplusplus`.

Popri spomínaných makrách každý z prekladačov môže definovať rôzne iné makrá, vyjadrujúce napr. typ prostredia, typ aplikácie, pamäťový model a iné.

Nabudúce sa budeme venovať štandardnej knižnici jazyka C. Nebude to podrobný opis ani prerozprávanie manuálu, skôr len naznačím okruhy funkcií, ktoré máme k dispozícii. To bude posledná časť zaoberajúca sa neobjektovou polovicou C++. Úmyselne som vynechal problematiku štruktúrovaných typov (`struct`, `union`), ktoré súvisia s objektovými typmi a bude vhodnejšie hovoriť o nich v tejto súvislosti.

## Pätnásta časť

Ako som minule sľúbil, naposledy sa budeme venovať neobjektovej polovici C++. Budeme si rozprávať o tzv. štandardnej knižnici jazyka C, ale skôr ako začneme, dovolil by som si ešte pár slov k otázke spustenia a ukončenia programu v C++.

### Prológ a epilóg

Vieme už, že beh programu v C++ sa točí okolo jednej špecifickej funkcie s názvom `main()`. Práve ona obsahuje našu „pridanú hodnotu“, teda kód, ktorý sme naprogramovali. Samozrejme, že časť kódu sa môže nachádzať v samostatných funkciách, ktoré v príhodnom okamihu z funkcie `main()` zavoláme (a tak ďalej, rekurzívne do takmer ľubovoľnej hĺbky). Dôležité však je, že nami ovplyvnená časť programu sa začína tam, kde sa začína funkcia `main()`, a z normálnych okolností sa končí tam, kde sa táto funkcia končí.

Pri návrhu štruktúry programu neraz dôjde k situácii, keď potrebujeme vykonávanie programu ukončiť predčasne – teda skôr, ako program prirodzene dospeje ku koncu funkcie `main()`. Ak sa táto situácia vyskytne v kóde, ktorý je súčasťou `main()`, môžeme použiť, tak ako v každej inej funkcii, príkaz `return`, voliteľne nasledovaný návratovou hodnotou. Na pohľad sa bude realizovať bežný návrat z funkcie; no návrat z `main()` sa rovná ukončeniu programu. Prípadná návratová hodnota sa odovzdá nejakým spôsobom operačnému systému, ktorého prostriedkami ju budeme môcť neskôr zistiť.

Oveľa častejšie sa však stane, že budeme potrebovať ukončiť program v niektorej z vnorených funkcií – v takom prípade nám pomôže funkcia `exit()`. Jej prototyp sa nachádza v hlavičkovom súbore `<stdlib.h>` a obyčajne aj v súbore `<process.h>` a vyzerá takto:

```
void exit(int);
```

Vidíme, že funkcia `exit()` má jediný argument typu `int`, tento argument je už spomínanou návratovou hodnotou programu. Obyčajne sa dodržuje konvencia, že nulová návratová hodnota indikuje ukončenie programu bez chýb, nenulová signalizuje nejakú chybu (napríklad program na kopírovanie súborov môže nenulovou návratovou hodnotou oznamovať, že požadovaný súbor neexistuje).

Volanie funkcie `exit()` bude mať za následok (okrem ukončenia programu) spláchnutie všetkých výstupných prúdov, zavretie všetkých otvorených súborov a zavolanie všetkých funkcií registrovaných pomocou volania funkcie `atexit()`. Táto funkcia s prototypom

```
void atexit(void (*)());
```

ktorý sa nachádza v `<stdlib.h>`, umožňuje definíciu „obslužných rutín“ ukončenia programu. Takéto obslužné rutiny môžu mať na starosti prípadné aditívne činnosti, ktoré treba vykonať skôr, než program odíde do „večných lovísk“ (typicky uvoľnenie alokovaných prostriedkov a pod.). Každá registrovaná funkcia musí mať prototyp

```
void fnc();
```

funkcii `atexit()` sa odovzdáva ukazovateľ na danú funkciu, resp. priamo identifikátor funkcie, ktorý sa v takomto prípade automaticky konvertuje na ukazovateľ. Ukážeme si príklad:

```
#include <stdio.h>
#include <stdlib.h>

void fnc1()
{
     printf(„Handler #1\n“);
}

void fnc2()
{
     printf(„Handler #2\n“);
}

void main()
{
     atexit(fnc1);
     atexit(fnc2);
     printf(„main()\n“);
     exit(0);
}
```

Po spustení programu si všimnime, že obslužné funkcie sa vyvolávajú v opačnom poradí, ako boli registrované (t. j. stratégia LIFO).

Ak vo funkcii `main()` použijeme príkaz `return`, bude jeho účinok ekvivalentný volaniu funkcie `exit()` s príslušným argumentom (`return` bez návratovej hodnoty je totožný s volaním `exit(0)`). V tejto súvislosti je vhodné spomenúť, že funkcia `main()` môže byť deklarovaná s návratovým typom `void` alebo `int`. V prvom prípade, ak chceme operačnému systému vrátiť nenulovú hodnotu, musíme *vždy* použiť funkciu `exit()`. Naproti tomu v druhom prípade nezávisle od toho, či chceme alebo nechceme vracať nejakú hodnotu, musíme *vždy* použiť buď funkciu `exit()`, alebo príkaz `return`, a to aj vtedy, keď za normálnych okolností by program sám dospel ku koncu funkcie `main()`. Niektoré benevolentnejšie prekladače pri chýbajúcom príkaze `return` vydajú iba varovanie, ide však o chybu, pretože funkcia, ktorá je deklarovaná s návratovým typom iným ako `void`, *musí* vrátiť nejakú hodnotu. Preto, ak si spomínate, náš prvý program v C++ vyzeral takto:

```
int main()
{
     printf(„Hello, world!\n“);
     return 0;
}
```

Ak by sme chceli vynechať príkaz `return`, museli by sme deklarovať funkciu `main()` ako

```
void main() { ... }
```

Popri funkcii `exit()` môžeme program ukončiť i podobnou funkciou `_exit()`. Prototypy oboch funkcií sú okrem názvu zhodné, líšia sa však účinkom – volanie funkcie `_exit()` ukončí program *bez* spláchnutia výstupných prúdov, uzavretia otvorených súborov či volania funkcií registrovaných pomocou `atexit()`, o čom sa ľahko môžete presvedčiť modifikáciou uvedeného programu.

Program sa dá ukončiť ešte jednou funkciou, `abort()`. Na rozdiel od predchádzajúcich dvoch však táto funkcia ukončuje program násilne, bez ohľadu na alokované prostriedky, na konzolu obyčajne vypíše správu o takomto násilnom ukončení a v operačných systémoch typu Unix vyprodukuje aj tzv. core dump, čo je, zjednodušene povedané, do súboru zaznamenaný obraz pamäte, ktorá bola pridelená danému procesu. Funkcia `abort()` sa používa obyčajne len v prípade, keď dôjde k takej závažnej chybe, že program jednoducho nemôže pokračovať ďalej a nedokáže sa ani z chyby zotaviť. Prototyp funkcie

`void abort()`

sa nachádza v hlavičkovom súbore `<stdlib.h>` či `<process.h>`.

Dosiaľ sme sa pohybovali v rámci funkcie `main()`. Program však skôr, než zavolá túto funkciu, musí vykonať ešte niekoľko prípravných prác. O čo presne pôjde, to závisí od prekladača, ktorým sme program preložili. My ako programátori do tohto úseku programu zasahovať priveľmi nemôžeme – vlastne jediný spôsob, ako zmeniť tento tzv. prológ programu, je zmeniť príslušný objektový súbor, ktorý sa prilinkuje k výslednému programu. Vo všeobecnosti prológ pripravuje prostredie na beh funkcie `main()` – inicializuje statické objekty, spracuje príkazový riadok a vytvorí z neho argumenty pre `main()`, otvorí štandardné súbory (`stdin, stdout, stderr`) a priradí im príslušné deskriptory a tak ďalej. Potom bežným spôsobom zavolá našu funkciu `main()`, po ktorej skončení nastupuje záverečný kód (epilóg). Tento kód robí práve to upratovanie, ktoré sme si opísali pri funkcii `exit()`. Po jeho skončení sa už riadenie vracia operačnému systému.

## Štandardná knižnica jazyka C

Termín štandardná knižnica jazyka C treba chápať pomerne voľne – myslím ním tie funkcie, ktoré dostaneme automaticky k dispozícii spolu s prekladačom a ktoré by sa mali vyskytovať v každej implementácii nezávisle od platformy či výrobcu. Navyše túto množinu obmedzíme na funkcie, ktoré boli k dispozícii už v jazyku C, a teda nebudeme uvažovať funkcie vyžadujúce C++.

Nie je veľmi jednoduché určiť, ktoré funkcie možno považovať za štandardné a ktoré sú rozšírením, špecifickým pre ten-ktorý prekladač. Mnohé funkcie „zľudoveli“ a vyskytujú sa vo viacerých prekladačoch, hoci sú napríklad závislé od platformy. V nasledujúcom prehľade sa teda pokúsim vymenovať a prípadne stručne opísať také funkcie, o ktorých si myslím, že sú dostatočne všeobecné a sú k dispozícii naozaj s každým prekladačom. (Pre jazyk C existuje norma ANSI/ISO, ktorá, myslím, opisuje aj množinu funkcií, ktoré sa musia povinne dodávať spolu s prekladačom, ale, bohužiaľ, nemôžem toto tvrdenie momentálne ničím podložiť, pretože nemám spomínanú normu k dispozícii.) Celkom určite niektoré menej používané funkcie vynechám, pretože cieľom tejto časti nie je vymenovať všetko, čo máme k dispozícii, ale skôr poskytnúť vám určitý náhľad na to, s čím môžete pri písaní vlastných programov rátať (a nemusíte to programovať sami).

## Matematické a konverzné funkcie

Prvou kategóriou funkcií, o ktorých si povieme, sú funkcie na výpočet rôznych matematických výrazov, prípadne na prevod medzi rôznymi typmi údajov. Prototypy týchto funkcií sa nachádzajú prevažne v hlavičkových súboroch `<math.h>, <stdlib.h>` a iných.

Na výpočet absolútnej hodnoty čísla slúži funkcia `abs()`. Existuje obyčajne v niekoľkých variáciách, ktoré sa líšia typom argumentu; keďže ide o funkcie jazyka C, musia sa tieto variácie líšiť aj názvom funkcie (takže máme funkcie `fabs()`, `labs()` a iné). Nebudem (ani v ďalšom texte) uvádzať prototypy funkcií, pretože by som asi zbytočne duplikoval manuály k prekladačom, takže presný opis typov argumentov a návratovej hodnoty a takisto hlbší opis činnosti funkcií nájdete v príslušnej dokumentácii.

Na výpočet maxima či minima z dvoch čísel máme logicky k dispozícii funkcie `max()` a `min()`. Pri ich použití pozor, pretože sú často definované aj ako makrá, ktoré, samozrejme, zneprístupnia skutočné funkcie. Ako tieto makrá odstrániť, to sa zvyčajne uvádza v dokumentácii k prekladačom (prinajhoršom napíšete niečo ako `#undef max`).

Na výpočet goniometrických funkcií môžeme použiť množstvo funkcií – základné `sin()`, `cos()`, `tan()`, ich inverzné protipóly (podľa správnosti ide o cyklometrické funkcie) `asin()`, `acos()`, `atan()` a hyperbolické funkcie `sinh()`, `cosh()`, `tanh()`.

Do triedy exponenciálnych a logaritmických funkcií patria funkcie `exp()` (výpočet $e^x$), `pow()` (výpočet $x^y$), `sqrt()` (odmocnina), `log()` (prirodzený logaritmus), `log10()` (dekadický logaritmus). Všeobecný logaritmus $\log_z x$ môžeme vypočítať podľa známeho vzorca ako `log(x)/log(z)`, prípadne `ln(x)/ln(z)`.

Často máme k dispozícii funkcie na prácu s komplexnými číslami, ale tie obyčajne používajú na ich reprezentáciu štruktúry, o ktorých sme si nehovorili, pretože sú veľmi podobné triedam a považoval som za vhodnejšie nechať si rozprávanie o nich práve na nasledujúcu časť seriálu. Obyčajne ide o funkcie ako `real()` a `imag()` na určenie zložiek komplexného čísla, `conj()` na výpočet konjugovaného čísla a `cabs()` a `arg()` na určenie absolútnej hodnoty a argumentu.

Medzi matematické funkcie patrí aj niekoľko ďalších funkcií, ako `ceil()` na zaokrúhľovanie nahor a `floor()` na zaokrúhľovanie nadol, `div()` a `ldiv()` na výpočet podielu a zvyšku po delení dvoch celých čísel, funkcie `rand()` a `random()` na generovanie pseudonáhodných čísel (rozdiel medzi nimi – pozri dokumentáciu) a `randomize()` a `srand()` na inicializáciu tohto generátora.

Nakoniec si spomenieme ešte tzv. konverzné funkcie. Tie používame v prípade, že potrebujeme previesť číslo na reťazec či reťazec na číslo. Pre prvý prípad máme k dispozícii funkcie `itoa()`, `ltoa()`, `ultoa()` na prevod celočíselných typov a `ecvt()`, `fcvt()`, `gcvt()` na prevod reálnych typov, pre druhý prípad zase funkcie `atoi()`, `atol()`, `atof()`, `strtol()`, `strtoul()`, `strtod()`. Názvy týchto funkcií ostatne hovoria samy za seba.

Medzi konverzné funkcie môžeme zaradiť aj funkcie `toupper()`, `tolower()` a `toascii()`, ktoré prevádzajú svoj znakový argument na veľké písmeno, malé písmeno, resp. 7-bitový znak ASCII (orezaním 8. bitu).

## Klasifikačné funkcie

Tieto funkcie sú často k dispozícii aj ako makrá, ktorým sa, samozrejme, pri použití dá prednosť – ak chceme naozaj použiť funkciu, musíme príslušné makro oddefinovať (`#undef`). Každá z týchto funkcií dostane ako argument znak (typ `char`) a vráti nenulovú hodnotu, ak tento znak spĺňa určité kritérium. Funkcií je spolu dvanásť: `isalnum()`, `isalpha()`, `isdigit()`, `isxdigit()`, `isascii()`, `iscntrl()`, `isprint()`, `isgraph()`, `islower()`, `isupper()`, `ispunct()`, `isspace()`; ich prototypy sa nachádzajú v súbore `<ctype.h>`. Prvá z nich, `isalnum()`, „rozpoznáva“ alfanumerické znaky, teda písmená a číslice, `isalpha()` iba písmená (nezávisle od ich veľkosti), `isdigit()` zase iba číslice, `isxdigit()` takisto číslice, ale hexadecimálne (nezávisle od veľkosti písmen A až F). `isascii()` kladne klasifikuje „čisté“ znaky ASCII (s nulovým 8. bitom), `iscntrl()` riadiace znaky (s kódom 0x00 až 0x1F, plus znak 0x7F), `isprint()` „tlačiteľné znaky“ s kódom 0x20 až 0x7E, `isgraph()` je to, čo `isprint()`, ale bez znaku 0x20 (t. j. bez medzery), `islower()` identifikuje malé písmená, `isupper()` veľké, `isspace()` rozpoznáva „biele“ znaky (s kódom 0x09 až 0x0D a 0x20) a `ispunct()` je konjunkciou `iscntrl()` a `isspace()` (tzv. „punctuation characters“).

## Funkcie na prácu s reťazcami a pamäťou

Manipulácia s reťazcami je v C++, jemne povedané, krkolomná, a preto nečudo, že máme k dispozícii veľké množstvo funkcií na jej uľahčenie. Všetky tieto funkcie považujú ako reťazec ľubovoľne dlhé pole znakov ukončené nulou (teda presnejšie znakom `'\0'`). Často sa pre takýto reťazec používa výraz ASCIIZ string alebo zero-terminated string. Prototypy funkcií na prácu s reťazcami sa prakticky všetky nachádzajú v súbore `<string.h>`.

Asi najznámejšou funkciou z tejto triedy je funkcia `strlen()`, ktorá vracia dĺžku reťazca (nerátajúc ukončovací nulový znak). Na spájanie reťazcov použijeme funkciu `strcat()`, ktorá pripojí jeden reťazec k druhému. Tento druhý reťazec musí mať, samozrejme, „za sebou“ dostatočne dlhý úsek pamäte, aby sa doň pripájaný prvý reťazec zmestil. Variantom je funkcia `strncat()`, ktorá umožňuje zadať max. počet pripájaných znakov (vtedy sa však môže stať, že sa nepripojí ukončovacia nula a reťazec zostane takpovediac „visieť vo vzduchu“).

Na kopírovanie reťazcov slúži funkcia `strcpy()`, ktorá skopíruje jeden reťazec do druhého (ten musí byť opäť dostatočne dlhý, inak si prepíšeme to, čo nechceme). Jej činnosť sa skončí prekopírovaním ukončovacej nuly zo zdrojového reťazca. Aj k tejto funkcii existuje reštriktívna alternatíva `strncpy()`, pre ktorú takisto platí poznámka z predchádzajúceho odseku. Ak chceme vytvoriť duplikát reťazca, použijeme funkciu `strdup()`, ktorá za nás aj alokuje príslušnú pamäť (tú však budeme musieť explicitne uvoľniť sami).

Na porovnávanie reťazcov máme k dispozícii funkciu `strcmp()`. Táto funkcia porovnáva reťazce znak po znaku a končí sa vtedy, ak nájde prvú nezhodu alebo ak narazí aspoň v jednom z reťazcov na ukončovaciu nulu. Porovnanie je bezznamienkové, na základe číselných hodnôt znakov (čiže na základe kódu ASCII). Funkcia vracia nulovú hodnotu pre rovnaké reťazce a zápornú či kladnú, ak je jeden z reťazcov lexikograficky menší či väčší ako druhý. Ak chceme obmedziť max. počet porovnávaných znakov, použijeme funkciu `strncmp()`, na porovnávanie nezávisle od veľkosti písmen v reťazcoch slúžia funkcie `stricmp()` a `strnicmp()`.

Ak potrebujeme nastaviť jednotlivé znaky reťazca na určitú hodnotu, môžeme použiť funkciu `strset()`, resp. `strnset()`. Na konverziu reťazca na veľké či malé písmená (in situ, teda v rámci samotného reťazca) slúžia funkcie `strupr()` a `strlwr()`. Revertovať reťazec môžeme pomocou funkcie `strrev()`.

Na vyhľadávanie znaku v reťazci máme k dispozícii funkcie `strchr()` (vracia prvý výskyt znaku) a `strrchr()` (vracia posledný výskyt). Na vyhľadávanie celého podreťazca môžeme použiť funkciu `strstr()`. Medzi ďalšie vyhľadávacie funkcie patria `strtok()` a `strspn()`.

Na prácu s pamäťou máme k dispozícii podobný komfort. Na kopírovanie úseku pamäte slúžia funkcie `memcpy()` a `memmove()`. Obe majú podobný účinok, tá druhá navyše ošetruje prípad, keď sa pôvodný úsek a jeho kópia v pamäti prekrývajú, takže by mohlo dôjsť k prepísaniu nežiaducich údajov (tí, čo si ešte pamätáte inštrukcie procesora Z80, spomeňte si, ako sa pomocou tohto faktu a inštrukcie `LDIR` dala krásne vynulovať pamäť… nedá mi, aby som tu ten kúsok kódu z nostalgie neuviedol:

```
ld   HL,start
ld   DE,start+1
ld   BC,length
ldir
```

(Ach jaj, to boli časy!) Rozdiel oproti podobným funkciám na prácu s reťazcami je ten, že kopírovanie sa

nekončí žiadnou nulou a dĺžka úseku sa musí explicitne uviesť v argumentoch týchto funkcií.

Porovnávanie dvoch úsekov pamäte realizuje funkcia `memcmp()`, resp. `memicmp()`, nastavenie úseku pamäte na nejakú hodnotu funkcia `memset()`. Vyhľadávať konkrétnu hodnotu v pamäti môžeme pomocou funkcie `memchr()`, ale aj pomocou funkcií `lsearch()` (lineárne hľadanie) a `bsearch()` (binárne hľadanie), ktoré majú trochu univerzálnejšiu sémantiku. Pri tejto príležitosti treba spomenúť aj funkciu `qsort()`, implementujúcu usporadúvací algoritmus Quick Sort.

Čo sa týka alokácie pamäte, pre ňu nebola v jazyku C nijaká konštrukcia podobná operátoru `new` z C++, takže bolo potrebné použiť knižničné funkcie `malloc()`, `calloc()` či `realloc()` na alokovanie bloku pamäte a `free()` na jej uvoľnenie. Na zistenie dostupnej takto alokovateľnej pamäte slúžila funkcia `coreleft()`. Používať tieto funkcie v C++ nemá veľký význam, preto ich ani nebudem bližšie opisovať.

## Vstupno-výstupné funkcie

Realizácia vstupu a výstupu údajov (z hľadiska programu) nie je súčasťou jazyka C++ ani C, preto nečudo, že štandardná knižnica obsahuje zrejme najväčší podiel práve vstupno-výstupných funkcií. Rozdeliť ich môžeme zhruba na funkcie pracujúce so štandardným vstupom a výstupom, funkcie na vstup a výstup údajov z a do údajových prúdov, funkcie na prácu s diskovými súbormi a funkcie na prácu s konzolou. Prototypy funkcií nájdeme predovšetkým v súbore `<stdio.h>` a aj v mnohých ďalších.

Prvou oblasťou sú funkcie, ktoré čítajú údaje zo štandardného vstupu, resp. zapisujú ich na štandardný výstup. Medzi ne patrí notoricky známa funkcia `printf()`, ktorá realizuje formátovaný výstup do súboru `stdout`, implicitne smerovaného na konzolu (t. j. na obrazovku). Syntax tejto funkcie už isto poznáte a viete, že prvým argumentom je formátovací reťazec, podľa jeho pokynov sa vypisujú nasledujúce argumenty. Súčasťou tohto reťazca sú riadiace špecifikátory, ktorých presný opis nájdete v dokumentácii ku každému prekladaču. Je dobré preštudovať si tento opis, objavíte tak nespočetné možnosti funkcie `printf()` a jej rôznych variácií (bude o nich reč o chvíľu). Z nich najpodobnejšia je funkcia `vprintf()`, ktorá má namiesto premenného počtu argumentov jediný argument (samozrejme, okrem formátovacieho reťazca) špeciálneho typu `va_list`, slúžiaceho na platformovo nezávislý prístup k argumentom funkcie deklarovanej s výpustkou. O chvíľu si o ňom povieme bližšie.

Na zápis neformátovaného reťazca na štandardný výstup máme k dispozícii funkciu `puts()`, na výstup jediného znaku zase funkciu `fputchar()` a makro `putchar()`, ktoré má rovnaký účinok. Tu dochádza k istej nekonzistencii v pomenovávaní, lebo hoci názov funkcie `fputchar()` sa začína znakom `f`, nejde o zápis do súboru.

Komplementom funkcie `printf()` je tiež kedysi spomínaná funkcia `scanf()`, ktorá realizuje formátované čítanie údajov zo štandardného vstupu. Pre jej formátovací reťazec platia mierne odlišné pravidlá, preto treba opäť preštudovať dokumentáciu, v ktorej sú opísané. Vieme už, že argumentmi tejto funkcie musia byť ukazovatele na premenné, do ktorých budú zapisované údaje vložené. Podobnou funkciou, ktorá pracuje s argumentom typu `va_list`, je funkcia `vscanf()`.

Čítanie reťazca zo štandardného vstupu má na starosti funkcia `gets()`. Treba mať na pamäti, že táto funkcia načíta celý riadok, končiaci sa znakom `'\n'`, ktorý nahradí ukončovacou nulou, teda znakom `'\0'`. Vzhľadom na krkolomnosť a pomerne veľkú nepružnosť funkcie `scanf()` je výhodnejšie používateľský vstup z konzoly realizovať práve pomocou `gets()` a následne získaný reťazec spracovať vo vlastnej réžii. Argumentom `gets()` je ukazovateľ, ktorý musí ukazovať na dostatočne veľký úsek pamäte. Teoreticky, pokiaľ operačný systém nelimituje dĺžku jedného riadka, napísaného na konzole, nikdy nevieme, aký dlhý reťazec naozaj bude. Vtedy si musíme pomôcť rôznymi trikmi (napríklad napísať vlastnú verziu funkcie `gets()`). Na načítanie jedného znaku zo štandardného vstupu máme k dispozícii funkciu `fgetchar()` a makro `getchar()`, ktoré však vzhľadom na fakt, že operačný systém vstup z klávesnice prakticky vždy ukladá do riadkovej vyrovnávacej pamäte, nemajú priveľký zmysel (musíme totiž každý vstup ukončiť stlačením klávesu Enter). Azda iba v prípade, že z toho vopred načítaného riadka potrebujeme získavať jednotlivé znaky.

Ďalšia skupina funkcií pracuje s pojmom údajový prúd. V podstate nejde o nič iné ako o pomyselnú „rúru“, z ktorej čítame alebo do ktorej zapisujeme údaje. Jeden koniec vidíme my (t. j. program), druhý je napojený na diskový súbor, na konzolu či napr. na tlačiareň. Dôležité je, že z hľadiska programu stále ide o jeden údajový typ. Takto môžeme rovnakými funkciami zapisovať napríklad do súboru i na obrazovku. Abstrakciu údajového prúdu vytvára používateľský typ `FILE`, ktorý je de facto štruktúrou (`struct`). Funkcie, ktoré s ním pracujú, vyžadujú ukazovateľ naň (teda typ `FILE*`). (*Poznámka*: Nemýľte si spomínanú „rúru“ so skutočnými rúrami, existujúcimi v normálnych operačných systémoch a slúžiacimi na komunikáciu medzi procesmi, hoci prístup k nim je rovnako možné realizovať pomocou údajových prúdov.)

Na štandardný vstup, výstup a chybový výstup máme automaticky v programe k dispozícii tri premenné (typu `FILE*`) `stdin`, `stdout` a `stderr`, s ktorými môžeme pracovať rovnako ako s akýmikoľvek inými prúdmi.

Prv, než budeme môcť pracovať s údajovým prúdom, musíme ho najprv otvoriť. Vtedy sa vlastne vytvorí spojenie s protiľahlým objektom a inicializujú sa určité polia štruktúry `FILE`. Na otvorenie prúdu slúži funkcia `fopen()`, ktorej argumentom je okrem názvu súboru či zariadenia (ako napr. „`PRN:`“ v DOS-e) aj mód otvorenia prúdu (čítanie, zápis, pridávanie, textový/binárny prenos údajov). Na „znovuotvorenie“ súboru, resp. na prepojenie rúry na iný objekt slúži funkcia `freopen()`, vhodná napríklad na presmerovanie `stdout` do súboru.

Podobné funkcie ako pre zápis na štandardný výstup existujú aj pre zápis do údajového prúdu. Formátovaný výstup tak realizujú funkcie `fprintf()` a `vfprintf()`, ktoré majú navyše argument predstavujúci príslušný prúd. Reťazec zapíšeme pomocou funkcie `fputs()` a znak pomocou funkcie `fputc()`, resp. makra `putc()`. Ďalej máme k dispozícii funkciu na zápis bloku údajov (resp. bloku bajtov) `fwrite()`, ktorej argumentmi sú okrem iného ukazovateľ na tento blok a jeho dĺžka.

Čítanie z dátového prúdu, naopak, zabezpečujú funkcie `fscanf()`, `vfscanf()` (formátovaný vstup), `fgets()` (čítanie reťazca), `fgetc()`, `getc()` (čítanie znaku – funkcia a makro) a čítanie bloku údajov funkcia `fread()`.

Niekoľko ďalších funkcií nám umožňuje operovať s aktuálnou pozíciou v súbore (t. j. miestom, odkiaľ sa budú čítať, resp. kam sa budú zapisovať nasledujúce dáta). Funkcia `ftell()` túto aktuálnu pozíciu vracia, `fseek()` ju zase nastavuje. Podobný význam majú funkcie `fgetpos()` a `fsetpos()`, ktoré však pracujú so špeciálnym typom `fpos_t`. Pomocou funkcie `feof()` zisťujeme, či sme už na konci súboru (to značí, či posledná operácia čítania/zápisu detegovala koniec súboru), funkciou `rewind()` previnieme ukazovateľ pozície späť na začiatok súboru.

Funkcia `ferror()` zisťuje, či pri poslednom čítaní/zápise došlo k nejakej chybe, `clearerr()` takýto príznak chyby vynuluje. V prípade, že potrebujeme explicitne „spláchnuť“ vyrovnávacie pamäte údajového prúdu [teda odpratať ich fyzicky na disk (to nie je celkom presné, údaje zapisované na disk sú totiž vyrovnávané ešte samotným operačným systémom)], použijeme funkciu `fflush()`.

Po skončení práce s údajovým prúdom ho musíme vždy explicitne zavrieť. To má za následok okrem iného vyprázdnenie vyrovnávacích pamätí a, samozrejme, zavretie súboru na úrovni operačného systému, teda aktualizáciu metaúdajov na disku (adresár, alokačná tabuľka a pod.). Na tieto účely slúži funkcia `fclose()`.

Na tomto mieste ešte spomeniem funkcie, ktoré umožňujú realizovať formátovaný vstup a výstup z a do reťazcov v pamäti. Efekt je rovnaký ako pri čítaní/zápise z/do súboru, ibaže sa pracuje s existujúcim reťazcom, resp. alokovaným miestom v pamäti. Funkcie sú štyri, a to: `sprintf()`, `vsprintf()`, `sscanf()` a `vsscanf()`. Jedným z ich argumentov je reťazec predstavujúci požadovaný zdroj či spotrebič údajov.

Okrem prístupu k súborom prostredníctvom údajových prúdov k nim môžeme pristupovať takmer na úrovni operačného systému, a to pomocou nasledujúcich funkcií. V nich je identifikátorom otvoreného súboru kladné celé číslo, tzv. deskriptor súboru (file handle). Za pozornosť stojí, že tak ako sú štandardné prúdy prístupné pomocou premenných `stdin`, `stdout` a `stderr`, sú im priradené po rade deskriptory `0`, `1` a `2`. Pri otváraní ľubovoľného súboru sa mu priradí automaticky najnižšie nepoužité číslo deskriptora.

Súbor otvoríme použitím funkcie `open()`. Aj tentoraz môžeme určiť mód otvoreného súboru a typ prístupu k nemu. Podobná funkcia `sopen()` berie do úvahy možnosť viacnásobného otvorenia (v multipoužívateľskom prostredí nič nezvyčajné) a umožňuje nastaviť mód zdieľania otvoreného súboru. Zaujímavé sú dve funkcie `dup()` a `dup2()`. Prvá z nich vytvára nový deskriptor pre už otvorený súbor, zatiaľ čo pomocou druhej môžeme existujúcemu deskriptoru priradiť iný súbor (tak sa dá presmerovať napr. štandardný výstup).

Vytvoriť nový súbor môžeme jednak pomocou vhodných argumentov funkcie `open()`, jednak pomocou špeciálnej funkcie `creat()`. Jej účinok je inak okrem vytvorenia nového súboru zhodný s funkciou `open()`, t. j. takisto dostaneme ako návratovú hodnotu deskriptor novo otvoreného súboru.

Na čítanie a zápis údajov z/do súborov nemáme priveľa funkcií– vlastne iba dve: `read()` a `write()`, ktoré sú sémantikou podobné funkciám `fread()` a `fwrite()`. Aktuálnu pozíciu v súbore zistíme pomocou `tell()` a nastavíme pomocou `lseek()`. Na detekciu konca súboru slúži funkcia `eof()`.

K dispozícii máme funkcie, pomocou ktorých môžeme zistiť údaje o otvorenom súbore: `filelength()` vracia jeho dĺžku, `fstat()` rôzne iné údaje v štruktúre `stat`. A nakoniec po skončení práce so súbormi ich uzavrieme pomocou funkcie `close()`.

V každom prostredí prekladača máme obyčajne k dispozícii funkcie na prácu s neotvorenými súbormi, adresármi, zisťovanie aktuálneho adresára, prácu s diskmi a podobne. Tieto funkcie však často bývajú minimálne platformovo, ak nie priamo prekladačovo závislé. Spomeniem preto len niekoľko – `rename()` na premenovávanie súborov, `unlink()` či `remove()` na ich mazanie, `chmod()` na zmenu prístupových práv, `mkdir()` a `rmdir()` na vytváranie a vymazávanie adresárov a mnohé iné. Bližšie informácie a hlavne ďalšie funkcie nájdete celkom určite vo vašej dokumentácii.

Konečne funkcie na prácu s konzolou sú rovnako

často závislé od prekladača a ešte výraznejšie od platformy. Preto ich tu nebudem menovať vôbec, aby som neznevýhodnil tých, ktorí pracujú napríklad v Linuxe a borlandovské špeciality nemôžu používať. Ostatne dve z nich, `getch()` a `kbhit()`, som v jednom zo svojich programov už použil aj opísal.

### Funkcie na prácu s procesmi

Dve z tejto triedy funkcií sme už v tomto článku preberali – sú to funkcie `exit()` a `abort()`. Popri nich najvýznamnejšími zástupcami sú funkcie `exec...()` a `spawn...()` (resp. ich všetky varianty). Obe slúžia na spúšťanie programov (lepšie povedané, nových procesov). Rozdiel medzi nimi je ten, že prvá z nich novým programom prekryje ten, v ktorom sa nachádzala, takže po jeho skončení sa pôvodný program k slovu vôbec nedostane (lebo už ani neexistuje v pamäti), zatiaľ čo druhá vytvorí pre nový program samostatný subproces a čaká na jeho skončenie. Nie som si istý, či v operačných systémoch typu Unix funkcia `spawn()` existuje, ale ak nie, tak jej použitie je zhruba ekvivalentné dvojici `fork()` a `exec()`.

Každá z funkcií má osem modifikácií, ktoré sa líšia príponou (preto tie tri bodky v názvoch funkcií): `execl()`, `execle()`,`execlp()`,`execlpe()`,`execv()`,`execve()`, `execvp()`, `execvpe()` a podobne pre `spawn()`. Tie funkcie, ktoré obsahujú písmeno ‚l', sú deklarované s premenným počtom argumentov, predstavujúcich argumenty príkazového riadka spúšťaného programu, naopak, tým, ktoré obsahujú písmeno ‚v', sa tieto argumenty odovzdávajú vo forme poľa ukazovateľov. Funkcie obsahujúce v názve písmeno ‚e' navyše musia dostať ako argument pole ukazovateľov na premenné prostredia. A konečne funkcie obsahujúce ‚p' pri hľadaní spúšťaného programu prezerajú okrem zadanej cesty, resp. aktuálneho adresára aj adresáre uložené v premennej prostredia `PATH`.

Medzi známejšie patria aj dve funkcie `signal()` a `raise()`, ktoré pochádzajú z Unixu a predstavujú v podstate niečo ako mechanizmus softvérových prerušení. Ide o tému na dlhšie vysvetľovanie, a ak vás zaujímajú bližšie detaily, skúste si preštudovať okrem dokumentácie k prekladaču nejakú knihu o programovaní v Unixe.

### Funkcie na prácu s dátumom a časom

V tejto oblasti vládne tak trochu chaos – čo prekladač, to iné funkcie. Medzi tie, ktoré nájdeme takmer všade, patrí funkcia `strftime()`, ktorá formátuje zadaný čas a dátum na reťazca podľa požadovaných pravidiel (pracuje so špeciálnou štruktúrou `tm`), ďalej `mktime()` na prevod medzi štruktúrou `tm` a typom `time_t`, `gmtime()` na opačný prevod, funkcie `ctime()` a `asctime()` na neriadený prevod času na reťazec, `difftime()` na zistenie rozdielu v sekundách medzi dvoma časmi a predovšetkým funkcia `time()` na zistenie systémového času a dátumu a `stime()` na jeho nastavenie. Popri nich isto nájdeme aj funkcie viac na mieru šité danému operačnému systému, ktoré budú často aj vhodnejšie svojou sémantikou.

### Funkcie na prácu so zoznamom argumentov

Poslednou skupinou, o ktorej si dnes povieme, sú tri funkcie (v skutočnosti sú to makrá) na prácu s argumentmi funkcie deklarovanej s výpustkou. Taká funkcia má vždy aspoň jeden známy (pevný) argument. Všetky tri funkcie pracujú so špeciálnym typom `va_list`, spolu s ním sú definované v súbore `<stdarg.h>`.

