Čo je to C++

Je to rozsiahly objektovo-orientovaný programovací jazyk. Napriek tomu, že vznikol postupným vývojom z jazyka C, na prechod od programovania v jazyku C do programovania v jazyku C++ nestačí iba zmeniť kompilátor a naučiť sa nové syntaktické prvky - je nutné zmeniť i spôsob programovania, t.j. prejsť od procedurálneho na objektovo-orientované programovanie. C++ nie je čisto objektovo-orientovaný jazyk (ako napr. Smalltalk), práve kvôli zachovaniu aspoň čiastočnej kompatibility s C jazykom. Pokiaľ sa o C jazyku hovorí, že je to *najnižší vyšší programovací programovací jazyk*, o C++ by sme mohli povedať, že je to najnižší *objektovo-orientovaný programovací jazyk*.

Celá táto časť je venovaná práve rozdielom medzi C a C++ jazykom, t.j. doplnkom, o ktoré jazyk C++ rozširuje jazyk C. Vyžaduje preto aspoň základnú znalosť jazyka C.

Rozšírenia môžeme rozdeliť do dvoch základných skupín:

- formát programu v C++
- syntaktické rozšírenia a vylepšenia jazyka
- prvky, ktoré podporujú objektovo-orientované programovanie

Proces jazykového prekladu

Všetky počítačové jazyky prevádzajú z niečoho, čo má tendenciu byť ľahké pre ľudské porozumenie (zdrojový kód) do niečoho, čo sa vykonáva na počítači (strojové inštrukcie). Tradične sa prekladače delia na dve skupiny: ***interpretery*** a ***kompilátory***.

Interpretery

Interpreter prekladá zdrojový kód do činností (ktoré môžu pozostávať zo skupiny strojových inštrukcií) a okamžite vykonáva tieto činnosti. Takýmot jazykom je napríklad populárny BASIC. Tradičný BASIC interpretery prekladajú a vykonávajú riadok po riadku a potom na vykonané riadky zabudnú. Toto ich spomaľuje, pretože prevádzať každý opakovaný kód. BASIC bol kvôli rýchlosti i kompilovaný. Modernejšie interpretery ako je napríklad jazyk Python, prekladajú celý program do medzijazyka, ktorý sa potom vykonáva rýchlejším interpreterom.

Interpretery majú mnoho výhod. Prechod z napísaného kódu do vykonateľného kódu je takmer okmažitý a zdrojový kód je vždy k dispozícií, takže interpreter môže byť podrobnejší, keď nastanú chyby. Často citovanou výhodou interpretera je jednoduchšia vzájomná interakcia rýchlejší vývoj (ale nie nevyhnutne vykonávanie) programov.

Interpretované jazyky majú často niekoľko vážných obmedzení pri budovaní rozsiahlých projektov (Python sa javí byť výnimkou). Interpreter (alebo redukovaná verzia) musí byť vždy v pamäti, aby sa kód mohol vykonávať a i najrýchlejší interpreter môže spôsobovať neočakávané obmedzenia rýchlosti. Väčšina interpreterov vyžaduje, aby bol celý zdrojový kód prenesený do interpretera naraz. Toto nielen že zavádza obmedzenie kapacity, ale môže to spôsobiť omnoho závažnejšie chyby, ak jazyk nemá schopnosť lokalizovať dôzledky rôznych častí kódu.

Kompilátory

Kompilátor prevádza zdrojový kód priamo do jazyka symbolických adries (assembler) alebo strojových inštrukcií. Potencionálnym koncovým produktom je súbor alebo súbory, obsahujúce strojový kód. Toto je zložitý porces a zvyčajne pozostáva z niekoľkých krokov. Prechod z napísaného kódu do vykonateľného kódu je pri kompilátore podstatne dlhší.

V závislosti od chytrosti tvorcu kompilátora, programy, generované kompilátorom vyžadujú menej priestoru na svoje vykonávanie a vykonávajú sa omnoho rýchlejšie. Hoci rozmer a rýchlosť sú pravdepodobne najčastejšie citované argumenty pre použitie kompilátora, v mnohých prípadoch to nie sú najdoležitejšie dôvody. Niektoré jazyky (ako je napríklad C) sú navrhnuté tak, že dovoľujú prekladať časti programu nezávisle. Tieto kúsky sa spájajú do finálneho *vykonateľného* programu nástrojom, nazývanym *linker*. Tento proces sa nazýva *oddelená kompilácia*.

Oddelená kompilácia má mnoho výhod. Program, ktorý by ako celok prekročil obmedzenia kompilátora alebo kompilačného prostredia, sa môže kompilovať po kúskoch. Programy sa môžu budovať a testovať po častiach. Akonáhle kúsok funguje. Môžeme hoc uložiť a pracovať s ním ako so stavebným blokom. Kolekcie otestovaných a fungujúcich častí môžeme spájať do ***knižníc***, ktoré môžu používať ďalší programátori. Po vytvorení časti je zložitosť ďalšieho kúska skrytá. Všetky tieto črty podporujú tvorbu veľkých programov.

Postupom času sa značne vylepšili ladiace vlastnosti kompilátora. Prvé kompilátory iba generovali strojový kód a programátor vkladal výpisové príkazy, aby vedel čo sa deje. Toto nie je vždy efektívne. Moderné kompilátory dokážu do vykonateľného kódu vkladať informácie o zdrojovom kóde. Tieto informácie využívajú výkonné ladiace nástroje na úrovni zdrojového kódu, aby presne ukázali, čo sa deje v programe pri trasovaní zdrojového kódu.

Niektoré kompilátory riešia problém rýchlosti kompilácie vykonávaním tzv. kompilácie v pamäti (in-memory-compilation). Väčšina kompilátorov narába so súbormi, ktoré číta a zapisuje v každom kroku kompilačného procesu. Kompilátory v pamäti uchovávajú program v RAM. Pre malé programy je to ako pri interpreteroch.