Prácu so zoznamom argumentov začneme volaním prvej funkcie, `va_start()`. Jej parametrami sú jednak deklarovaná premenná typu `va_list` (interne to je obyčajne nejaký ukazovateľ na zásobník), jednak posledný pevný argument funkcie. Jednotlivé argumenty potom sprístupňujeme volaním druhej funkcie, `va_arg()`. Táto funkcia okrem parametra typu `va_list` (ktorý by mal byť pred tým inicializovaný pomocou `va_start()`, inak je výsledok nepredvídateľný) očakáva typ sprístupňovaného argumentu (keďže ide o makro, je to v poriadku – normálne by sme názov typu nemohli odovzdať do funkcie ako jej argument). Samozrejme, nerobí sa nijaká kontrola, či to, čo nám `va_arg()` vráti, je naozaj platná premenná požadovaného typu, túto kontrolu musíme prípadne robiť sami. Jednoducho sa vezme kus pamäte (na zásobníku), pretypuje sa na daný typ a vráti ako výsledok. Súčasne sa modifikuje premenná typu `va_list` tak, aby ďalšie volanie `va_arg()` sprístupnilo zase ďalší argument. Takisto nikto nekontroluje, či to s volaniami `va_arg()` „neprešvihneme" a nezačneme čítať nejaké nezmysly.

Po prečítaní všetkých argumentov by sme mali zavolať tretiu funkciu, `va_end()`, ktorá má jediný argument, a to našu premennú typu `va_list`. Obyčajne táto funkcia, resp. makro nič nerobí, ale človek nikdy nevie a podľa normy je volanie `va_end()` povinné.

Na záver si ešte ukážeme krátky príklad na použitie spomínaných funkcií, a to funkciu, ktorá spočíta svoje argumenty typu `int` a vráti ich súčet. Prvým argumentom funkcie bude počet sčítancov:

```
#include <stdio.h>
#include <stdarg.h>

int sum(int n, ...)
{
    va_list arg;
    int s = 0;

    va_start(arg, n);
    while (n--)
        s += va_arg(arg, int);
    va_end(arg);

    return s;
}

int main()
{
    printf(„sum1 = %i\n", sum(3, 1, 2, 3));
    printf(„sum2 = %i\n", sum(5, 7, 6, 5, 4, 3));

    return 0;
}
```

Dúfam, že vám je program jasný a nepotrebuje komentár.

## Šestnásta časť: ŠTRUKTÚRY A UNIONY

Na úvod tohto pokračovania si dovolím malú skrytú reklamu. Naďabil som nedávno v kníhkupectve na knihu, ktorá na našom trhu výrazne chýbala. Vydalo ju vydavateľstvo Grada Publishing, s. r. o., jej názov je *Jazyky C a C++ podle normy ANSI/ISO* a podtitul *Kompletní kapesní průvodce*. Túto knihu vrelo odporúčam každému, kto berie C++ aspoň trochu vážne, pretože ide o jedinú publikáciu tohto typu v češtine (o slovenčine taktne pomlčím), ktorá navyše obsahuje opis C++ podľa aktuálnej normy ANSI, resp. ISO. Od kúpy by vás možno mohla odradiť cena, ktorá je asi 500 Sk (a to ide o pomerne malý paperbackový formát…), ale myslím si, že takáto investícia sa vám veľmi rýchlo vráti, aj keď v trochu inej podobe.

Ale teraz poďme k nášmu seriálu. Pomyselným spojivom medzi oboma jeho polovicami sú štruktúrované údajové typy – štruktúry, uniony a triedy. Ako uvidíme neskôr, štruktúry sú de facto zvláštnym prípadom tried. Uniony na rozdiel od štruktúr či tried predstavujú špecifické typy s mierne odlišnou sémantikou.

### Štruktúry

Kedysi dávno, v jednej z prvých častí seriálu, sme si hovorili, že údajové typy v C++ môžeme rozdeliť na základné a odvodené. Základné typy, ako napríklad `int` či `double`, predstavujú základné stavebné kamene na vytváranie zložitejších typových konštrukcií. Každý zo základných typov je deklarovaný pomocou svojho kľúčového slova, má svoj rozsah hodnôt a definovanú množinu operácií. Naproti tomu odvodené typy sú rôznymi spôsobmi skonštruované z viacerých základných typov, z viacerých prvkov jedného základného typu alebo sú určitými sémantickými modifikáciami základných typov. Zatiaľ z odvodených typov poznáme polia, t. j. vektory prvkov (jediného!) základného typu, a, samozrejme, ukazovatele a referencie. (Funkcie, ktoré sú vlastne tiež údajovými typmi, môžeme považovať za odvodené typy alebo ich z tohto prehľadu môžeme vylúčiť.)

Pri návrhu programu veľmi často prídeme k záveru, že by sme potrebovali údajový typ, ktorý by bol *nehomogénnym* združením viacerých prvkov *rozdielnych* základných typov. Takémuto typu sa často hovorí „agregovaný" a v C++ je jeho najjednoduchšou realizáciou údajový typ *štruktúra* (`struct`). Každý objekt, ktorý bude deklarovaný ako objekt typu štruktúry, bude predstavovať spomínanú kolekciu združených objektov základných alebo aj odvodených typov. Tieto združené objekty budeme nazývať členmi, resp. členskými objektmi danej štruktúry.

Deklarácia údajového typu *štruktúra* je pomerne jednoduchá. Predovšetkým si treba spomenúť na rozdiel medzi deklaráciou a definíciou a na zloženie deklarácií. Špecifikátor, ktorý použijeme pri deklarácii štruktúry, pozostáva z kľúčového slova `struct` a zo zoznamu deklarácií jednotlivých členov štruktúry uzavretého v krútených zátvorkách. Ak uvedieme za špecifikátorom deklarátor(y), deklarujeme súčasne s novým údajovým typom aj jeden či niekoľko objektov tohto typu. V praxi sa však oddeľuje deklarácia typu štruktúry, od deklarácie objektov typu tejto štruktúry a to z dôvodov, ktoré si spomenieme o chvíľu. Toto rozdelenie je možné vďaka tomu, že z deklarácie štruktúry môžeme bez problémov vynechať deklarátor(y). Bežne potom v programe uvidíme nasledujúci zápis:

```
struct S
{
    int a;
    double b;
    char c;
};

S s1, s2;
```

V príklade deklarujeme nový agregovaný údajový typ `S` (všimnite si, že nepotrebujeme mechanizmus `typedef`!) a následne dve premenné `s1` a `s2` tohto typu. Samotný typ `S` je v programe dostupný buď pod svojím krátkym názvom `S`, alebo v prípade, že neskôr deklarujeme nejaký iný objekt (lokálnu premennú, funkciu a pod.) s názvom `S`, pod úplným názvom `struct S`. Oba objekty `s1` a `s2` by sme mohli preto deklarovať aj takto:

```
struct S s1, s2;
```

Deklaráciu typu `S` aj deklaráciu objektov `s1` a `s2` môžeme, samozrejme, spojiť:

```
struct S { ... } s1, s2;
```

Tento spôsob je však oveľa menej prehľadný a navyše neumožňuje vloženie deklarácie štruktúry do hlavičkového súboru (kde je jej najčastejšie miesto, aby sme ju mohli používať vo viacerých zdrojových súboroch).

Nakoniec je tu ešte jedna možnosť, keď z deklarácie vypustíme názov typu `S`. V takom prípade deklarujeme objekty `s1` a `s2` ako objekty neznámeho (nepomenovaného) typu, nekompatibilného so žiadnym iným typom, hoci by mal aj rovnaké zloženie. To znamená, že napríklad v nasledujúcej deklarácii:

```
struct { int x; } r1;
struct { int x; } r2;
```

sú `r1` a `r2` objekty **rôznych (!)** typov, hoci majú na pohľad úplne rovnaké zloženie.

Čo sa týka zoznamu deklarácií členov štruktúry, ide o deklarácie v klasickom zmysle slova s niekoľkými obmedzeniami. Členmi štruktúr môžu byť premenné, funkcie, iné štruktúry či triedy, enumerácie, typy a bitové polia. O bitových poliach si povieme o chvíľu, všetko ostatné už poznáme. Členy štruktúry nemôžu byť deklarované so špecifikátormi `auto`, `register` alebo `extern`. Naopak, pri deklarácii môžeme použiť špecifikátor `static`, pomocou ktorého odlišujeme tzv. statické členy (bude o nich reč neskôr).

Štruktúra nesmie obsahovať údajový člen typu samej seba, môže však obsahovať ukazovateľ, resp. referenciu na svoj typ (teda aj ukazovateľ na samu seba, ak nám to nejako pomôže). Údajové členy, ktoré sú poľami, musia mať deklarované všetky rozmery.

Deklarácie členov štruktúry nesmú obsahovať inicializátory, inicializácia sa vykonáva pomocou špeciálnej členskej funkcie. Keďže nie je priveľmi bežné, aby štruktúry (t. j. typy `struct`) obsahovali členské funkcie či vnorené typy, obmedzíme sa zatiaľ len na štruktúry obsahujúce klasické premenné (t. j. údajové členy) a o členských funkciách budeme hovoriť až v súvislosti s triedami.

## Prístup k členom štruktúry

Jednotlivé členy štruktúry musia byť nejakým spôsobom prístupné zvonka, inak by celý koncept štruktúr stratil zmysel. Toto sprístupnenie sa vykonáva pomocou dvoch operátorov. Oba sú binárne a infixové a majú v podstate najvyššiu prioritu (rovnakú ako operátory `()` a `[]`). Prvým z týchto dvoch operátorov je operátor `.` (bodka). Jeho ľavým operandom je objekt typu štruktúry, pravým operandom je identifikátor niektorého člena tejto štruktúry. Majme definovanú štruktúru `S` a objekty `s1` a `s2` ako predtým:

```
struct S
{
    int a;
    double b;
    char c;
};

S s1, s2;
```

Potom jednotlivé členy objektu `s1` naplníme napríklad takto:

```
s1.a = 123;
s1.b = 4.567;
s1.c = ‚D‘;
```

Vidíme, že takto sprístupnené členy sú l-hodnotami, keďže aj objekt `s1` je sám osebe l-hodnotou. Okrem toho môžeme objekty typu štruktúry priraďovať aj medzi sebou:

```
s2 = s1;
```

Toto priradenie sa vzhľadom na neexistenciu určitých členských funkcií realizuje člen po člene (lepšie povedané, bajt po bajte) – ide o tzv. shallow copy (plytkú kópiu). Jeho primárnou nevýhodou je, že ak je členom typu štruktúry ukazovateľ, skopíruje sa jeho hodnota z jednej štruktúry do druhej tak, ako je, čo vedie k tomu, že obe štruktúry budú de facto zdieľať jedny údaje (tie, na ktoré ukazuje ukazovateľ). To nám nemusí vždy vyhovovať, väčšinou chceme, aby každý objekt typu štruktúry obsahoval ukazovateľ na vlastné, privátne údaje. V takom prípade situáciu vyriešia spomínané členské funkcie, menovite konštruktor a operátor priradenia. Ale o tom až neskôr.

Druhým operátorom prístupu k členom štruktúry je operátor `->` (pomlčka a znak „je väčšie“). Jeho jediný rozdiel oproti operátoru `.` je ten, že jeho ľavým operandom nie je priamo objekt typu štruktúry, ale ukazovateľ na tento objekt. Ak teda máme deklarovaný takýto ukazovateľ:

```
S* ps = &s1;
```

môžeme uvedené priradenia realizovať aj takto:

```
ps->a = 123;
ps->b = 4.567;
ps->c = ‚D‘;
```

Je očividné, že výraz `ps->a` je ekvivalentný výrazu `(*ps).a`, len je oveľa prehľadnejší.

Objekty typu takýchto jednoduchých štruktúr (bez konštruktorov, bez neverejných členov, bez nadradených štruktúr a bez virtuálnych funkcií) môžeme spolu s deklaráciou aj inicializovať. Za znamienko `=` uvedieme v krútených zátvorkách uzavretý zoznam inicializátorov jednotlivých členov štruktúry, oddelených čiarkami. Hore uvedená inicializácia priradením po zložkách sa teda dá zapísať aj takto:

```
S s1 = { 123, 4.567, ‚D‘ };
```

V prípade vnorených štruktúr postupujeme podobne – ako príslušný inicializátor uvedieme opäť v zátvorkách uzavretý zoznam inicializátorov vnorenej štruktúry.

Inicializácia štruktúr, resp. tried takýmto spôsobom nie je veľmi používaná, je oveľa výhodnejšie inicializovať jednotlivé členy štruktúry pomocou samostatnej funkcie so špeciálnym postavením – tzv. konštruktora. Bližšie sa o konštruktoroch dozviete v ďalších častiach.

## Bitové polia

Zvláštnym typom členských údajov štruktúry sú tzv. bitové polia. Ide o bežné celočíselné premenné, pri ktorých však môžeme navyše určiť ich bitovú šírku. Ak máme také členské premenné, ktorých obsah sa vojde do menšieho počtu bitov, ako je najmenší typ, ktorý C++ poskytuje (t. j. `char`), ušetríme pomocou bitových polí často drahocenné miesto v pamäti (a nielen v pamäti, ale aj napríklad v súboroch a pod.). Okrem toho sa bitové polia využívajú vtedy, ak pracujeme s údajmi, ktoré získavame externe, mimo nášho programu a vieme o nich, že majú príslušnú štruktúru – typicky napríklad pri práci s hardvérovými registrami, ktoré bývajú delené po bitoch.

Deklarácia bitového poľa obsahuje za deklarátorom príslušného člena dvojbodku, nasledovanú celočíselnou konštantou, vyjadrujúcou šírku tohto poľa. Typ takéhoto člena musí byť `int` alebo `unsigned` (podľa toho sa, samozrejme, s jeho obsahom pracuje ako so znamienkovou či neznamienkovou hodnotou). Príklad:

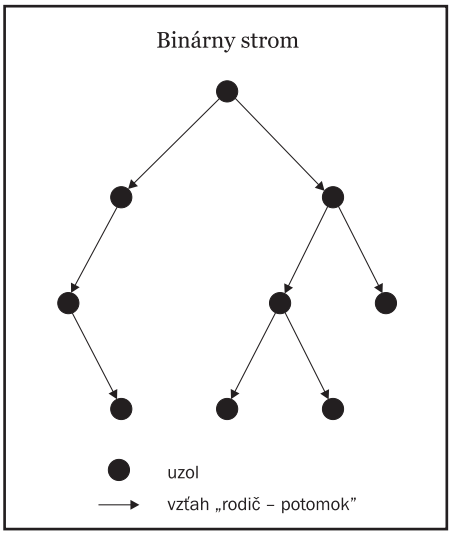
```
struct Reg
{
    unsigned in : 3;
    unsigned out : 3;
    unsigned : 1;
    unsigned ok : 1;
}
```

Štruktúra `Reg` predstavuje fiktívny register, ktorého tri bity slúžia na čítanie údajov (člen `in`), tri bity na zápis údajov (člen `out`) a jeden bit pre pomyselný príznak chyby (člen `ok`). Zaujímavosťou v deklarácii `Reg` je nepomenované bitové pole so šírkou jedného bitu. Takéto pole nie je nijako prístupné a slúži len ako „vypchávka“. Prístup k jednotlivým členom sa realizuje pomocou bežných operátorov (`. a ->`). Ak sa snažíme do bitového poľa vložiť väčšiu hodnotu, ako umožňuje jeho šírka, výsledok závisí od implementácie – obyčajne sa hodnota oreže na šírku bitového poľa.

Treba poznamenať, že aj presný spôsob reprezentácie bitových polí v pamäti je implementačne závislý – nevieme vopred povedať, či napr. pole `in` bude zaberať tri najnižšie alebo tri najvyššie bity z celého priestoru, ktorý obsadzuje štruktúra `Reg`, a nevieme dokonca ani povedať, či bude štruktúra `Reg` v pamäti zaberať ten jeden bajt, ktorý by jej bohato stačil, alebo či bude vzhľadom na požiadavky zarovnávania objektov zaberať viacej bajtov (napríklad štyri).

## Binárny strom

Pre názornejšiu ukážku práce so štruktúrami si uvedieme krátky príklad. V ňom budeme používať abstraktný údajový typ „binárny strom“, ktorý je definovaný veľmi jednoducho: 1. uzol bez potomkov je binárnym stromom, 2. uzol, ktorého ľavým a/alebo pravým potomkom je binárny strom, je aj sám osebe binárnym stromom (kde uzol je nejaká pomyselná entita, predstavujúca „stavebnú jednotku“ stromov). Príklad binárneho stromu je na obr. 1.




*Obr. 1 Binárny strom*

Táto pomerne neformálna definícia s rekurzívnym nádychom nás vedie k nasledujúcej predstave: Reprezentujme uzol stromu ako štruktúru, ktorá bude popri nejakom užitočnom informačnom obsahu zahŕňať dva ukazovatele na ľavého a pravého potomka. Oba ukazovatele budú, samozrejme, ukazovať opäť na ďalšie uzly, ktoré môžu prípadne ukazovať zase na ďalšie uzly, a tak stále ďalej a ďalej do hĺbky stromu. Dohodneme sa takisto, že ak daný potomok neexistuje, príslušný ukazovateľ bude mať hodnotu `0`. Celý strom bude reprezentovaný jedným, tzv. koreňovým uzlom, z ktorého budú viesť ukazovatele na nižšie a nižšie úrovne – až k jednotlivým listom stromu.

Napíšeme teda deklaráciu štruktúry `Bnode`, predstavujúcej uzol stromu:

```
struct Bnode
{
  char* str;
```

```
  int count;
  Bnode* left, right;
};
```

Povedzme si ešte, na čo binárny strom využijeme. Jednou z pekných a názorných aplikácií je jeho použitie ako prostriedku na počítanie výskytu slov v texte. Ako informačný obsah budeme uchovávať v každom uzle jedno slovo (ako textový reťazec) a jeho počet výskytov v prehľadávanom texte. Algoritmus takéhoto počítania slov je v skratke nasledujúci:

1. Načítaj slovo.

2. Ak je koreňový strom prázdny, vytvor nový uzol, nastav počet výskytov slova na 1 a choď na krok 1.

3. Ak je slovo zhodné so slovom z koreňa stromu, inkrementuj počet výskytov a choď na krok 1.

4. Ak je slovo menšie ako to z koreňa stromu, pokračuj od kroku 2 s ľavým podstromom.

5. Ak je slovo väčšie ako to z koreňa stromu, pokračuj od kroku 2 s pravým podstromom.

V krokoch 4 a 5 jednoducho zoberieme ľavý alebo pravý podstrom a ďalej s ním rekurzívne pracujeme ako s pôvodným stromom. S každým novým načítaným slovom teda traverzujeme stromom od koreňa smerom nadol, až kým ho nenájdeme v niektorom uzle alebo kým neurčíme miesto, kam slovo vložíme v prípade, že ho ešte v strome nemáme. Algoritmus zaručí, že z pohľadu každého uzla budú v jeho ľavom podstrome všetky slová menšie a v pravom podstrome všetky slová väčšie ako slovo v tomto uzle. (Čo sa týka pojmov väčší a menší, ide o lexikografické porovnanie znak po znaku.) Pre každé slovo teda bude existovať *jednoznačná* cesta, ktorá nás dovedie buďd k uzlu s týmto slovom, alebo k miestu, kam slovo patrí. Je jasné, že táto cesta bude závisieť od doterajšieho spektra načítavaných slov a pre rôzne texty bude strom vyzerať rôzne. V krajnom prípade dokonca môžeme dostať tzv. degenerovaný strom, v ktorom každý uzol má napríklad len pravých potomkov – táto situácia nastane vtedy, keď budú slová v načítavanom texte zoradené podľa abecedy.

Nasleduje niekoľko funkcií, ktoré realizujú algoritmus:

```
int get_word(char*);
Bnode* create_node(char*);
void add_word(Bnode*, char*);

void do_count()
{
  char wrd_buf[80];

  get_word(wrd_buf);
  Bnode* tree = create_node(wrd_buf);

  while (get_word(wrd_buf))
    add_word(tree, wrd_buf);

  // spracovanie výsledkov
}

void add_word(Bnode* root, char* word)
{
  if (!strcmp(word, root->str))
    root->count++;
  else if (strcmp(word, root->str) < 0)
    if (root->left)
      add_word(root->left, word);
    else
      root->left = create_node(word);
  else /* (strcmp(word, root->str) > 0) */
    if (root->right)
      add_word(root->right, word);
    else
      root->right = create_node(word);
}

Bnode* create_node(char* word)
{
  Bnode* node = new Bnode();
  node->str = new char[strlen(word) + 1];
  strcpy(node->str, word);
  node->count = 1;
  node->left = node->right = 0;
  return node;
}
```

Funkcia `do_count()` je hlavnou funkciou celého algoritmu. Obsahuje premennú `wrd_buf`, predstavujúcu pomocný buffer pre načítavané slová. Jeho dĺžka je obmedzená na 80 znakov; v prípade, že by sme chceli algoritmus spraviť dostatočne robustným a schopným počítať aj dlhšie slová, museli by sme pre každé načítané slovo vytvoriť miesto v pamäti dynamicky, čím by sa celý príklad trochu skomplikoval. Prípadné úpravy týmto smerom ponechávam preto na vás. V podstate stačí zabezpečiť, aby sa alokované miesto vo vhodnom okamihu uvoľnilo pomocou operátora `delete` – dá sa, samozrejme, využiť fakt, že pri vytváraní nového uzla potrebujeme nové slovo nejako uchovať, na čo môžeme použiť preň alokované miesto, ktoré v takom prípade uvoľňovať nebudeme.

Samotné načítanie nového slova má na starosti funkcia `get_word()`. Jej implementáciu v príklade schválne nenájdete, pretože bude závisieť od toho, z akého zdroja slová čítame – zo súboru, z konzoly a pod. Takže opäť miesto pre vašu realizáciu. Uvedený máte len prototyp funkcie, ktorá ako argument dostane ukazovateľ na miesto v pamäti, kam sa má nové slovo skopírovať (predpokladá sa, že je tam dostatok voľného priestoru). Funkcia musí vracať nenulovú hodnotu, kým je čo čítať, a nulovú, keď dôjdeme na koniec spracúvaného textu. Jeden z možných spôsobov realizácie jednoducho načítava znaky a kopíruje ich do vyhradeného buffera, až kým nedospeje k znaku, ktorý nie je súčasťou slova – typicky biele znaky, rôzne interpunkčné znamienka a iné.

Skôr než funkcia `do_count()` začne v cykle načítavať jednotlivé slová, pripraví si celý strom vytvorením jeho koreňového uzla na základe prvého načítaného slova. Strom je reprezentovaný premennou `tree` typu ukazovateľ na štruktúru uzla `Bnode`. Vytvorenie nového uzla má na starosti funkcia `create_node()`, ktorá alokuje novú inštanciu štruktúry `Bnode` a inicializuje jej polia. Pole `str`, v ktorom má byť uložený ukazovateľ na príslušné slovo, sa inicializuje priradením ukazovateľa na novo alokované miesto v pamäti, získaného pomocou operátora `new`. Toto miesto musí byť, vzhľadom na ukončovaciu nulu o jeden znak dlhšie ako samotné slovo. V príklade sa pre jednoduchosť netestuje, či prvé volanie `get_word()` nevrátilo náhodou nulu ani či bolo vytvorenie uzla úspešné (to vlastne ani funkcia `create_node()` nijako neindikuje).

Jadrom funkcie `do_count()` je cyklus `while`, v ktorom sa v každej iterácii načíta jedno nové slovo a zaznačí sa jeho výskyt do stromu. Toto zaznačenie realizuje funkcia `add_word()` s dvoma argumentmi. Prvým z nich je ukazovateľ na koreň stromu, do ktorého slovo pridávame, druhým je ukazovateľ na pridávané slovo. Ak sa toto slovo zhoduje so slovom uloženým v koreňovom uzle, nerobíme nič iné, len inkrementujeme príslušnú premennú `count`. V opačnom prípade testujeme, ktorým podstromom sa budeme ďalej zaoberať, pomocou štandardnej funkcie `strcmp()`. Vieme, že jej návratová hodnota je v prípade nezhody porovnávaných reťazcov kladná alebo záporná podľa toho, ktorý z oboch reťazcov je väčší. Po zistení, ktorým smerom sa máme vydať, najprv testujeme, či príslušný podstrom vôbec existuje (porovnaním príslušného ukazovateľa s nulou). Ak nie, jednoducho vytvoríme nový uzol pomocou funkcie `create_node()`, v opačnom prípade zavoláme funkciu `add_word()` rekurzívne s príslušným podstromom, čím sa celá procedúra zopakuje odznova, tentoraz však na nižšej úrovni.

Príklad nie je kompletným programom, chýba mu jednak spomenuté načítavanie slov, jednak nejaké spracovanie získaných výsledkov, t. j. výpis výskytu jednotlivých slov. Takýto výpis môžeme realizovať jednoduchým rekurzívnym prechádzaním celého stromu pomocou tzv. in-order stratégie, keď sa strom spracúva takto: 1. spracujeme ľavý podstrom, 2. spracujeme koreňový uzol (vypíšeme slovo a jeho počet výskytov) a 3. spracujeme pravý podstrom. Kroky 1 a 3 robíme, samozrejme, iba v prípade, že príslušné podstromy existujú. Spracovanie každého podstromu robíme rovnakým algoritmom (tu je práve rekurzia). Výhodou in-order stratégie je fakt, že jednotlivé vypísané slová budú zoradené podľa abecedy. Okrem nej existujú ešte dve používané stratégie spracovania binárneho stromu –pre-order stratégia (vymenené kroky 1 a 2) a post-order stratégia (vymenené kroky 2 a 3). Oblasti ich použitia sú však dosť špecifické.

V uvedenom programe je použitý z dôvodu jednoduchosti a názornosti štýl programovania, ktorý za žiadnych okolností nesmiete používať – totiž nikde sa neuvoľňujú objekty alokované pomocou `new`. Takéto upratovanie by pre celý strom znamenalo opätovne jeho rekurzívne traverzovanie a postupné uvoľňovanie jednotlivých alokovaných reťazcov a uzlov smerom zdola nahor. Za domácu úlohu si môžete vyskúšať napísať funkciu, ktorá bude túto dealokáciu realizovať, ale oveľa efektívnejšie je definovať pre štruktúru špeciálnu členskú funkciu, tzv. deštruktor (opak konštruktora), ktorý bude mať na starosti likvidáciu objektu štruktúry. Onedlho si o ňom povieme viac.

## Uniony

Druhým agregovaným typom, podobným štruktúram, ale s mierne odlišnou sémantikou, sú uniony. Úmyselne používam originálny anglický termín union, hoci napríklad v českej literatúre sa bežne stretnete s prekladom únia. Myslím si však, že netreba za každú cenu prekladať výrazy, ktoré si to nevyžadujú. Ešte stále (a vlastne dnes možno viac ako kedysi) je programovanie veľmi úzkou špecializáciou, ktorá má nárok na vlastné odborné výrazy, ktorým programátori bez problémov rozumejú aj bez prekladu. A neraz som už spomínal, že angličtina by mala byť „the second native language for all software and/or hardware engineers".

Deklarácia unionov je veľmi podobná deklarácii štruktúr, s tým rozdielom, že namiesto kľúčového slova `struct` použijeme kľúčové slovo `union`. Aj uniony môžu obsahovať deklaráciu členských funkcií, nesmú však obsahovať virtuálne funkcie ani sa nesmú podieľať na akomkoľvek vzťahu dedičnosti. Takisto uniony nesmú mať statické členy.

Čo sa týka sémantiky, union si môžeme predstaviť ako štruktúru, ktorá obsahuje všetky svoje členy akoby „nad sebou" – to znamená, že všetky jeho členy sa začínajú na rovnakej adrese, a teda všetky zdieľajú tú istú pamäťovú oblasť. Ak zmeníme hodnotu niektorého z členov unionu, zmení sa automaticky aj hodnota všetkých ostatných členov. Ukážme si podobný príklad:

```
union U
{
  int a;
  double b;
  char c;
};

U u;
```

K jednotlivým členom unionu pristupujeme klasickým spôsobom. Nech sme do členskej premennej `a` objektu `u` uložili hodnotu `123` príkazom:

```
u.a = 123;
```

Ak teraz zmeníme hodnotu člena `b` napr. takto:

```
u.b = 4.56;
```

zmení sa aj hodnota členov `a` a `c`, o čom sa môžeme ľahko presvedčiť výpisom obsahu premennej `u.a`.

Je zrejmé, že pri prístupe k ľubovoľnému z členov

sa jednoducho vezme príslušná oblasť pamäte a interpretuje sa ako premenná zodpovedajúceho typu (bežná vlastnosťt C++). To nám umožňuje používať nasledujúci trik (inak veľmi častý – v rôznych obmenách):

```
union WORD
{
  unsigned short w;
  struct {
    unsigned char l;
    unsigned char h;
  } x;
};
```

Vidíme, že deklarovaný union `WORD` obsahuje dva členy – `w` typu `unsigned short`, čo je obyčajne 16-bitové celé číslo bez znamienka, a `x` typu štruktúra, obsahujúce zase dva členy `l` a `h` typu `unsigned char`, čo sú 8-bitové celé čísla bez znamienka. Oba členy `w` a `x` sa prekrývajú, čo nám umožňuje pristupovať k ľubovoľnému 16-bitovému číslu jednak ako k celku, jednak k jeho vyššiemu a nižšiemu bajtu samostatne. S pomocou unionu `WORD` je rozloženie čísla na vyšší a nižší bajt veľmi jednoduché:

```
WORD wrd;
wrd.w = 0xABCD;
printf(„lo = 0x%2X\n“, wrd.x.l);
printf(„hi = 0x%2X\n“, wrd.x.h);
```

Samozrejme, musíme vedieť, ako sú 16-bitové čísla ukladané v pamäti. Na bežných intelovských procesoroch sa bez výnimky používa Little-Endian notácia, teda najprv nižší a potom vyšší bajt. Preto je vo vnorenej štruktúre najprv člen `l` a za ním člen `h`.

Zvláštnym typom unionov sú tzv. anonymné uniony. V ich deklarácii chýba jednak názov unionu, a jednak akékoľvek deklarátory, takže dostávame niečo ako:

```
union
{
  int a;
  double b;
  char c;
};
```

Takéto uniony nedefinujú nový typ, ale predstavujú samostatné, nepomenované objekty. Prístup k nim sa nerealizuje pomocou operátorov `.` či `->`, namiesto toho sú jednotlivé členy anonymných unionov prístupné priamo ako bežné premenné, s tým rozdielom, že zdieľajú spoločnú pamäť. Je očividné, že členy globálnych anonymných unionov sa nachádzajú v rovnakom priestore mien, takže nemôžeme mať v dvoch unionoch dva členy s rovnakým názvom. Ďalej anonymné uniony nesmú obsahovať neverejné členy a nesmú mať členské funkcie. Deklaráciou, ktorá obsahuje aspoň jeden deklarátor, nedeklarujeme anonymný union, nasledujúci kód je teda chybou:

```
union { int a; char* p; } obj;
a = 7;
```

k členom unionu `obj` musíme pristupovať klasickým spôsobom:

```
obj.a = 7;
```

Objekt `obj` je inštanciou normálneho unionu, ktorý však nie je deklarovaný vopred ako samostatný typ, ale jeho je deklarácia spojená s deklaráciou jednej z jeho inštancií.

### Úvod do OOP

Záver tejto časti seriálu by som chcel venovať jemnému a veľmi neformálnemu úvodu do princípov objektovo-orientovaného programovania. Nebude to vyčerpávajúci opis a už vôbec nie učebnica OOP, pretože náš seriál sa nezaoberá otázkou „ako programovať“, ale takmer výhradne otázkou „ako (efektívne) programovať v C++“. Tento úvod pomôže tým z vás, ktorí nemajú OO koncept programovania v malíčku. O problematike OOP existuje veľké množstvo kníh, mne osobne sa veľmi páčila kniha *Základy OOP* od I. Kravala, ktorú vydal Computer Press a ktorú odporúčam do vašej pozornosti. Autor síce vysvetľuje princípy OOP s použitím Visual Basicu, ale tak zrozumiteľne a všeobecne, že nie je problém ich úspešne aplikovať v iných jazykoch, napríklad v C++.

Základnou entitou v OOP je objekt. Môžeme si ho predstaviť ako malé abstraktné niečo, čo má svoj vlastný stav. Jednotlivé zložky stavu nazývame atribúty objektu. Dovnútra objektu nevidíme – nevieme, čo ho tvorí ani ako je jeho stav zakódovaný. Toto je jeden zo základných konceptov OOP, hovorí sa mu *zapuzdrenie* (encapsulation). Každý objekt má mať svoj stav skrytý pred okolím, aby nemohlo dôjsť k takým jeho modifikáciám, ktoré by uviedli objekt do nekonzistentného stavu.

Interakcia medzi objektmi sa deje pomocou posielania správ. Každý objekt dáva k dispozícii akýsi zoznam správ, na ktoré dokáže reagovať. Napríklad objekt Zamestnanec by mohol reagovať na správy „Nastav meno“, „Vráť plat“ a podobne. Dôležité je, že objekt, ktorý správu posiela, sa nemusí starať o to, akým spôsobom sa jeho správa spracuje – to má na starosti objekt, ktorý správu dostane a len on jediný pozná svoj stav, môže ho modifikovať, prípadne na zabezpečenie nejakej funkčnosti môže poslať správu iným objektom.

Máme teda predstavu objektu ako entity zapuzdrujúcej nejaký informačný obsah a exportujúcej kolekciu správ, na ktoré je schopná reagovať. Druhým veľmi dôležitým konceptom OOP je *polymorfizmus*. Táto vlastnosť znamená, že môžeme jednu správu poslať viacerým (rôznym) objektom a každý na ňu zareaguje inak, svojím vlastným spôsobom. Klasickým prípadom môže byť správa „Serializuj sa“. Serializácia objektu je zakódovanie jeho stavu tak, aby sa dal uložiť napríklad do súboru, s tým, že niekedy v budúcnosti sa z tohto uloženého stavu objekt dokáže rekonštruovať. Je zrejmé, že každý objekt sa bude serializovať inak; nás to však nemusí zaujímať, jednoducho pošleme príslušnú správu všetkým objektom a je ďalej len na nich, akým spôsobom ju spracujú.

Zapuzdrenie a polymorfizmus sú naozaj základnými princípmi OOP. Bez nich by celá koncepcia strácala akýkoľvek význam. Obyčajne však máme k dispozícii ešte pojmy trieda a dedičnosť. Triedy si môžeme predstaviť ako akési šablóny pre objekty, resp. ako metaobjekty slúžiace na generovanie nových objektov. V C++ sa pojem triedy stavia výrazne do popredia, pretože neexistuje spôsob, ako deklarovať objekt bez toho, aby sme najprv deklarovali jeho triedu. OOP však triedy nevyžaduje a je možné mať jazyk, ktorý je objektovoorientovaný a predsa koncept tried neobsahuje. Čo sa týka dedičnosti o tej si povieme neskôr spolu s konkrétnymi príkladmi. V skratke ide o možnosť usporiadať objekty, resp. triedy do hierarchie na základe vzťahu generalizácie/špecializácie.

## Sedemnásta časť: OD ČLENSKÝCH FUNKCIÍ K TRIEDAM

V predošlej časti sme prebrali štruktúry a uniony. Povedali sme si o nich, že sú to agregované typy, združujúce pod jednou strechou a jedným názvom množinu (obyčajne nehomogénnu) iných typov. Vieme, že členmi štruktúry môžu byť klasické premenné, ktoré sa takto stávajú členskými premennými. Pristupujeme k nim za pomoci binárneho operátora `.` resp. `->`. O čom sme však bližšie nehovorili, len sme to naznačili, je skutočnosť, že členmi štruktúr môžu byť aj funkcie. V tomto pokračovaní sa preto budeme venovať členským funkciám, ďalej sesterským typom štruktúr – triedam a riadeniu prístupu k členom tried.

### Členské funkcie

Po preštudovaní predchádzajúcej časti už vieme, ako sa štruktúra deklaruje. Po kľúčovom slove `struct` a názve štruktúry uvedieme v krútených zátvorkách postupnosť deklarácií jednotlivých členov. Deklarácie sú, samozrejme, vždy ukončené bodkočiarkou. Bodkočiarku musíme uviesť aj za pravú krútenú zátvorku, ukončujúcu deklaráciu štruktúry. Pozabudnutie na túto maličkosť vedie k veľmi častej chybe pri preklade, ktorá bude obyčajne ohlásená na úplne inom mieste, a my sa nestačíme diviť, čo to ten náš prekladač zase vyvádza.

Členmi štruktúry však nemusia byť iba premenné, môžu nimi byť aj funkcie, tak ako ich poznáme doteraz. Takéto funkcie nazývame členskými funkciami, prípadne aj metódami. V ďalšom texte budeme používať termín členská funkcia (member function) a metódy si ponecháme pre Javu.

Členské funkcie deklarujeme jednoducho, stačí ich definíciu vložiť do deklarácie príslušnej štruktúry:

```
struct Account
{
  double balance;

  void deposit(double amount)
  {
    if (amount > 0)
      balance += amount;
  }
};
```

V tomto jednoduchom príklade máme štruktúru `Account`, reprezentujúcu (veľmi zjednodušene) účet v banke. Jej člen `balance`, ktorý je typu `double`, predstavuje aktuálny zostatok na účte. Na uľahčenie realizácie operácie vkladania na účet sme do štruktúry zabudovali funkciu `deposit()`. Jej argumentom je hodnota, ktorú chceme vložiť na účet. Funkcia `deposit()` skontroluje, či naozaj vkladáme zmysluplnú sumu (hodnota musí byť kladná), a potom o túto sumu zvýši aktuálny stav účtu.

Musíme si ešte ukázať, akým spôsobom členskú funkciu zavoláme. Spôsob je v princípe zhodný s prístupom k údajovým členom štruktúry, teda pomocou operátorov `.` a `->`:

```
Account* myAcc;
myAcc = new Account();
myAcc->balance = 50000.0;
myAcc->deposit(11000.0);
myAcc->deposit(45000.0);
myAcc->deposit(20000.0);
...
delete myAcc;
```

Nový objekt typu `Account` sme vytvorili dynamicky pomocou operátora `new`. Získaný ukazovateľ sme uložili do premennej `myAcc`. „Počiatočný vklad“ sme zatiaľ pre nedostatok iných možností museli realizovali jednoduchým priradením hodnoty 50 000 do členskej premennej `balance`. No a potom sme už jednotlivé vklady realizovali výhradne volaním funkcie `deposit()`. Z nášho príkladu nie je zatiaľ jasné, načo zavádzať členské funkcie, keď môžeme operovať priamo s členom `balance`. Tento člen, reprezentujúci aktuálny zostatok na účte, však môžeme považovať za súčasť vnútorného stavu objektu typu `Account` a z predošlej časti vieme, že v zmysle zapuzdrenia tento vnútorný stav objektu nesmie byť prístupný zvonka. O chvíľu si ukážeme, ako prostriedkami

jazyka C++ dosiahnuť, aby člen `balance` zvonka naozaj nebol prístupný a aby jediným spôsobom zmeny stavu objektu bolo volanie jeho členských funkcií (ekvivalentné posielaniu správ objektu, o ktorom sme hovorili minule).

Členské funkcie štruktúr sú niečím špecifické. Predstavme si situáciu, keď máme dva rôzne účty, a teda dva rôzne objekty typu `Account` a na každý z nich chceme vložiť inú sumu:

```
Account a1, a2;
a1.deposit(1000.0);
a2.deposit(2200.0);
```

Keď sa pozrieme na deklaráciu funkcie `deposit()`, vidíme, že v jej tele odkazujeme jednak na formálny argument `amount` (ktorý sa inicializuje jedinečným spôsobom pri volaní funkcie), a jednak na údajový člen `balance` našej štruktúry `Account`. Ale my vyvolávame funkciu `deposit()` nad dvoma rôznymi objektmi. Ako teda táto funkcia vie, s ktorým objektom pracuje (a na ktorý účet má vlastne vložiť peniaze)? Logicky môžeme predpokladať, že funkcia bude pracovať vždy s tým objektom, pomocou ktorého bola zavolaná. To však znamená, že pri jej volaní musíme nejakým spôsobom odovzdať odkaz na tento objekt. A presne toto sa deje aj v skutočnosti – každá členská funkcia dostáva ako jeden z argumentov ukazovateľ na ten objekt, nad ktorým bola zavolaná. Tento argument je na pohľad skrytý a nijako sa nedeklaruje, v tele funkcie je však vždy dostupný pomocou kľúčového slova `this`. Každý odkaz na ľubovoľný údajový člen štruktúry sa v tele členskej funkcie realizuje prostredníctvom ukazovateľa `this`. Vďaka tomu, že členská funkcia je členom príslušnej štruktúry a teda má prístup k údajovým členom štruktúry priamo, je jeho písanie nepovinné. Zvyčajne ukazovateľ `this` používame len na sprehľadnenie zápisu kódu, prípadne na odlíšenie rovnako pomenovanej členskej premennej a formálneho argumentu funkcie (nič nezvyčajné). Našu funkciu `deposit()` by sme teda mohli rovnako dobre zapísať aj takto:

```
void deposit(double amount)
{
  if (amount > 0)
    this->balance += amount;
}
```

Je vhodné uvedomiť si, že ukazovateľ `this` je konštantný, pre členské funkcie triedy `X` je jeho typ `X* const`, preto akákoľvek jeho zmena je zakázaná.

Dovolím si teraz malú vložku, v ktorej sa pokúsim opraviť prípadné nesprávne chápanie konceptu štruktúr (ergo tried), objektov, členských premenných a funkcií. Deklaráciou štruktúry (triedy, unionu) zavádzame do programu nový typ. Nedefinujeme tým však nijaký nový objekt a prekladač nepridelíuje nijakú pamäť. Ak ďalej v programe deklarujeme objekt typu tejto štruktúry, prípadne ak ho vytvoríme dynamicky pomocou operátora `new`, hovoríme, že vytvárame novú inštanciu štruktúry, resp. triedy (v angličtine je na to výraz „to instantiate“, doslova niečo ako „inštancionalizovať“). Novému objektu sa v pamäti vyhradí miesto, v ktorom budú obsiahnuté všetky údajové členy danej štruktúry, nie však jej členské funkcie (!). Z tohto hľadiska sa členské funkcie správajú ako bežné funkcie C++, iba s tým rozdielom, že pri volaní dostávajú ako argument skrytý ukazovateľ na príslušný objekt a z hľadiska linkera sú prísne typovo označkované, čiže ich naozaj nemožno volať iným spôsobom, iba prostredníctvom objektov danej štruktúry. Všetky inštancie jedného typu teda zdieľajú jeden kód pre každú členskú funkciu, ale každá si uchováva vlastný privátny stav. To nám presne harmonizuje s predstavou, ktorú sme si opísali minule, že každý objekt má svoj vlastný stav a okrem toho exportuje zoznam správ, na ktoré dokáže reagovať. Je zrejmé, že správanie objektov rovnakého typu bude rovnaké, preto je kód členských funkcií zdieľaný. (A mimochodom, je to úplne logické – predstavte si desaťtisíc objektov, z ktorých každý by mal vlastnú kópiu jedného a toho istého kódu – to je, samozrejme, nezmysel).

Už sme spomenuli, že členské funkcie deklarujeme uvedením ich definície v zozname deklarácií členov štruktúry. Nie je to celkom presné, pretože existuje ešte jedna alternatíva. Namiesto definície funkcie vložíme do štruktúry iba jej deklaráciu, t. j. prototyp tejto členskej funkcie a samotné telo uvedieme niekde mimo (samozrejme, až za deklaráciu štruktúry). V takom prípade však musíme meno členskej funkcie, modifikovať, aby bolo prekladaču jasné, ku ktorej štruktúre ktorá funkcia patrí. Úplné, tzv. *kvalifikované* meno členskej funkcie získame predradením mena jej rodičovskej štruktúry spolu so známym operátorom štvorbodky `::`. Deklarácia štruktúry `Account` bude potom vyzerať takto:

```
struct Account
{
 double balance;
  void deposit(double);
};

void Account::deposit(double amount)
{
  if (amount > 0)
    balance += amount;
}
```

Členská funkcia `deposit()` štruktúry `Account` má teda kvalifikované meno `Account::deposit()`. Oba spôsoby deklarácie sú v zásade ekvivalentné okrem jedného významného detailu. Funkcie, definované prvým spôsobom, t. j. priamo v deklarácii štruktúry, sú implicitne `inline`. Čo to znamená, to sme si už hovorili, pre tých, ktorí zabudli – ide o funkčný ekvivalent makra, keď sa volanie funkcie nahradí priamo jej telom. Druhý spôsob nám `inline`-ovosť funkcie nijako nezabezpečuje, a teda pokiaľ ju požadujeme, musíme kľúčové slovo `inline` uviesť explicitne:

```
inline void Amount::deposit(double amount)
{
  if (amount > 0)
    balance += amount;
}
```

(V princípe je uvedenie definície členskej funkcie priamo v deklarácii štruktúry úplne ekvivalentné jej zápisu za touto deklaráciou so špecifikátorom `inline`.)

Je dobrým zvykom členské funkcie s veľmi jednoduchým telom definovať priamo v deklarácii triedy (resp. štruktúry, ale prakticky môžeme bez ujmy na všeobecnosti hovoriť o triedach z dôvodov, ktoré uvidíme o chvíľu). Tým automaticky zabezpečíme, že takéto funkcie budú linkované ako `inline`. Zložitejšie funkcie definujeme samostatne, aby zbytočne nezneprehľadňovali samotnú deklaráciu triedy. V praxi bývajú deklarácie tried uložené v samostatných hlavičkových súboroch (samozrejme, pre každú triedu nemusí nevyhnutne existovať jeden súbor i keď neraz sa to robí práve takto). Každý hlavičkový súbor môže byť pomocou direktívy `#include` vložený do tých zdrojových súborov, v ktorých s príslušnou triedou potrebujeme pracovať. Samotná implementácia členských funkcií tej - ktorej triedy sa však nachádza v samostatných súboroch. Takto môžeme dať iným programátorom na prípadné znovupoužitie našich tried k dispozícii sériu objektových a hlavičkových súborov bez toho, aby sme zverejňovali detaily implementácie týchto tried. A programátor, ktorému nevyhovuje správanie určitej triedy, si jednoducho pomocou mechanizmu dedičnosti odvodí novú triedu s upraveným správaním. Ale o tom až v kapitole venovanej dedičnosti.

## Riadenie prístupu k členom

Pomyselným mostíkom medzi štruktúrami a triedami je kontrola prístupu k jednotlivým členom agregovaných typov. Každý jeden člen štruktúry, triedy či unionu, či je to premenná, alebo funkcia, má pridelený zvláštny atribút, ktorý určuje, kedy a za akých okolností je tento člen viditeľný a prístupný.

V C++ existujú tri úrovne oprávnení k prístupu, charakterizované kľúčovými slovami `private`, `protected` a `public`. Tieto prístupové špecifikátory sa uvádzajú v deklarácii štruktúry v rovnakom tvare ako návestia `case` v príkaze `switch`. Každý špecifikátor určuje prístupovú úroveň pre všetky nasledujúce členy až do výskytu ďalšieho špecifikátora, resp. až po koniec deklarácie štruktúry. Špecifikátory sa môžu v jednej deklarácii použiť aj viackrát. Demonštrujeme si ich použitie na príklade:

```
   struct X
{
public:
  int a;
  double b;

private:
  char c;
  void out();

protected:
  long d;

public:
  char* str;
  void print();
};
```

V príklade deklarujeme štruktúru `X`, ktorá obsahuje premennú `c` a funkciu `out()` typu `private`, ďalej premennú `d` typu `protected` a nakoniec premenné `a`, `b`, `str` a funkciu `print()` typu `public` . Aký je však význam jednotlivých špecifikátorov? To nám čiastočne napovie ich preklad. Členy deklarované ako `private` sa považujú za *súkromné* pre danú štruktúru a sú prístupné iba členským funkciám tejto štruktúry. Iný spôsob prístupu je zakázaný. Majme nasledujúcu štruktúru:

```
struct A
{
private:
  int x;

public:
  void set(int i)
  {
    x = i;
  }
};
```

Zmena hodnoty privátneho člena `x` je prostredníctvom členskej funkcie `set()` povolená – tento člen je v rámci štruktúry prístupný hocijakej členskej funkcii. Ak by sme však chceli zmeniť hodnotu `x` priamo, prekladač ohlási chybu:

```
A a;
a.set(5); // OK
a.x = 5;  // chyba!
```

Privátne môžu byť aj členské funkcie, ktoré v takom prípade nemôžeme volať priamo, ale len z iných členských funkcií.

Členy deklarované ako `public` sú, presne naopak, *verejnými* členmi danej triedy. Aj keď to odporuje princípu zapuzdrenia, je možné takto niektoré členské premenné štruktúry

sprístupniť okoliu priamo. Táto voľnosť však takmer vždy vedie k zamedzeniu dôslednej kontroly a udržiavania konzistencie vnútorného stavu objektov. Ako `public` sa preto prevažne deklarujú členské funkcie. Zoznam všetkých verejných funkcií tej – ktorej štruktúry/triedy predstavuje jej externé programátorské rozhranie, a teda de facto zoznam správ, na ktoré je každá inštancia tejto štruktúry schopná reagovať. Ideálny stav predstavuje taká štruktúra, ktorej všetky údajové členy sú privátne, pretože spolu vytvárajú vnútorný stav objektu, a ktorý exportuje určitú pevne definovanú množinu verejných funkcií predstavujúcich API objektu.

Tretí typ prístupu k členom, reprezentovaný kľúčovým slovom `protected`, nachádza uplatnenie až po zavedení vzťahov dedičnosti medzi triedami. `Protected` údajové členy či funkcie takisto nie sú prístupné zvonka. Prístup k nim však majú povolený okrem členských funkcií danej triedy aj členské funkcie všetkých tried z nej odvodených. Bližšie si o tomto špecifikátore povieme neskôr.

## Triedy verzus štruktúry

Konečne máme dostatok informácií na to, aby sme si objasnili zásadný rozdiel medzi štruktúrami a triedami. Okrem čisto kozmetického faktu, že triedy deklarujeme s kľúčovým slovom `class` namiesto `struct` pri štruktúrach používaného, rozdiel spočíva v implicitnom prístupe k jednotlivým členom oboch typov. Zatiaľ čo štruktúry (a takisto uniony) majú všetky svoje členy implicitne verejné (`public`), čo nám umožnilo v našich príkladoch týkajúcich sa štruktúr špecifikátory prístupu vynechať, členy tried sa implicitne považujú za privátne (`private`). To znamená, že deklarácia typu:

```
class A { int x, y; };
```

je nanič, pretože členy `x` a `y` z tejto triedy nijakým spôsobom „nevydolujeme“.

Teraz, keď vieme, čo sú to triedy a ako sa líšia od dosiaľ preberaných štruktúr, dohodneme sa, že na štruktúry zabudneme. Ich použitie v OO programoch je zbytočné, pretože sa dajú rovnako dobre realizovať pomocou tried. (V skutočnosti sú štruktúry dedičstvom po jazyku C, v ktorom však ich použitie bolo výrazne obmedzené oproti C++ – nebolo napríklad možné ako členy štruktúry uviesť funkcie). Odteraz budeme preto pracovať výhradne s pojmom trieda.

Pre lepšie podchytenie konceptu tried si ukážeme príklad takého malého ideálneho objektu. Bude to trieda `Word`, zapuzdrujúca 16-bitové celé číslo bez znamienka. O užitočnosti tejto triedy by sme mohli polemizovať, ale na výučbové účely nám celkom postačí. Tu je jej deklarácia:

```
typedef unsigned short WORD;

class Word
{
  WORD w;

public:
  void set(WORD val)
  { w = val; }
  WORD get()
  { return w; }
};
```

Vidíme, že trieda `Word` obsahuje jediný dátový člen `w`, ktorý bude predstavovať uchovávanú hodnotu. Tento člen je implicitne privátny, preto sme špecifikátor `private` vynechali. Okrem toho máme k dispozícii dve funkcie – `Word::set()` a `Word::get()`. Ich názvy napovedajú, na čo ich budeme používať. Prvá z nich slúži na nastavenie či zmenu uloženého čísla, keď sa jednoducho priradí členu `w` hodnota argumentu tejto funkcie. Druhá z funkcií, naopak, slúži na zistenie uloženej hodnoty, ktorú vráti ako návratovú hodnotu. Poviete si, načo sme vôbec skrývali člen `w` pred svetom, keď sme vyexportovali dve funkcie, pomocou ktorých si môžeme s obsahom triedy `Word` robiť, čo sa nám zapáči. Ale v tom je práve skryté celé tajomstvo. Môžeme síce pracovať s obsahom triedy `Word` ľubovoľne, ale **len** pomocou funkcií `set()` a `get()`. Ak sa v budúcnosti rozhodneme, že zmeníme vnútornú reprezentáciu triedy `Word`, okolie sa to nedozvie, pretože nezmeníme programátorské rozhranie tejto triedy. Nuž a v tom je vlastne podstata celého OOP. Každý objekt sa stará sám o seba a služby iných objektov využíva len prostredníctvom ich rozhrania. Pokiaľ sa toto rozhranie nezmení, môžeme vnútro objektov meniť podľa ľubovôle, a predsa nebudeme musieť meniť všetky funkcie, ktoré s daným objektom pracujú. V tejto na pohľad triviálnej vete je skrytá obrovská, predovšetkým časová úspora pri vývoji programov.

## Statické členy tried

Rovnako ako nečlenské premenné alebo funkcie aj jednotlivé členy triedy môžu byť deklarované s kľúčovým slovom `static`. Tieto tzv. statické členské premenné, resp. statické členské funkcie, však majú odlišný význam od svojich nečlenských ekvivalentov. Zopakujme si pre osvieženie pamäti, čo spôsobí doplnenie špecifikátora `static` k deklarácii (nečlenského) objektu. Globálna statická premenná či funkcia (so súborovým rozsahom platnosti) je neviditeľná mimo zdrojového súboru, v ktorom je deklarovaná. V podstate ide o taký nie príliš dokonalý a neobjektový ekvivalent špecifikátora `private`, ibaže nie v rámci triedy, ale v rámci súboru (t. j. modulu). Lokálna statická premenná zase existuje aj vtedy, keď sa program práve nenachádza v bloku, v ktorom je deklarovaná.

Deklarácia statického člena triedy naproti tomu „odoberá“ vlastníctvo tohto člena inštancii triedy a prenáša ho na samotnú triedu. To znamená, že statická členská premenná nebude súčasťou vnútorného stavu každej inštancie, nebude existovať jej samostatná kópia pre každú vytvorenú inštanciu, ale bude existovať jediná kópia, ktorá bude de facto súčasťou samotnej triedy. Okrem iného to znamená, že na prístup k statickej premennej nepotrebujeme existujúcu inštanciu triedy. Ukážeme si príklad. Predstavme si, že chceme implementovať zariadenie známe z rôznych bánk, ktoré každému návštevníkovi vytlačí lístok s jeho poradovým číslom. Od zariadenia požadujeme, aby bolo poradové číslo pre každý lístok jedinečné a o jednotku väčšie ako číslo na predchádzajúcom lístku. Nech je každý lístok inštanciou triedy `Ticket`. Keďže nám zatiaľ chýbajú potrebné znalosti o špeciálnej členskej funkcii zodpovednej za inicializáciu objektu po jeho vytvorení (tzv. konštruktor), implementujeme len jednu členskú funkciu `nextNumber()`, ktorá bude vracať číslo nového lístka. Aby sme splnili uvedené požiadavky, musíme si pamätať číslo, ktoré bolo vygenerované naposledy. Na tento účel do triedy zavedieme statickú členskú premennú `lastNumber`. Funkcia `nextNumber()` pri každom volaní túto premennú inkrementuje a jej obsah vráti. Tu je deklarácia triedy:

```
class Ticket
{
  static int lastNumber;

  // ...
  // iné premenné
  // ...

public:
  int nextNumber()
  {
    return ++lastNumber;
  }

  // ...
  // iné funkcie
  // ...
};
```

Oproti nestatickým údajovým členom sa statické premenné vyznačujú ešte jednou zvláštnosťou. Nestačí ich totiž len deklarovať v rámci triedy, musíme ich navyše *povinne* definovať mimo triedy, akoby to boli bežné globálne premenné. Pri tejto definícii, rovnako ako pri definícii členských funkcií mimo triedy treba použiť úplné, kvalifikované meno danej premennej. Našu deklaráciu teda musíme doplniť ešte riadkom:

```
int Ticket::lastNumber;
```

Tento riadok nesmie byť súčasťou deklarácie triedy, musí sa nachádzať mimo nej, na úrovni súboru.

Zostáva už len otázka, ako premennú `lastNumber` inicializovať. Na tomto mieste si musíme uviesť fakt, ktorý som dosiaľ nijako nezdôrazňoval – deklarácie členských premenných tried **nesmú** (na rozdiel napríklad od Javy) za žiadnych okolností obsahovať inicializátory! Nasledujúca deklarácia je preto chybná:

```
class C
{ int a = 123; };
```

Členské premenné sa inicializujú výhradne pomocou špeciálnej členskej funkcie (už spomínaného konštruktora). Čo však so statickými premennými? Tie nie sú súčasťou vytváraných objektov, existujú dávno pred vytvorením prvej inštancie. Našťastie ich je možné (dokonca nutné) inicializovať jednoducho doplnením inicializátora do ich samostatne uvedenej definície. Premenná `lastNumber` z našej triedy `Ticket` by mohla byť inicializovaná napríklad takto:

```
int Ticket::lastNumber = 0;
```

Ako sme už uviedli, statické údajové členy existujú nezávisle od inštancií danej triedy, a teda je možné k nim pristupovať priamo, bez použitia operátorov `.` či `->`. Napríklad majme triedu:

```
class A
{ public: static int a; };

int A::a;
```

Člen `a` je verejný, a teda prístupný aj mimo triedy. Pracovať s ním môžeme jednoducho – použitím jeho plne kvalifikovaného mena `A::a`:

```
A::a = 10;
```

K statickým členom, samozrejme, môžeme pristupovať aj prostredníctvom jednotlivých inštancií triedy `A`, no v takom prípade sa ľavý operand operátora `.` či `->` nevyhodnocuje:

```
A a1, *pa = &a1;

a1.a = 111;
pa->a = 222;
(++pa)->a = 333;
```

Vo všetkých troch prípadoch sa pracuje s jedným a tým istým členom `a`, navyše sa v treťom prípade hodnota ukazovateľa `pa` nijako nemení.

Statické môžu byť aj členské funkcie. Od nestatických sa líšia veľmi podstatne v tom, že nedostávajú pri volaní ako jeden z argumentov ukazovateľ `this`. Každé použitie tohto kľúčového slova v rámci statickej členskej funkcie je chybou. Keď sa nad tým zamyslíme, je to nanajvýš logické, pretože volanie statickej funkcie sa nevzťahuje na žiadnu inštanciu. Podobne ako k statickej premennej možno k statickej funkcii pristupovať (teda

ju volať) priamo, s použitím jej kvalifikovaného mena a takisto ju možno volať aj v okamihu, keď žiadna inštancia príslušnej triedy neexistuje. Je, samozrejme, dovolené volať statickú funkciu aj klasicky – prostredníctvom operátorov . alebo ->, ale ani v tomto prípade sa ľavý operand oboch operátorov nevyhodnocuje. Statické funkcie smú používať len statické premenné danej triedy (resp. všetko, čo netreba sprístupňovať pomocou ukazovateľa `this`).

Ako príklad statickej členskej funkcie si paradoxne uvedieme našu funkciu `nextNumber()`. Keď ste sa nad triedou `Ticket` poriadne zamysleli, isto ste dospeli k názoru, že funkcia ako `nextNumber()` sa nevzťahuje na nijakú konkrétnu inštanciu. Jej služby potrebujeme len v okamihu vytvorenia nového lístka. Len čo je lístok vytlačený, už by sme jeho číslo nemali meniť, preto každé ďalšie volanie `nextNumber()` stráca zmysel. Z tohto dôvodu je vcelku logické urobiť funkciu `nextNumber()` statickou a nezávislou od jednotlivých inštancií. Zmeníme preto triedu `Ticket` takto:

```
class Ticket
{
  static int lastNumber;

  // ...
  // iné premenné
  // ...

public:
  static int nextNumber()
  {
    return ++lastNumber;
  }

  // ...
  // iné funkcie
  // ...
};
```

Odteraz môžeme funkciu `nextNumber()` volať priamo (a triedu `Ticket` zneužiť na generovanie postupnosti celých čísel):

```
int n = Ticket::nextNumber();
```

Statická funkcia nemusí byť nevyhnutne definovaná mimo triedy, jej telo sa môže nachádzať aj priamo v deklarácii triedy.

## Vnorené a lokálne triedy

Triedu môžeme deklarovať aj vnútri inej triedy. Takáto trieda sa nazýva *vnorená*. Rozsah platnosti vnorenej triedy je trieda, ktorá jej deklaráciu obsahuje. Vnorená trieda môže zo svojej „nadradenej“ triedy (nie nadradenej v rámci dedičnosti!) priamo používať len prípadné statické členy, typy a enumerátory, všetky ostatné členy musí sprístupňovať bežným spôsobom cez existujúce inštancie. Pozrime sa na príklad:

```
class outer
{
public:
  int x;
  static int s;

  class inner
  {
    static int t;
    void f(int i, outer* p);
  };
};
```

V rámci funkcie `f` triedy `inner` nemáme prístup k členu `x` triedy `outer`, priradenie

```
x = i;
```

je preto chybou. Naproti tomu k členu s prístup máme, preto priradenie

```
s = i;
```

je povolené. Ak však máme k dispozícii platnú inštanciu triedy `outer`, napríklad prostredníctvom ukazovateľa, môžeme bez problémov pristupovať aj k jej členu `x`:

```
p->x = i;
```

Názov triedy `inner` je sám osebe súčasťou triedneho rozsahu platnosti triedy `outer`, preto deklarácia

```
inner *ptr = new inner();
```

je chybná, musíme použiť plne kvalifikované meno:

```
outer::inner *ptr = new outer::inner();
```

Statické členy a členské funkcie triedy `inner` môžeme definovať na globálnej úrovni, ale takisto len s použitím kvalifikovaných mien:

```
int outer::inner::t = 0;
void outer::inner::f(int i, outer* p)
{ ... }
```

Medzi triedami `outer` a `inner` neexistuje nijaký iný vzájomný vzťah, čo sa týka možnosti prístupu k členom – trieda `outer` nemá prístup k privátnym členom triedy `inner` a trieda `inner` takisto nemá prístup k privátnym členom triedy `outer`.

Typy definované pomocou mechanizmu `typedef` vo vnorených triedach sú takisto prístupné len pomocou kvalifikovaných mien:

```
class A
{
  class B
  {
  public:
    typedef int I;
    // ...
  };
  // ...
};

I i1;      // chyba
B::I i2;   // chyba
A::B::I i; // OK
```

Trieda deklarovaná v rámci definície nejakej funkcie sa nazýva *lokálna*. Jej názov je lokálny pre danú funkciu a táto trieda sama nesmie používať lokálne premenné danej funkcie. Všetky členské funkcie lokálnej triedy musia byť definované priamo v jej deklarácii. Lokálna trieda nesmie obsahovať statické údajové členy. Funkcia obsahujúca deklaráciu lokálnej triedy nemá špeciálny prístup k členom tejto triedy (t. j. nemá prístup k neverejným členom triedy).

## „Priateľské“ deklarácie

Povedali sme si, že v C++ existujú tri úrovne prístupu k členom tried: `private`, `protected` a `public`. Často sa však dostaneme do situácie, keď by sme potrebovali riadiť povolenie prístupu k neverejným členom v závislosti od funkcie, ktorá ho požaduje. Na tieto účely máme k dispozícii kľúčové slovo `friend`. Ide o funkčný špecifikátor, ktorý povoľuje nečlenskej funkcii prístup k `private` a `protected` členom nejakej triedy. Funkcia sa však nestáva členom tejto triedy ani nie je volaná prostredníctvom jej inštancií. Takisto sa na ňu nevzťahujú špecifikátory prístupu, uvedené v deklarácii triedy.

Nasledujúci príklad názorne ukazuje rozdiel medzi prístupom k privátnemu členu prostredníctvom členskej funkcie a prostredníctvom spriatelenej funkcie:

```
class A
{
  int x;
  friend void friendSet(A*, int);

public:
  void memberSet(int)
  {
    x = i;
  }
};

void friend_set(A* pa, int i)
{
  pa->x = i;
}
```

V príklade máme dve funkcie slúžiace na nastavenie hodnoty člena `x` triedy `A`. Jedna z nich, `memberSet()`, je členská a má, samozrejme, prístup k privátnemu členu `x`. Druhá z nich, `friendSet()`, síce nie je členská, ale zato je deklarovaná ako spriatelená (`friend`), a preto tiež môže pristupovať k členu `x`. Obe nasledujúce volania sú preto správne:

```
A a;
friend_set(&a, 10);
a.member_set(10);
```

Ako spriatelenú funkciu môžeme uviesť aj členskú funkciu inej triedy (použitím kvalifikovaného mena) a takisto môžeme určiť za spriatelenú celú cudziu triedu. V takom prípade budú mať všetky členské funkcie tejto triedy prístup k neverejným členom triedy, ktorá priateľstvo udelila:

```
class X
{
  void f();
  // ...
};

class Y
{
  friend void X::f();
  // ...
};

class Z
{
  friend class X;
};
```

Spriatelené funkcie sú obyčajne definované mimo triedy, ktorá im priateľstvo udelila, ale v princípe je možné ich definíciu uviesť aj priamo v deklarácii tejto triedy. V takom prípade sú tieto funkcie implicitne `inline`. Priateľstvo nie je ani dedičné, ani tranzitívne, t. j. ak je trieda `B` priateľom triedy `A` a trieda `C` priateľom triedy `B`, neznamená to, že trieda `C` musí byť automaticky priateľom triedy `A`.

Priateľstvo môže byť deklarované aj symetricky, v takom prípade však musíme použiť tzv. predbežnú deklaráciu triedy:

```
class Y;

class X
{
  void f();
  friend void Y::g();
  // ...
};

class Y
{
  void g();
  friend void X::f();
  // ...
};
```

Na to, aby sme mohli funkciu `Y::g()` deklarovať ako spriatelenú pre triedu `X`, ktorej deklarácia sa nachádza *pred* deklaráciou triedy `Y`, musíme prekladaču oznámiť, že bude niekedy v budúcnosti deklarovaná nejaká trieda `Y`, inak dostaneme len chybové hlásenie.

### Konštantné členské funkcie

Na záver tejto časti si ešte povieme o aplikácii špecifikátorov `const` a `volatile` na členské funkcie. Predstavme si, že máme inštanciu nejakej triedy, deklarovanú ako konštantnú (t. j. so špecifikátorom `const`). Ak prostredníctvom tejto inštancie zavoláme niektorú členskú funkciu, odovzdá sa tejto funkcii ukazovateľ `this` na danú inštanciu. Prostredníctvom ukazovateľa by však funkcia mohla meniť obsah inštancie bez ohľadu na to, či bola inštancia deklarovaná ako konštantná, alebo nie! Nekonštantná členská funkcia triedy `X` očakáva ukazovateľ `this` typu `X* const` (čo je, ešte raz zdôrazňujem, konštantný ukazovateľ, nie ukazovateľ na konštantu). Ukazovateľ na našu inštanciu je však typu `const X* const`, a preto nám prekladač takéto volanie nepovolí. Z tohto dôvodu máme možnosť špecifikovať ľubovoľnú členskú funkciu ako konštantnú pridaním kľúčového slova `const` *za* jej deklarátor, ale ešte *pred* jej telo. Takáto členská funkcia, samozrejme, nemá povolené meniť údajové členy inštancie, nad ktorou bola zavolaná, môžeme ju však volať aj pre konštantné objekty. Najlepšie bude vysvetliť celú situáciu na príklade:

```
class A
{
  int x;

public:
  void set(int i)
  {
    x = i;
  }
  int get() const
  {
    return x;
  }
};
```

Trieda `A` obsahuje jedinú celočíselnú premennú `x`. Na zmenu hodnoty tejto premennej slúži funkcia `set()`, na jej čítanie funkcia `get()`. Keďže `get()` nemení obsah `x` a teoreticky by mohla byť volaná aj pre konštantné objekty, deklarujeme ju ako konštantnú. Naproti tomu `set()` mení obsah `x`, preto ju pre konštantné objekty nemôžeme volať:

```
A a1;
const A a2;
a1.set(5);                    // OK
int i1 = a1.get(); // OK
a2.set(8);          // chyba
int i2 = a2.get(); // OK
```

Konštantnú funkciu môžeme volať pre konštantné i nekonštantné objekty, nekonštantnú iba pre nekonštantné. Konštantnosť je súčasťou typu funkcie, preto je možné mať dve členské funkcie s rovnakým názvom, počtom a typmi argumentov, z ktorých jedna bude konštantná a jedna nie. Typický príklad (aj keď silne oklieštený):

```
class String
{
  char* str;

public:
  char* get()
  {
    return str;
  }
  const char* get() const
  {
    return str;
  }
};
```

Obe funkcie `get()` sa inak líšia ešte aj v návratovej hodnote, to však za normálnych okolností nestačí na rozlíšenie dvoch funkčných synoným. Ako vidno z príkladu, telá oboch funkcií sú zhodné, líšia sa len tým, že jedna vracia nekonštantný a druhá konštantný ukazovateľ na zapuzdrený reťazec.

Rovnaké pravidlá ako pre špecifikátor `const` platia aj pre špecifikátor `volatile`. Typ ukazovateľa `this` sa, samozrejme, mení na `volatile X* const`. Okrem toho je možné deklarovať členskú funkciu ako konštantnú a `volatile` zároveň uvedením oboch špecifikátorov, typ ukazovateľa `this` bude potom `const volatile X* const`.

## Osemnásta časť: ŠPECIÁLNE ČLENSKÉ FUNKCIE

Už v predchádzajúcich častiach sme sa zmieňovali o dvoch členských funkciách so zvláštnymi menami a špecifickým postavením. Tušíte správne, ide o konštruktor a deštruktor. Teraz si o nich povieme viac.

### Konštruktor

*Konštruktor* je členská funkcia, ktorá sa poväčšine stará o inicializáciu údajových členov inštancie triedy. Vieme, že na rozdiel od nečlenských deklarácií nemôžeme inicializovať členy tried priamo v deklarácii. A priradovať zakaždým po vytvorení nového objektu explicitne jednotlivým členom východiskové hodnoty pomocou operátora priradenia, to by nás asi veľmi rýchlo omrzelo. A to už vôbec nehovoríme o prípadoch, keď by takáto „inicializácia“ nebola možná pre neprístupnosť privátnych údajových členov.

Z tohto dôvodu existuje v C++ možnosť doplniť do deklarácie triedy funkciu, ktorá požadovanú inicializáciu zrealizuje. Ako, to už je na nás. Konštruktor nikdy nevoláme explicitne, jeho vyvolanie má na starosti prekladač a dochádza k nemu vtedy, keď vzniká nový objekt danej triedy. Automatické objekty sú konštruované vždy – keď program vstúpi do bloku, v ktorom sú deklarované (napr. funkcia), statické objekty sa konštruujú len raz, pred spustením programu (globálne), resp. pri prvom prechode deklaráciou (lokálne). Objekty vytvárané dynamicky, pomocou operátora `new`, sú konštruované v rámci tohto operátora (samozrejme, až *po* pridelení pamäte).

Konštruktor sa od ostatných členských funkcií odlišuje v princípe dvoma spôsobmi: predovšetkým musí mať rovnaké meno ako trieda, ktorej je členom, a okrem toho nesmie mať deklarovanú návratovú hodnotu (ani ako typ `void`). Vďaka tomu nie je možné v tele konštruktora použiť príkaz `return` s argumentom. Čo sa týka zoznamu argumentov konštruktora, ten závisí len a len od našich potrieb. Pochopiteľne, v jednej triede môžeme mať viacero prekrytých konštruktorov, každý s iným zoznamom argumentov. Akým spôsobom zariadime, aby konštruktor dostal nami požadované argumenty, to si ukážeme neskôr na príkladoch.

V tele konštruktora môžeme bez problémov volať ostatné členské funkcie. Konštruktor nesmieme deklarovať s kľúčovým slovom `const` či `volatile` – kompilátor pri jeho volaní neberie ohľad na `const/volatile` deklaráciu inštancií. Konštruktor nesmie byť statický ani virtuálny (vysvetlíme neskôr).