Proces kompilácie

Aby sme mohli programovať v C a C++, potrebujeme pochopiť kroky a nástroje kompilačného procesu. Niektoré jazyky (konkrétne C a C++) začínajú kompiláciu odštartovaním ***preprocesora*** zdrojového kódu. Preprocesor je jednoduchý program, ktorý v zdrojovom kóde nahradí šablóny inými šablónami, ktoré programátor definoval (pomocou direktív preprocesora). Direktívy preprocesora šetria písanie a zvyšujú čitateľnosť kódu (aj keď C++ odradzuje od používania preprocesora, pretože to môže spôsobovať zákerné chyby). Predspracovaný kód sa často zapisuje do medzisúboru.

Kompilátory robia svoju činnosť v dvoch prechodoch. Prvý prechod analyzuje predspracovaný kód. Kompilátor rozdelí kód do malých jednotiek a organizuje ich do štruktúr, nazývaných ***strom***. Vo výraze "**A + B**" prvky '**A**', '+' a '**B**' sú listami stromu.

Medzi prvým a druhým prechodom sa občas používa ***globálny optimalizátor***, aby sa vytvoril menší a rýchlejší kód.

V druhom prechode prechádza ***generátor kódu*** cez strom a generuje buď kód v jazyku symbolických adries (asembleri) alebo strojový kód pre uzly stromu. Ak generátor kódu vytvára asemblerovský kód, potom sa musí odštartovať asembler. Výsledkom v obidvoch prípadoch je cieľový modul (object module – súbor má zvyčajne koncovku **.o** alebo **.obj**). V druhom prechode sa občas aplikuje optimalizátor, ktorý vyhľadáva kúsky kódu, obsahujúceho redundantné príkazy asembleru.

Použitie pojmu „object" na popis kúskov strojového kódu je nanešťastie artefaktom. Pojem vznikol pred rozšírením obejktovo-orientovaného programovania. „Object" sa používa v rovnakom zmysle ako „cieľ", keď hovoríme o kompilácií, zatiaľ čo v objektovo-orientovanom programovaní znamená „vec s ohraničením".

Linker spája zoznam cieľových modulov do vykonateľného programu, ktorý môžeme zaviesť a spustiť v operačnom systéme. Keď sa funkcia v jednom cieľovom module odkazuje na funkciu alebo premennú v inom cieľovom module, linker takýto odkaz vyrieši. Zabzpečí, že všetky externé funkcie a dáta, ktorých existunciu požadujeme počas kompilácie i skutočne existujú. Linker zároveň pridáva špeciálny cieľový modul, ktorý zabezpečuje štartovacie aktivity.

Linker dokáže prehľadávať špeciálne súbory, nazývané ***knižnice***, aby analyzoval všetky jej odkazy. Knižnica obsahuje kolekciu cieľových modulov v jedinom súbore. Knižnica sa vytvára a udržiava programom, nazývaným ***knihovník***.

Kontrola statických typov

Kompilátor vykonáva počas prvého prechodu ***kontrolu typov***. Kontrola typov testuje správne použitie argumentov vo funkciách a predchádza mnohým druhom programovacích chýb. Pretože kontrola typov sa robí počas kompilácie a nie počas vykonávania programu, nazýva sa ***statická kontrola typov***.

Niektoré objektovo-orientované jazyky (najmä Java) robia nejakú kontrolu typov za chodu (***dynamická kontrola typov***). Spojenie statickej a dynamickej kontroly typov je výkonnejšie než samotná statická kontrola typov. Avšak pridáva do vykonávania programu určitú réžiu.

C++ používa statickú kontrolu typov, pretože jazyk nedokáže predpokadať nejakú konkrétnu podporu zlých operácií za chodu programu. Statická kontrola typov oznamuje programátorovi chybné použitie typov počas kompilácie a tak maximalizuje rýchlosť vykonávania. Časom uvidíme, že C++ väčšina rozhodnutí návrhu jazyka uprednostňuej rovnaký druh rýchleho, výrobno-orientovaného programovania, ktoré preslávilo jazyk C.

V C++ ***môžeme*** statickú kontrolu typu vypnúť. Tiež môžeme vytvoriť vlastnú dynamickú kontrolu typu . akurát musíme napísať kód.

Nástroje na oddelenú kompiláciu

Oddelená kompilácia je obzvlášť dôležitá pri budovaní rozsiahlých projektov. V C a C++ môže byť program vytvorený z malých, ovládateľných, nezávisle testovaných kúskov. Najzákladnejším nástrojom na rozdelenie programu na kúsky je schopnosť vytvárať pomenované podprogramy. V C a C++ sa podporgram ***nazýva funkcia*** a funkcie sú kúsky kódu, ktoré môžu byť umiestnené v rôznych súboroch, ktoré môžu byť kompilované oddelene. Povedané inak, funkcia je atomická jednotka kódu, pretože nemôžeme mať časť funkcie v jednom súbore a ďalšiu jej časť v inom súbore. Celá funkcia musí byť umiestnená v jedinom súbore (hoci súbory môžu a obsahujú viac ako jednu funkciu).

Keď zavoláme funkciu, prenášame do nej zvyčajne nejaké ***argumenty***, čo sú hodnoty, s ktorými chceme aby funkcia pracovala počas svojho vykonávania. Keď funkcia skončí zvyčajne vraciame ***návratovú hodnotu***, t.j. hodnotu, ktorú nam funkcia dáva ako výsledok. Môžeme však písať funkcie, ktoré nemajú žiaden argument a nevracajú ani žaidnu hodnotu.

Aby sme napísal program z viacerých súborov, funkcie v jednom súbore musia pristupovať k funkciám a dátam v iných súboroch. Pri kompilácií músi C alebo C++ kompilátor poznať funkcie a dáta z ostatných súborov, konkrétne ich mená a správne použitie. Kompilátor zabezpečuje, že funkcie a dáta sú použité ***správne***. Tento proces „oznamovania kompilátoru“ mená externých funkcií a dát a ako by mali vypadať sa nazýva ***deklaracia***. Akonáhle deklarujeme funkciu alebo premennú, kompilátor pozná ako ma skontrolovať a zabezpečiť, že bola použitá správne.