Dva typy konštruktorov sa používajú častejšie ako ostatné, preto majú samostatné názvy. Prvým je *implicitný konštruktor* (default constructor), ktorý možno zavolať bez argumentov (to dosiahneme buď prázdnym zoznamom argumentov, alebo deklaráciou všetkých argumentov konštruktora ako implicitných). Takýto konštruktor sa uplatní pri deklarácii explicitne neinicializovaných objektov. Spomeňme si, že ak sme pri deklarácii premennej nejakého jednoduchého typu, napríklad `int`, vynechali inicializátor, vytvorená premenná obsahovala nejaký náhodný obsah, závislý od stavu pamäťového miesta, kde vznikla. Ale v prípade objektov, pri ktorých chceme dodržať nejakú vnútornú podmienku konzistencie, zrejme nebude vhodné ponechať jednotlivým údajovým členom náhodné hodnoty. Implicitný konštruktor preto obyčajne priraďuje členom triedy implicitné, vopred dohodnuté hodnoty (napríklad samé nuly).

Druhým zvláštnym typom konštruktora je *kopírovací konštruktor* (copy constructor), ktorý slúži na vytvorenie kópie existujúceho objektu. Preto ho musí byť možné zavolať s jediným argumentom – kopírovaným objektom. Keďže konštruktor triedy `X` nemôže mať argument typu `X`, kopírovací konštruktor musí byť deklarovaný s argumentom typu `X&`, resp. `const X&` (a prípadnými ďalšími implicitnými argumentmi).

Oba konštruktory, implicitný aj kopírovací, sú špecifické aj tým, že v prípade ich neprítomnosti si ich dokáže kompilátor vygenerovať sám. Implicitný konštruktor sa generuje len v prípade, že nie je definovaný žiaden iný konštruktor. V takom prípade treba minimálne umožniť deklarovať inštanciu triedy bez inicializácie. Takto vygenerovaný implicitný konštruktor je verejný (`public`) a dokopy nič nerobí, iba prípadne volá implicitné konštruktory základných tried a/alebo implicitné konštruktory vnorených objektov. Kopírovací konštruktor zase kompilátor doplní v prípade potreby vytvorenia kópie nejakého objektu, či už pri inicializácii iného objektu, pri použití operátora priradenia, pri odovzdávaní argumentov funkciám alebo pri spracovaní návratovej hodnoty. Takýto implicitne doplnený kopírovací konštruktor vytvorí už minule spomínanú plytkú kópiu objektu, t. j. skopíruje jednotlivé členy bajt po bajte do nového objektu. Treba podotknúť, že prekladačom generované konštruktory v programe existujú len vtedy, keď sú potrebné.

Aby sme sa však nepohybovali stále len v teoretickej rovine, ukážeme si teraz malý príklad. Deklarujeme triedu `complex`, ktorú budeme používať na reprezentáciu komplexných čísel. Trieda bude (zatiaľ) obsahovať dva údajové členy – reálnu zložku `re` a imaginárnu zložku `im`, obe typu `double` – a štyri konštruktory:

```
class complex
{
  double re, im;

public:
  complex()
  { re = im = 0.0; }
  complex(double r)
  { re = r; im = 0.0; }
  complex(double r, double i)
  { re = r; im = i; }
  complex(complex& c)
  { re = c.re; im = c.im; }
};
```

Trieda, samozrejme, nie je úplná, minimálne chýbajú funkcie na sprístupnenie oboch zložiek komplexného čísla. Tie si môžete doplniť sami. Ďalšie užitočné členské funkcie budeme dopĺňať priebežne počas výkladu.

Prvý konštruktor, `complex::complex()`, je spomínaným implicitným konštruktorom. Ako vidíme, nerobí nič iné, iba vynuluje obe zložky, reálnu aj imaginárnu. Takýto konštruktor sa uplatní napríklad pri nasledujúcich deklaráciách:

```
complex c1, c2;
complex* pc1 = new complex();
```

Oba objekty `c1` a `c2`, ako aj objekt, na ktorý ukazuje ukazovateľ `pc1`, budú mať po vytvorení obe zložky nulové. Všimnite si použitie operátora `new`. Vieme už, že jeho operandom je typ, ktorého inštanciu chceme vytvoriť. Do zátvoriek za typ môžeme napísať inicializačnú hodnotu (napríklad `new int(3)` vytvorí novú premennú typu `int` a nastaví ju na hodnotu

3). No v prípade inštancií tried máme trochu širšie možnosti – do zátvoriek zapisujeme argumenty, pomocou ktorých chceme objekt skonštruovať. Na základe ich počtu a typu prekladač vyberie ten správny konštruktor, ktorý následne zavolá. Ak nijaký vhodný nenájde, ohlási chybu. Ak nechceme dodať žiadne argumenty, nemusíme pár zátvoriek vôbec písať (ako pri jednoduchých typoch):

```
complex* pc1 = new complex;
```

Druhý konštruktor, `complex::complex(double`, vhodne slúži na inicializáciu takých komplexných čísel, ktoré majú imaginárnu zložku nulovú. V rámci šetrenia miestom ich potom môžeme inicializovať takto:

```
complex c3(4);
complex* pc2 = new complex(3.14);
```

Objekt `c3` bude mať po inicializácii reálnu zložku rovnú štyrom, imaginárnu konštruktor vynuluje. Podobne objekt `*pc2`. Vidíme, že argument konštruktora v prvom prípade uvedieme do zátvoriek za identifikátor deklarovanej premennej, v druhom prípade za názov vytváraného typu.

Tretí konštruktor, `complex::complex(double, double)`, konečne umožňuje plnohodnotnú inicializáciu reálnej aj imaginárnej zložky celého10 objektu. Argumenty uvádzame podobne ako v predchádzajúcom prípade:

```
complex c4(9.2, 5.7);
complex* pc3 = new complex(4, 2.1);
```

Ak sa dobre zahľadíte na deklaráciu predchádzajúcich troch konštruktorov, zrejme si uvedomíte, že je ich možné spojiť do jedného, ktorý sa bude dokonca tváriť ako implicitný:

```
complex(double r = 0.0, double i = 0.0)
{ re = r; im = i; }
```

Možno ste si až teraz uvedomili, na čo sú dobré implicitné argumenty funkcií.

Posledným konštruktorom v triede `complex` je kopírovací konštruktor `complex::complex(complex& c)`. V jeho tele, samozrejme, máme prístup k privátnym členom argumentu `c`, pretože sa nachádzame stále v tej istej triede. Použitie konštruktora je podobné predchádzajúcim príkladom:

```
complex c5(c3);
complex* pc4 = new complex(c2);
```

Ale navyše môžeme vytvárať kópie objektov triedy `complex` aj takýmto spôsobom:

```
complex c6 = c4;
```

Pravdu povediac, tento spôsob inicializácie je možný pri použití ľubovoľného konštruktora s jedným argumentom, ale celá problematika je trochu komplikovanejšia a budeme sa jej venovať o chvíľu. Podstatné je teraz, že kopírovací konštruktor sa volá ako dôsledok vytvorenia *nového* objektu. Ak by sme z predchádzajúceho riadku kódu vynechali špecifikátor `complex`, dostali by sme priradenie `c6 = c4`, pri ktorom sa uplatňuje priraďovací operátor `=`.

## Dočasné objekty

Počas vykonávania jednotlivých príkazov programu je často výhodné, či dokonca nevyhnutné používať *dočasné objekty*. Takéto objekty nie sú nijako deklarované a existujú obyčajne ako vedľajší účinok vyhodnocovania niektorých výrazov, odovzdávania argumentov funkciám, ukladania návratovej hodnoty a pod. Môžeme ich však vytvoriť aj explicitne, uvedením názvu triedy spolu s argumentmi konštruktora v zátvorkách. Ukážme si príklad:

```
class C
{
public:
  C(int);
  C(C&);
  // ...
};

C f(C);

void g()
{
 C c1(5);
  C c2 = f(c1);
  C c3 = f(C(11));
  c1 = f(c1);
}
```

V príklade máme torzo triedy `C`, ktorá obsahuje jeden klasický konštruktor s jediným argumentom typu `int` a jeden kopírovací konštruktor. Funkcia `f()`, ktorej prototyp sa nachádza v príklade, má jediný argument typu `C` a vracia návratovú hodnotu rovnakého typu.

Prvý objekt `c1` deklarujeme a inicializujeme známym spôsobom, po jeho vytvorení prekladač známym spôsobom zavolá konštruktor `C::C(int)` s argumentom `5`. Druhý objekt `c2` je inicializovaný návratovou hodnotou volania `f(c1)`. Pri odovzdávaní argumentu `c1` funkcii sa obyčajne (je to závislé od implementácie) vytvorí dočasný objekt typu `C`, ktorý bude kópiou `c1`. Tento dočasný objekt sa ako každá kópia inicializuje pomocou kopírovacieho konštruktora `C::C(C&)`. Funkcii sa potom dodá známym spôsobom – skopírovaním na zásobník. Návratová hodnota funkcie, ktorá je v podstate takisto dočasným objektom, vytvoreným pri opúšťaní jej tela príkazom `return` s patričným argumentom, potom bude slúžiť ako vzor, na základe ktorého sa vytvorí objekt `c2`. Vzhľadom na tvar inicializácie `c2` sa opäť použije kopírovací konštruktor. Je, samozrejme, možné, že na základe návratovej hodnoty sa vytvorí nový dočasný objekt a až ten bude slúžiť ako vzor na vytvorenie `c2`, presné detaily sú implementačne závislé. Ak chcete vedieť, aké rôzne dočasné objekty vznikajú, stačí doplniť do oboch konštruktorov nejaký ladiaci výpis – koľkokrát tento výpis uvidíte na obrazovke, toľkokrát sa konštruktor volal. Tento investigatívny spôsob výčby vám vrelo odporúčam, pretože len vďaka nemu získate presnú predstavu o tom, čo všetko program robí.

Tretí objekt `c3` sa inicializuje podobným spôsobom ako `c2`, s tým rozdielom, že argumentom funkcie `f()` je nami explicitne vytvorený dočasný objekt triedy `C`. Pri jeho vytvorení sa použije vzhľadom na argument `11` konštruktor `C::C(int)`. Takýto dočasný objekt môžeme vytvoriť hocikde, kde potrebujeme objekt triedy `C`, jeho životnosť je však obmedzená na výraz, v ktorom sa nachádza. Ak ho teda chceme použiť aj neskôr, budeme ho musieť buď priradiť inému objektu, alebo vytvoriť jeho kópiu (na báze kopírovacieho konštruktora). V každom prípade však dočasný objekt po použití zanikne.

V príklade máme ešte jeden riadok, na ktorom objektu `c1` priraďujeme výsledok volania funkcie `f()` s týmto objektom ako argumentom. Je očividné, že pri vyhodnocovaní tohto výrazu dôjde ku vzniku jedného či viacerých dočasných objektov – typicky sa vytvorí pomocou kopírovacieho konštruktora kópia `c1`, tá sa odovzdá funkcii `f()`, ktorá vráti nejakú hodnotu typu `C` a táto hodnota sa pomocou priraďovacieho operátora (ktorý môže byť prekrytý! – o tom si ešte budeme hovoriť) priradí objektu `c1`.

Na tomto mieste je vhodné objasniť, akým spôsobom obyčajne vraciame z funkcie hodnotu typu triedy. Najprv predpokladajme, že návratovou hodnotou nejakej funkcie je „čistý“ (t. j. nemodifikovaný) typ triedy, napríklad `complex`. Návratovú hodnotu určíme ako argument príkazu `return`. Môže ním byť už existujúci globálny či lokálny objekt alebo môžeme použiť spomínaný explicitne vytvorený dočasný objekt. Druhý spôsob sa veľmi často používa v jednoduchších funkciách; ako príklad si doplníme do našej triedy `complex` členskú funkciu `conj()`, ktorá bude vracať číslo komplexne konjugované (pre tých, čo zabudli, k číslu *a* + *b*i je komplexne konjugované číslo *a* – *b*i). „Uvravenejšia“ verzia bude vyzerať asi takto:

```
complex complex::conj()
{
  complex c;
  c.re = re;
  c.im = -im;
  return c;
}
```

Načo však robiť veci zložito, keď to ide oveľa jednoduchšie:

```
complex complex::conj()
{
  complex c(re, -im);
  return c;
}
```

A úplne najjednoduchšie je to takto:

```
complex complex::conj()
{
  return complex(re, -im);
}
```

Oba posledné príklady sú si veľmi podobné a v podstate využívajú existenciu príslušného konštruktora. Opakovane upozorňujem, že presné detaily odovzdania návratovej hodnoty sú implementačne závislé, nedá sa teda dopredu povedať, koľko dočasných medziobjektov sa použije.

Objekt nejakej triedy však môžeme z funkcie vrátiť aj referenciou. V takom prípade si musíme dať pozor na to, aby sme nevracali automatickú premennú alebo dočasný objekt, pretože by sme dostali buď referenciu nikam, alebo tzv. dočasnú referenciu, čo je vlastne referencia na vytvorený dočasný objekt, ktorého životnosť bude rovnaká ako životnosť danej referencie. Pokiaľ naozaj potrebujeme vrátiť referenciou novo vytvorený objekt, pomôže nám operátor `new`. Nesmieme však zabudnúť po skončení práce objekt aj zrušiť pomocou `delete`. Malý príklad (bez výrazného zmyslu):

```
complex& complex::fnc()
{
  // ...
  return *(new complex(1, 2));
};
```

## Konverzie

Nie je to tak dávno, čo sme sa zaoberali štandardnými konverziami medzi údajovými typmi C++. Pri tej príležitosti sme naznačili, že okrem nich C++ poskytuje možnosť tzv. používateľských konverzií, slúžiacich prakticky výhradne na prevod medzi objektovými a štandardnými typmi, resp. medzi objektovými typmi navzájom. Používateľské konverzie sú vyvolávané implicitne v podobných situáciách ako štandardné, teda. pri konverzii inicializátorov, argumentov a návratových hodnôt funkcií, pri konverzii operandov vo výrazoch, výrazov používaných v iteračných či vetviacich príkazoch a pod.

Používateľské konverzie možno definovať dvoma spôsobmi: pomocou konštruktorov a pomocou konverzných funkcií. Prvý spôsob zabezpečuje konverzie z iných typov *na* daný objektový typ, druhý spôsob konverzie presne opačné.

### Konverzie pomocou konštruktorov

Tento spôsob konverzií je v podstate triviálny. Stačí do deklarácie triedy doplniť konštruktor, ktorý možno zavolať s jediným argumentom toho-ktorého typu. Príklad:

```
class C
{
public:
  C(int);
  C(char*, int = 0);
  // ...
};
```

S takto deklarovanou triedou môžeme používať nasledujúce príkazy:

```
C c1 = 1;
```

Objekt `c1` je inicializovaný pomocou kopírovacieho konštruktora, ktorého argumentom je dočasný objekt `C(1)`, vzniknutý konverziou čísla `1` na objekt triedy `C` pomocou konštruktora `C::C(int)`. Riadok je v podstate ekvivalentný riadku:

```
C c1 = C(1);
```

Iná konverzia:

```
C c2 = „Niki“;
```

V tomto prípade sa vyvolá konštruktor `C::C(char*, int)`. Deklarácia je ekvivalentná tejto:

```
C c2 = C(„Niki“, 0);
```

Konverzie nie sú obmedzené len na deklarácie:

```
c1 = 123;
```

V tomto priradení sa číslo `123` pomocou konštruktora konvertuje na dočasný objekt `C(123)`. Tento objekt sa potom pomocou priraďovacieho operátora skopíruje do `c1`.

Ak sa pri snahe o konverziu nenájde vhodný konštruktor, kompilátor neskúša, či sa náhodou nedá vykonať konverzia okľukou cez iný typ. Nasledujúci kód je preto chybný:

```
class A
{
public:
  A(int);
  // ...
};

class B
{
public:
  B(A);
  // ...
};

B b = 10;
```

V tomto prípade sa neskúša volanie `B(A(10))`, pokiaľ to sami prekladaču nenaznačíme zápisom:

```
B b = A(10);
```

### Konverzné funkcie

Opačný smer konverzie majú na starosti konverzné funkcie. V skutočnosti ide o prekryté operátory pretypovania (o prekrývaní operátorov sme ešte nehovorili). Deklarujeme ich ako bežné členské funkcie, ale so špecifickým názvom, tvoreným kľúčovým slovom `operator`, za ktoré zapíšeme cieľový typ (štandardný či objektový, povolené sú aj kombinácie s operátormi `*` a `&`). Ako príklad si doplníme našu triedu `complex` o operátor konverzie na typ `double`. Výsledkom konverzie bude absolútna hodnota komplexného čísla:

```
#include <math.h>

class complex
{
  // ...
  operator double()
  { return sqrt(re * re + im * im); }
};
```

Všimnite si, že pri konverznej funkcii neuvádzame typ návratovej hodnoty – ten je už raz uvedený v názve funkcie. Vďaka doplnenej členskej funkcii môžeme bez problémov konvertovať typ `complex` na `double` a používať objekty triedy `complex` všade tam, kde sa očakáva typ `double`:

```
complex c;
double d = double(c);
d = (double)c;
d = c;
double e = c ? (c + 1) : c;
if (c) { ... };
```

Podobne ako pri konverzných konštruktoroch prekladač nebude hľadať možnosti konverzie okľukou cez iné typy:

```
class A
{
public:
  operator int();
  // ...
};

class B
{
public:
  operator A();
  // ...
};

B b;
int i = b;
```

V poslednom riadku je chyba, prekladač nebude skúšať konverziu `b.operator A().operator int()`. Môžeme mu to však naznačiť:

```
int i = A(b);
```

Povedali sme už, že prekladač v mnohých prípadoch vyvoláva konverzné funkcie implicitne. Môže sa však stať, že použijeme objekt nejakej triedy v takom kontexte, keď nebude jasné, ktorú z konverzných funkcií treba použiť. Vtedy sa nebude implicitne volať žiadna a prekladač ohlási chybu. Typický príklad:

```
class C
{
public:
  operator int();
  operator void*();
  // ...
};

C c;
if (c)
{ ... }
```

Z kontextu nie je jasné, či sa má objekt `c` v rámci podmienkového výrazu príkazu `if` konvertovať na typ `int` alebo na typ `void*`. Jediným riešením je v tomto prípade explicitné pretypovanie.

### Inicializácia objektov

Zopakujme si, čo už vieme o možnostiach inicializácie objektov. V prípade, že trieda má iba verejné nestatické členy, nemá nijaký konštruktor, neobsahuje virtuálne funkcie a nie je odvodená od iných tried, môžeme jej inštancie inicializovať podobne ako štruktúry zoznamom inicializačných hodnôt jednotlivých členov. Trieda, ktorá má konštruktor, musí byť buď inicializovaná explicitne, alebo musí mať implicitný konštruktor, ktorý sa použije v prípade, že objekt nie je inicializovaný explicitne.

V prípade, že inicializujeme objekt pomocou jeho konštruktora, môžeme do zátvoriek uviesť zoznam argumentov tohto konštruktora. Alternatívny spôsob je uvedenie jedinej hodnoty za operátor `=`. Táto hodnota sa (po prípadných konverziách) uvedie ako argument kopírovacieho konštruktora.

Majme vyššie deklarovanú triedu `complex` so štyrmi konštruktormi. Uvedieme si pre zhrnutie rôzne možnosti inicializácie:

```
complex c1(2);
```

Objekt `c1` sa inicializuje pomocou konštruktora `complex::complex(double)`.

```
complex c2 = c1;
```

Objekt `c2` sa inicializuje pomocou kopírovacieho konštruktora kópiou objektu `c1`.

```
complex c3 = complex(9, 11);
```

Pomocou konštruktora `complex::complex(double, double)` sa skonštruuje dočasný objekt `complex(9, 11)`, ktorým sa na základe kopírovacieho konštruktora inicializuje objekt `c3`.

```
complex c4;
```

Objekt `c4` sa inicializuje implicitným konštruktorom `complex::complex()`.

```
complex c5 = 42;
```

Prostredníctvom konštruktora `complex::complex(double)` sa skonštruuje dočasný objekt `complex(42)`. Ním sa pomocou kopírovacieho konštruktora inicializuje objekt `c5`.

Dôležité je, že všetky uvedené príklady sú deklarácie, preto ani uvedenie operátora `=` v nich neznamená priradenie, ale inicializáciu. To je dôležité vedieť vtedy, keď potrebujeme rozlišovať medzi kopírovacím konštruktorom a prekrytým priraďovacím operátorom.

Inicializácia, ktorá nastáva ako dôsledok odovzdávania argumentov a návratových hodnôt funkcií, je ekvivalentná tvaru:

```
T x = a;
```

kde `T` je príslušný objektový typ, `x` je formálny argument/návratová hodnota a `a` je skutočný argument/skutočne vracaná hodnota. Inicializácia spojená s alokáciou dynamického objektu operátorom `new` je ekvivalentná tvaru:

```
T x(a);
```

Inicializovať môžeme aj polia objektov. Ak neuvedieme zoznam inicializačných hodnôt, použije sa implicitný konštruktor, ktorý musí v triede existovať, inak prekladač ohlási chybu. Ak zoznam uvedieme, musí v triede existovať konštruktor, ktorý prijíma jediný argument. Prekladač pri inicializácii postupne berie prvok za prvkom z inicializačného zoznamu a na ich základe konštruuje jednotlivé objekty poľa. Pokiaľ potrebujeme skonštruovať objekty na základe viacargumentových konštruktorov, použijeme takúto fintu:

```
complex cc[4] = {
  3,
  complex(),
  5,
  complex(7.6, 1.1)
};
```

Prvky `cc[0]` a `cc[2]` sa inicializujú pomocou `complex::complex(double)`, prvok `cc[1]` pomocou `complex::complex()` a konečne prvok

cc[3] pomocou complex::complex(double, double).

V prípade, že inicializačný zoznam má menej prvkov, ako je deklarovaný rozmer poľa objektov, musí mať trieda implicitný konštruktor, ktorý sa použije na skonštruovanie zostávajúcich prvkov.

Inicializovať uvedeným spôsobom nemôžeme pole objektov vytvorené dynamicky, pomocou operátora new. V takom prípade musia mať objekty implicitný konštruktor, ktorý sa vyvolá pre každý prvok novo vytvoreného poľa.

### Deštruktor

Tak ako máme možnosť objekt po jeho vytvorení inicializovať pomocou konštruktora, C++ nám poskytuje možnosť objekt pred jeho zrušením aj správne „zlikvidovať". Takáto likvidácia obyčajne spočíva v uvoľnení alokovanej pamäte, prostriedkov a prípadne v iných upratovacích činnostiach. Zjednodušene povedané, čo konštruktor vytvoril, to deštruktor zruší. K volaniu deštruktora dochádza vždy vtedy, keď príslušný objekt končí svoju životnú púť: automatické objekty pri opustení príslušného bloku, statické po skončení programu a dynamicky alokované zavolaním operátora delete.

Deklarácia deštruktora je v princípe podobná deklarácii konštruktora. Meno deštruktora dostaneme pripojením znaku ~ (tilda) pred meno triedy. Deštruktor nesmie mať nijaké argumenty a podobne ako konštruktor nemá deklarovaný typ návratovej hodnoty (ani ako void). Takisto nemôže byť deštruktor deklarovaný ako const alebo volatile a nesmie byť ani statický. Na rozdiel od konštruktora však môže mať trieda virtuálny deštruktor. Z tela deštruktora možno volať ostatné členské funkcie. Prvky poľa sa deštruujú v opačnom poradí, ako boli konštruované.

Ako príklad na deštruktor si uvedieme triedu string, ktorá bude zapuzdrovať reťazec znakov. Jediným jej údajovým členom bude ukazovateľ na obsiahnutý reťazec. Do triedy zavedieme aj niekoľko konštruktorov:

```
class string
{
  char* str;

public:
  string(const char* s)
  {
    str = new char[strlen(s) + 1];
    strcpy(str, s);
  }

  string(const string& s)
  {
    str = new char[strlen(s.str) + 1];
    strcpy(str, s.str);
  }

  ~string()
  {
    delete str;
  }
};
```

Trieda, samozrejme, pre jednoduchosť netestuje, či nie sú argumenty konštruktorov náhodou nulové ani či sa alokácia podarila. Vidíme, že jedinou starosťou deštruktora string::~string() je delokácia miesta, na ktoré ukazuje ukazovateľ str.

Možno si spomeniete, ako sme kedysi dávno, pri výklade o dynamickej alokácii a dealokácii pamäte, hovorili o tom, že pre zrušenie alokovaného poľa objektov musíme použiť mierne odlišnú syntax operátora delete:

```
string* as = new string[10];
// ...
delete [] as;
```

(Pozor, tento príklad nie je celkom správny, trieda string, ako sme ju zatiaľ definovali, neobsahuje implicitný konštruktor!) Za operátor delete musíme v tomto prípade pripísať ešte dvojicu zátvoriek [], aby sme zabezpečili, že sa pre každý prvok poľa as pri rušení tohto poľa vyvolá jeho deštruktor.

## Deväťnásta časť: DEDIČNOSŤ A POLYMORFIZMUS

Povieme si o tom, ako triedy dokážu jedna od druhej dediť svoje vlastnosti a ako je možné, že inštancie tried dokážu byť polymorfné.

### Keď triedy dedia

V reálnom živote akt dedenia obyčajne predpokladá ukončenie životnosti jedného objektu (bohatého strýka z Ameriky) a prevod jeho vlastníctva (ťažko zarobených miliónov) na iný objekt (najlepšie na nás). Nie tak v C++. Tu vzťah dedičnosti predstavuje statickú väzbu medzi dvoma triedami, pri ktorej je z existujúcej triedy špecifickým spôsobom *odvodená* nová trieda, ktorá automaticky získava všetky atribúty a metódy nadradenej triedy. Statická väzba preto, lebo vzťah dedičnosti je známy už v čase kompilácie programu (compile-time). V C++ nie je možné odvodiť novú triedu, resp. nový objekt dedením za behu programu (run-time), čo ostatne vyplýva z nutnosti pre každý objekt, ktorý chceme v programe použiť, povinne deklarovať triedu.

Trieda, ktorej atribúty sú dedené, sa nazýva aj *základná* trieda, rodičovská trieda, resp. trieda predka. Odvodená trieda je dcérska trieda či trieda potomka.

Prvoradým zmyslom existencie dedenia medzi triedami je vyjadrenie vzťahu špecializácie/generalizácie. Predstavme si, že máme definovanú triedu zivocich. Z nej môžeme odvodiť triedy cicavec, vtak, hmyz a iné, ktoré sú špecifickými druhmi živočíchov. Každú z týchto tried môžeme ďalej rozvíjať, čím dostávame stromovú hierarchiu tried, v ktorej každá odvodená trieda je špeciálnym prípadom základnej triedy a naopak – základná trieda je zovšeobecnením hociktorej odvodenej triedy. V praxi takto dosiahneme, že vlastnosti, ktoré sú spoločné viacerým odvodeným triedam, budú sústredené do základnej triedy, čím sa ušetrí často zbytočné opakovanie deklarácií atribútov či metód. Navyše, ako uvidíme, C++ nám umožní všade tam, kde sa bude vyžadovať inštancia základnej triedy, použiť inštanciu ľubovoľnej odvodenej triedy, a to nielen v priamom vzťahu dedičnosti, ale aj v nepriamom, cez viacero stupňov hierarchie. Toto je ostatne nevyhnutná podmienka realizácie polymorfizmu. Ale k tomu sa ešte dostaneme.

Odvodená trieda automaticky dedí všetky verejné (public) a chránené (protected) údajové členy a členské funkcie základnej triedy, ku ktorým môžeme pristupovať tak, akoby boli deklarované v odvodenej triede. Už sme si spomenuli, že chránené členy sú prístupné jednak členským a spriateleným funkciám danej triedy, jednak členským a spriateleným funkciám všetkých odvodených tried, ibaže doteraz sme nevedeli, čo sú to odvodené triedy. Súkromné (private) členy nie sú odvodeným triedam prístupné.

Základné triedy (môže ich byť aj viac, vtedy hovoríme o viacnásobnej dedičnosti – o dôsledkoch si povieme neskôr) špecifikujeme pri deklarácii odvodenej triedy – za jej názov doplníme dvojbodku a čiarkami oddelený zoznam základných tried:

```
class Base
{
public:
  int a, b;
  void f();
};

class Derived : public Base
{
public:
  double c, d;
  int g(int);
};
```

V našom príklade trieda Base obsahuje dva údajové členy a a b typu int a jednu členskú funkciu f typu void(). Trieda Derived všetky tieto zložky dedí a navyše pridáva svoje vlastné: ďalšie dva údajové členy c a d typu double a členskú funkciu g typu int(int). Nasledujúce výrazy sú teda správne:

```
Base b;
Derived d;
b.a = 10;
b.b = 14;
b.f();
d.a = 17;
d.b = 900;
d.c = 8.854;
d.d = 1e40;
d.f();
int i = d.g(83);
```

zatiaľ čo tieto sú chybné:

```
b.c = 24.9;
b.d = 0.001;
int j = b.g(0);
```

čo je zrejmé, pretože trieda Base členy c, d a g() neobsahuje.

### Zmena prístupu

V deklarácii triedy Derived vidíme, že pred názvom jej základnej triedy Base je uvedený špecifikátor public, ktorý aj v tomto prípade determinuje prístupové práva, no tentoraz pre zdedené členy. Vo všeobecnosti pred názvom každej základnej triedy môžeme uviesť jeden zo špecifikátorov private, protected, public. Ako sa na ich základe mení prístup k dedeným členom, vidíme z tabuľky č. 1.

*Tabuľka č. 1 Zmena prístupu k zdedeným členom*

| Deklarovaný prístup k členu základnej triedy | Špecifikátor prístupu k základnej triede | | |
|---|---|---|---|
| | **public** | **protected** | **private** |
| **public** | public | protected | private |
| **protected** | protected | protected | private |
| **private** | – | – | – |

V ľavom stĺpci sú uvedené všetky možnosti prístupu k údajovým členom základnej triedy, v hornom riadku sú takisto uvedené všetky špecifikátory, ktoré môžeme použiť pri deklarácii odvodenej triedy (za dvojbodkou). V jednotlivých políčkach tabuľky potom nájdeme výsledný typ prístupu. Je zrejmé, že ak je trieda verejnou (public) základnou triedou, zachovajú si jej verejné a chránené členy v odvodenej triede svoje prístupové práva bez zmeny. Zdedené členy chránenej (protected) základnej triedy budú v odvodenej triede chránené a podobne zdedené členy privátnej (private) základnej triedy budú v odvodenej triede privátne. Pre privátne členy základnej triedy zmena prístupu nemá zmysel, pretože tieto členy v odvodenej triede nie sú prístupné.

Pokiaľ špecifikátory prístupu vynecháme, použijú sa implicitné hodnoty – ak je odvodená trieda štruktúrou (struct), doplní sa public, ak je triedou (class), doplní sa private.

V prípade, že nám nevyhovuje spôsob zmeny prístupových práv k zdedeným členom, môžeme ho v deklarácii odvodenej triedy upraviť uvedením kvalifikovaného

mena príslušného člena v sekcii s vhodným špecifikátorom:

```
class B
{
  // ...
public:
  int a, b;
};

class D : private B
{
  // ...
public:
  B::a;
  int c;
};
```

Trieda D dedí od triedy B člen a, ktorý je v triede B verejný. Vďaka tomu, že B je privátnou základnou triedou D, bol by za normálnych okolností člen a v triede D privátny. Uvedením jeho kvalifikovaného mena B::a (bez špecifikátora typu!) v sekcii za špecifikátorom public mu „vraciame“ jeho verejnosť aj v triede D.

Úprava prístupu týmto spôsobom nie je povolená v prípade, že by sme chceli obmedziť prístup k údajovému členu prístupnému v základnej triede, a ani v prípade, že by sme chceli povoliť prístupu k členu neprístupnému v základnej triede:

```
class B
{
private:
  int a;
public:
  int b;
};

class D : private B
{
protected:
  B::b;   // chyba
public:
  B::a;   // chyba
};
```

V príklade chceme privátny člen a triedy B urobiť verejným v triede D, okrem toho chceme verejný člen b triedy B urobiť chráneným v triede D. Ani jedno, ani druhé nám, samozrejme, prekladač nedovolí.

## Prístup k chráneným členom

Vráťme sa ešte k spôsobu prístupu ku chráneným členom. Vieme, že k nim majú prístup okrem členských a spriatelených funkcií základnej triedy aj členské a spriatelené funkcie všetkých odvodených tried. Ale pozor! Tie môžu k nestatickým chráneným členom základnej triedy pristupovať *len* prostredníctvom objektu odvodenej triedy, resp. ukazovateľa či referencie naň (statické sú, samozrejme, prístupné pomocou kvalifikovaného mena):

```
class B
{
protected:
  int a;
};

class D
{
  void f(B&, D&);
  friend void g(B*, D*);
};

void D::f(B& b, D& d)
{
  b.a = 1;        // chyba
  d.a = 2;        // OK
  ((B&)d).a = 3; // chyba
}

void g(B* pb, D* pd)
{
  pb->a = 4;        // chyba
  pd->a = 5;        // OK
  ((B*)pd)->a = 6; // chyba
}

void h(B& b, D& d)
{
  b.a = 7;     // chyba
  d.a = 8;     // chyba
}
```

Členská funkcia D::f(), ako aj funkcia g(), ktorá je „priateľom“ triedy D, má prístup k zdedenému chránenému členu a triedy B, ale len prostredníctvom referencie, resp. ukazovateľa na objekt triedy D. Pri prístupe cez objekt typu B je člen a, samozrejme, skrytý. Všimnite si aj to, že pri pretypovaní argumentov d, resp. pd na referenciu, resp. ukazovateľ na triedu B (čo je dovolené – vysvetlíme o chvíľu) strácame k členu a prístup, hoci očividne pracujeme stále s objektom triedy D. V príklade je ešte pre ilustráciu nečlenská a nespriatelená funkcia h(), ktorej nie je člen a prístupný nijakým spôsobom.

## Prekrývanie členov

C++ dovoľuje v deklarácii odvodenej triedy prekryť ľubovoľný člen základnej triedy. Nemusí ísť v tomto prípade o priameho predka, takisto je možné prekryť ľubovoľný člen hociktorej triedy, ktorá je v hierarchii „vyššie“. Prekryté členy sú v odvodenej triede dostupné pod svojím kvalifikovaným menom:

```
class B
{
public:
  int a, b;
};

class D : public B
{
public:
  int b, c;
};

void f()
{
  D d;
  d.a = 10;
  d.B::b = 20;
  d.b = 30;
  d.c = 40;
}
```

Trieda D v našom príklade má štyri dátové členy: a a b zdedené od triedy B a vlastné, b a c. Vlastný člen b triedy D prekrýva zdedený člen b triedy B, ku ktorému máme prístup len pomocou jeho kvalifikovaného mena B::b. Použité meno triedy v kvalifikovanom mene člena však neznamená nevyhnutne triedu, v ktorej tento člen musí byť definovaný, ale skôr triedu, v ktorej sa začne „hľadať“ požadovaný člen smerom nahor v hierarchii dedičnosti:

```
class A { public: int x; };
class B : public A {};
class C : public B { public: int x; };

C c;
c.x = 10;
c.B::x = 20;
c.A::x = 30;
```

Posledné dva riadky priraďujú hodnotu tomu istému členu x zdedenému z triedy A, pretože v triede B nie je nijaký člen s menom x definovaný.

## Ekvivalencia tried

Dôležitou vlastnosťou C++ je, že ukazovateľ, resp. referencia na odvodenú triedu môže byť vždy konvertovaný(á) na ukazovateľ, resp. referenciu na hociktorú *prístupnú* základnú triedu. Je to nanajvýš logické, pretože každý objekt triedy D odvodenej z triedy B je súčasne platným objektom triedy B – obsahuje všetky jej členy a členské funkcie. Mohli by ste namietnuť: veď privátne členy sa nededia. Áno, to je pravda, ale len v tom zmysle, že k nim odvodená trieda nemá prístup. V skutočnosti sa v inštancii odvodenej triedy nachádza kompletná inštancia základnej triedy, o čom sa môžete presvedčiť v nasledujúcom príklade:

```
class B
{
private:
    int a;
public:
    int b;
    int f() { return a + b; }
};

class D : public B {};

void g(B& b)
{
    int x = b.f();
}

void main()
{
    D d;
    g(d);
}
```

Funkcia g() prijíma ako argument referenciu na objekt triedy B, nad ktorým zavolá jeho členskú funkciu f(). Funkcia f() vracia súčet jedného privátneho a jedného verejného člena triedy B. My však funkcii g() odovzdáme ako argument referenciu na objekt triedy D. To, samozrejme, podľa uvedeného pravidla môžeme, referencia sa automaticky konvertuje. Lenže nad týmto objektom sa takisto vyvolá funkcia f(), ktorá sa bude snažiť pristupovať k privátnemu členu a, mysliac si, že pracuje s objektom triedy B. Keďže uvedený príklad funguje, je zrejmé, že inštancia triedy D musí tento člen obsahovať.

## Inicializácia odvodených tried

Vieme už, že inicializáciu inštancií objektových typov majú na starosti špeciálne členské funkcie, nazývané konštruktory. Povedali sme si, za akých okolností prekladač môže automaticky vygenerovať vlastné verzie konštruktorov, a vieme aj to, že každá trieda môže mať niekoľko konštruktorov. Zatiaľ však nevieme, ako je to s inicializáciou odvodených tried – tie obsahujú inštancie svojich predkov, ktoré treba takisto nejakým spôsobom inicializovať.

Začneme triedou bez konštruktorov. Pri vzniku novej inštancie takejto triedy sa vyvolá implicitný konštruktor, vygenerovaný prekladačom. Jeho jedinou úlohou je vyvolanie implicitných konštruktorov všetkých základných tried (priamych, lebo tie následne budú volať konštruktory *svojich* základných tried atď.). Pokiaľ niektorá základná trieda nemá implicitný konštruktor (a súčasne má definovaný aspoň jeden *iný* konštruktor), máme smolu, lebo v takom prípade sa preklad skončí chybou. Pokiaľ je trieda inicializovaná pomocou kopírovacieho konštruktora, vytvoreného prekladačom, ten skôr, než inicializuje vlastné dátové členy, zavolá kopírovacie konštruktory základných tried.

Ak trieda obsahuje jeden či viacero konštruktorov, máme možnosť explicitne ovplyvniť, aké konštruktory základných tried sa použijú a s akými parametrami budú volané. Princíp je jednoduchý – za deklarátor konštruktora, ale ešte pred krútenú zátvorku, ktorou sa začína jeho telo, uvedieme za dvojbodku čiarkami oddelený zoznam inicializátorov. Každý z týchto inicializátorov má tvar

*meno-triedy* ( *zoznam-výrazov* )

alebo

*identifikátor* ( *zoznam-výrazov* )

Syntax s *menom triedy* sa používa na inicializáciu zdedených členov. V zátvorke uvedený zoznam výrazov nie je nič iné ako zoznam parametrov, na základe ktorých sa vyberie a zavolá príslušný konštruktor základnej triedy. Meno triedy musí predstavovať niektorého z priamych predkov odvodenej triedy. Druhý tvar – s *identifikátorom* – umožňuje takto zjednodušene inicializovať členy odvodenej triedy. Identifikátor predstavuje meno člena, v zátvorke uvedený zoznam jeho inicializačné hodnoty. Zoznam z toho dôvodu, že môžeme takto inicializovať aj vnorené objekty iných tried. Mimochodom, tento druhý tvar nám poskytuje jedinú možnosť, ako inicializovať konštantné údajové členy a členy typu referencie.

Ale dosť teórie, ukážme si radšej príklad:

```
class B1
{
  // ...
public:
  B1(int);
};

class B2
{
  // ...
public:
  B2(double);
};

class D : public B1, public B2
{
  const double c;
  B1 d;
public:
  D(int a, double b) : B1(a + 5),
                       B2(b * 3.45),
                       c(9.999),
                       d(a << 3)
  { /* ... */ }
};

D d(4, 11.23);
```

Trieda `B1` má jediný konštruktor s jediným argumentom typu `int`, podobne trieda `B2` (okrem typu `double`). Trieda `D` má dvoch predkov, `B1` aj `B2`, jej konštruktor má dva argumenty typu `int` a `double`. Vyvolaním tohto konštruktora sa inicializuje zdedený podobjekt triedy `B1` hodnotou `a+5`, podobjekt triedy `B2` hodnotou `b*3.45`. Okrem toho sa inicializujú dva vlastné údajové členy triedy `D` – člen `c` (ktorý je konštantný a ďalej nemenný) hodnotou `9.999` a člen `d`, ktorý je inštanciou triedy `B1` (ale inou ako tá zdedená). Ten sa inicializuje hodnotou `a<<3` typu `int`, preto sa vyvolá jeho konštruktor `B1::B1(int)`.

Zoznam inicializátorov v konštruktore nie je povinný; ak chýba, volajú sa implicitné konštruktory predkov (ktoré musia existovať a byť prístupné) a údajové členy triedy sa inicializujú implicitne (teda vlastne nijako, jedine že by išlo o statické objekty, ktoré sa vynulujú).

Poradie, v ktorom sa uplatňujú jednotlivé inicializátory, je presne stanovené: najprv sa inicializujú zdedené podobjekty základných tried v poradí deklarácie (*nezávisle* od poradia inicializátorov v konštruktore), potom sa inicializujú vlastné údajové členy (samozrejme, len tie nestatické!), opäť v poradí ich deklarácie v triede a nakoniec sa vykoná samotné telo konštruktora. Inak povedané, najprv sa vyvolajú konštruktory predkov, potom sa nastaví východiskový stav objektu, ktorý sa nakoniec prípadne upraví podľa aplikačných špecifík, definovaných telom konštruktora. Jedinou výnimkou z toho pravidla sú virtuálne základné triedy, ale o tých si povieme až neskôr.

Pri zániku inštancií odvodených tried sa vyvoláva deštruktor, ktorý okrem iného automaticky zabezpečí zrušenie vnorených objektov a zdedených podobjektov. Poradie volania deštruktorov je presne opačné ako pri inicializácii: najprv sa vykoná samotné telo deštruktora, potom sa volajú deštruktory nestatických členských objektov (v opačnom poradí, ako boli deklarované) a nakoniec sa zavolajú deštruktory základných tried (takisto v opačnom poradí, ako boli deklarované).

## Virtuálne funkcie

Ľubovoľná členská funkcia s výnimkou konštruktora a niektorých operátorových funkcií môže byť deklarovaná ako virtuálna pripojením špecifikátora `virtual` k jej deklarácii. Predstavme si, že máme triedu `B`, ktorá obsahuje virtuálnu funkciu `f()`, a triedu `D`, odvodenú od triedy `B`, ktorá obsahuje funkciu `f()` s rovnakou signatúrou (t. j. počtom a typmi argumentov a typom návratovej hodnoty). Potom je aj funkcia `D::f()` virtuálna (nezávisle od toho, či je deklarovaná ako `virtual`, alebo nie) a je zaručené, že každé volanie `f()` nad objektom, ktorý je inštanciou triedy `D`, vyvolá funkciu `D::f()` bez ohľadu na to, či k nemu pristupujeme pomocou ukazovateľa, resp. referencie na typ triedy `B` (t. j. jej základnej triedy), alebo nie. Majme teda tento kód:

```
class B
{
public:
  virtual void f() { printf(„in B\n"); }
};

class D : public B
{
public:
  void f() { printf(„in D\n"); }
};

void g1(B* pb) { pb->f(); }
void g2(D* pd) { pd->f(); }

B b;
D d;
g1(&b);  // vypíše ‚in B'
g2(&d);  // vypíše ‚in D'
g1(&d);  // vypíše ‚in D' !
```

V príklade sme definovali dve pomocné funkcie `g1()` a `g2()`. Obe volajú členskú funkciu `f()` svojho argumentu, ibaže jedna z nich používa ukazovateľ na typ `B` a druhá na typ `D`. Ak si program vyskúšate, uvidíte, že hoci funkcia `g1()` pracuje so svojím argumentom ako s objektom triedy `B`, pokiaľ jej odovzdáme ukazovateľ na objekt triedy `D`, vyvolá *jeho* funkciu `f()`, a nie funkciu `f()` zdedenú z triedy `B`.

Virtuálna funkcia odvodenej triedy v podstate „maskuje" rovnakú funkciu základnej triedy. Slovo „maskuje" sa veľmi nehodí, ale to je problém s prekladom pôvodných anglických výrazov *overload* a *override*. Prvý z nich opisuje situáciu, keď máme viacero funkcií s rovnakým názvom, ale s rôznou signatúrou. V takom prípade sme hovorili, že sa funkcie prekrývajú. Druhý z nich naproti tomu vyjadruje vzťah medzi virtuálnymi funkciami (s rovnakým názvom aj signatúrou) z tried zúčastňujúcich sa vzťahu dedičnosti.

Funkcia, ktorá má v odvodenej triede rozdielny počet a/alebo typy argumentov ako virtuálna funkcia s rovnakým menom v základnej triede, nie je virtuálna. Líšiť sa môže len návratovým typom, aj to len vtedy, ak virtuálna funkcia základnej triedy `B` vracia typ `B*` alebo `B&` a virtuálna funkcia odvodenej triedy `D` vracia typ `D*` alebo `D&`. Toto pravidlo však niektoré staršie prekladače nepodporujú.

Prístup k virtuálnej funkcii sa riadi špecifikátorom uvedeným v tej triede, prostredníctvom ktorej (teda vlastne ukazovateľa či referencie, na ktorú) je volanie realizované:

```
class B
{
public:
  virtual void f();
};

class D : public B
{
private:
  void f();
};

D d;
B* pb = &d;
D* pd = &d;
pb->f();  // OK
pd->f();  // chyba
```

Volanie `f()` prostredníctvom ukazovateľa `pb` je v poriadku, funkcia `f()` je v triede `B` verejná (i keď v skutočnosti sa volá funkcia `D::f()`). Volanie `f()` prostredníctvom `pd` už je však chybné, pretože `f()` je v triede `D` deklarovaná ako privátna.

Mimoriadne vhodné je používanie virtuálnych deštruktorov, čo nám zabezpečí vyvolanie vždy toho správneho deštruktora pre každý rušený objekt. Musíme však dať pozor na skutočnosť, že v rámci tela deštruktora, ale aj konštruktora sa mechanizmus virtuálnych funkcií neuplatňuje a volajú sa vždy iba členské funkcie danej triedy. Je to koniec koncov logické – v okamihu volania konštruktora základnej triedy ešte nie je kompletne skonštruovaný objekt odvodenej triedy a volanie jeho virtuálnej funkcie by operovalo nad nekonzistentným stavom objektu. Podobne v okamihu volania deštruktora základnej triedy je objekt odvodenej triedy už zrušený a volanie virtuálnej funkcie by v takomto prípade operovalo nad de facto zrušeným objektom.

Zmyslom virtuálnych funkcií je umožniť rozlišovanie skutočného typu objektov za behu programu, na rozdiel od rozlišovania typu vo fáze prekladu na základe typu ľavého operandu operátorov `.` alebo `->`, používaných pri prístupe k členským funkciám. Z toho vyplýva aj skutočnosť, že virtuálnymi nemôžu byť statické funkcie, pretože tie sa nevzťahujú na nijakú existujúcu inštanciu typu. Čo sa týka praktického použitia virtuálnych funkcií, predstavme si, že máme spomínanú triedu `zivocich`, ktorá obsahuje virtuálnu funkciu `vydaj_zvuk()` (ospravedlňujem sa za trochu krkolomné príklady :). Každý živočích vydáva iné zvuky, preto bude mať každá odvodená trieda túto funkciu implementovanú inak. Virtuálnosť funkcie nám však umožní vypočuť si hlasy všetkých živočíchov v našej súkromnej ZOO jednoduchým spôsobom:

```
zivocich* zoo[MAX_ZIV];
// ...
for (int i = 0; i < MAX_ZIV; i++)
  zoo[i]->vydaj_zvuk();
```

Ukazovatele na jednotlivých obyvateľov ZOO sú uložené v poli `zoo[]`. Rôzne ukazovatele budú zrejme ukazovať na živočíchy rôznych tried, ale vďaka dedičnosti a vďaka virtuálnosti funkcie `vydaj_zvuk()` dosiahneme, že každý živočích sa ozve tak, ako mu „zobák narástol".

V jednej z predchádzajúcich častí sme si hovorili o niektorých základných princípoch OOP. Jedným z nich je polymorfizmus, ktorý je v C++ realizovaný práve pomocou virtuálnych funkcií. Jedinou podmienkou jeho úspešného používania je pristupovať k rôznym objektom pomocou ukazovateľa či referencie na nejakú spoločnú triedu, ktorej exportované funkcie musia byť virtuálne a implementované v každej odvodenej triede „po svojom". Takto dosiahneme, že rôzne objekty budú na jednu a tú istú správu (t. j. volanie virtuálnej funkcie) reagovať rôznym spôsobom.

## Abstraktné triedy

Pri návrhu objektovej hierarchie často dochádza k situácii, keď jednotlivé uzly hierarchického stromu pred-

stavujú čisto všeobecné triedy a konkrétne triedy sa nachádzajú v listoch celého stromu. Z tohto dôvodu by bolo vhodné, keby sa dalo zakázať inštancionalizovať uzlové triedy. Prirovnaním k nášmu príkladu – nemá zmysel vytvoriť inštanciu triedy `zivocich`; ani v praxi neexistuje konkrétny tvor, nazývaný *živočích*. Zmysel majú len konkrétne triedy predstavujúce konkrétnych živočíchov (napr. `vrana`, `pavuk` a pod.).

C++ obsahuje na tento účel koncept tzv. abstraktných tried. Z takýchto tried nie je povolené vytvárať inštancie, nemôžu byť odovzdávané do a z funkcie hodnotou (hoci môžu byť odovzdávané ukazovateľom alebo referenciou) a nemôžu byť ani cieľom explicitného či implicitného pretypovania. Možno ich, samozrejme, použiť ako základné triedy. Abstraktná je každá trieda, ktorá obsahuje aspoň jednu *čisto virtuálnu funkciu* (pure virtual function). Takáto funkcia musí za svojím deklarátorom obsahovať ešte špecifikátor `= 0`. Príklad s našou triedou `zivocich` a jej virtuálnou funkciou `vydaj_zvuk()`:

```
class zivocich
{
  // ...
public:
  virtual void vydaj_zvuk() = 0;
};
```

Funkciu `vydaj_zvuk()` zrejme bude ťažké definovať pre všeobecného živočícha. Keď ju však spravíme čisto virtuálnou, môžeme jej konkrétnu definíciu ponechať na odvodené triedy.

Čisto virtuálne funkcie, samozrejme, nemusia byť v abstraktnej triede definované. Na druhej strane to však ani nie je zakázané, takže sa teoreticky môžeme stretnúť aj so zápisom:

```
virtual void vydaj_zvuk() = 0
{
  // ...
}
```

Takto definovanú čisto virtuálnu funkciu môžeme volať len použitím jej plne kvalifikovaného mena `zivocich::vydaj_zvuk()`.

Čisto virtuálne funkcie sú dedené odvodenou triedou takisto ako čisto virtuálne. Pokiaľ odvodená trieda nedoplní definíciu všetkých zdedených čisto virtuálnych funkcií, sama sa stáva abstraktnou.

### Viacnásobná dedičnosť

Ako sme už spomínali, trieda môže byť odvodená od viac ako jednej základnej triedy. Pre túto tzv. *viacnásobnú dedičnosť* v princípe platia rovnaké pravidlá ako pre jednoduchú. Spôsob prístupu k zdedeným členom je možné špecifikovať pre každú základnú triedu zvlášť. Odvodená trieda nemôže priamo dediť od jednej základnej triedy viackrát, môže tak však urobiť nepriamo, cez medzitriedy:

```
class B { /* ... */ };
class M1 : public B { /* ... */ };
class M2 : public B { /* ... */ };
class D : public M1, public M2 { /* ... */ };
```

V tomto príklade bude trieda obsahovať dva zdedené podobjekty typu `B`. Ak takémuto dôsledku chceme zabrániť, musíme deklarovať príslušné základné triedy pri dedení ako virtuálne (pozor! nemýliť si s virtuálnymi funkciami), pridaním špecifikátora `virtual`:

```
class V { /* ... */ };
class M1 : virtual public B { /* ... */ };
class M2 : virtual public B { /* ... */ };
class D : public M1, public M2 { /* ... */ };
```

Tentoraz bude trieda `D` obsahovať iba jedinú kópiu objektu triedy `B`.

Trieda môže mať virtuálnych aj nevirtuálnych predkov súčasne (a to dokonca aj rovnakého typu):

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : public A { /* ... */ };
class E : public B, public C, public D { /* ... */ };
```

Trieda `E` obsahuje jednu nevirtuálnu kópiu objektu triedy `A`, zdedenú cez triedu `D`, a jednu virtuálnu, zdedenú cez triedy `B` a `C`.

Pri používaní viacnásobnej dedičnosti môžeme naraziť na problém existencie viac ako jedného člena s rovnakým menom, ale z rôznych základných tried. V takom prípade jednotlivé členy musíme rozlišovať pomocou ich kvalifikovaných mien:

```
class B1 { public: int a; };
class B2 { public: int a; };
class D : public B1, public B2 { public: void
f(); }

void D::f()
{
  a = 1;       // chyba: B1::a alebo B2::a ?
  B1::a = 2;  // OK
  B2::a = 3;  // OK
}
```

V prípade použitia virtuálnych základných tried môže byť jeden údajový člen dosiahnuteľný cez viacero tried (viacerými cestami v tzv. orientovanom acyklickom grafe, predstavujúcom hierarchiu tried), nejde tu však o dvojznačnosť ako v predchádzajúcom prípade, pretože ide o jeden a ten istý člen:

```
class V { public: int a; };
class B1 : virtual public V {};
class B2 : virtual public V {};
class D : public B1, public B2 { public:
void f(); }

void D::f()
{
  a = 1;       // OK: B1::a = B2::a = V::a
}
```

Virtuálne základné triedy prinášajú so sebou ešte jeden problém: ich použitie môže viesť k tomu, že jedno meno môže byť v grafe tried nájdené viacerými spôsobmi (viacerými cestami). K dvojznačnosti nedôjde vtedy, keď jedno z nájdených mien dominuje nad ostatnými. Meno `B::f` dominuje nad menom `A::f`, ak `A` je predkom `B`. V takom prípade sa „nájde“ meno `B::f` a meno `A::f` sa ignoruje.

Dosiaľ opisovaná možná dvojznačnosť sa týka aj pretypovania ukazovateľa, resp. referencie na odvodený objekt na ukazovateľ, resp. referenciu na základný objekt:

```
class V {};
class N {};
class M1 : public N, virtual public V {};
class M2 : public N, virtual public V {};
class D : public M1, public M2 {};

D d;
M1* pm1 = &d;  // OK
M2* pm2 = &d;  // OK
N* pn = &d;    // chyba!
V* pv = &d;    // OK
```

Pri pretypovaní ukazovateľa na objekt `d` na typ `N`, čo je inak povolené, keďže `N` je predkom `D`, dochádza k dvojznačnosti, pretože nie je jasné, či má pretypovaný ukazovateľ ukazovať na objekt `N` zdedený cez triedu `M1` alebo cez triedu `M2`.

Virtuálne triedy trochu menia poradie volania konštruktorov, resp. deštruktorov pri inicializácii objektov odvodených tried, pretože ich konštruktory sú volané *pred* konštruktormi nevirtuálnych predkov v poradí najlepšie charakterizovanom slovami „depth-first left-to-right traversal“ spomínaného orientovaného acyklického grafu (DAG). Myslím, že by bolo zbytočné vysvetľovať, čo znamená depth-first prechod grafu, nemáme tu na to priestor ani čas. V prípade, že by to niekoho vážne zaujímalo, isto si dokáže nájsť príslušnú literatúru. Deštruktory sú, samozrejme, volané v presne opačnom poradí.

## Dvadsiata časť: PREKRÝVANIE FUNKCIÍ

V závere predošlej časti sme si sľúbili, že sa v tomto pokračovaní budeme zaoberať prekrývaním štandardných operátorov pre objektové údajové typy. Tento sľub aj splníme, ale budeme hovoriť o prekrývaní vo všeobecnom zmysle, t. j. o prekrývaní funkcií ako takých. Iste, čo – to sme si už o možnosti existencie funkčných homoným v minulosti povedali, ale až teraz vďaka znalostiam tried a práce s nimi môžeme celú problematiku zosumarizovať.

### Základné informácie

Prekrývanie funkcií – čo to vlastne znamená? O dvoch (a viacerých) funkciách hovoríme, že sa prekrývajú, ak majú rovnaké meno, ale rôznu deklaráciu, resp. rôznu funkčnú signatúru. Signatúra každej funkcie je daná počtom a typmi jej formálnych argumentov a typom návratovej hodnoty. Na základe počtu a typov skutočných argumentov prekladač pri použití volania funkcie vyberie jej správnu verziu. Príklad:

```
int max(int, int);
double max(double, double);

int i = max(1, 2);
double d = max(3.0, 4.0);
```

V prvom prípade sa zavolá funkcia `int max(int, int)`, v druhom, naopak, funkcia `double max(double, double)`.

Funkčné homonymá, ako sa prekrytým funkciám niekedy hovorí, musia mať svoje signatúry „dostatočne“ rozdielne. Čo znamená dostatočne, to si hneď vysvetlíme. Prekladač musí pri určovaní správnej verzie funkcie pre každý skutočný argument určiť jeho typ, na základe ktorého bude vyberať, ktorú prekrytú funkciu použije.

Majme ľubovoľný typ `T`. Dve funkcie, ktorých argument(y) sa líšia iba v tom, že jeden je typu `T` a druhý typu `T&`, nemôžu mať rovnaké meno, pretože oba typy `T` aj `T&` sú inicializované rovnakou množinou hodnôt a prekladač medzi nimi nedokáže rozlišovať:

```
void f(double);
void f(double&);  // chyba
```

Keby sme funkciu `f()` zavolali napríklad takto:

```
double d = 1.234;
f(d);
```

prekladač nemá ako zistiť, ktorú verziu `f()` vlastne voláme a či má argument `d` odovzdať hodnotou alebo odkazom.

Podobne sa dve prekryté funkcie nesmú líšiť iba tým, že jedna má argument typu `T` a druhá typu `const T`, resp. `volatile T`. Dôvod je rovnaký ako v predchádzajúcom prípade: všetky tri variácie sú inicializované rovnakou množinou hodnôt (nezabúdajme, že premennú typu `const T`, resp. `volatile T` môžeme bez

problémov inicializovať premennou typu `T` a naopak). Navyše špecifikácia formálneho argumentu ako `const`/`volatile` hovorí len o tom, že chceme, aby sa daný argument bral v tele funkcie ako konštantná/`volatile` lokálna premenná nezávisle od toho, akou hodnotou bol inicializovaný.

Je však možné, aby sa dve prekryté funkcie líšili tak, že jedna bude mať argument typu `T&` a druhá typu `const T&`, resp. `volatile T&`. V takomto prípade sa skutočný argument odovzdáva odkazom. Prekladač vie, či je tento argument konštantný/`volatile`, a príslušný variant funkcie vyberie podľa toho. Napríklad majme takýto kód:

```
void f(int&);
void f(const int&);

int a = 1;
const int b = 2;
volatile int c = 3;

f(a);   // OK
f(b);   // OK
f(c);   // chyba
```

Pri prvom volaní funkcie `f()` prekladač vyberie verziu `f(int&)`, pri druhom verziu `f(const int&)` (oboje podľa typu premenných `a`, resp. `b`), pri treťom volaní by mal ohlásiť chybu – neexistenciu príslušnej funkcie. Niektoré staršie prekladače však budú hlásiť niečo ako nejednoznačnosť pri výbere funkcie. Premennou c typu `volatile int` môžeme totiž rovnako dobre inicializovať formálny argument typu `int&` aj typu `const int&`, takže prekladač nedokáže určiť, ktorú verziu funkcie `f()` chceme volať. Podobná situácia nastane v prípade, že definujeme druhú verziu funkcie `f()` s argumentom typu `volatile int&` – prekladač ohlási chybu pri volaní `f(b)`. Ak chceme eliminovať takéto nejednoznačnosti, musíme definovať tri verzie `f()`, každú pre príslušný špecifikátor.

Všetko, čo sme uviedli v predchádzajúcom odseku, platí rovnakým spôsobom pre typy `T*`, `const T*` a `volatile T*`.

Funkcie, ktoré sa líšia len typom návratovej hodnoty, nesmú byť prekryté. Je to triviálne, pretože stačí nepoužiť návratovú hodnotu funkcie a prekladač nemá ako zistiť, ktorú verziu voláme.

Trieda nesmie mať dve členské funkcie, líšiace sa len v tom, že jedna je statická a druhá nie. Opäť totiž neexistuje spôsob, ako ich odlíšiť v prípade, že ich voláme prostredníctvom operátora `.` alebo `->` nad inštanciou tejto triedy.

Typy, definované pomocou špecifikátora `typedef`, nie sú samostatnými typmi, ale len synonymami alebo aliasmi pre už existujúce typy. Preto dve funkcie, líšiace sa len takýmto spôsobom, nemôžu mať rovnaké meno:

```
typedef char* pchar;

void f(char*);
void f(pchar);  // chyba
```

Typy, definované ako enumerácie, však už sú samostatnými typmi, a preto môžu byť použité na rozlišovanie funkčných homoným:

```
enum E { a, b, c };

void f(int i);
void f(E i);  // OK
```

Isto si spomínate, že pri deklarácii ukazovateľov ako argumentov funkcií môžeme použiť dvojakú syntax – zápis `T[]` je ekvivalentný zápisu `T*`. Preto funkcie, ktorých argumenty sa líšia len takýmto spôsobom, nemôžu byť prekryté. Pri viacrozmerných poliach sa samozrejme toto pravidlo vzťahuje len na prvý rozmer, ostatné rozmery sú pri rozlišovaní typov signifikantné:

```
void f(char*);
void f(char[]);    // chyba
void f(char[10]); // chyba

void g(char(*)[5]);
void g(char[3][5]);  // chyba
void g(char(*)[10]); // OK
```

Pozor ale na správnu deklaráciu; ako vidíme v poslednom príklade, ekvivalentom typu `char[3][5]` (pole troch päťprvkových znakových polí) je typ `char(*)[5]` (ukazovateľ na päťprvkové znakové pole) a nie typ `char*[5]` (pole piatich ukazovateľov na `char`, inak tiež reprezentované typom `char**` – ukazovateľom na ukazovateľ na `char`).

Základným predpokladom pre prekrývanie dvoch funkcií je ich existencia v rovnakom rozsahu platnosti. Už minule sme si spomínali, že členská funkcia odvodenej triedy s rovnakým menom ako funkcia základnej triedy túto zdedenú funkciu neprekrýva, ale ju namiesto toho skrýva. (Napohľad ide o hru so slovíčkami, ale to je len určitá neobratnosť slovenčiny pri preklade z angličtiny – termín *prekrývanie* je ekvivalentom anglického *overloading*, ktoré sa bežne prekladá ako preťažovanie, čo však podľa mňa nevystihuje podstatu celého mechanizmu. Problém vzniká pri snahe o preklad anglického *overriding*, ktoré opisuje existenciu virtuálnych funkcií s rovnakým menom, ktoré majú rovnakú signatúru a pri volaní sa rozlišujú až za behu programu podľa skutočného typu inštancie, nad ktorou boli vyvolané. Pre tento jav som v predošlých častiach použil výraz *maskovanie*. Tretí výraz, prekladaný ako zakrývanie či skrývanie, je ekvivalentom anglického *hiding*. Koniec jazykového okienka.) Majme príklad:

```
class B
{
public:
  void f(int);
};

class D : public B
{
public:
  void f(char*);
};
```

Keďže obe funkcie `f()` sú každá v inom rozsahu platnosti, nie sú prekryté. Funkcia `D::f()` zakrýva funkciu `B::f()`, ktorá je dostupná len pod svojím kvalifikovaným menom:

```
D* pd = new D;
pd->f(5);       // chyba
pd->B::f(5);    // OK
pd->f(„foo“);  // OK
```

Dve členské funkcie v tej istej triede môžu byť prekryté, pretože sa nachádzajú v rovnakom rozsahu platnosti. Každá z oboch funkcií dokonca môže mať rozdielne prístupové práva:

```
class C
{
private:
  void f(int);
public:
  void f(double);
};
```

## Párovanie argumentov

Predstavme si, čo treba urobiť pri výbere správnej verzie prekrytej funkcie. Jediné, čo má prekladač k dispozícii, je meno prekrytej funkcie a počet a typy skutočných argumentov. Na základe mena funkcie určí množinu prekrytých funkcií, z ktorých si bude v ďalších krokoch vyberať.

Prekladač sa snaží pre každý skutočný argument vybrať takú množinu funkcií, ktoré vykazujú najlepšiu zhodu, čo sa týka typu formálneho argumentu na danej pozícii. Prienik týchto množín určuje funkciu, ktorá sa zavolá. Ak sa stane, že prienik množín obsahuje viac ako jednu funkciu, prekladač ohlási chybu nejednoznačnosti volania. V prípade, že prienikom je prázdna množina, dôjde takisto k chybe, spôsobenej neexistenciou vhodného funkčného homonyma. Vybraná funkcia musí však navyše aspoň pre jeden argument vykazovať striktne lepšiu zhodu ako všetky jej „kolegyne“.

Zostáva osvetliť, čo považujeme za najlepšiu zhodu formálneho a skutočného argumentu. Zrejme úplne najlepší prípad nastane vtedy, keď je skutočný argument rovnakého typu ako príslušný formálny argument. To však vo všeobecnosti nemusí byť pravda a vtedy sa hľadá najlepšia sekvencia konverzií skutočného argumentu na typ formálneho argumentu. Najlepšia preto, lebo konverzných sekvencií môže byť viac, podľa toho, aké typy, konverzné konštruktory či operátory máme v programe definované. Pri hľadaní konverzných sekvencií sa nebude uvažovať taká sekvencia, ktorá obsahuje viac ako jednu používateľskú konverziu (t. j. konverziu pomocou konštruktora alebo pomocou prekrytého operátora pretypovania), a ani sekvencia, z ktorej možno vynechať niektoré kroky bez zmeny sémantiky (napríklad sekvenciu `int->float->double` možno skrátiť na `int->double`).

Všetky uvažované sekvencie možno zoradiť do usporiadanej postupnosti, určujúcej ich prioritu pri výbere najlepšej z nich. Táto postupnosť má päť úrovní:

1. Sekvencie, ktoré obsahujú buď žiadnu, alebo jednu a viac tzv. triviálnych konverzií, sú lepšie ako všetky ostatné. Žiadna konverzia znamená, že ide o presnú zhodu typu formálneho a skutočného argumentu. Za triviálne konverzie sa považujú nasledujúce: z typu `T` na typ `T&` a naopak; z typu `T[]` na typ `T`; z typu `T()` na typ `T(*)()`; z typu `T` na typ `const T`, resp. `volatile T` a konečne z typu `T*` na typ `const T*`, resp. `volatile T*`.

2. Zo sekvencií iných ako v bode 1 tie, ktoré obsahujú iba celočíselné rozšírenia, konverzie z `float` na `double` a triviálne konverzie, sú lepšie ako všetky ostatné.

3. Zo sekvencií iných ako v bode 2 tie, ktoré obsahujú iba štandardné a triviálne konverzie, sú lepšie ako ostatné. V rámci štandardných konverzií sa považuje za lepšiu konverzia ukazovateľa na odvodenú triedu na ukazovateľ na základnú triedu ako na ukazovateľ na typ `void*` a ďalej je lepšia konverzia ukazovateľa/referencie na odvodenú triedu na ukazovateľ/referenciu na bližšiu základnú triedu (v zmysle hierarchie tried) ako na vzdialenejšiu.

4. Zo sekvencií iných ako v bode 3 tie, ktoré obsahujú iba používateľské, štandardné a triviálne konverzie, sú lepšie ako ostatné.

5. A nakoniec sekvencie, ktoré zahŕňajú konverziu na formálny argument zahrnutý vo výpustke (`...`), sa považujú za najhoršie.

Priznávam, že predchádzajúce riadky nie sú práve najzrozumiteľnejšie, ale, bohužiaľ, postup pri párovaní argumentov je normatívne daný a nedá sa nijako oklamať. Bolo by asi vhodné uviesť nejaký príklad, ale vzhľadom na variabilitu typov C++ existuje nekonečné množstvo variácií, ktoré všetky nemožno pokryť pár príkladmi, preto skutočne len na ilustráciu:

```
void f(double, char*);
void f(long, char*);

f(1, „foo“);
f(100U, „bar“);
f(3.5, „xyz“);
```

Prvé volanie funkcie `f()` vyberie jej verziu `f(long, char*)`, pretože konverzia konštanty `1` typu `int` na typ `long` je lepšia ako konverzia na `double`. Druhé volanie vyberie rovnakú verziu, pretože konverzia konštanty `100` typu `unsigned int` na typ `long` je stále lepšia ako konverzia na typ `double`. Nakoniec tretie volanie vyberie verziu `f(double, char*)`, pretože tentoraz konštanta `3.5` vykazuje presnú zhodu s typom `double`.

## Prekrývanie operátorov

Konečne sa dostávame k trochu zaujímavejšej a opäť objektovej téme – prekrývaniu štandardných operátorov vlastnými, používateľskými verziami. Prekrytie štandardného operátora v podstate modifikuje jeho význam pri použití s vlastnými, objektovými typmi. Prekrývaním operátorov však nemôžeme zmeniť ich význam pre štandardné typy C++, takisto nemôžeme zmeniť ich aritu (t. j. to, či sú unárne, binárne, ternárne), prioritu, asociativitu ani prefixovosť/postfixovosť. Nemôžeme si ani zadefinovať nové operátory (napr. nemôžeme zobrať znak `$` a definovať ho ako operátor).

Prekryť môžeme takmer všetky operátory C++ s výnimkou nasledujúcich:

```
.
.*
::
?:
sizeof
```

Operátor `.*` zatiaľ ešte nepoznáme, ale povieme si o ňom na záver tejto časti. Prekryť takisto nemôžeme operátory preprocesora `#`, `##` a `defined`.

Funkcie, ktorými prekrývame pôvodné definície operátorov, musia mať tvar `operator` *meno* `(` *argumenty* `)`, kde *meno* je symbol príslušného operátora. Počet, poradie a typy argumentov sú špecifické pre jednotlivé operátory.

Prekryté operátorové funkcie musia byť buď členskými funkciami nejakej triedy, alebo musia mať jeden z argumentov typu triedy/enumerácie alebo referencie na triedu/enumeráciu. Členské operátorové funkcie sa dedia bežným spôsobom s výnimkou funkcie `operator=()`. Nečlenské operátorové funkcie sa často deklarujú ako spriatelené danej triede/triedam.

Štandardné operátory vykazujú určitú ekvivalenciu v použití, napr. výraz `++x` je ekvivalentný výrazu `x+=1`. Táto ekvivalencia nemusí byť dodržaná pre používateľské verzie operátorov, a preto je dobré nespoliehať sa na ňu pri používaní cudzích tried.

V nasledujúcich odsekoch si preberieme jednotlivé typy operátorov a špecifiká ich prekrývania. Väčšinu operátorov budeme dopĺňať do triedy `complex`, ktorú sme si definovali už minule a ktorá reprezentuje údajový typ komplexné číslo. Tu je jej základná deklarácia:

```
class complex
{
  double re, im;
public:
  complex(double r = 0.0, double i = 0.0)
: re(r), im(i) {}
  complex(complex& c) : re(c.re), im(c.im) {}
  double real() { return re; }
  double imag() { return im; }
};
```

## Unárne operátory

Prefixové unárne operátory môžeme pretypovať buď nestatickou členskou funkciou bez argumentov, alebo nečlenskou funkciou s jedným argumentom. V prvom prípade bude výraz `@x` pre ľubovoľný unárny operátor `@` interpretovaný ako `x.operator@()`, v druhom prípade ako `operator@(x)`. V prípade existencie oboch verzií sa uplatní klasické rozlišovanie prekrytých funkcií na základe párovania argumentov.

Zaveďme do našej triedy `complex` unárny operátor `-`, ktorý bude vracať opačné číslo k danému komplexnému číslu (opačné číslo k číslu $a + bi$ je číslo $-a - bi$):