Deklarácie versus definície

Je dôležité pochopiť rozdiel medzi ***deklaráciami*** *a* ***definíciami***, pretože tieto termíny sa musia používať presne. V zásade všetky C a C++ programy vyžadujú deklarácie. Skôr ako napíšeme prvý vlastný program, musíme pochopiť správny spôsob zápisu deklarácie.

Deklarácia predstavuje kompilátoru meno - identifikátor. Oznamuje kompilátoru „Táto funkcia alebo premenná niekde existuje a bude vypadať takto.“ Na druhej strane ***definícia*** hovorí: „Vytvor túto premenú tu“ alebo „ Vytvor túto funkciu tu.“ Alokuje pamäť pre meno. Tento význam platí či už hovoríme o premennej alebo funkcii. V obidvoch prípadoch v okamihu definície kompilátor alokue pamäť. Pre premennú kompilátor určuje veľkosť premennej a to je dôvod na generovanie pamäti na uchovanie dát tejto premennej. Pre funkciu kompilátor generuje kód, ktorý tiež skončí okupáciou priestoru v pamäti.

Premennú alebo funkciu môžeme deklarovať na mnohých miestach, ale definícia musí byť v C a C++ iba jedna (toto sa nazýva ODR – one-definition-rule). Keď linker spája všetky cieľové moduly, zvyčajne sa sťažuje, ak nájde viac ako jednu definíciu tej istej funkcie alebo premennej.

Definícia môže byť i deklaráciou. Ak kompilátor ešte nepoznal meno **x** a použijeme definíciu **int x;** komiplátor považuje meno sa deklaráciu a súčasne i alokuje priestor pre túto premennú.

Syntax deklarácie funkcie

Deklarácia funkcie v C a C++ poskytuje funkcii meno, typy argumentov, prenášaných do funkcie a návratovú hodnotu funkcie. Napríklad deklarácie funkcie nazvanej **func1( )** ktorá má dva argumenty (celé čísla sa v C/C++ označujú kľúčovým slovom **int**) a vracia celé číslo vypadá takto:

```
int func1(int,int);
```

Prvé kľúčové slovo je samotná návratová hodnota: **int**. Argumenty sú obklopené okrúhlymi zátvorkami za menom funkcie v takom poradí, v akom sú použité. Bodkočiarka indikuje koniec príkazu. V tomto prípade oznamuje kompilátoru,že „to je všetko – definícia funkcie tu nie je!“.

Deklarácie v C a C++ sa snažia napodobňovať formu použitia položky. Napríklad ak **a** je ďalšie celé číslo, vyššie uvedená funkcia sa môže použiť nasledujúcim spôsobom:

```
a = func1(2,3);
```

Pretože func1( ) vracia celé číslo, C a C++ kompilátor skontroluje použitie funkcie **func1( )**, aby zabezpečil, že **a** dokáže akceptovať návratovú hodnotu a že argumenty sú zodpovedajúce.

Argumenty funkcie môžu byť pomenované. Kompilátor ignoruje mená, ale môžu byť nápomocné ako menmonika pre užívateľa. Napríklad môžeme deklarovať **func1( )** iným spôsobom ale s rovnakým významom:

```
int func1(int length, int width);
```

Medzi funkciami s prázdnym zoznamom argumentov je v C a C++ významný rozdiel. V C deklarácia:

```
int func2();
```

predstavuje „funkciu s ľubovoľným počtom argumentov ľubovoľného typu“. Toto zamedzuje typovú kontrolu, takže v C++ toto znamená „funkciu bez argumentov“.

Definície funkcií

Definície funkcií sa podobajú deklaráciam funkcií, avšak majú telá. Telo je súbor príkazov, ohraničených zloženými zátvorkami. Zložené zátvorky označujú začiatok a koniec bloku kódu. Ak chceme funkcii **func1( )** dodať definíciu, ktorým je prázdne telo (telo, neobsahujúce žiaden kód), napíšeme:

```
int func1(int length, int width) { }
```

Všimnime si, že v definícií funkcie zložené zátvorky nahradili bodkočiarku. Pretože zložené zátvorky obklopujú príkaz alebo skupinu príkazov, nepotrebujeme bodkočiarku. Všimnime si tiež, že argumenty v definícii funkcie musia byť pomenované, ak chceme argumenty použiť v tele funkcie (pretože tu nie sú použité, sú nepovinné).

Syntax deklarácie premennej

Význam prisúdený fráze „deklarácia premennej“ bol historicky mätúci a protichodný a je dôležité pochopiť jeho správnu definíciu,a by sme mohli čítať kód správne. Deklarácia premennej oznamuje kompilátoru ako bude prtemenná vypadať. Hovorí „Viem, že toto meno si ešte nevidel, ale sľubujem, že kdesi existuje a je to premenná typu X“.

V deklarácií funkcie zadávame typ (návratovú hodnotu), meno funkcie, zoznam argumentov a bodkočiarku. To kompilátoru stačí, aby zistil že sa jedná o deklaráciu a ako by funkcia mala vypadať. Z toho odvodíme, že deklarácia premennej by mal byť typ za ktorým nasleduje meno. Napríklad:

```
int a;
```

by mohla byť deklarácia premennej **a** ako celého čísla na základe vyššie uvedenej logiky. A tu je konflikt: táto informácia kompilátoru postačuje na vytvorenie priestoru pre celočíslenú premennú nazvanú **a**, a to i nastane. Aby sme vyriešili túto situáciu, C a C++ potrebuje kľúčové slovo, ktoré bude hovoriť „toto je len deklarácia, definícia je na inom mieste.“ Týmto kľúčovým slovom je slovo **extern**. Znamená, že definícia je buď externou pre tento súbor alebo sa definicía nájde ´dalej v súbore.

Deklarovanie premennej bez definície znamená použitie kľúčového slova **extern** pred popisom premennej:

```
extern int a;
```

Kľúčové slovo **extern** môžeme tiež aplikovať na deklarácie funkcií. Pre **func1( )** by to vypadalo nasledovne:

```
extern int func1(int length, int width);
```

Tento príkaz je ekvivalentný s predchádzajúcou deklaráciou funkcie **func1( )**. Pretože nie je zadané telo funkcie, kompilátor musí toto považovať za deklaráciu funkcie a ne za jej definíciu. Kľúčové slovo **extern** je nadbytočné a nepovinné pre deklaráciu funkcie. Pravdepodobne je nešťastím, že tvorcovia C nevyžadovali použitie slova **extern** pre deklarácie funkcie. Mohlo by to byť konzistentnejšie (jednotnejšie) a menej mätúce (ale vyžadovalo by si to viac písania, čo pravdepodobne vysvetľuje rozhodnutie).