```
complex complex::operator-()
{ return complex(-re, -im); }
```

(Nezabudnite doplniť do deklarácie triedy príslušný prototyp!) Všimnite si, že sme návratovú hodnotu našej verzie operátora `-` deklarovali ako `complex`. Je dobrým zvykom dodržiavať pravidlá zavedené štandardnými operátormi v tom zmysle, že ak štandardná verzia vracia l-hodnotu, mala by ju vracať aj používateľská verzia, takisto ak štandardná verzia nevracia l-hodnotu, nemala by ju vracať ani používateľská verzia. L-hodnotu obyčajne vracajú tie operátory, ktoré akýmkoľvek spôsobom menia svoje operandy (ako napríklad `=`, `+=`, `++` a pod.). V našom príklade vraciame ako návratovú hodnotu dočasnú inštanciu triedy `complex`, takže ani nemá zmysel vracať ju odkazom.

Vďaka prekrytému operátoru `-` teraz môžeme získať opačné číslo k nejakému komplexnému číslu veľmi jednoducho:

```
complex c1(1, 2);
complex c2 = -c1;
```

Teraz si skúste sami do triedy `complex` doplniť vlastnú verziu operátora `~`, ktorá bude vracať k danému komplexnému číslu číslo k nemu komplexne združené. Ak by ste mali problémy, v predposlednej časti seriálu sme definovali členskú funkciu `complex::conj()` s rovnakým významom.

## Operátory `++` a `--`

S unárnymi operátormi `++` a `--` je to o niečo zložitejšie, pretože oba majú dve verzie – prefixovú a postfixovú. Prefixovú verziu prekrývame buď pomocou členskej funkcie bez argumentov, alebo pomocou nečlenskej funkcie s jedným argumentom typu danej triedy, teda rovnakým spôsobom ako iné unárne operátory. Pri definícii prekrytej postfixovej verzie však musíme doplniť fiktívny argument typu `int`, ktorý sa nijako nepoužíva, pri volaní má nulovú hodnotu a slúži len na rozlíšenie oboch verzií operátorov.

Prefixová a postfixová štandardná verzia oboch operátorov sa líši aj v tom, že prvá z nich vracia l-hodnotu, zatiaľ čo druhá nie. Ak chceme dodržať túto konvenciu, musíme patrične deklarovať návratovú hodnotu prekrytých verzií. Vo všeobecnosti, ak chceme vracať l-hodnotu, stačí deklarovať návratovú hodnotu ako referenciu. Musíme si však dať pozor na to, aby sme touto referenciou skutočne vracali objekt, nad ktorým pracujeme, a nie nejakú dočasnú lokálnu premennú. V členských funkciách je to jednoduché – aktuálnu inštanciu vrátime výrazom `return *this;`. V prípade nečlenských funkcií však pozor: ak chceme objekt modifikovať, musí príslušná operátorová funkcia mať za argument referenciu na menený objekt. Túto referenciu potom vrátime bežným spôsobom.

Pre objasnenie si teraz ukážeme na príklade rozdiely medzi jednotlivými verziami. Operátory, ktoré definujeme, budú tak trochu za vlasy pritiahnuté, pretože budú inkrementovať/dekrementovať len reálnu zložku daného komplexného čísla, ale to na celom príklade nič nemení. Najprv teda obe verzie operátora `++` ako členské funkcie:

```
complex& complex::operator++()
{
  re++;
  return *this;
}

complex complex::operator++(int)
{
  complex tmp(*this);
  re++;
  return tmp;
}
```

Všimnite si, že v prípade postfixovej verzie musíme vytvoriť pomocnú premennú `tmp`, ako kópiu súčasného stavu, ktorú vrátime po tom, čo modifikujeme objekt `*this`. Tak je totiž definovaná sémantika postfixového operátora `++`.

A tu sú obe verzie operátora `--`, tentoraz ako nečlenské funkcie:

```
complex& operator--(complex& c)
{
  c.re--;
  return c;
}

complex operator--(complex& c, int)
{
  complex tmp(c);
  c.re--;
  return tmp;
}
```

Na to, aby prekladač obe funkcie správne preložil, je však potrebné do deklarácie triedy `complex` doplniť tieto dva riadky:

```
friend complex& operator--(complex&);
friend complex operator--(complex&, int);
```

## Binárne operátory

Binárne operátory môžeme pretypovať buď nestatickou členskou funkciou s jedným argumentom, alebo nečlenskou funkciou s dvoma argumentmi. V prvom prípade bude výraz `x@y` pre ľubovoľný binárny operátor `@` interpretovaný ako `x.operator@(y)`, v druhom prípade ako `operator@(x,y)`. V prípade existencie oboch verzií sa opäť uplatní rozlišovanie prekrytých funkcií na základe párovania argumentov.

Ako príklad si zavedieme do našej triedy `complex` operátor `*` realizujúci bežné násobenie komplexných čísel. Pre zopakovanie: súčin čísel $a + bi$ a $c + di$ je rovný $(ac - bd) + (ad + bc)i$:

```
complex complex::operator*(complex c)
{
  double newre = re * c.re - im * c.im;
  double newim = re * c.im + im * c.re;
  return complex(newre, newim);
}
```

Premenné `newre` a `newim` sme zaviedli iba pre sprehľadnenie zápisu. Všimnite si, že návratová hodnota nášho operátora nie je l-hodnotou, tak ako je to zvykom pri štandardnom operátore `*`. Argument nášho operátora je odovzdávaný hodnotou. Ak sa chceme vyhnúť v podstate zbytočnému volaniu konštruktora, môžeme ho odovzdať aj odkazom, i keď v tele operátora pôvodný objekt nemeníme. Aby sme však aj v budúcnosti zabránili nechcenej modifikácii pôvodného objektu, zmeníme deklaráciu operátora `*` takto:

```
complex complex::operator*(const complex& c)
{ ... }
```

Takto zabezpečíme, že sa pri vyvolaní operátora `*` nad dvoma komplexnými číslami nebude zbytočne konštruovať nový objekt, predstavujúci kópiu jedného z operandov, ktorý aj tak slúži iba na čítanie údajových členov. Podobný prístup môžeme použiť pri nečlenskej verzii operátorovej funkcie.

Samozrejme, že nemusíme definovať iba operátor pre násobenie dvoch komplexných čísel. Rovnakým spôso-

bom môžeme zadefinovať napríklad operátor násobenia komplexného čísla reálnym. Prototyp takej funkcie bude nasledujúci:

```
complex complex::operator*(double d);
```

Definíciu tela ponechám na vás. Ale pozor: takto definovaný operátor funguje iba pre násobenie komplexného čísla reálnym sprava. Pokiaľ chceme realizovať aj násobenie zľava, musíme už definovať nečlenskú funkciu s prototypom:

```
complex operator*(double d, complex c);
```

Iste ste si všimli, že nám tento mechanizmus umožňuje vytvoriť dva rôzne varianty násobenia komplexného a reálneho čísla podľa ich vzájomného poradia, čím úspešne zrušíme komutatívnosť operátora *. Takýto prístup sa vo všeobecnosti neodporúča.

Medzi binárne operátory, ktoré možno pretypovať uvedeným spôsobom, patria ďalej operátory +, -, /, %, ^, &, |, <, >, <=, >=, ==, !=, &&, ||, <<, >> a operátor , (čiarka). Pozor však pri operátoroch &&, || a čiarka. Ich používateľské verzie totiž nemajú zaručené špeciálne poradie vyhodnocovania operandov!

Na precvičenie si skúste do triedy `complex` doplniť tie binárne operátory, ktoré majú pre komplexné čísla nejaký praktický význam.

## Operátor priradenia

Priraďovací operátor má zvláštne postavenie. Môže byť definovaný iba ako členská funkcia s jediným argumentom, nededí sa a v prípade jeho neexistencie si ho prekladač dokáže doplniť sám. Implicitný operátor priradenia vytvára plytkú kópiu, t. j. kopíruje stav objektu člen po člene, použijúc pre objektové členy ich operátory priradenia. Jeho prototyp bude pre triedu `X` vyzerať takto:

```
X& X::operator=(const X&);
```

V prípade, že trieda obsahuje objektové členy, ktorých priraďovacie operátory neakceptujú konštantné argumenty, mení sa tento prototyp na nasledujúci:

```
X& X::operator=(X&);
```

Implicitný operátor priradenia nie je možné vytvoriť v prípade, že trieda obsahuje konštantné členy, referencie alebo objektové členy s privátnym priraďovacím operátorom.

Operátor priradenia sa vyvolá vždy, keď prekladač narazí na výraz `x = e`, kde `x` je inštancia objektového typu, `e` je výraz. Ako už vieme, deklarácia spojená s inicializáciou má za následok vyvolanie kopírovacieho konštruktora, a nie priraďovacieho operátora.

Pre našu triedu `complex` nemá zmysel deklarovať samostatný priraďovací operátor, úplne postačí ten implicitný. Pre ilustráciu si však ukážeme, ako by taká deklarácia vyzerala:

```
complex& complex::operator=(const complex& c)
{
  re = c.re;
  im = c.im;
}
```

Všimnime si, že ak budeme vracať výsledok nie odkazom, ale iba hodnotou, nebude možné reťaziť priraďovacie operátory spôsobom `x = y = z`.

## Operátor volania funkcie

Volanie funkcie v tvare:

*výraz* ( *zoznam-argumentov* ) ;

sa považuje za binárny operátor, ktorého prvým operandom je *výraz* a druhým *zoznam-argumentov*. Prekryť ho môžeme iba nestatickou členskou funkciou s názvom `operator()`. Volanie `x(arg)` je potom ekvivalentné zápisu `x.operator()(arg)`, kde `x` je inštancia triedy.

Ako príklad si uvedieme triedu `matrix`, reprezentujúcu maticu údajov. Jej podstatným údajovým členom bude dvojrozmerné pole prvkov typu `double`. Pre jednoduchosť nebudeme zavádzať komplikované konštruktory s dynamickou alokáciou poľa, ale budeme mať pevne dané rozmery matice. Dôležitý bude totiž v triede `matrix` prekrytý operátor volania funkcie, ktorý bude slúžiť na prístup k jednotlivým prvkom matice. Jeho dva argumenty budú predstavovať požadované indexy:

```
const int N = 5;

class matrix
{
  double mm[N][N];
public:
  double& operator()(int i, int j)
  { return mm[i][j]; }
};
```

Pre jednoduchosť netestujeme, či oba indexy nepresahujú povolený rozsah. Zavedený operátor `()` nám umožňuje pracovať s prvkom $m_{ij}$ matice pomocou zápisu `m(i, j)`:

```
matrix m;
m(3, 2) = 10;
```

Priradenie je možné, pretože operátor `()` vracia l-hodnotu (typ `double&`).

## Operátor indexovania

Podobne ako v predchádzajúcom prípade je indexovanie poľa v tvare:

*výraz* [ *výraz* ] ;

považované za binárny operátor. Výraz `x[y]` je interpretovaný ako `x.operator[](y)`, kde `x` je inštancia objektového typu. Prekrytý operátor `[]` musí byť nestatickou členskou funkciou.

Ako príklad si uvedieme triedu `vector`, čo je vlastne jednorozmerný prípad predchádzajúcej triedy `matrix`:

```
class vector
{
  double vv[N];
public:
  double& operator[](int i)
  { return vv[i]; }
};
```

Podobnosť s triedou `matrix` je zrejmá. Pre prístup k prvkom vektora však tentoraz používame operátor `[]`. V triede `matrix` sme museli použiť operátor `()` z toho dôvodu, že prvky matice sú prístupné pomocou dvoch indexov. Je, samozrejme, možné aj také riešenie, že vytvoríme triedu `matrix`, ktorá bude obsahovať jednorozmerné pole prvkov typu `vector`. Potom môžeme prekryť operátor `[]`, ktorý vráti príslušný riadok (či stĺpec – podľa toho, ako maticu definujeme). Na tento výsledok typu `vector` sa potom aplikuje operátor `[]` s druhým indexom.

## Operátor prístupu k členom triedy

Ďalším špecifickým operátorom je operátor `->`. Ten je považovaný za unárny v tom zmysle, že výraz `x->m` je interpretovaný ako `(x.operator->())->m`. Operátorová funkcia `operator->()` musí byť nestatickou členskou funkciou a musí vracať buď ukazovateľ na nejakú triedu, alebo objekt takej triedy, pre ktorú je definovaný operátor `->` (resp. referenciu na takúto triedu).

Prekrytý operátor `->` sa obyčajne používa pri triedach, ktoré zapuzdrujú nejaké ukazovatele. Typický príklad:

```
class cptr
{
  complex* p;
public:
  complex* operator->()
  { return p; }
};
```

Z deklarácie triedy `cptr` síce nevidieť, prečo by sme nemohli rovno používať typ `complex*`, ale zmysel táto trieda dostane v okamihu, keď doplníme konštruktor a deštruktor, ktoré budú automaticky alokovať/dealokovať objekt triedy `complex`. Trieda `cptr` potom bude predstavovať tzv. automatický ukazovateľ. Takéto ukazovatele sa používajú pomerne často a šablóny pre ne sú dokonca súčasťou štandardnej knižnice C++.

## Operátory alokácie a dealokácie

Na alokáciu, resp. dealokáciu dynamických objektov slúžia v C++ operátory `new` a `delete`, s ktorými sú zviazané operátorové funkcie `operator new()` a `operator delete()`. Tieto funkcie však pracujú trochu ináč, ako je to zvykom pri ostatných prekrytých operátoroch.

Operátor `new` volá na alokovanie príslušného miesta v pamäti funkciu `operator new()`. Pre neobjektové typy sa použije jej globálna verzia `::operator new()`, pre inštancie triedy `X` sa použije verzia `X::operator new()` – samozrejme, len vtedy, ak sme ju definovali. Použitie globálnej verzie aj v prípade objektových typov si môžeme vynútiť pripojením operátora `::` pred kľúčové slovo `new`.

Funkcia `operator new()` je v prípade objektových typov vždy statická, aj keď takto nie je deklarovaná. Jej návratová hodnota musí byť typu `void*` a jej prvý argument, predstavujúci veľkosť alokovaného miesta v pamäti, musí byť typu `size_t`. Prípadné ďalšie argumenty sú voliteľné, pri volaní operátora `new` ich uvádzame v zátvorke za kľúčovým slovom `new` (pred prípadným inicializátorom). Klasickým príkladom je definícia takého operátora `new`, ktorý umiestni nový objekt na nami určené miesto v pamäti:

```
void* operator new(size_t, void* ptr)
{ return ptr; }
```

Ak teraz chceme vytvoriť novú premennú typu `complex`, umiestnenú v nejakej pamäťovej oblasti `buf`, použijeme takýto zápis:

```
char* buf[512];
complex* cp = new(buf) complex(1.2, 3.4);
```

Na takto získaný ukazovateľ však nesmieme použiť štandardný operátor `delete`, ten by sa totiž snažil danú oblasť pamäte bežným spôsobom dealokovať.

Ak funkcia `operator new()` nedokáže alokovať dostatočne veľkú oblasť pamäte, vracia `0`. Vtedy aj samotný operátor `new` vracia nulu, ako oznámenie neúspechu pri alokácii.

Podobne operátor `delete` volá pre dealokáciu obsadenej pamäte funkciu `operator delete()`. Opäť sa pre neobjektové typy použije jej globálna verzia `::operator delete()` a pre inštancie triedy `X` verzia `X::operator delete()`, pokiaľ je definovaná. Zápis `::delete` si vynúti použitie globálnej verzie aj pre objektové typy.

Aj funkcia `operator delete()` je statická, jej návratová hodnota musí byť `void` a jej prvý argument,

reprezentujúci ukazovateľ na dealokovanú pamäť, musí byť typu `void*`. Navyše môžeme pridať druhý argument typu `size_t`, ktorý bude predstavovať veľkosť dealokovaného objektu. V rámci jednej triedy môže existovať iba jediná funkcia `operator delete()`.

Keďže sú členské funkcie `operator new()` a `operator delete()` statické, nemôžu byť virtuálne. Nájdenie ich správnych verzií pre daný objekt je zabezpečené jedinečnosťou konštruktora a virtuálnosťou deštruktora.

Podľa najnovšej normy ANSI C++ môžeme v programe definovať aj vlastnú verziu funkcií pre alokáciu/dealokáciu polí. Tieto funkcie majú názov `operator new[]()`, resp. `operator delete[]()`.

### Ukazovatele na členy tried

Dosiaľ sme si nič nehovorili o ukazovateľoch na členy triedy. Tento údajový typ je zvláštnym prípadom klasických ukazovateľov, s tým obmedzením, že nemôže ukazovať na ľubovoľnú premennú, resp. oblasť pamäte, ale len na člena (údajového či funkčného) tej – ktorej triedy. Každý takýto ukazovateľ je zviazaný so „svojou" triedou a ukazovateľ do triedy `A` nemôže ukazovať na členy triedy `B`.

Deklarácia členských ukazovateľov je na prvý pohľad trochu mätúca. Jej všeobecný zápis vyzerá takto:

T *meno-triedy* :: * *cv-kvalifikátory* D1

`T` je bežný špecifikátor typu. *Meno-triedy* určuje triedu, do ktorej daný ukazovateľ bude ukazovať, *cv-kvalifikátory* (`const` a `volatile`) umožňujú deklarovať ukazovateľ ako konštantný (nie na konštantu!), resp. `volatile`. `D1` je ďalší deklarátor – používame rovnakú schému zápisu ako v časti venovanej deklaráciám. Celý typ deklarovaného identifikátora vnoreného v `D1` je „... *cv-kvalifikovaný* ukazovateľ na člena triedy *meno-triedy* typu `T`".

Ukážme si príklad. Majme jednoduchú triedu:

```
class C
{
public:
  int a, b;
  int f(double);
};
```

Teraz môžeme deklarovať napríklad ukazovateľ na celočíselný člen triedy `C`:

```
int C::* pi;
```

a prinútiť ho ukazovať na člen `a` alebo na člen `b` triedy `C`:

```
pi = &C::a;
pi = &C::b;
```

Všimnite si, že sme dosiaľ nedeklarovali žiadnu inštanciu triedy `C`, ukazovateľ `pi` ukazuje všeobecne na člen `a`, resp. `b` triedy `C`. Podobne môžeme deklarovať ukazovateľ na niektorú členskú funkciu:

```
int (C::* pf)(double) = &C::f;
```

Ukazovateľ `pf` je ukazovateľom na takú členskú funkciu triedy `C`, ktorá má jediný argument typu `double` a vracia typ `int`. Týmto podmienkam vyhovuje členská funkcia `C::f()`, preto sme aj ukazovateľ na ňu priradili do `pf`. Keď si dobre preštudujete spôsob deklarácie členských ukazovateľov, všimnete si, že sa vlastne deklarujú ako bežné ukazovatele, len pred hviezdičkou je navyše meno ich „domácej" triedy so štvorbodkou.

Samozrejme, na to, aby nám členské ukazovatele boli na niečo užitočné, musíme ich dokázať aj dereferencovať. Na to máme k dispozícii dva operátory `.*` a `->*`. Oba sú binárne, infixové, asociujú sa zľava doprava a majú prioritu menšiu než unárne operátory, ale väčšiu než multiplikatívne operátory. Ak ich pravý operand je ukazovateľom na člena triedy `C`, ich ľavý operand musí byť typu triedy `C` alebo jej dostupného predka (to pre operátor `.*`), resp. typu ukazovateľa na triedu `C` alebo jej dostupného predka (pre operátor `->*`).

Použitie oboch operátorov sa už musí vzťahovať na konkrétnu inštanciu objektového typu. Výsledkom aplikácie operátorov je príslušný údajový člen či členská funkcia tej inštancie, na ktorú boli aplikované. Majme deklarovanú triedu `C` a ukazovatele `pi` a `pf` ako predtým:

```
C c;
c.*pi = 12;
C* pc = &c;
int i = (pc->*pf)(3.14);
```

V tomto príklade priraďujeme hodnotu `12` tomu údajovému členu inštancie `c` triedy `C`, na ktorý ukazuje ukazovateľ `pi`. Ďalej voláme s parametrom `3.14` tú členskú funkciu, na ktorú ukazuje ukazovateľ `pf`.

Na ukazovatele do tried sa vzťahuje niekoľko pravidiel. Predovšetkým tieto ukazovatele nemôžu ukazovať na statické členy tried. Ak ukazovateľ ukazuje na funkciu, jeho dereferencovanú hodnotu možno použiť iba na realizáciu funkčného volania. Ukazovateľ na člena základnej triedy možno bez problémov konvertovať na ukazovateľ na člena triedy odvodenej za predpokladu, že ukazovateľ na odvodenú triedu môžeme konvertovať na ukazovateľ na základnú triedu. Inými slovami, ukazovatele na členy tried je dovolené konvertovať presne opačným smerom ako ukazovatele na samotné triedy. Prečo je to tak? Zamyslite sa nad tým.

## Dvadsiata prvá časť: ŠABLÓNY A VÝNIMKY

Šablóny sú súčasťou jazyka už dávno a aj staršie prekladače dokážu s nimi pracovať; okrem toho je takmer celá štandardná knižnica C++ postavená na šablónach, takže porozumieť práci s nimi je nevyhnutné na využívanie všetkých možností, ktoré nám dnešné moderné prekladače C++ poskytujú. Okrem toho si povieme niečo o mechanizme generovania a spracovania výnimiek v C++ a aj o práci so štruktúrovanými výnimkami v jazyku C.

### Deklarácia šablón

Šablóna v C++ predstavuje množinu príbuzných deklarácií. Pomocou šablón môžeme deklarovať množinu funkcií alebo množinu tried. Každá takáto množina je parametrizovaná jedným či viacerými parametrami. Na rozdiel od parametrov funkcií parametrami šablón môžu byť aj názvy typov.

Deklarácia šablóny vyzerá takto:

`template <` *zoznam-argumentov* `>` *deklarácia*

*Zoznam-argumentov* je čiarkami oddelený zoznam deklarácií argumentov šablóny. Môže ísť jednak o klasické deklarácie argumentov, totožné s deklaráciami argumentov funkcií, a jednak o deklarácie typových argumentov v tvare:

`class` *identifikátor*

alebo

`typename` *identifikátor*

Druhý spôsob je novší a niektoré staršie prekladače ho nemusia podporovať.

*Deklarácia* je bežnou deklaráciou funkcie alebo triedy (pozri ďalej). Deklarácia šablóny sa môže vyskytovať iba na úrovni súboru (t. j. ako globálna deklarácia). Pre novo deklarované meno platia všetky bežné pravidlá pre rozsah platnosti a spôsob prístupu.

### Šablóny funkcií

Najprv sa budeme zaoberať deklaráciou šablón funkcií. Takáto deklarácia je v podstate ekvivalentná deklarácii nekonečnej (z hľadiska počtu existujúcich typov a ich variácií) množiny navzájom sa prekrývajúcich funkcií s rovnakým názvom, ale rozdielnym typom argumentov.

Typickým príkladom je funkcia `max()` s dvoma argumentmi rovnakého typu, ktorá vráti väčší z nich. Často potrebujeme viacero verzií takejto funkcie pre rôzne doménové typy. Samozrejme, vždy môžeme definovať niekoľko prekrytých funkcií, ako `int max(int, int)`, `double max(double, double)` atď., ale všetky tieto funkcie budú mať rovnaké telo – líšiť sa budú len typom argumentov a návratovej hodnoty. V takomto prípade je oveľa elegantnejšie deklarovať jedinú funkciu `max()` ako šablónu s jedným typovým argumentom:

```
template <class T>
T max(T a, T b)
{
  return (a > b) ? a : b;
}
```

Táto deklarácia (a súčasne definícia) negeneruje nijaký kód, je to skutočne iba šablóna na vytvorenie príslušnej funkcie v okamihu, keď ju bude treba. Takýto okamih nastane vtedy, keď použijeme funkciu `max()` niekde v rámci programu. Typ argumentu šablóny sa implicitne odvodí z typu použitých argumentov funkcie `max()`:

```
int c = max(2, 1);
double d = max(7.0, 8.0);
```

V prvom prípade sa vygeneruje a použije funkcia `max<int>()`, v druhom funkcia `max<double>()`. Tu vidíme výhodu šablónových funkcií: kód ich tela sa vygeneruje prekladačom iba vtedy, keď ho skutočne potrebujeme a použijeme.

Niekedy chceme prekladaču explicitne naznačiť, ktorú inštanciu má v danom prípade použiť. To môžeme zabezpečiť uvedením požadovaného typového argumentu do špicatých zátvoriek za názov funkcie:

```
int e = max<int>(3, 'z');
```

V tomto príklade je druhý argument funkcie `max()` typu `char`. Prekladač je schopný vygenerovať funkciu `max(int, int)` alebo `max(char, char)`, ale nie `max(int, char)`, preto by za normálnych okolností ohlásil chybu. Explicitné uvedenie `max<int>` spôsobí, že prekladač použije (zhodou okolností už predtým vygenerovanú) funkciu `max(int, int)`.

Vygenerovanie príslušnej funkcie môžeme zabezpečiť aj inak, bez toho, aby sme museli túto funkciu niekde volať. V našom príklade vytvoríme inštanciu `long max(long, long)` jednoducho takto:

```
template long max(long, long);
```

alebo aj takto:

```
template long max<long>(long, long);
```

Takéto explicitné vytváranie inštancií má zmysel napríklad vtedy, ak vytvárame .LIB súbory, v ktorých chceme

mať príslušnú inštanciu „natvrdo“, aby si ju nikto nemohol neskôr predefinovať.

Pýtate sa, či je možné predefinovať existujúcu funkciu? Odpoveď znie: áno, pokiaľ ide o takú inštanciu šablóny, ktorá dosiaľ nebola použitá a teda ani vygenerovaná. Ak totiž niekde v programe klasickým spôsobom deklarujeme a definujeme funkciu, ktorej prototyp sa bude zhodovať s prototypom inštancie niektorej šablónovej funkcie, bude túto novú funkciu prekladač považovať za právoplatnú inštanciu danej šablóny. V našom príklade môžeme teda pred použitím funkcie `max()` s argumentmi typu `double` uviesť takúto deklaráciu:

```
double max(double, double);
```

a prekladač namiesto vygenerovania novej inštancie `double max<double>(double, double)` podľa šablóny použije nezávisle definovanú funkciu `double max(double, double)`.

Zavedenie šablónových funkcií mierne modifikuje pravidlá hľadania tej verzie prekrytých funkcií, ktorá sa použije pri danom volaní (pozri predchádzajúcu časť seriálu). V prvom rade prekladač hľadá funkciu, ktorej počet a typy argumentov sa presne zhodujú s počtom a typmi argumentov použitých pri volaní funkcie. Ak takú nenájde, pokúsi sa nájsť takú deklaráciu šablóny, z ktorej je možné vygenerovať funkciu s presnou zhodou argumentov. Ak ani teraz neuspeje, pokračuje v hľadaní bežným spôsobom. Ak v druhom kroku nájde viac ako jednu vhodnú šablónu, ohlási chybu nejednoznačnosti. Pri hľadaní vhodnej šablóny sa neuvažujú nijaké konverzie (ani triviálne), ako sme videli pri volaní `max(3, 'z')`.

Pre deklaráciu šablónových funkcií platí určité obmedzenie: pokiaľ nie je explicitne určená konkrétna inštancia šablóny, musia všetky typové argumenty šablóny vystupovať v deklaráciách formálnych argumentov šablónovej funkcie. Prekladač totiž musí na základe volania funkcie vedieť, ktorú inštanciu má vygenerovať. To sa mu nepodarí napríklad v prípade:

```
template <class T>
T foo() { /* ... */ }

int a = foo();
```

I keď vidíme, že by sa mala vygenerovať inštancia `foo<int>()`, prekladač si to, bohužiaľ, nie je schopný domyslieť a ohlási chybu. Jediná možnosť, ako ho informovať o tom, čo chceme, je takáto:

```
int a = foo<int>();
```

Podobné pravidlo platí pre netypové argumenty, ale tie sa používajú skôr pri deklaráciách šablón tried, preto sa tu nimi nebudeme hlbšie zaoberať.

## Šablóny tried

Deklarácia šablón tried má podobný význam ako deklarácia šablón funkcií. Šablóna triedy predstavuje parametrizovaný vzor na tvorbu nových tried. Na rozdiel od šablón funkcií majú šablóny tried oveľa širšiu oblasť použitia, umožňujúc pomerne rýchly vývoj programov na vysokej úrovni abstrakcie.

Ako príklad si vytvoríme šablónu pre triedu `vector`, ktorá bude predstavovať dynamicky alokované pole prvkov nejakého typu. Tento typ prvku bude parametrom šablóny a aby sme si ukázali aj iné ako typové parametre, budeme veľkosť poľa zadávať nie ako parameter konštruktora (čo by sme spravili za normálnych okolností), ale ako parameter šablóny. Uvedomte si však, že takto definovaná šablóna by vytvárala samostatnú triedu pre každú použitú veľkosť poľa v programe, čo obyčajne nepotrebujeme a nechceme.

Deklarácia šablóny triedy `vector` bude vyzerať takto:

```
template <class T, int n>
class vector
{
  T* ptr;
public:
  vector();
  ~vector();
  T& operator[](int i)
  { return ptr[i]; }
};
```

Pre jednoduchosť sme do triedy `vector` zaviedli len jedinú členskú funkciu: prekrytý operátor indexovania pre prístup k jednotlivým prvkom vektora. Parameter šablóny `T` je použitý ako doménový typ ukazovateľa `ptr`, reprezentujúceho dynamické pole prvkov.

Takto deklarovaná šablóna, samozrejme, nie je úplná, prekladač dosiaľ nevie, ako konštruovať a deštruovať inštancie triedy `vector`. Jednotlivé členské funkcie, pokiaľ ich nedefinujeme v rámci deklarácie triedy, treba definovať ako šablónové, s rovnakými parametrami šablóny ako ich materská trieda:

```
template <class T, int n>
vector<T, n>::vector()
{
  ptr = new T[n];
}

template <class T, int n>
vector<T, n>::~vector()
{
  delete [] p;
}
```

Všimnite si, že členské funkcie kvalifikujeme menom ich triedy, ktoré však v tomto prípade nie je `vector`, ale `vector<T, n>`, pretože ide o šablónu triedy. Aj tu však existuje výnimka – v názvoch konštruktora a deštruktora už parametre šablóny nesmieme uvádzať.

Triedu `vector<T, n>` môžeme odteraz veselo používať, len musíme uviesť oba parametre šablóny, t. j. napríklad takto:

```
vector<int, 20> v;
v[10] = 2178;
```

Premenná `v` bude predstavovať inštanciu triedy `vector<int, 20>`, čo je inštancia šablóny `vector<T, n>`. Prekladač automaticky po zhliadnutí deklarácie `v` vygeneruje telo členských funkcií a prípadné statické premenné triedy `vector<int, 20>`. Meno novej triedy je plnohodnotným menom triedy a môžeme ho ďalej používať ako každé iné meno typu. Dokonca môžeme toto meno použiť ako parameter ľubovoľnej šablóny – aj tej istej:

```
vector<vector<int, 20>, 5> v2;
```

Premenná `v2` bude predstavovať vektor piatich prvkov, z ktorých každý bude sám osebe vektorom dvadsiatich celých čísel.

Vzhľadom na skutočnosť, že šablónová trieda je trieda ako každá iná, môžeme od nej bez problémov odvodzovať nové triedy:

```
class newvector : public vector<char, 100>
{ /* ... */ };
```

Častejšie však bude odvodená trieda sama deklarovaná pomocou šablóny.

Pre explicitné vygenerovanie inštancie nejakej šablóny triedy platí to isté, čo sme uviedli v odseku o šablónach funkcií. Ak teda chceme prekladaču prikázať, aby vygeneroval inštanciu šablóny `vector<double, 200>`, použijeme zápis:

```
template class vector<double, 200>;
```

Akákoľvek deklarácia triedy, ktorej meno sa zhoduje s menom konkrétnej inštancie niektorej šablóny, je považovaná za inštanciu tejto šablóny. V praxi teda môžeme deklarovať napríklad triedu `vector<float, 99>`, ktorá bude mať úplne iné údajové členy a členské funkcie ako generická trieda `vector<T, n>`. Napriek tomu bude trieda `vector<float, 99>` považovaná za inštanciu šablóny `vector<T, n>`.

Keďže sa všetky členské funkcie šablónovej triedy automaticky považujú za šablónové funkcie, nič nám nebráni definovať si ich podľa potreby inak. Musíme to však spraviť skôr, než prekladač vygeneruje ich telo automaticky na základe vytvorenia inštancie príslušnej triedy. Toto predefinovanie bude vyzerať takto:

```
vector<long, 3>::vector()
{ /* ... */ }
```

a spôsobí, že pri vytváraní objektov triedy `vector<long, 3>` sa nepoužije generický konštruktor zo šablóny, ale táto jeho špeciálna verzia.

Ak v rámci deklarácie šablóny triedy uvedieme nejakú funkciu ako spriatelenú (`friend`), nestáva sa táto funkcia automaticky šablónovou. Ukážme si príklad:

```
template <class T>
class foo()
{
  // ...
  friend void fnc1();
  friend void fnc2(foo<T>*);
  friend void fnc3(foo*);
};
```

Funkcia `fnc1()` bude spriatelenou funkciou všetkým triedam `foo<T>`, ktoré prekladač vygeneruje. Funkcia `fnc2()` musí byť deklarovaná ako šablóna, pretože jej parametrom je ukazovateľ na šablónovú triedu `foo<T>`. Jej deklarácia môže vyzerať napríklad takto:

```
template <class T>
void fnc2(foo<T>*)
{ /* ... */ }
```

Každá vygenerovaná inštancia šablóny `foo<T>` bude mať vlastnú spriatelenú funkciu `fnc2(foo<T>*)`. Konečne deklarácia funkcie `fnc3()` je chybou, pretože neexistuje samostatná trieda `foo`, iba jej konkrétne varianty, ako `foo<int>`, `foo<double>` a pod.

## Statické premenné a členy tried

V tele šablónovej funkcie môžeme deklarovať statické premenné. Každá inštancia šablónovej funkcie bude mať vlastnú množinu statických premenných, rovnako bude mať vygenerovaný vlastný kód, predstavujúci telo funkcie. Statické premenné, samozrejme, môžu byť parametrizované pomocou parametrov šablóny. Malý príklad:

```
template <class T>
void swap(T& a, T& b)
{
  static T tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

Ak funkciu `swap()` použijeme napríklad takto:

```
int x, y;
double m, n;
swap(x, y);
swap(m, n);
```

bude v inštancii `swap(int&, int&)` existovať statická premenná `tmp` typu `int` a v inštancii `swap(double&, double&)` statická premenná

`tmp` typu `double`. Obe premenné `tmp` budú od seba nezávislé.

Podobne to platí pre statické členy tried, len tu musíme dať pozor: statické údajové členy je potrebné definovať samostatne, mimo deklarácie triedy. Preto pre každú vygenerovanú inštanciu danej šablóny (automaticky či explicitne) musíme zabezpečiť definíciu všetkých jej statických členov, ovplyvňovaných parametrami šablóny, na globálnej úrovni:

```
template <class T>
class foo()
{
  // ...
  static T s;
};

int foo<int>::s = 123;
char* foo<char*>::s = "hello";
```

Ak deklarácia statických členov nie je ovplyvňovaná parametrami šablóny, postačí samozrejme každý statický údajový člen definovať raz, ale musíme použiť podobnú syntax, ako pri deklarácii samotnej šablóny:

```
template <class T>
class bar()
{
  // ...
  static int s;
};

template <class T>
int bar<T>::s = 216;
```

Takáto deklarácia zabezpečí, že každé vygenerovanie inštancie šablóny triedy `bar<T>` spôsobí automatickú definíciu a inicializáciu jej statického člena `s`. Pokiaľ chceme pre konkrétnu inštanciu inicializovať jej člen `s` inou hodnotou ako implicitných 216, použijeme zápis:

```
int bar<double*>::s = 78;
```

## Výnimky v C++

Ďalšou pokročilou črtou jazyka C++ je existencia výnimiek. Mechanizmus výnimiek v podstate slúži na dokonalejšie a prirodzenejšie ošetrovanie výnimočných stavov, ku ktorým môže dôjsť počas behu programu (typicky nedostatok voľnej pamäte, neexistencia súboru na disku a pod.). Klasický spôsob detekcie chýb vyzerá asi takto: majme funkciu, ktorá má na starosti nejakú činnosť. Táto funkcia nech oznamuje výskyt prípadnej chyby špeciálnou návratovou hodnotou. Po každom zavolaní takejto funkcie musíme zisťovať, čo nám vrátila a či náhodou nedošlo k chybe. Teda ak funkciu voláme za sebou päťkrát, musíme do kódu doplniť päť príkazov `if`. Zdrojový text programu sa stáva neprehľadným, nehovoriac o tom, že ľahko môžeme niektorý z testov omylom vynechať. Program potom nie je dostatočne robustný, pri výskyte chyby môže jednoducho padnúť a chudák používateľ bude mať na čo nadávať. Okrem toho nie každá funkcia si môže dovoliť vyhradiť jednu špeciálnu návratovú hodnotu na indikáciu chyby.