A tu je niekoľko príkladov deklarácií:

```
// príklady deklarácií a definícií
extern int i; // Declarácia bez definície
extern float f(float); // Declarácia funkcie

float b;  // Declarácia & definícia
float f(float a)  // Definícia
{
  return a + 1.0;
}
```

```
int i; // Definícia
int h(int x) // Declarácia & definícia
{
  return x + 1;
}

int main()
{
  b = 1.0;
  i = 2;
  f(b);
  h(i);
}
```

V deklaráciach funkcií sú identifikátory argumentov nepovinné. V definíciach sú požadované (identifikátory sú požadované iba v C, nie v C++).

Včleňovanie hlavičiek

Väčšina knižníc obsahuje značné množstvo funkcií a premenných. Aby sme si ušetrili prácu a zabezpečili konzistenciu externých deklarácií týchto položiek C a C++ používajú nástroj, nazývaný ***hlavičkový súbor***. Hlavičkový súbor je súbor obsahujúci externé deklarácie knižnice. Podľa dohody má príponu 'h', napríklad **headerfile.h**. (Niektoré staršie kódy používajú odlišné prípony, napríklad **.hxx** alebo **.hpp**, ale sú zriedkavé.)

Programátor, ktorý vytvára knižnicu dodáva i hlavičkový súbor. Ak chce užívateľ deklarovať funkcie a externé premenné z knižnice, jednoducho iba včlení hlavičkový súbor. Na včleňovanie hlavičkových súborov sa používa direktíva preprocesora **#include**. Táto direktíva prikáže preprocesoru otvoriť menovaný hlavičkový súbor a vložiť jeho obsah tam, kde sa nachádza príkaz **#include**. Príkaz **#include** môže menovať súbor dvomi spôsobmi: v ostrých zátvorkách (< >) alebo v úvodzovkách (" ").

Meno v špicatých zátvorkách, napríklad

```
#include <header>
```

donúti preprocesor hľadať súbor takým spôsobom, ktorá je špecifická pre implementáciu, ale zvyčajne existuje nejaký spôsob „vyhľadávacej cesty včleňovacieho súboru", ktorý špecifikujeme v prostredí alebo v príkazovom riadku kompilátora. Mechanizmus nastavovania vyľadávacej cesty závisí od počítača, operačného systému, C++ implementácie a môže od nás vyžadovať určité znalosti.

Mená súboru v úvodzovkách, napríklad:

```
#include "local.h"
```

prikazujú preprocesoru hľadať súbor (podľa špecifikácie) „implementačne-definovaným" spôsobom. Zvyčajne to znamená, že súbor sa hľadá relatívne na aktuálny adresár. Ak sa súbor nenájde, potom včleňovacia direktíva spracuje znova ako keby mala namiesto úvodzoviek špicaté zátvorky.

Na včlenenie hlavičkového súboru iostream napíšeme:

```
#include <iostream>
```

Preprocesor nájde hlavičkový súbor iostream (často v podadresári, nazvanom „include") a vloží ho.

Štandardný C++ formát včleňovania

Počas vývoja C++ rôzni tvorcovia kompilátorov volili rôzne prípony pre mená súborov. Okrem toho rozmanité operačné systémy kladú rôzne obmedzenia na mená súborov, konkrétne na ich dĺžku. Tieto záležitosti spôsobujú problémy s kompatibilitou zdrojového kódu. Na zmiernenie týchto ostrých hrán norma používa formát, ktorý dovoľuje mená súborov dlhšie, než je notorických osem znakov a aliminuje príponu. Napríklad namiesto starého štýlu včlenenia **iostream.h**, ktorý vypadá nasledovne:

```
#include <iostream.h>
```

môžeme písať:

```
  #include <iostream>
```

Prekladač môže implementovať včleňovacie príkazy takým spôsobom, ktorý vyhovuje požiadavkam konkrétneho kompilátora a operačného systému, v prípade potreby orezaním mena a pridaním prípony. Samozrejme môžeme hlavičkové súbory, dodávané kompilátorom, skopírovať do súborov bez prípon, ak chceme použiť tento štýl skôr, než ho tvorca kompilátora bude podporovať.

Knižnice, ktoré boli zdedené z C sú stále k dispozícií s tradičnou príponou '**.h**'. Avšak môžeme ich používať i modernejším C++ včleňovacím štýlom pridaním "**c**" pred meno. A tak s :

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

vznikne:

```
#include <cstdio>
#include <cstdlib>
```

A tak ďalej pre všetky štandardné c hlavičkové súbory. Toto poskytne čitateľovi jednoznačnú indikáciu, kedy používate C a kedy C++ knižnice.

Výsledok nového včleňvacieho formátu nie je identický so starým: použitie **.h** dáva staršiu nešáblonovú verziu a vynechanie **.h** dáva novú šablónovú verziu. Ak sa pokúsiem zmiešať obidva spôsoby v jednom programe, zvyčajne budemem mať problémy.

Linkovanie

Linker pozbiera cieľové moduly (ktoré majú často prípony**.o** alebo **.obj**), vygenerované kompilátorom do vykonateľného programu, ktorý operačný systém dokáže zaviesť a odštartovať. Je to posledná fáza procesu kompilácie.

Vlastnosti linkera sa menia od systému k systému. Vo všeobecnosti linkeru akurát povieme mená cieľových modulov a knižníc, ktoré chceme z linkovať a meno vykonateľného súboru a linker začne pracovať. Niektoré systémy vyžadujú, aby sme linker aktivovali explicitne. Vo väčšine sa linker aktivuje prostredníctvom C++ kompilátora. V mnohých situáciách sa linker aktivuje skryte.

Niektoré strašie linkery prehľadávajú cieľové súbory a knižnice viackrát a prehľadávajú zoznam, ktorý zadáme zľava doprava. Znamená to, že poradie cieľových súborov a knižníc môže byť dôležité. Ak máme záhadný problém, ktorý sa objavil až počas linkovania, jednou z množnosti je poradie súborov, v ktorom sú zdané pre linker.

Používanie knižníc

Teraz keď už sme oboznámení so základnou terminológiou, dokážeme pochopiť používanie knižníc. Ak chceme použiť knižnicu:

1. Včleníme hlavičkový súbor knižnice
2. Použijeme funkcie a premenné z knižnice.
3. Prilinkujeme knižnicu k vykonateľnému programu.

Tento postup platí i v prípade, že cieľové moduly nie sú spojené do knižnice. Včleňovanie hlavičkového súboru a linkovanie cieľových modulov sú základné kroky oddlenej kompilácie v C i C++.

Ako linker prehľadáva knižnicu

Ak vytvoríme externý odkaz na funkciu alebo premennú v C alebo C++, linker keď narazí na tento odkaz, môže urobiť dve veci. Ak ešte nenarazil na definíciu funkcie alebo premennej, pridá identifikátor do zoznamu „neznámych odkazov“. Ak linker už narazil na definíciu, odkaz je vyriešený.

Ak linker nemôže nájsť definíciu v zozname cieľových modulov, prehľadáva knižnice. Knižnice sú čiastočne indexované, takže linker nemusí prechľadávať všetky cieľové moduly knižnice – prehľadáva iba index. Keď linker nájde definíciu v knižnici, celý cieľový modul, nie iba definícia funkcie, sa pripojí k vykonateľnému programu. Všimnime si, že sa nelinkuje celá knižnica, iba cieľový modul knižnice, ktorý obsahuje definíciu, ktorú potrebujeme (inak by programy boli zbytočne rozsiahlé). Ak by sme chceli minimalizovať veľkosť vykonateľného programu, mohli by sme uvažovať o vložení jedinej funkcie do každého súboru zdojového kódu pri budovaní vlstnej knižnice. Toto si vyžaduje viac editácie, ale pre užívateľa to môže byť užitočné.

Keďže linker prehľadáva zdrojové súbory v poradí, v akom ich zadáme, môžeme použitie knižničnej funkcie „prebiť“ vlastnou funkciou s rovnakým menom tak, že meno súboru, ktorý túto funkciu obsahuje bude pred menom knižničného súboru. Pretože linker vyrieši všetky odkazy na túto funkciu použitím našej funkcie pred prehľadávaním knižnice, použije sa namiesto knižničnej funkcie naša funkcia. Poznemnajme však, že toto môže byť tiež chyba, ktorej C++ predchádza používaním tzv. menopriestoru.

Tajomné dodatky

Pri vytváraní vykonateľného C alebo C++ programu sa k nemu pripájajú tajomné dodatky. Jedným z nich je štartovací modul, ktorý obsahuje inicializčné rutiny, ktoré sa musia vykonať na začiatku vykonávania každého C alebo C++ programu. Tieto rutiny nastavujú zásobník a inicializujú určité premenné programu.

Linker vždy prehľadáva štandardnú knižnicu kvôli kompilovaným verziám akejkoľvek „štandardnej“ funkcii, volanej v programe. Pretože štandardná knižnica sa vždy prehľadáva, hocičo z knižnice môžeme použiť iba včlenením vhodného hlavičkového súboru do programu. Nemusíme prikazovať prehľadávať štandardnú knižnicu. V štandrdnej knižnici sú napríklad iostream funkcie. Ak chceme použiť, stačí včleniť hlavičkový súbor **<iostream>**.

Ak používame externé knižnice, musíme explicitne pridať meno knižnice k zoznamu súborov, spracovávaných linkerom.

Čo je to objektové programovanie ?

Skôr ako začneme hovoriť o C++, je nutné povedať niečo o objektovo-orientovanom programovaní. *Objektovo-orientované programovanie* je dôležitá množina metód, ktoré zefektívňujú vývoj programu a zároveň zvyšujú spoľahlivosť výsledných počítačových programov. Vzniklo postupným vývojom, počínajúc bežným procedurálnym programovaním až po funkcionáln programovanie, ktoré je z historického hľadiska priamym predchodcom objektovo-orientovaného programovania. Objektovo-orientované programovanie *zlúčilo* procedurálne programovanie, ktorého podstatou je *funkčná abstrakcia* (rozčlenenie celej aplikácie do malých modulov podľa druhu vykonávaných činností) s tzv. *dátovou abstrakciou*. Čo je to dátová abstrakcia? Dátová abstrakcia robí s dátami to, čo funkčná s operáciami. Podobne ako pri funkčnej abstrakcií nás zaujíma iba to, čo daná funkcia robí a nie ako to robí, v dátovej abstrakcií nás zaujíma čo môžeme s dátovou štruktúrou robiť a nie ako je implementovaná. Čiastočná dátová abstrakcia je i v procedurálnych programovacích vyšších jazykoch, napríklad, čísla v pohyblivej rádovej čiarke sú abstrahované vo všetkých programovacích jazykoch. Nemusíme sa zaoberať presnou binárnou reprezentáciou čísla v pohyblivej rádovej čiarke, keď mu priraďujeme hodnotu. Nie je nutné, aby sme rozumeli zložitosti binárneho násobenia, keď chceme násobiť dve hodnoty v pohyblivej rádovej čiarke. Jedinou dôležitou vecou je, že čísla v pohyblivej rádovej čiarke fungujú správnym a zrozumiteľným spôsobom. Nuž a objektovo-orientované programovanie dovoľuje definovať *vlastné abstraktné dátové typy*.

Základná zmena objektovo-orientovaného programovania v porovnaní s procedurálnym spočíva v tom, že program je navrhnutý okolo dát, s ktorými pracuje, a nie na samotných operáciách. Toto je celkom prirodzené, keď si uvedomíme, že zmyslom programu je spracovanie dát. Nakoniec celá činnosť, vykonávaná počítačmi, sa nazýva spracovanie dát. Dáta a činnosti sú spojené na základnej úrovni; aby mali obidve zložky zmysel, jedna potrebuje druhú. Nuž a objektovo-orientované programy vyjadrujú tento vzťah explicitne. V procedurálnom programovaní sú dátové štruktúry pasívne, v objektovo-orientovanom programovaní sa stávajú aktívnymi.