Výnimky boli vymyslené práve preto, aby nám pomohli vyriešiť problémy naznačené v predchádzajúcom odseku. Ich princíp je jednoduchý: chybový stav sa bude indikovať vygenerovaním (aj „vyhodením“, z anglického throw) výnimky. Blok príkazov, v ktorom môže dôjsť k výnimke, budeme strážiť a v prípade, že k nej naozaj dôjde, budeme na túto skutočnosť patrične reagovať pomocou tzv. handlera výnimky (odmietam toto slovo preložiť do slovenčiny, pretože „ovládač“ to nie je a „obslužný kód“ je trochu krkolomné).

V C++ celá situácia bude vyzerať takto. Strážený blok príkazov (tiež nazývaný try-blok) má tvar:

`try` *zložený-príkaz* *zoznam-handlerov*

Zoznam-handlerov je bielymi znakmi oddelená postupnosť handlerov v tvare:

`catch` ( *deklarácia-výnimky* ) *zložený-príkaz*

V rámci strážového bloku môžeme výnimku vygenerovať pomocou výrazu:

`throw` *výraz*

Tento výraz je typu `void`.

Ukážme si na jednoduchom príklade, ako celý mechanizmus vyzerá:

```
void main()
{
  try
  {
    // ...
    throw "chyba!";
    // ...
  }
  catch (const char* str)
  {
    printf("výnimka: %s\n", str);
  }
}
```

Len čo program dospeje k príkazu `throw`, skočí na koniec bloku `try` a hľadá vhodný handler. Pri hľadaní jednoducho porovnáva typ vyhodeného objektu (v našom prípade `const char*`) s deklaráciou uvedenou v hlavičke handlera. Ak nájde vhodný handler, vykoná jeho telo a ďalej pokračuje kódom, ktorý nasleduje za všetkými handlermi príslušného try bloku. No ak taký handler nenájde, ukončí funkciu, v ktorej sa nachádza, a prejde do nadradenej (volajúcej) funkcie, v ktorej opätovne hľadá vhodný handler rovnakým spôsobom. Ak v nadradenej funkcii nie je volanie funkcie, ktorá výnimku vyhodila, uzavreté do try bloku, rovno túto funkciu opustí. Takto prechádza celý zásobník volaných funkcií, až kým nenájde vhodný handler alebo kým nedôjde k pokusu o opustenie funkcie `main()`. Vtedy zavolá funkciu `terminate()`, ktorá implicitne ukončí program.

Objekt, ktorý je vyhodený pomocou príkazu `throw`, môže byť ľubovoľného typu, najčastejšie asi to bude inštancia nejakej vhodnej triedy. Inicializácia formálneho objektu v deklarácii hlavičky handlera je prakticky zhodná s inicializáciou formálneho argumentu funkcie. Niekedy nepotrebujeme do handlera odovzdať konkrétny objekt, ale iba informáciu o type výnimky. Preto je možné v hlavičke handlera vynechať identifikátor vyhodeného objektu – v takom prípade však k nemu nemáme nijaký prístup:

```
class dummy {};

try
{
  // ...
  throw dummy();
  // ...
}
catch (dummy)
{
  // ...
}
```

Ako sme už povedali, pri hľadaní vhodného handlera sa porovnáva typ vyhodeného objektu s typom deklarovaným v hlavičke handlera. Platí tu nasledujúce pravidlo: handler s deklarovaným typom `T`, `const T`, `T&`, `const T&` zachytí objekt typu `E`, ak

- `T` a `E` sú rovnaké typy,
- ak `T` je prístupnou základnou triedou `E`,
- ak `T` aj `E` sú ukazovatele a `E` možno pretypovať na `T` pomocou štandardných konverzií.

Handlery sú testované v poradí ich deklarácie, preto je chybou uviesť handler zachytávajúci objekt odvodenej triedy za handlerom zachytávajúcim objekt základnej triedy – druhý z handlerov sa k slovu už nedostane.

Zvláštnym prípadom handlera je tzv. univerzálny handler, deklarovaný podobne ako pri funkciách s premenným počtom argumentov pomocou výpustky (`...`). Takýto handler zachytí akúkoľvek výnimku. Je dobré ho dávať až na koniec celej hierarchie, aby sme si omylom neodchytili nesprávnu výnimku. Ak nechceme, aby nám nejaká výnimka ukončila program predčasne, môžeme univerzálny handler dať na koniec funkcie `main()`:

```
void main()
{
  try {
    // ...
  }
  catch (...)
  {
    // ...
  }
}
```

Niekedy treba odovzdať výnimku ďalej, vyššiemu handleru. Aj na takúto situáciu C++ pamätá – pokiaľ uvedieme kľúčové slovo `throw` bez argumentov, aktuálne vyhodený objekt sa pošle vyššie v celej hierarchii. Toto však môžeme spraviť iba v rámci nejakého handlera, inokedy by totiž nebolo čo „poslať ďalej“.

Najdôležitejšou črtou celého mechanizmu výnimiek je však skutočnosť, že v okamihu, keď sa nájde príslušný handler, program automaticky deštruuje všetky automatické objekty, ktoré boli skonštruované od vstupu do try-bloku, patriaceho k nájdenému handleru až po okamih výskytu výnimky a to aj cez niekoľko úrovní funkcií! Tento proces sa nazýva aj „odrolovanie“ zásobníka (v angličtine stack unwinding). Ukážme si príklad:

```
class C
{
public:
  C();
  ~C();
};

C::C()
{
  printf("constructing C\n");
}

C::~C()
{
  printf("destructing C\n");
}

void fnc2()
{
  throw 1;
}

void fnc1()
{
  C c2;
  fnc2();
}

void main()
{
  try
  {
    C c1;
    fnc1();
  }
  catch (...)
  {
    printf("caught exception!");
  }
}
```

Ak si vyskúšate tento príklad preložiť a spustiť (budete však potrebovať niektorý z novších prekladačov podporujúcich výnimky – starý dobrý Borland C++ 3.1 vám nebude stačiť), uvidíte, že skôr, než sa vykoná telo univerzálneho handlera, zavolajú sa deštruktory objektov `c1` aj `c2`.

### Deklarácia povolených výnimiek

Deklaráciu každej funkcie môžeme doplniť zoznamom výnimiek, ktoré táto funkcia môže generovať. Tento zoznam sa skladá z kľúčového slova `throw` a v zátvorkách uzavretej množiny typov objektov, ktoré môžu byť z funkcie „vyhodené“. Príklad:

```
class C { /* ... */ };

void fnc() throw (int, C, char*)
{
  // ..
}
```

Funkcia, ktorá takúto špecifikáciu neobsahuje, môže vyhodiť ľubovoľnú výnimku. Funkcia, ktorá obsahuje prázdnu špecifikáciu, `throw()`, nesmie vyhodiť nijakú výnimku. Funkcia, ktorá má povolené vyhodiť výnimku typu triedy C, môže vyhodiť aj výnimku typu triedy verejne odvodenej od `C`. Ak počas behu programu nastane situácia, že by funkcia mala vyhodiť výnimku, ktorú nemá povolenú, zavolá sa funkcia `unexpected()`.

### Špeciálne funkcie

V súvislosti s výnimkami v C++ sme spomenuli dve funkcie so zvláštnym postavením – `terminate()` a `unexpected()`. Prvú z nich prekladač zavolá za týchto okolností:

- keď nenájde vhodný handler pre vygenerovanú výnimku,
- keď počas hľadania handlera zistí porušenie zásobníka,
- keď počas odrolovania zásobníka niektorý z volaných deštruktorov sám vyhodí výnimku

Implicitne funkcia `terminate()` volá funkciu `abort()`, čo má za následok okamžité a násilné ukončenie programu. Pomocou funkcie `set_terminate()` s argumentom typu `void (*)()` môžeme nastaviť vlastnú obsluhu, ktorá však musí program ukončiť; pokus o návrat z `terminate()` je chybou. Funkcia `set_terminate()` vracia ako návratovú hodnotu ukazovateľ na predchádzajúcu obslužnú funkciu; umožňuje tak zreťaziť niekoľko vlastných funkcií.

Druhú z funkcií, `unexpected()`, zavolá prekladač vtedy, keď zistí, že sa niektorá funkcia snaží vyhodiť výnimku, ktorú explicitne nedeklarovala v svojom zozname povolených výnimiek. Implicitne funkcia `unexpected()` volá funkciu `terminate()`. Vlastné správanie môžeme definovať pomocou funkcie `set_unexpected()`, ktorá má opäť jediný argument typu `void (*)()` a takisto vracia ukazovateľ na predchádzajúcu obslužnú funkciu.

### Štruktúrované výnimky

Na záver tejto časti si povieme pár slov o téme, ktorá nie je súčasťou normy C++. Štruktúrované výnimky sú určené pre jazyk C, ale ani tam nie sú nijako normatívne definované. Ich podpora jednotlivými prekladačmi závisí čisto od „dobrej vôle“ ich výrobcov, ale prácu s nimi umožňujú všetky moderné prekladače, hoci stopercentne to môžem tvrdiť iba o Microsoft Visual C++. Nasledujúce informácie sú preto založené na dokumentácii k tomuto produktu a berte ich len ako stručnú informáciu o tom, ako je možné pracovať s výnimkami aj v jazyku C.

Na prácu so štruktúrovanými výnimkami máme k dispozícii niekoľko kľúčových slov: `__try`, `__except`, `__finally`, `__leave`. Všimnite si, že všetky sa začínajú dvoma znakmi podčiarknutia, čo vyjadruje ich neštandardnosť a závislosť od implementácie. Základná schéma stráženého bloku je podobná ako v C++:

*__try zložený-príkaz*$_1$
*__except ( výraz ) zložený-príkaz*$_2$

Prvý zložený príkaz predstavuje strážený blok. Ak sa počas vykonávania príkazov v tomto bloku nevyskytne výnimka, program pokračuje za zloženým príkazom v klauzule `__except`. Výskyt výnimky v bloku spôsobí vyhodnotenie výrazu, na základe ktorého sa rozhodne, čo ďalej. Možnosti sú tri (ide o preddefinované konštanty):

- `EXCEPTION_CONTINUE_SEARCH` – výnimka nie je rozpoznaná, hľadanie vhodného handlera pokračuje na vyššej úrovni (ekvivalentné príkazu `throw` bez argumentov);
- `EXCEPTION_CONTINUE_EXECUTION` – výnimka je rozpoznaná, ale ignoruje sa, program pokračuje ďalej od toho bodu, v ktorom k výnimke došlo;
- `EXCEPTION_EXECUTE_HANDLER` – výnimka je rozpoznaná, vykoná sa príslušný handler (druhý zložený príkaz) a na rozdiel od C++ vykonávanie programu pokračuje od bodu, v ktorom výnimka nastala.

Uprostred bloku `__try`/`__except` je možné použiť kľúčové slovo `__leave`, ktoré spôsobí okamžité opustenie bloku a presmerovanie programu na prvý príkaz za klauzulou `__except`.

Druhý variant stráženého bloku používa kľúčové slovo `__finally` a má tvar:

*__try zložený-príkaz*$_1$
*__finally zložený-príkaz*$_2$

Na rozdiel od predchádzajúceho prípadu tu nie je nijaký handler pre výnimky. Zmysel sekcie uvedenej slovom `__finally` je jej zaručené vykonanie nezávisle od toho, či bude strážený blok ukončený riadnym spôsobom až do konca alebo bude opustený skôr v dôsledku vzniku výnimky. Do tejto ukončovacej sekcie sa preto obyčajne dáva kód slúžiaci na uvoľnenie alokovaných prostriedkov a iné upratovacie činnosti. Aj uprostred bloku `__try`/`__finally` je možné použiť kľúčové slovo `__leave`.

Na záver zostáva povedať, že hoci je možné miešať výnimky jazyka C++ a štruktúrované výnimky jazyka C, nie je to práve najrozumnejší nápad. Štruktúrované výnimky sú vždy typu `unsigned int` a sú zachytiteľné bežným C++ handlerom (`catch`), ale vzhľadom na nepomerne širšie možnosti C++ výnimiek nemá veľmi zmysel ich používať (jedine, keby ste mali niečo do činenia s existujúcim kódom v jazyku C).

### Efektivita

Na začiatku rozprávania o výnimkách sme sa zmienili o tom, čo viedlo k zavedeniu mechanizmu výnimiek do C++ a aké výhody nám plynú z ich používania. Bohužiaľ, každá minca má dve strany a programovanie za pomoci výnimiek nás niečo stojí. To niečo je výkon programu, ktorý pri nadmernom používaní výnimiek dosť radikálne klesá – uvedomte si rozdiel medzi bežným návratom z funkcie (niekoľko inštrukcií) a návratom na základe vyhodenia výnimky (zložitý kód – nájdenie handlera, odrolovanie zásobníka). Ak vás preto napadlo vracať výsledok volania nejakej funkcie výnimkou, tak na to rýchlo zabudnite, cena je privysoká. Výnimky majú zmysel pri rozsiahlych programoch, kde by klasické ošetrovanie chybových stavov neúmerne zneprehľadnilo program. Určité spomalenie takýchto programov je nevyhnutnou daňou za nezbláznenie sa pri ich tvorbe.

## Dvadsiata druhá časť: ČO JE NOVÉ V C++?

V predposlednej časti seriálu budeme hovoriť o najnovších črtách jazyka C++, o vlastnostiach, ktoré sa stali de iure súčasťou jazyka až nedávno, a to na základe kodifikácie normou ANSI, o používaných i menej používaných prvkoch, jednoducho o všetkom tom, o čom zatiaľ ani mnohí skúsení programátori v C++ nemajú potuchy (dúfam, že tento stav je len dočasný a vďaka tomuto článku sa rýchlo zmení). O niekoľkých črtách, ktoré by sa dali považovať za viac-menej nové, som sa v priebehu seriálu explicitne zmienil. Veci, o ktorých sa dozviete dnes, môžete brať skutočne ako „crčme de la crčme“ toho, čo sa možno o C++ dozvedieť. Bohužiaľ, neznamená to však, že budete môcť všetky tieto nové vlastnosti aj používať – nijaký dnes dostupný prekladač nie je úplne konformný s normou ANSI, v implementácii každého z nich existujú menšie či väčšie odchýlky od štandardu. Útechou nech vám je opojný pocit, že sa po prečítaní článku budete so svojimi vedomosťami nachádzať, ako sa hovorí, „on the cutting edge“ alebo, po slovensky povedané, na špici súčasného stavu jazyka C++.

Skôr než začneme, chcel by som vás ešte upozorniť, že vývoj jazyka C++ prebiehal a prebieha kontinuálne, mnohé črty sa stali jeho de facto súčasťou dávno predtým, než boli podchytené normou ANSI. Je preto možné, že niektoré tu opisované vlastnosti budú pre niekoho z vás známe, prípadne ste o nich počuli už dávnejšie. V takom prípade ma, prosím, nekameňujte za to, že som si dovolil označiť ich ako „nové“. Pre väčšinu z vás novými budú.

### Kľúčové slová

Súčasťou C++ je niekoľko nových kľúčových slov, ktoré môžeme použiť ako ekvivalenty niektorých operátorov. Ich zavedenie sa odôvodňuje potenciálne obťažným zápisom niektorých ASCII znakov, tvoriacich tieto operátory, v určitých znakových sadách. Priznám sa, pokladám to za dosť pochybný dôvod, ale dajme tomu. Nové kľúčové slová spolu s ekvivalentnými operátormi sú v tabuľke 1.

*Tabuľka č. 1 Slovné ekvivalenty operátorov.*

| Kľúčové slovo | Operátor |
|---|---|
| and | && |
| and_eq | &= |
| bitand | & |
| bitor | \| |
| compl | ~ |
| not | ! |
| not_eq | != |
| or | \|\| |
| or_eq | \|= |
| xor | ^ |
| xor_eq | ^= |

V niektorých prekladačoch sú tieto kľúčové slová implementované pomocou makier, #definovaných v štandardnom hlavičkovom súbore `<iso646.h>`.

### Údajové typy

Na reprezentáciu znakov, ktorých číselná reprezentácia sa nevojde do rozsahu typu `char` (typicky znaky ázijských abecied, resp. znaky v kóde Unicode), je k dispozícii údajový typ `wchar_t`. Jeho šírka je obyčajne 16 bitov, hoci to norma nijako nepredpisuje. Širokoznakové literály môžeme zadávať buď klasickým spôsobom v apostrofoch,

ale s predradenou predponou L, alebo pomocou escape sekvencie \u*xxxx* (znaky Unicode). Príklady:

```
wchar_t c1 = L'ab';
wchar_t c2 = L'\x0F\x3A';
wchar_t c3 = ,\u109D';
```

Podpora escape sekvencie \u*xxxx* je však zatiaľ veľmi zriedkavá.

Druhým novým údajovým typom je typ bool, ktorý slúži, ako správne predpokladáte, na reprezentáciu logických hodnôt. Jeho povolený rozsah tvoria dve hodnoty – true a false –, ktoré sú súčasne kľúčovými slovami C++. Nad typom bool môžeme používať bežné logické operátory &&, || a !. Je možné, samozrejme, použiť aj bitové operátory &, |, ^ a ~, ale tie majú za následok celočíselné rozšírenie typu bool na typ int (intuitívne – false sa rozšíri na hodnotu 0 a true na hodnotu 1). Navyše môžeme na premennú typu bool aplikovať prefixový či postfixový operátor ++, ktorý spôsobí, že sa premenná nastaví na hodnotu true. Bohužiaľ, opačný spôsob – aplikácia operátora -- – nie je povolený. Ľubovoľný celočíselný typ môžeme konvertovať na typ bool: výsledkom konverzie nulovej hodnoty je hodnota false, výsledkom konverzie nenulovej hodnoty je, naopak, hodnota true.

So zavedením typu bool sa zmenil aj výsledný typ podmienkového výrazu. Podmienkový výraz je výraz obsahujúci niektorý z operátorov ==, !=, <, <=, >, >=. Teda napríklad výraz i==0 bude mať hodnotu true za predpokladu, že obsah premennej i bude nulový, a hodnotu false v ostatných prípadoch. Podmienkové výrazy sú súčasťou príkazov if, while, do a for, takže napríklad nekonečný cyklus je možné zapísať aj takto:

```
while (true)
{ /* ... */ }
```

## Operátory pretypovania

Pretypovanie premenných a konštánt v C++ sa považuje za neveľmi bezpečný prvok jazyka, pretože pri jeho nevhodnom použití môže dôjsť k nesprávnej interpretácii údajov, s ktorými program pracuje, a následne k závažným chybám, často vedúcim až k zrúteniu programu.

Klasický spôsob pretypovania sa opiera o operátor pretypovania známy ešte z jazyka C, prípadne o syntax podobnú volaniu funkcie, špecifickú pre C++:

```
a = (int)b;
a = int(b);
```

Z dôvodu zvýšenia bezpečnosti bola úloha pretypovacieho operátora rozdelená medzi štyri nové operátory dynamic_cast, static_cast, const_cast a reinterpret_cast. V nasledujúcich odsekoch si ich preberieme podrobnejšie.

### Operátor dynamic_cast

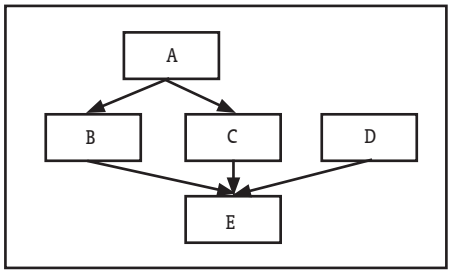
Prvý operátor dynamic_cast slúži na dynamické pretypovanie ukazovateľov, resp. referencií na objektové typy. Pretypovanie je možné iba v rámci hierarchie dedičnosti. Vieme, že je bez problémov možné pretypovať ukazovateľ na potomka na ukazovateľ na predka, to isté platí pre referencie. Často však bude potrebné pretypovať ukazovateľ či referenciu opačným smerom, keď celkom určite vieme, že ukazovateľ na predka v skutočnosti ukazuje na inštanciu niektorého jeho potomka. Práve na tieto účely máme k dispozícii operátor dynamic_cast. Jeho syntax je takáto:

```
dynamic_cast < cieľový typ > ( ukazovateľ )
dynamic_cast < cieľový typ > ( referencia )
```

Pretypovávaný ukazovateľ, resp. referencia musia odkazovať na inštanciu objektového typu. Cieľový typ musí byť trieda, ktorá je predkom alebo potomkom triedy pretypovávanej inštancie.

Operátor dynamic_cast využíva tzv. dynamickú identifikáciu typu (RTTI – real-time type identification), pomocou ktorej si overí, či je pretypovanie možné a má zmysel. Už z názvu vyplýva, že táto kontrola sa deje za behu programu, a teda je možné pretypovávať iba polymorfné triedy s virtuálnymi metódami.

Mnohorakosť použitia operátora dynamic_cast naznačí nasledujúci príklad: Predstavme si, že máme objekt triedy E, ktorá je súčasťou hierarchie tried, znázornenej na obr. 1. V rámci tejto hierarchie môžeme ukazovateľ na E pretypovať na ukazovateľ na ľubovoľnú inú triedu A až D v smere aj proti smeru šípok vyjadrujúcich vzťahy dedičnosti.




*Obrázok č. 1 Príklad hierarchie tried*

Predpokladajme napríklad, že máme objekt triedy E prístupný prostredníctvom ukazovateľa pe. Tento ukazovateľ môžeme pretypovať na ukazovateľ na triedu C priamo (to je dovolené – pretypujeme potomka na predka):

```
C* pc = pe;
```

alebo pomocou nového operátora:

```
C* pc = dynamic_cast<C*>(pe);
```

Ak máme objekt triedy E prístupný pomocou ukazovateľa pa na triedu A (ktorý de facto ukazuje na zdedený podobjekt triedy A v objekte triedy E), môžeme sa k ukazovateľu na celý objekt E dostať takto:

```
E* pe = dynamic_cast<E*>(pa);
```

Tu už na rozdiel od predchádzajúceho príkladu priamy spôsob pretypovania nemôžeme použiť.

Operátor dynamic_cast však umožňuje aj pretypovanie ukazovateľa na podobjekt triedy B na ukazovateľ na podobjekt triedy D (stále za predpokladu, že v skutočnosti pracujeme s objektom triedy E):

```
B* pb = pe;
D* pd = dynamic_cast<D*>(pb);
```

Cieľovým typom operátora dynamic_cast môže byť aj typ void*. V takomto prípade je výsledkom pretypovania ukazovateľ na celý objekt, ktorého sa pretypovanie týka. Ak teda máme napríklad ukazovateľ pd z predchádzajúceho príkladu, po pretypovaní:

```
void* pv = dynamic_cast<void*>(pd);
```

bude v ukazovateli pv adresa celého objektu triedy E.

V prípade, že pretypovanie nie je možné, operátor vráti nulovú hodnotu (pri pretypovaní ukazovateľov) alebo vyhodí výnimku bad_cast (pri pretypovaní referencií).

Uveďme si ešte príklad, ktorý ukáže použitie operátora dynamic_cast v praxi:

```
class B
{
public:
  virtual void f() {}
};

class D : public B
{
public:
  virtual void f() {}
  void g() { printf(„Hello\n"); }
};
void fnc(B* pb)
{
  D* pd = dynamic_cast<D*>(pb);
  if (pd)
    pd->g();
}

void main()
{
  D d;
  fnc(&d);
}
```

Funkcia fnc() testuje, či jej argument, ukazovateľ s doménovým typom B, náhodou neukazuje na objekt triedy D. Ak zistí, že áno, zavolá nad týmto objektom jeho členskú funkciu g(). Ak si vo funkcii main() zmeníte typ premennej d na triedu B, uvidíte, že k volaniu funkcie D::g() nedôjde.

### Operátor static_cast

Na rozdiel od predchádzajúceho operátora nepoužíva operátor static_cast dynamickú identifikáciu typu. To ostatne naznačuje už jeho názov. Pri použití operátora static_cast sa prekladač riadi len statickým, v dobe prekladu známym typom pretypovávaného objektu. Syntax operátora je podobná ako pri dynamic_cast:

```
static_cast < cieľový typ > ( výraz )
```

Pokus o pretypovanie objektovej inštancie na triedu, ktorá nie je ani jej predkom, ani potomkom, prekladač bez problémov povolí, ale výsledok nie je definovaný a jeho použitie môže viesť až k zrúteniu programu. Ukážme si príklad (predpokladáme triedy B a D z predchádzajúceho odseku):

```
void f(B* pb)
{
  D* pd = static_cast<D*>(pb);
  pd->g();
}
```

Vo funkcii f() sa snažíme pretypovať ukazovateľ na triedu B na ukazovateľ na jej potomka, triedu D. Keďže je v programe zapísané (a známe už počas prekladu), že chceme pretypovať ukazovateľ pb, ktorý je typu B*, na ukazovateľ pd typu D*, prekladač sa nebude sťažovať a hlavne si nijako nebude overovať, či pb naozaj ukazuje na inštanciu triedy D (to napokon počas prekladu ani nejde). Teraz si predstavme, že funkcii f() odovzdáme ako skutočný argument ukazovateľ na inštanciu triedy B. Pretypovaním dostaneme ukazovateľ typu D*, v skutočnosti však ukazujúci na typ B. Volanie členskej funkcie D::g() nad objektom triedy B sa skončí pravdepodobne zle.

Operátor static_cast nie je obmedzený len na pretypovanie objektových typov. V skratke možno povedať, že tento operátor je ekvivalentom rozumného použitia pôvodného pretypovacieho operátora a mal by sa namiesto neho používať všade tam, kde chceme explicitne vyjadriť konverziu medzi dvoma typmi. Nasledujúce príklady ukazujú jeho použitie:

```
void* pv = „Hello";
char* pc = static_cast<char*>(pv);
float f = 123.456;
double d = static_cast<double>(f);
```

```
char c = ‚Z‘;
int i = static_cast<int>(c);
```

### Operátor const_cast

Všetci veľmi dobre poznáme význam a použitie špecifikátorov `const` a `volatile`. Takisto všetci vieme, že na to, aby sme obišli zábrany, ktoré z ich použitia vyplývajú, nám stačí použiť vhodne pretypované ukazovatele či referencie. Operátor `const_cast` je určený práve na takéto pretypovávanie. Ako jediný zo všetkých štyroch nových operátorov totiž dokáže pridávať či odstraňovať špecifikátory `const` a `volatile` z existujúcich výrazov. Syntax jeho použitia asi nikoho neprekvapí:

`const_cast <` *cieľový typ* `> (` *výraz* `)`

Cieľový typ a typ pretypovávaného výrazu sa môžu líšiť maximálne v počte a umiestnení špecifikátorov `const` a `volatile`.

Použitie operátora `const_cast` v praxi je zrejmé. Jeden príklad za všetky:

```
const int ci = 10;
const int* pci = &ci;
*pci = 20;  // chyba
int* pi = const_cast<int*>(pci);
*pi = 20;   // OK
```

### Operátor reinterpret_cast

Posledný zo štvorice operátorov má na starosti doslova „špinavú“ robotu. Je určený na pretypovanie ukazovateľov medzi sebou, na konverziu ukazovateľov na celé čísla a naopak a všeobecne na všetky tie implementačne závislé, neprenosné, nízkoúrovňové prevody, ktoré tak často a radi používame, pretože nám ušetria kopu starostí. Jeho použitie nie je príliš bezpečné, v podstate jediným zaručene bezpečným výsledkom je pretypovanie „tam a naspäť“, po ktorom dostaneme opäť pôvodnú hodnotu.

Syntax operátora `reinterpret_cast` nie je ničím výnimočná:

`reinterpret_cast <` *cieľový typ* `> (` *výraz* `)`

Pre istotu je tu ešte malý príklad použitia:

```
char* p = new char[100];
int addr = reinterpret_cast<int>(p);
```

V premennej `addr` bude celočíselná reprezentácia adresy miesta v pamäti, kde je uložené alokované pole, 100 znakov.

### Dynamická identifikácia typu

Pri opise operátora `dynamic_cast` sme si spomenuli, že na zisťovanie skutočného typu objektu sa používa dynamická identifikácia typu. V tejto súvislosti máme v C++ k dispozícii nový operátor `typeid`, ktorý vracia pre daný objekt referenciu na konštantnú inštanciu triedy `type_info`, opisujúcu typ tohto objektu. Syntax operátora `typeid` je dvojaká:

`typeid (` *výraz* `)`
`typeid (` *typ* `)`

Prvý spôsob slúži na zisťovanie typu určitého výrazu, druhý je v podstate ekvivalentom klasických literálov – pre daný explicitne vyjadrený typ vracia jeho identifikáciu.

Trieda `type_info` je deklarovaná v hlavičkovom súbore `<typeinfo.h>` a obsahuje okrem iného konštantnú členskú funkciu `name()`, vracajúcu názov daného typu (ako `char*`), a dva prekryté operátory `==` a `!=` na porovnávanie inštancií tejto triedy. Pokiaľ je možné vyhodnotiť operátor `typeid` už počas prekladu, stane sa tak. Dynamicky, za behu programu, sa vyhodnocuje len typ polymorfných objektov, prezentovaných tomuto operátoru pomocou referencie alebo dereferencovaného ukazovateľa. V prípade, že by operátor `typeid` mal dereferencovať nulový ukazovateľ, vyhodí výnimku typu `bad_typeid`.

### Operátory new a delete

Pri rozprávaní o dynamickej alokácii premenných sme si povedali, že operátor `new` pri neúspešnej alokácii vráti nulový ukazovateľ. Norma ANSI mu však stanovuje trochu odlišné správanie. Štandardná verzia operátora `new` (resp. operátorovej funkcie `operator new()`) pri nemožnosti alokovať požadované množstvo pamäte musí vyhodiť výnimku `bad_alloc`. V záujme zachovania možnosti používať pôvodný spôsob alokácie existuje aj verzia, ktorá vracia nulový ukazovateľ. Tú špecifikujeme pomocou dodatočného argumentu `nothrow`. Tento argument je v skutočnosti prázdna štruktúra, definovaná len na odlíšenie oboch verzií operátora `new`.

Vzhľadom na existenciu uvedených rozdielov sa trochu menia prototypy operátorových funkcií `operator new()`. Keď ich chceme prekryť vlastnými funkciami, mali by sme dodržať nasledujúcu sémantiku:

```
void* operator new(size_t) throw(bad_alloc);
void* operator new(size_t, const nothrow_t&)
throw();
```

Typ `nothrow_t` je typom spomínanej štruktúry `nothrow`.

Druhá novinka sa týka alokácie polí. Operátor `new` pri alokácii poľa objektov volá funkciu `operator new[]()`. Túto funkciu, ktorá zabezpečuje pridelenie dostatočne veľkého úseku pamäte, môžeme, samozrejme, prekryť vlastnou funkciou. Jej presný prototyp (bez pridaných voliteľných argumentov) je:

```
void* operator new[](size_t);
```

Povinný argument tejto funkcie typu `size_t` predstavuje celkovú veľkosť alokovaného poľa, návratová hodnota predstavuje ukazovateľ na alokované miesto v pamäti. (V skutočnosti existujú dva rôzne prototypy, líšiace sa zoznamom výnimiek, ktoré funkcia vyhadzuje – pozri predchádzajúci text).

Podobne operátor `delete` pri dealokácii poľa objektov volá funkciu `operator delete[]()`. Jej dva základné prototypy (opäť bez pridaných voliteľných argumentov) sú:

```
void operator delete[](void*);
void operator delete[](void*, size_t);
```

Na zopakovanie: Prvý argument predstavuje ukazovateľ na dealokovanú oblasť, prípadný druhý určuje jej veľkosť.

### Príkazy selekcie

Medzi príkazy selekcie čiže výberu (myslí sa výber medzi viacerými alternatívami budúceho toku programu) patria príkazy `if`, `switch`, `for`, `while` a `do`. Podľa normy ANSI je povolené v podmienkových výrazoch týchto príkazov (s výnimkou príkazu `do`) deklarovať nové premenné. Rozsah platnosti takto deklarovaných premenných sa obmedzuje na blok tvoriaci telo príslušného príkazu. Ukážme si príklad:

```
while (char c = getchar())
{
  // ...
}
c = ‚A‘;  // chyba!
```

Pokus o prístup k premennej `c` mimo bloku `while` má za následok chybu pri preklade.

Rovnaké pravidlá platia pre ostatné príkazy selekcie. Za zmienku stojí hádam iba príkaz `for`, v ktorom bolo možné deklarovať premenné už predtým, ale iba v inicializačnom príkaze (prvá časť zátvorky pred bodkočiarkou). Podľa normy ANSI môžeme deklarovať nové premenné aj v testovacom výraze (v strede medzi oboma bodkočiarkami). Okrem toho niektoré prekladače nedodržujú presne pravidlo o obmedzení rozsahu platnosti premenných na telo príkazu `for`, takže je možné takto deklarované premenné používať aj v ďalšom kóde.

### Členy tried

Vieme, že premenné objektových typov môžeme definovať so špecifikátorom `const`. Takéto premenné sa považujú za konštantné, ich zložky nemôžeme meniť a dokonca môžeme nad nimi vyvolať iba tie členské funkcie, ktoré sme deklarovali ako konštantné (opäť pomocou kľúčového slova `const`). Tieto členské funkcie nesmú za normálnych okolností nijako modifikovať stav objektu.

Môže sa nám však pri návrhu objektových typov stať, že budeme potrebovať zaviesť do definície triedy taký údajový člen, ktorý by mal byť modifikovateľný za všetkých okolností, teda aj pre konštantné inštancie danej triedy. Pre takúto situáciu máme v C++ k dispozícii nový špecifikátor `mutable`, použiteľný len pri deklarácii údajových členov tried, ktorý povoľuje modifikáciu týchto členov aj z konštantných členských funkcií. Tu je krátky príklad:

```
class C
{
  int a;
  mutable int b;
public:
  void f();
  void g() const;
};

void C::f()
{
  a = 1;  // OK
  b = 2;  // OK
}

void C::g() const
{
  a = 3;  // chyba!
  b = 4;  // OK
}

void main()
{
  C c1;
  const C c2;
  c1.f();  // OK
  c1.g();  // OK
  c2.f();  // chyba
  c2.g();  // OK
}
```

V konštantnej členskej funkcii `g()` nesmieme modifikovať zložku `a`, môžeme však ľubovoľne pracovať so zložkou `b`, deklarovanou v triede ako `mutable`. Ostatné pravidlá pre volanie členských funkcií nad konštantnými a nekonštantnými objektmi zostávajú v platnosti, t. j. nad premennou `c2` môžeme zavolať len členskú funkciu `g()`, volanie `c2.f()` prekladač odmietne preložiť.

### Explicitné konštruktory

Isto si spomeniete, že v časti venovanej konštruktorom sme sa zmienili o tzv. implicitných konverziách na objektové typy. Vždy, keď prekladač potrebuje konvertovať hodnotu jedného (objektového či neobjektového) typu na iný objektový typ, môže tak urobiť buď pomocou konverznej funkcie, čo nie je nič iné ako prekrytý operátor pretypovania, alebo pomocou konštruktora s práve jedným pevným argumentom vhodného typu. Na osvieženie pamäti príklad:

```
class C
{
public:
  C(int);
};

void f(C) {}

void main()
{
  f(1);
  C c = 2;
}
```

Trieda `C` obsahuje konštruktor s jedným argumentom typu `int`. V programe máme definovanú funkciu `f()` s jedným argumentom typu `C`, ktorú vo funkcii `main()` zavoláme s argumentom typu `int`. Prekladač pri konverzii hodnoty `1` typu `int` na typ `C` použije spomínaný konštruktor `C::C(int)`. Druhý prípad implicitnej konverzie nastáva pri inicializácii premennej `c` typu `C` hodnotou `2`. Vieme, že riadok

```
C c = 2;
```

je ekvivalentný riadku

```
C c = C(2);
```

takže pri inicializácii objektu `c` sa implicitným volaním konštruktora `C::C(int)` vytvorí dočasný objekt `C(2)`. Tento objekt sa odovzdá ako argument kopírovaciemu konštruktoru, ktorý na základe neho vytvorí objekt `c` (v našom prípade sme nedefinovali kopírovací konštruktor, použije sa preto implicitný, vytvorený prekladačom).