Základná terminológia objektovo-orientovaného programovania

Objektovo-orientované programovanie umožňuje organizovať dáta v programe podobným spôsobom ako napríklad biológovia organizujú rôzne druhy rastlín. V žargóne objektovo-orientovaného programovania autá, stromy, inventárne záznamy, komplexné čísla, atď. budeme nazývať *triedy* (po anglicky class). Trieda je šablóna, ktorá popisuje jednak dátovú štruktúru, a jednak právoplatné činnosti s jej dátovými položkami. Dátovú položku deklarovanú, ako prvok triedy, budeme nazývať *dátový člen*. Funkcie, ktoré sú v triede deklarované budeme nazývať *členské funkcie* alebo *metódy*; sú to jediné funkcie, ktoré pracujú s dátovými členmi triedy.

Každá kópia triedy sa nazýva *inštancia triedy* alebo *objekt*. Objekty sú navzájom nezávislé, a preto zmeny jedného objektu nemajú vplyv iný objekt.

Triedy môžu ako svoje stavebné bloky využívať iné triedy. Nové triedy zo starých sa dajú vytvárať prostredníctvom *dedičnosti*. Nová trieda, označovaná ako *odvodená trieda*, dedí vlastnosti, t.j. dátové i funkčné členy pôvodnej, resp. *základnej triedy*. Nová trieda môže pridávať ďalšie dátové prvky i metódy k tým, ktoré zdedila zo základnej triedy. Akákoľvek trieda (vrátane odvodenej triedy) môžu obsahovať ľubovoľný počet odvodených tried. Mechanizmus dedičnosti zaručuje možnosť tvorby *hierarchie tried*. Hierarchie tried sa často podobajú rodokmeňom, a preto sa základná trieda nazýva i *rodičovská trieda* a odvodená trieda zasa *potomok*.

Niekedy triedy odvodené od základnej triedy potrebujú zmeniť spôsob fungovania metód základnej triedy. Objektovo-orientované programovanie poskytuje na spracovanie takýchto záležitostí elegantnú vlastnosť, nazvanú *polymorfizmus*.

Polymorfizmus závisí od väzby medzi triedou a metódou. Pri použití polymorfných metód kompilátor nedokáže určiť, ktorú funkčnú metódu má volať. Volanie špecifickej funkcie závisí od konkrétnej triedy, a preto sa volaná funkcia určuje počas chodu programu. Toto sa nazýva *neskorá väzba*, pretože nastáva počas vykonávania programu. V niektorých objektovo-orientovaných jazykoch sa vyskytuje *skorá väzba* pre nepolymorfné (známe ako *statické*) metódy. Kompilátor vie presne, ktorá funkčná metóda sa má vyvolať a preto môže vytvoriť priame volanie už počas kompilácie. Aj keď je skorá väzba veľmi výkonná, väčšina objektovo-orientovaných jazykov používa pre všetky metódy neskorú väzbu.

Prečo objektovo-orientované programovanie

Objektovo-orientované programovanie zavádza do programovania mnoho nových termínov a pojmov. Čo však bolo dôvodom vzniku objektovo-orientovaného programovania? Medzi jeho hlavné ciele môžeme zaradiť zrýchlenie vývoja programovového vybavenia a zvýšenie jeho spoľahlivosti. Zároveň značne spriehľadňuje zdrojový kód.

Objektovo-orientované programy sú vytvorené z opakovane použiteľných programových komponentov. Akonáhle je trieda dokončená a otestovaná, môžeme ju použiť ako stavebný blok pri konštrukcii programu. Ak sú potrebné zmeny alebo nové vlastnosti, z existujúcich tried sa dajú odvodiť nové; odskúšané a správne schopnosti základnej triedy nie je nutné znova vyvíjať. Namiesto prepisovania už niečoho existujúceho sa môžeme venovať tvorbe nového kódu. Programy sa ľahšie testujú, pretože chyby programu sa dajú izolovať v rámci nového kódu odvodených tried.

Po vytvorení množiny tried sa z týchto tried ľahšie vyvíjajú nové aplikácie. Programátori, ktorí pracujú v skupine môžu aplikáciu budovať ako postupnosť tried, ktoré spojením dohromady vytvoria finálny program. Môžu používať a vylepšovať triedy, vyvinuté inými členmi týmu, bez toho, aby menili skutočné implementácie týchto tried, ba čo viac nemusia ich ani vidieť.
Jedným z programovacích jazykov, ktorý objektovo-orientované programovanie podporuje je i jazyk *C++.*

Syntaktické rozšírenia

Táto časť popisuje malé, avšak významné rozdiely medzi súčasnou ANSI definíciou jazyka C a definíciou AT&T C++, a rozdiely v používaní funkcií v C++.

Nové kľúčové slová

Aby bolo možné jazyk C doplniť o nové vlastnosti, C++ definuje niekoľko nových kľúčových slov. Akýkoľvek C program, ktorý používa identifikátory s rovnakým názvom ako sú uvedené kľúčové slová je nutné pred kompiláciou v C++ zmeniť. Patria sem:

```
asm          catch       class       delete      friend
inline       new         namespace   operator    private
protected    public      template    this        throw
try          virtual
```

Význam a použitie týchto slov je vysvetlený v ďalších častiach pri popise jednotlivých súčastí C++.

Poznámky //

C++ podporuje dva typy poznámok, jednak C programátorom dobré známe poznámky ohraničené symbolmi ***/**** a ****/***. Všetko, čo sa nachádza za symbolom /* kompilátor ignoruje až kým nenájde inverzné symbol */. Okrem tohto druhu poznámok jazyk C++ pozná i tzv. *jednoriadkové poznámky*, ktoré začínajú symbolom *//*:

```
for(i=0; i<10; ++i) {  // začiatok cyklu
   cout << i << "\n";   // zobraz i
}
```

V uvedenom príklade, všetko, čo sa nachádza za znakmi // až do konca riadku sa považuje za poznámku a kompilátor túto časť ignoruje.

Bežne sa poznámkový spôsob /* ...*/ používa pre veľké bloky poznámok a štýl // pre kratšie poznámky.

Pretypovanie (typecast)

C++ podporuje dve rôzne formy explicitného pretypovania:

```
int  i = 0;
long l = (long)f; // tradičné C-pretypovanie
long m = long(f); // nový C++ štýl pretypovania
```

Typ void

Typ *void* reprezentuje prázdnu množinu, t.j. objekt typu *void* nemá žiadnu hodnotu. Na prvý pohľad sa môže zdať, že mať typ, pre ktorý nie sú definované žiadne hodnoty, nemá význam. Typ *void* je však v praxi veľmi užitočný, a to najmä v týchto dvoch prípadoch:

- Funkcia, ktorá nemá návratovú hodnotu, môže byť deklarovaná s návratovou hodnotou typu *void*. Toto vylučuje možnosť, že takéto funkcie by museli zbytočne vracať hodnotu typu *int*, t.j. nemusí sa generovať kód pre návrat hodnoty.
- Typ *void* môžeme používať na definovanie všeobecného smerníka. Pred zavedením tohto typu bolo nutné na adresovanie pamäti bez toho aby nás zaujímal typ dát v nej uložených,

použiť smerník typu *char* alebo *int*. Pretože *void* * sa dá priradiť akémukoľvek smerníku, vyhneme sa mnohým zbytočným pretypovaniam. Deklaráciu *void* * si môžeme predstaviť ako smerník na akýkoľvek typ.

Flexibilné deklarácie

Jazyk C vyžaduje, aby všetky deklarácie v rámci daného rozsahu platnosti boli na začiatku. Programátorsky povedané, všetky globálne deklarácie sa musia objaviť pred akoukoľvek funkciou, a všetky lokálne deklarácie musia byť pred akýmkoľvek výkonným príkazom. C++ dovoľuje miešať deklarácie dát medzi funkcie a výkonný kód.

Pozrime si takúto, i keď trochu nesprávne napísanú C funkciu:

```
void fun(void)
{
   float i;
   char *cp;
/* tu si predstavme 500 riadkov kódu */
/* alokuj 100 bytov pre cp */
   cp = malloc(100);     /* prvé použitie cp */
   for(i=0; i<100; ++i) { /* prvé použitie i */
                          /* nejaká činnosť */
  }
/* ďalší kód */
}
```

Po 500 riadkoch kódu sa alokuje priestor pre *cp* a *i* je prvýkrát použité až v cykle. Nanešťastie medzitým programátor zabudne, ako bolo *i* deklarované. A pretože *i* je typu *float*, cyklus *for* je vytvorený s iteračnou premennou typu *float*, ktorá proces cyklu spomaľuje. Deklarácie, urobené ďaleko od použitia dátovej položky vedú k zmätkom a chybám.

C++ dovoľuje umiestňovať deklarácie bližšie k bodu ich skutočného použitia. V C++ by *fun* mohla vypadať takto:

```
void fun(void)
{
// 500 riadkov kódu
// alokuj 100 bytov pre cp
char *cp = new char[100];

// vykonaj cyklus
for (int i=0; i<100; ++i) {
/* nejaká činnosť */
  }
}
```

C++ verzia je zrozumiteľnejšia, pretože *i* aj *cp* sú deklarované pri ich prvom použití, a nie o 500 riadkov skôr.

Kvalifikátor const

Aj keď si ANSI C požičalo koncept *const* hodnôt od C++, neimplementuje ich rovnakým spôsobom. V obidvoch jazykoch C i C++ je hodnota deklarovaná ako *const* nenarušiteľná; program ju žiadnym spôsobom nemôže zmeniť. Najbežnejším spôsobom použitia hodnôt typu *const* v C je náhrada literálových konštánt, definovaných pomocou *#define*.

V C++ sa *const* hodnoty môžu používať namiesto akejkoľvek literálovej konštanty. Toto je nesmierne užitočná schopnosť, ktorá dovoľuje vytvárať **typové konštanty** namiesto použitia *#define* na tvorbu konštánt, ktoré neobsahujú žiadnu typovú informáciu.

Nasledovný fragment kódu je prípustný v C++, avšak v ANSI C bude označený ako chybný:

```
const int ArraySize = 100;
int Array[ArraySize];
```

Odlišný je i rozsah platnosti *const* hodnôt v ANSI C a C++. V ANSI C *const* hodnoty majú globálny rozsah platnosti, čo znamená, že sú viditeľné mimo súboru, v ktorom sú deklarované, pokiaľ nie sú zároveň deklarované i ako *static.* Toto môže narobiť veľa problémov, keď *const* definície umiestníme do hlavičkového súboru, ktorý je včleňovaný do viacej ako jedného zdrojového modulu programu. C++ rieši tento problém tým, že všetky *const* objekty sú implicitne statické.

Typová kompatibilita

Čo sa týka typovej kompatibility, C je dosť pružné. C++ je omnoho vyberavejšie. Napríklad C++ definuje typy *short int*, *int* a *long int* ako rôzne typy. Dokonca i keď *short int* je čo do veľkosti a formátu identické s obyčajným *int*, C++ ich stále považuje za rôzne typy, ktoré treba pretypovať pri vzájomnom priraďovaní hodnôt týchto typov. Toto je bezpečnostné opatrenie - skutočne prenositeľný kód musí tieto typy spracovávať ako rozdielne, pretože rôzne systémové architektúry môžu definovať implementácie týchto typov odlišne.