Nie vždy nám však bude takéto správanie prekladača vyhovovať. V takom prípade máme možnosť pomocou kľúčového slova `explicit` zakázať akékoľvek implicitné konverzie pomocou daného konštruktora. Špecifikátor `explicit` môžeme použiť iba v rámci deklarácie triedy a len pre konštruktory. Pridať ho môžeme aj k iným ako jednoparametrovým konštruktorom, nemá to však zmysel, pretože tieto iné konštruktory sa nemôžu zúčastňovať implicitných konverzií.

Prepíšeme teda deklaráciu triedy `C` z predchádzajúceho príkladu takto:

```
class C
{
public:
  explicit C(int);
};
```

Teraz oba pokusy o implicitnú konverziu zlyhajú a prekladač ohlási chybu. Ak chceme, môžeme prekladaču explicitne povedať, že danú konverziu požadujeme:

```
f(C(1));
C c = C(2);
```

Zmysel explicitných konštruktorov je predovšetkým v zabránení konštruovania nežiaducich objektov prekladačom v rozpore s úmyslami programátora.

### Výnimky

V kategórii výnimiek je súčasťou normy jedna nová vlastnosť, o ktorej však mám minimum informácií, a pokiaľ viem, dosiaľ neexistuje prekladač, ktorý by ju implementoval. Ide o to, že pri definícii konštruktora môžeme explicitne uviesť celé jeho telo priamo ako `try` blok. Rozdiel oproti bežnému spôsobu, keď je `try` blok vnorený v tele konštruktora, je v tom, že v prvom prípade dokážeme zachytiť aj výnimky, ktoré vzniknú pri inicializácii členov danej triedy na základe zoznamu inicializátorov, uvedeného za dvojbodkou a názvom konštruktora.

### Priestory mien

Poslednou novinkou, o ktorej si dnes povieme, sú priestory mien. Predstavte si väčší tím vývojárov, ktorí pracujú na nejakom rozsiahlejšom programe. Bežná situácia, ktorá si však vyžaduje pomerne veľkú disciplínu pri dodržiavaní pomenovacích konvencií pre funkcie či premenné, aby nedošlo ku kolíziám medzi názvami z rôznych modulov od rôznych členov tímu. Priestory mien pomáhajú zabraňovať týmto kolíziám veľmi jednoduchým spôsobom: každé deklarované meno môže byť priradené do svojho príslušného priestoru mien. Z hľadiska prístupu a viditeľnosti obsiahnutých mien sa priestory mien podobajú triedam. Mená deklarované v rozdielnych priestoroch mien môžu byť rovnaké a napriek tomu nedôjde k nijakej kolízii.

Deklarácia priestoru mien je veľmi priamočiara:

```
namespace meno { deklarácie }
```

Kľúčové slovo `namespace` je nasledované nepovinným menom deklarovaného priestoru mien. V zložených zátvorkách sa nachádzajú bežné deklarácie globálnych premenných, funkcií, tried či šablón. Ak sa v jednom súbore nachádza niekoľko `namespace` deklarácií s rovnakým menom, prekladač ich spojí do jednej. Priestory mien môžu byť vnorené. Ak neuvedieme meno priestoru, vytvoríme tzv. anonymný priestor mien. Všetky deklarácie v takomto priestore sa považujú za statické, a teda neviditeľné mimo daného súboru.

Pre už deklarovaný priestor mien môžeme zaviesť ľubovoľný počet jeho aliasov (synoným) pomocou deklarácie:

```
namespace meno1 = meno2 ;
```

Od tejto deklarácie bude $meno_2$ ekvivalentným názvom priestoru mien $meno_1$.

Prístup k menám deklarovaným v rámci nejakého priestoru mien je podobný prístupu k členom triedy: pred sprístupňované meno musíme uviesť meno priestoru spolu s operátorom `::` (štvorbodka). Funkcie deklarované vnútri priestoru mien môžeme definovať (t. j. uviesť ich telo) aj mimo deklarácie `namespace`, opäť však musíme explicitne pomocou štvorbodky k názvu funkcie doplniť meno priestoru, do ktorého patrí. Telo takýchto funkcií je takisto súčasťou príslušného priestoru mien.

Pokiaľ chceme kdekoľvek v programe používať mená deklarované v nejakom priestore mien bez nutnosti zakaždým uviesť ich plne kvalifikované meno, pomôže nám kľúčové slovo `using`. Použiť ho môžeme dvoma spôsobmi – rozlišujeme tzv. deklaráciu `using` a direktívu `using`.

Syntax deklarácie `using` je takáto:

```
using meno :: identifikátor ;
```

Uvedením tejto deklarácie si sprístupňujeme identifikátor deklarovaný v priestore mien s názvom meno priamo, bez nutnosti plne ho kvalifikovať.

Direktíva `using` naproti tomu takto sprístupňuje všetky identifikátory v danom priestore mien:

```
using namespace meno ;
```

Napríklad všetky deklarácie, ktoré sú súčasťou štandardnej knižnice C++, patria do jedného priestoru mien s názvom `std`. Ak teda chceme so štandardnou knižnicou pracovať bez nutnosti zakaždým uvádzať kvalifikačný reťazec `std::`, použijeme na začiatku programu direktívu:

```
using namespace std;
```

Aby sme však len neteoretizovali, nakoniec si ukážeme krátky príklad:

```
int i = 1;

namespace A
{
  int i = 2;
  void f(double);
}

void A::f(double d)
{ /* ... */ }

void main()
{
  int i = 3;
  printf(„%n\n"; i);      // lokálne i
  printf(„%n\n"; A::i);   // A::i
  printf(„%n\n"; ::i);    // globálne i
}
```

Funkcia `f()`, ktorá je deklarovaná v rámci priestoru mien `A`, má svoje telo uvedené mimo deklarácie `namespace`, musíme preto pri jej definícii uviesť plne kvalifikované meno `A::f()`. Ďalej v príklade máme deklarovanú globálnu premennú `i` s hodnotou `1` mimo akéhokoľvek priestoru mien. Okrem nej máme ešte jednu globálnu premennú `i` s hodnotou `2`, ktorá je však súčasťou priestoru mien `A`. Vo funkcii `main()` sa nachádza deklarácia lokálnej premennej `i` s hodnotou `3`. Jednotlivé premenné `i` sú prístupné spôsobom zrejmým z príkladu. Všimnite si, že aj tu na prístup k zakrytému globálnemu menu môžeme použiť unárny operátor `::`.

## Dvadsiata tretia časť: ŠTANDARDNÁ KNIŽNICA C++

Tak, a je to tu. Dnes sa pri seriáli o strastiach a slastiach C++ stretávame naposledy. Necelé dva roky som sa snažil podeliť sa s vami o svoje vedomosti a pomôcť tak rozšíriť rady programátorov v tomto pravdepodobne najmocnejšom a súčasne najnáročnejšom jazyku (aspoň podľa môjho názoru). Či sa mi to podarilo, to musíte vedieť predovšetkým vy. Za tie dva roky som od vás dostal nespočetne veľa ohlasov, z ktorých výrazná väčšina bola pozitívna, a preto pevne verím, že seriál splnil svoj cieľ.

Skôr však, než výklad o jazyku C++ ukončíme, musíme si ešte niečo povedať o štandardnej knižnici C++. Iste tušíte, o čo ide. Tak ako bola súčasťou jazyka C určitá štandardná množina funkcií, štruktúr, typov a makier, ktoré boli (alebo aspoň mali byť) k dispozícii v každej implementácii, i v C++ si nemusíme všetko programovať sami a mnoho potrebných funkcií a vzhľadom na objektový charakter C++ predovšetkým tried a šablón dostaneme v rámci implementácie práve v podobe štandardnej knižnice C++.

Pri uvažovaní nad objemom informácií, ktoré by som vám chcel sprostredkovať v tejto časti seriálu, som bol postavený pred dilemu: na jednej strane je štandardná knižnica C++ natoľko rozsiahla a komplikovaná, že preberať ju tu podrobne nemá zmysel – napokon knižnica je dôkladne zdokumentovaná v elektronických príručkách k dostupným prekladačom, kde si nájde informácie v detailnejšom opise každý, kto ich bude potrebovať. Na druhej strane si nemôžem dovoliť jej opis úplne vynechať, pretože je to súčasť normy ANSI a patrí k C++ ako chvost k mačke. Rozhodol som sa teda pre podobný spôsob výkladu ako pri rozprávaní o knižnici jazyka C – povieme si o jednotlivých oblastiach, ktoré štandardná knižnica C++ pokrýva, o najužitočnejších funkciách či triedach z každej oblasti a prípadne o špecifikách ich použitia. Všetky ďalšie informácie už budete musieť vyhľadať sami.

### Hlavičkové súbory

Kedysi dávno, na začiatku seriálu, sme si hovorili, že hlavičkové súbory v C++ majú obyčajne príponu `.H`, prípadne v niektorých implementáciách aj `.HPP`. Štandardná

knižnica C++ túto konvenciu dosť radikálnym spôsobom ruší – jej hlavičkové súbory sú totiž bez prípony. Keďže C++ už nejaký čas existuje a norma ANSI bola kodifikovaná len nedávno, je možné, že v niektorých implementáciách budú k dispozícii aj verzie súborov s príponou, ale vo všeobecnosti bude vhodné zvyknúť si na nový spôsob.

Ďalšia vec, o ktorej musíme vedieť skôr, než si začneme hovoriť o štandardnej knižnici, sa týka priestorov mien. V predchádzajúcom pokračovaní sme si okrem iného povedali, že všetky deklarované mená zo štandardnej knižnice C++ sú súčasťou priestoru mien s názvom `std`. Ak teda chceme použiť niektorú zo štandardných funkcií či tried, buď ju explicitne kvalifikujeme ako `std::identifikátor`, alebo použijeme deklaráciu `using` s príslušným identifikátorom (`using std::identifikátor;`), prípadne použijeme rovno direktívu `using namespace std;`, čím si sprístupníme všetky identifikátory z priestoru `std`. Toto však platí iba v prípade, že chceme pracovať s objektom, ktorého deklarácia sa nachádza vo vloženom hlavičkovom súbore bez prípony, napríklad `iostream`. Ak použijeme klasický variant, teda `iostream.h` (samozrejme, implementácia to musí umožňovať), direktívu `using` nepotrebujeme použiť, lebo ju prekladač bude predpokladať implicitne za nás.

Štandardná knižnica C++ preberá niekoľko (presne 18) hlavičkových súborov známych z jazyka C. Mierne však upravuje ich názov: jednak odstraňuje príponu `.H` (to sa dalo čakať), jednak predraďuje pred názov znak `c`. Je, samozrejme, možné v programe C++ použiť hlavičkový súbor jazyka C (napríklad `stdio.h`), ale odporúčam vám radšej používať nový tvar (t. j. napríklad `cstdio`). Rozdiel medzi `stdio.h` a `cstdio` by mal byť rovnaký ako medzi `iostream.h` a `iostream` (pozri vyššie), ale napríklad vo Visual C++ 6.0 sa obidva súbory správajú rovnako.

Pre úplnosť si uveďme zoznam spomínaných osemnástich hlavičkových súborov prebratých z jazyka C: `cassert`, `cctype`, `cerrno`, `cfloat`, `ciso646`, `climits`, `clocale`, `cmath`, `csetjmp`, `csignal`, `cstdarg`, `cstddef`, `cstdio`, `cstdlib`, `cstring`, `ctime`, `cwchar` a `cwctype`. Ich obsah je prakticky totožný s ich staršími ekvivalentmi – implementované sú často tak, že obsahujú okrem bežnej „omáčky“ jedinú direktívu `#include`. Nebudeme si o nich hovoriť nič podrobnejšie, opakovali by sme totiž len pätnástu časť seriálu, kde sa možno dozvedieť o štandardnej knižnici jazyka C viac.

Okrem deklarácií prebratých z jazyka C obsahuje štandardná knižnica C++ mnoho nových funkcií, tried a šablón, ktorých deklarácie nájdeme v týchto hlavičkových súboroch: `algorithm`, `bitset`, `complex`, `deque`, `exception`, `fstream`, `functional`, `iomanip`, `ios`, `iosfwd`, `iostream`, `istream`, `iterator`, `limits`, `list`, `locale`, `map`, `memory`, `numeric`, `ostream`, `queue`, `set`, `sstream`, `stack`, `stdexcept`, `streambuf`, `string`, `strstream`, `utility`, `valarray` a `vector`. V ďalšom texte si jednotlivé súbory rozdelíme do niekoľkých oblastí.

## Prúdové triedy

Hádam najdôležitejšími a najčastejšie používanými triedami zo štandardnej knižnice sú triedy na prácu s prúdmi. Už pri opisovaní štandardnej knižnice jazyka C sme si uviedli, akým mocným prostriedkom sú prúdy a čo je ich výhodou. Na zopakovanie: prúdy predstavujú generalizované zdroje či spotrebiče údajov, ktoré môžu byť napojené na diskové súbory, pomenované rúry (?!), reťazce v pamäti, klávesnicu a pod. Neoddiskutovateľnou prednosťou prúdov je jednotný spôsob prístupu k nim – údaje zo súboru i z konzoly môžeme čítať rovnakými programovými prostriedkami a to isté platí pre zápis.

Štandardná knižnica C++ obaľuje koncepciu prúdov do objektovej podoby a poskytuje nám hierarchiu tried, pomocou ktorých môžeme pristupovať k údajom. Na dokonalú možnosť kontroly práce s prúdmi poskytuje množstvo riadiacich funkcií a pre zjednodušenie pretypuje niektoré základné operátory.

Prúdové triedy a funkcie na prácu s nimi sú deklarované v hlavičkových súboroch `fstream`, `iomanip`, `ios`, `iosfwd`, `iostream`, `istream`, `ostream`, `sstream`, `streambuf` a `strstream`. Názvy jednotlivých hlavičkových súborov korešpondujú s triedami, ktoré sú v nich obsiahnuté.

Základom celej hierarchie je trieda `ios_base`. Táto trieda obsahuje predovšetkým údajové členy na formátovanie vstupu a výstupu údajov cez prúdy a na zaznamenanie informácií o vnútornom stave prúdu a niekoľko funkcií na prístup k týmto členom. Samostatne sa nepoužíva.

Od triedy `ios_base` je odvodená trieda `ios`. V skutočnosti je táto trieda inštanciou šablóny `basic_ios`, ktorá umožňuje určiť, či budú prúdy pracovať s jednobajtovými alebo multibajtovými znakmi. Pre jednoduchosť budeme v ďalšom texte hovoriť len o jednobajtových verziách tried. Všetky prúdové triedy, o ktorých si povieme, budú teda inštanciami príslušných šablón, ktorých mená dostaneme predradením reťazca `basic_` pred názov triedy.

Trieda `ios` vytvára nadviazanie prúdu na vyrovnávaciu pamäť (inštanciu triedy `streambuf`) a poskytuje funkcie na riadenie tejto vyrovnávacej pamäte, ako aj na riadenie stavu prúdu. Ani táto trieda sa samostatne nepoužíva.

Trieda `streambuf` predstavuje všeobecný typ pre vyrovnávaciu pamäť prúdu. Nie je inštanciovateľná, pretože neobsahuje nijaký verejný konštruktor. Obsahuje mnoho členských funkcií na riadenie vyrovnávacej pamäte, za bežných okolností však s ňou ako programátori veľmi do styku neprídeme. Trieda `streambuf` je vrcholom inej hierarchie tried, nemá spojitosť (z hľadiska dedičnosti) s triedami odvodenými od `ios`.

Prvou použiteľnou triedou v hlavnej hierarchii je trieda `istream`, ktorá je priamym potomkom triedy `ios`. Ako jej názov evokuje, je určená na vstup údajov, ktoré číta z pridruženej vyrovnávacej pamäte. Ak však chceme triedu `istream` reálne použiť, musíme sa o pridelenie vhodnej vyrovnávacej pamäte postarať sami. Preto je táto trieda vhodná skôr v prípade, že potrebujeme čítať údaje z iných zdrojov, ako nám poskytuje štandardná knižnica (napríklad zo sieťových socketov). Trieda `istream` obsahuje niekoľko členských funkcií na čítanie jedného či viacerých znakov [`get()`, `read()`], na vrátenie naposledy prečítaného znaku späť do prúdu [`putback()`] a na zmenu aktuálnej pozície ukazovateľa vstupu [`seekg()` – má, samozrejme, zmysel len pri podobných zdrojoch údajov, ako sú súbory]. Navyše poskytuje pretypovaný operátor `>>` na formátované čítanie údajov. Príklad použitia uvedieme o chvíľu. Počas behu programu máme k dispozícii automaticky vytvorenú inštanciu triedy `istream` s názvom `cin`, ktorá je naviazaná na vstupnú konzolu, a teda je ekvivalentom neobjektového prúdu `stdin`.

Komplementárnou k triede `istream` je (prekvapujúco) trieda `ostream`, takisto odvodená od triedy `ios`. Aj táto trieda je združená s vyrovnávacou pamäťou a pre spôsoby jej využitia platí to, čo sme uviedli v predchádzajúcom odseku. Členské funkcie triedy `ostream` majú podobný význam – zápis jedného či viacerých znakov [`put()`, `write()`], zmena ukazovateľa výstupnej pozície [`seekp()`], spláchnutie (vyčistenie??) vyrovnávacej pamäte [`flush()`]. Okrem toho môžeme na formátovaný výstup údajov používať pretypovaný operátor `<<`. Pri štarte programu sa automaticky vytvoria tri inštancie triedy `ostream` s názvami `cout`, `cerr` a `clog`, napojené na výstupnú konzolu. Prvá z nich je ekvivalentom prúdu `stdout`, druhé dve nahrádzajú prúd `stderr`. Rozdiel medzi nimi je v tom, že `cerr` je bez vyrovnávacej pamäte a `clog` ju obsahuje.

Od tried `istream` a `ostream` je odvodená trieda `iostream`, ktorá slúži na implementáciu prúdov, nad ktorými potrebujeme súčasne vykonávať operácie čítania aj zápisu. Sama osebe nezavádza nijakú novú funkčnosť, je prostým zložením oboch svojich rodičovských tried.

Štandardná knižnica poskytuje dve špecializácie prúdových tried pre konkrétne údajové zdroje či spotrebiče. Prvou sú prúdy pracujúce s reťazcami. Vyrovnávaciu pamäť nadviazanú na reťazec v pamäti predstavuje trieda `stringbuf`. Táto trieda je potomkom triedy `streambuf` a reťazec, s ktorým je spojená, musí byť inštanciou triedy `string` (opíšeme neskôr). S funkciami triedy `stringbuf` podobne ako pri jej rodičovi do kontaktu veľmi neprichádzame.

Vstupný prúd na čítanie z reťazca je realizovaný triedou `istringstream`. Táto trieda je odvodená od triedy `istream`, ktorej funkčnosť bezo zvyšku preberá a navyše pridáva len možnosť globálneho prístupu k reťazcu, na ktorý je nadviazaná.

Výstup do reťazca má na starosti trieda `ostringstream`. Ako potomok triedy `ostream` umožňuje rovnaký spôsob zápisu údajov, pridaná funkčnosť je totožná s triedou `istringstream`.

Logickým spojením oboch predchádzajúcich tried je trieda na vstup aj výstup z/do reťazca s predpokladaným názvom `stringstream`. Pozor, táto trieda je odvodená od triedy `iostream`, ku ktorej pridáva rovnakú funkčnosť ako `istringstream` a `ostringstream`.

V niektorých implementáciách nájdeme aj prúdy, ktoré budú pracovať nad klasickými, nulou ukončenými reťazcami jazyka C. Triedy, ktoré ich realizujú, majú podobné názvy: `strstreambuf`, `istrstream`, `ostrstream` a `strstream`.

Druhou špecializáciou sú prúdy na prácu so súbormi. Ich základom je opäť vyrovnávacia pamäť, ktorú tentoraz predstavuje trieda `filebuf`. Táto trieda je potomkom triedy `streambuf` a z jej verejne prístupných členských funkcií sú zaujímavé funkcie `open()` a `close()` na otvorenie a zavretie príslušného súboru a funkcie na nastavenie aktuálnej pozície [`seekpos()`, `seekoff()`]. V praxi je trieda `filebuf` opäť viac-menej v pozadí.

Na čítanie zo súboru máme k dispozícii triedu `ifstream`. Jej predkom je podľa očakávania trieda `istream`, a tak z hľadiska funkčnosti sú v triede `ifstream` nové len funkcie `open()` a `close()`, ktoré volajú svoje ekvivalenty z triedy `filebuf`. Súbor, nad ktorým trieda `ifstream` pracuje, sa zadáva buď ako argument konštruktora, alebo ako argument funkcie `open()`.

Trieda `ofstream` naproti tomu má na starosti výstup do súboru. Asi netreba nijako zdôrazňovať, že je odvodená od triedy `ostream` a že obsahuje podobné funkcie `open()` a `close()` ako trieda `ifstream`. Meno súboru, do ktorého zapisujeme, sa zadáva rovnako ako v predchádzajúcom prípade.

Poslednou triedou, ktorá zapadne do celkovej mozaiky, je trieda `fstream`, slúžiaca, ako iste tušíte, na súčasné čítanie aj zápis z/do súboru. Táto trieda je odvodená od triedy `iostream` a takisto zavádza funkcie `open()` a `close()` na otvorenie, resp. zavretie súboru.

V hlavičkovom súbore `iomanip` nájdeme niekoľko tzv. manipulátorov, čo sú funkcie, ktoré určitým spôsobom upravujú formátovanie výstupu. Používajú sa v súvislosti s pretypovanými operátormi `<<` a `>>`.

Ukážme si teraz niekoľko príkladov na použitie prúdov. Prvý úsek programu vytlačí na konzolu vhodne formátovanú tabuľku druhých mocnín prvých desiatich prirodzených čísel:

```
#include <iostream>
#include <iomanip>
using namespace std;

void main()
{
  for (int i = 1; i <= 10; i++)
      cout << setw(2) << i << setw(4) <<
i*i << endl;
}
```

V programe vidíme, akým spôsobom sa používa pretypovaný operátor << na zápis údajov do prúdu. Ľavým jeho operandom musí byť príslušný prúd, pravým premenná či konštanta, ktorú chceme do prúdu zapísať. Pozor, číselné hodnoty sa zapisujú ako reťazce, a nie ako binárne čísla! Vypísanie obsahu premennej `i` na konzolu zabezpečíme teda jednoduchým zápisom `cout << i`. Aký je to rozdiel oproti klasickému `printf(„%d", i)`, kde musíme navyše vo formátovacom reťazci explicitne vyjadriť typ premennej `i`! Čítanie údajov z prúdu sa realizuje podobne: `cin >> i` spôsobí, že sa z prúdu `cin` prečíta reťazec číslic tvoriaci celé číslo a uloží sa po konverzii do premennej `i`. Pravidlá rozoznávania číselných údajov sú rovnaké ako pri rodine funkcií `scanf()`.

Pretypovaný operátor <<, resp. >> vracia ako návratovú hodnotu referenciu na prúd, s ktorým pracoval, preto je možné operátory reťaziť ako v našom príklade. Operátory sú deklarované ako nečlenské funkcie. Pre výstup vlastných údajových typov (obyčajne objektových) môžeme v programe oba operátory prekryť vlastnými funkciami, ale musíme dodržať prototyp, ktorý je v prípade operátora << takýto:

```
ostream& operator<<(ostream&, typ);
```

a v prípade operátora >> takýto:

```
istream& operator>>(istream&, typ&);
```

Prekryté funkcie dostávajú ako jeden z argumentov referenciu na príslušný prúd, s ktorým môžu pracovať podľa svojich potrieb. Na účel možného zreťazenia operátorov musia vlastné funkcie vracať referenciu na ten istý prúd, ktorý dostali ako argument.

V programe vidíme aj použitie manipulátorov prúdu: `setw()` nastavuje šírku výstupného poľa, zapisovaný údaj bude v tomto poli zarovnaný doprava. Manipulátory tohto typu [okrem `setw()` sú to ešte `setprecision()`, `setfill()`, `setbase()`, `setiosflags()`] sú vlastne náhradou možností formátovacieho reťazca funkcie `printf()`. Zvláštnym manipulátorom je funkcia `endl`, ktorá zapisuje do prúdu znak nového riadka. Na rozdiel od priameho zápisu znaku `'\n'` navyše spláchne (vyčistí??) výstupnú vyrovnávaciu pamäť prúdu. Spolu s podobnou funkciou `ends`, ktorá zapisuje nulový ukončovač reťazca, sú deklarované v hlavičkovom súbore `ostream`. K dispozícii máme aj niekoľko manipulátorov, ktoré sú skratkovými vyjadreniami uvedených parametrických manipulátorov, ako napríklad `dec`, `hex`, `left`, `right`, `scientific` a pod. Zoznam všetkých možných manipulátorov nájdete v dokumentácii k prekladaču.

Druhý príklad bude čítať údaje z jedného súboru (`in.txt`) a zapisovať ich do druhého súboru (`out.txt`):

```
#include <iostream>
#include <fstream>
using namespace std;

void main()
{
  ifstream in(„in.txt");
  if (!in)
  {
       cerr << „Input file open error" <<
endl;
    return;
  }

  ofstream out(„out.txt");
  if (!out)
  {
       cerr << „Output file open error"
<< endl;
    in.close();
    return;
  }

  while (true)
  {
    unsigned char c;

    in >> c;
    if (in.eof())
      break;
    if (!in)
    {
      cerr << „Read error" << endl;
      break;
    }

    out << c;
    if (!out)
    {
      cerr << „Write error" << endl;
      break;
    }
  }

  in.close();
  out.close();
}
```

Program je doplnený aj ošetrením prípadných chybových stavov. Vidíme, že prúdové triedy majú prekrytý operátor `!`, ktorý svojou nenulovou hodnotou indikuje chybu pri práci s prúdom. Koniec súboru detegujeme pomocou členskej funkcie `eof()`. Pre zaujímavosť si skúste program napísať pomocou klasických funkcií jazyka C – uvidíte, ktorý spôsob je čitateľnejší a jednoduchší.

## Kontajnerové triedy

Druhou skupinou tried v štandardnej knižnici sú kontajnerové triedy. Kontajnerom sa v tomto prípade myslí zvláštny údajový typ, ktorý slúži na uchovávanie hodnôt rovnakého typu. Kontajnery sú v podstate tým, čo absolventi informatiky (ale nielen oni) poznajú z programovacích techník pod názvom abstraktný údajový typ (ADT). Všetky ďalej uvedené triedy sú šablónami, umožňujú tak pracovať s ľubovoľným typom. Tento typ ukladaných údajov je vždy parametrom šablóny.

Prvou kontajnerovou triedou je `bitset`. Táto trieda predstavuje bitovú mapu, čo je postupnosť bitov, ktoré môžeme po jednom nastavovať, nulovať, invertovať; prístup k nim je možný takisto pomocou pretypovaného indexovacieho operátora. Okrem neho je pretypované aj množstvo iných operátorov na zjednodušenie práce s bitovou mapou. Veľkosť mapy je parametrom šablóny.

Trieda `deque` je „obojstranným" frontom (z anglického double-ended queue). Obojstranný front umožňuje na rozdiel od bežného frontu (pozri ďalej) prístup k údajom na obidvoch koncoch a aj náhodný prístup k ľubovoľnému prvku frontu. Na pridávanie prvkov obsahuje funkcie `insert()`, `push_front()`, `push_back()`, na odoberanie `erase()`, `pop_front()` a `pop_back()`. Na priamy prístup máme k dispozícii pretypovaný operátor indexovania.

Ďalšou triedou je trieda `list`, ktorá implementuje obojsmerný zreťazený zoznam. Vkladanie a vyberanie prvkov sa realizuje podobnými funkciami ako v triede `deque`, ale nie je možný priamy indexovaný prístup. Zoznam možno, samozrejme, usporiadať a pomocou funkcie `merge()` aj vložiť do iného zoznamu pri zachovaní usporiadania.

Triedy `map` a `multimap` realizujú typ, obyčajne nazývaný asociatívne pole, hash tabuľka alebo slovník. Prístup k jednotlivým prvkom takéhoto poľa sa deje prostredníctvom kľúčov, ku ktorým sú asociované uložené hodnoty. Je zrejmé, že pri ukladaní prvkov do poľa musíme uviesť vždy dvojicu kľúč – hodnota. Typ kľúča je jedným z parametrov šablóny. Trieda `multimap` na rozdiel od triedy `map` povoľuje pre jeden kľúč viacero asociovaných hodnôt a neumožňuje prístup a vkladanie hodnôt pomocou indexovacieho operátora. Hodnoty môžeme vkladať aj funkciou `insert()`, mazanie má na starosti funkcia `erase()`.

Variáciou na tému front je trieda `priority_queue`, ktorá predstavuje prioritný front, čo je front, ktorého prvky sú zoradené podľa priority a vkladanie toto zoradenie zachováva. Inak sa nelíši od nasledujúcej triedy.

Trieda `queue` je implementáciou klasickej FIFO štruktúry, nazývanej front. Do frontu sa údaje pridávajú na jednej strane a odoberajú na druhej strane. Iný prístup k prvkom frontu nie je možný. Pridávanie má na starosti funkcia `push()`, odoberanie funkcia `pop()`.

Triedy `set` a `multiset` sú príbuzné triedam `map` a `multimap`, na rozdiel od nich však obsahujú iba kľúče. Trieda `set` zaručene obsahuje rôzne kľúče, zatiaľ čo pre triedu `multiset` toto obmedzenie neplatí. Obe triedy v podstate realizujú všeobecný koncept množiny.

Zásobník je implementovaný pomocou triedy `stack`. Ide o klasickú štruktúru typu LIFO, z ktorej môžeme vyberať prvky len presne v opačnom poradí, ako boli do nej vložené. Podobne ako pri triede `queue` vkladanie a vyberanie majú na starosti triedy `push()` a `pop()`.

Poslednou kontajnerovou triedou je trieda `vector`. Tá predstavuje jeden z najpoužívanejších typov, ktorý si môžeme predstaviť ako dynamické pole, ktorého veľkosť sa prispôsobuje aktuálnemu počtu prvkov. Je podobná triedam `deque` a `list`, každá z týchto troch tried je však optimalizovaná pre iný spôsob prístupu k prvkom. Bližšie detaily presahujú rámec nášho rozprávania a nájdete ich v dokumentácii.

S kontajnerovými triedami sú neoddeliteľne späté tzv. iterátory. Ide o niekoľko tried, ktoré predstavujú objektové typy podobné ukazovateľom, pomocou ktorých môžeme k jednotlivým prvkom kontajnera pristupovať podobným spôsobom, ako môžeme pomocou bežných ukazovateľov pristupovať k prvkom bežných, neobjektových polí. Menovite ich môžeme dereferencovať, inkrementovať a niektoré i dekrementovať, porovnávať medzi sebou a pod. V hlavičkovom súbore `algorithm` nájdeme niekoľko funkcií, ktoré realizujú množstvo často používaných operácií nad kontajnermi a na svoju činnosť využívajú práve iterátory.

Iterátormi sa tu bližšie zaoberať nebudeme. Ide o priveľmi sofistikovanú tému na to, aby sa dala objasniť na takom malom priestore, takže ak sa nenahneváte, odkážem vás opäť na elektronickú dokumentáciu, kde nájdete presný opis jednotlivých druhov iterátorov a príklady ich použitia.

### Ďalšie triedy

ANSI C++ poskytuje sústavu tried na podporu národných prostredí, pomocou ktorých môže program používať správne formátovanie čísel, dátumu, času a meny. Základom tejto sústavy je trieda `locale`, ktorá definuje všetky potrebné charakteristiky prostredia a umožňuje jeho dynamickú zmenu. Okrem tejto triedy nájdeme v hlavičkovom súbore s rovnakým názvom `locale` niekoľko ďalších pomocných tried na opis konverzie znakov, opis formátovacích pravidiel a pod.

V štandardnej knižnici C++ máme k dispozícii niekoľko nových tried s uplatnením v matematike. Pravdepodobne najdôležitejšia je trieda `complex`, ktorá reprezentuje komplexné čísla, definuje pre ne množstvo operácií a pretypuje najbežnejšie operátory. S inštanciami triedy `complex` pracujú nové verzie matematických funkcií ako `sin()`, `exp()` a pod. (áno, funkcia sínus je definovaná aj na komplexnom obore).

Ostatné matematické triedy sú menej dôležité – trieda `valarray` slúži na reprezentáciu usporiadaných množín, s ňou spolupracujú triedy `slice` a `gslice` na výber z množiny. Trieda `numeric_limits` opisuje implementačne závislé vlastnosti číselných údajových typov.

Pre prácu s reťazcami bola zavedená trieda `string` (v skutočnosti inštancia šablóny `basic_string` pre typ `char`; možné sú, samozrejme, aj viacbajtové reťazce). Členské funkcie triedy `string` pokrývajú širokú škálu operácií nad reťazcami – od rozširovania, skracovania reťazca cez úpravy častí reťazca až po prehľadávanie a porovnávanie rôznych reťazcov.

Výnimky majú v štandardnej knižnici vlastnú hierarchiu tried. Na jej vrchole je trieda `exception`, od ktorej sú odvodené dve triedy `logic_error` a `runtime_error` pre dva rôzne okruhy výnimiek. Od týchto tried sú potom odvodené konkrétne výnimkové triedy špecifické pre tú-ktorú časť štandardnej knižnice. Za zmienku stoja ešte výnimky vyhadzované priamo konštrukciami jazyka, ako `bad_alloc` (pri chybe alokácie operátorom `new`), `bad_exception` (pri výskyte neočakávanej výnimky), `bad_cast` (pri nedovolenom pretypovaní pomocou operátora `dynamic_cast`) a `bad_typeid` (pri nesprávnom použití operátora `typeid`). Všetky výnimkové triedy sú deklarované v hlavičkovom súbore `stdexcept`.

Štandardná knižnica C++ obsahuje okrem všetkých týchto tried množstvo ďalších, na ktoré sa tu dnes nedostalo. Bohužiaľ, nie je možné opísať tu celú knižnicu podrobne, zabralo by to podstatne viac miesta, ako poskytuje jedno celé číslo PC REVUE. Vy však viete, že najlepším učiteľom je vlastná prax a najviac sa vždy dozviete samostatným hľadaním v dostupných manuáloch, referenčných príručkách a v neposlednom rade aj opakovaným skúšaním vo vlastných programoch. Tento prístup k učeniu som razil počas celého seriálu a verím, že sa ním riadite a budete riadiť aj naďalej.

## Čo dodať na záver?

Rozprávaním o štandardnej knižnici sme teda vyčerpali celú problematiku C++. Bez zbytočnej samochvály si myslím, že dosiaľ neexistuje v slovenčine materiál, ktorý by sa venoval ANSI C++ v takomto rozsahu. Pevne verím, že tým, ktorí C++ neovládali, seriál pomohol tento veľmi krásny jazyk zvládnuť, tým, ktorí už s C++ prišli do styku, zase pomohol osviežiť pamäť a zopakovať zabudnuté vedomosti. Skôr než skončím, dovoľte mi poďakovať sa vám za priazeň i za ohlasy, ktoré ma presvedčili, že to, čo robím, má zmysel. Aby som bol korektný voči tým, ktorí pomohli zase mne, uvádzam tu ešte zoznam literatúry, z ktorej som čerpal. Základným a neoceniteľným zdrojom informácií bola kniha C++ Reference Manual od Bjarne Stroustrupa. V novších veciach mi nesmierne pomohla kniha Jazyky C a C++ podle normy ANSI/ISO z vydavateľstva Grada. Jeden z jej autorov, Miroslav Virius, má „na svedomí“ mnoho článkov v časopisoch Chip či Softwarové noviny, ktoré mi často pomohli objasniť to, čomu som dosiaľ nie celkom rozumel. No a, samozrejme, nemôžem vynechať dokumentáciu k prekladačom, predovšetkým k výbornému Visual C++ 6.0, ktorého som, podotýkam, legálnym vlastníkom (ak by to niekoho zaujímalo ;–).

■ **Vladimír Klimovský**