Znakové typy sú ďalším príkladom typov, ktoré kompilátor považuje za rozdielne, i keď programátor si myslí, že sú rovnaké. *unsigned char, char* a *signed char* majú veľkosť rovnú 1, avšak C++ ich nepovažuje za identické. Opäť je to podpora prenositeľnosti kódu. V C++ sa typy hodnôt musia presne zhodovať, aby bola zaručená úplná kompatibilita. V opačnom prípade sa musí použiť pretypovanie.

sizeof(char)

V C sú všetky znakové konštanty uložené ako *int*. Znamená to, že v C:

```
sizeof(´1´) == sizeof(int);
```

V C++ je jeden *char* spracovaný ako jeden byte a nerozširuje sa na veľkosť *int*. A tak v C++:

```
sizeof(´1´) == 1;
```

struct a union označenia

Medzi kľúčovými slovami *struct* a *union* v jazyku C a C++ sú dva podstatné rozdiely:

V C++ sa kľúčové slová *struct* a *union* deklarujú triedy, t.j môžu obsahovať definície dát i funkcií.

struct a union označenia sú považované za meno typu, ako keby boli deklarované príkazom *typedef*. V C môžeme použiť nasledovný fragment kódu:

```
struct foo {int a; float b };
struct foo f;
```

Uvedený kód deklaruje *struct* s označením *foo*, a potom vytvára inštanciu *foo*, nazvanú *f*.
V C++ je to jednoduchšie:

```
struct foo {int a; float b; };
foo f;
```

Taká istá konvencia platí i pre *union*. Avšak aby bola zachovaná kompatibilita s C, C++ stále akceptuje i staršiu syntax.

Anonymné union

C++ je doplnené o špeciálny druh *union*. Nazýva sa *anonymný union*, ktorý iba deklaruje množinu položiek, ktoré zdieľajú tú istú pamäťovú adresu. *Anonymný union* nie je pomenovaný a k jeho položkám sa pristupuje priamo pomocou ich mena. Anonymný union môže vyzerať napríklad takto:

```
union {
   int i;
   float f;
}
```

i aj *f* zdielajú ten istú pamäťovú adresu a dátový priestor. Na rozdiel od obyčajných, pomenovaných *union*, sa k hodnotám *anonymných union* pristupuje priamo. Napríklad po deklarácií vyššie uvedeného *union* je prípustný nasledovný kód:

```
i = 10;
f = 2.2;
```

Typy *enum*

S vymenovacími typmi (enum) C++ pracuje trochu odlišne než ANSI C. Napríklad pomenovanie *enum* je považované za meno typu, podobne ako to bolo u *union* a *struct*.

C definuje typ *enum* ako hodnoty typy *int*. Avšak v C++ má každý vymenovací typ svoj vlastný samostatný typ. Znamená to, že C++ nedovoľuje automatickú konverziu hodnoty typu *int* na hodnotu typu *enum*. Avšak vymenovacia hodnota sa namiesto *int* použiť môže. Nasledovný C fragment kódu by bol v C++ nesprávny:

```
enum Miesto {Prvy, Druhy, Treti};
Miesto Jano = Prvy;       // toto je v poriadku ...
int Vitaz = Jano;         // ...a toto tiež ...
Miesto Tom = 1;           // ...ale toto je chyba ...
Miesto Marek = Miesto(1); // ...pomocou typecast ju ...
// ...opravíme!
```

V jazyku C by všetky štyri uvedené príkazy boli prípustné.

Operátor ::

Operátor *::* je operátorom **rozsahu platnosti** a používa sa na sprístupňovanie položky, ktorá je v aktuálnom rozsahu skrytá. Napríklad:

```
#include "stream.hpp"

int a;
int main()
{
   float a;
   a = 1.5;
   ::a = 2;
   cout << "lokálne a = " << a << "\n";
   cout << "globálne a = " << ::a << "\n";
}
```

Oper*á*tor *::* v podstate hovorí, „Nepouži lokálne a; použi a, deklarované mimo tento rozsah.“ A tak vyššie uvedený program vypíše:

```
lokálne a = 1.5
globálne a = 2
```

Operátor rozsahu platnosti sa tiež používa pri definícií metód triedy na deklarovanie skutočnosti, že daná trieda je vlastníkom danej metódy .Tento operátor sa dá použiť na rozlíšenie členov základnej triedy od členov inej triedy s identickými menami. Avšak tieto pojmy objasní ďalšia časť.

Operátory *new* a *delete*

V tradičných C programoch sa všetka alokácia dynamickej pamäti robí pomocou štandardných knižničných funkcií ako sú *malloc* a *free*. C++ definuje nový spôsob dynamickej alokácie prostredníctvom operátorov *new* a *delete*.

Tradičnú C funkcia, ktorá využíva dynamickú pamäť vypadá asi takto:

```
void func(void)
{
   int *i;
   i = malloc(sizeof(int));
   *i = 10;
   printf("%d",*i);
   free(i);
}
```

V C++ nahradzuje C funkciu *malloc* operátor *new* a namiesto *free* je určený operátor *delete*. A tak V C++ by sme mohli vyššie uvedenú funkciu prepísať nasledovne:

```
void func(void)
{
   int *i = new int;
   *i = 10;
   cout << *i;
   delete i;
}
```

Na prvý pohľad je zrejmé, že C++ syntax je omnoho zrozumiteľnejšia a ľahšia. Na alokáciu poľa desiatich celých čísel 10 (*int*) by sme mohli použiť nasledovný príkaz:

```
int *i = new int[10];
```

a na ich dealokáciu zasa

```
delete []i;
```

Všimnime si syntax uvoľňovania alokovaného poľa - pred meno premennej musíme vložiť dvojicu hranatých zátvoriek. Táto syntax zabezpečí, že sa zavolá deštruktor pre každý uvoľňovaný objekt alokovaného poľa.

Mimochodom, ak operátor *new* nemôže alokovať požadované množstvo pamäti a nebol definovaný obslužný program chýb pamäťovej haldy (pozri nižšie), jeho návratová hodnota je NULL. Je to rovnaké ako vo funkcii *malloc*, ktorá tiež vracia NULL, keď nastane chyba pamäťovej haldy.

Možno sa zdá čudné, prečo boli do C++ pridané operátory *new* a *delete*, keď existujúce C knižničné funkcie fungujú rovnako. Odpoveď znie *flexibilita správy dynamickej pamäti*. Ktorákoľvek trieda môže si môže tieto operátory predefinovať a vytvoriť tak svoje vlastné verzie *new* a *delete*. Ak trieda operátory *new* a/alebo *delete* nedefinuje, automaticky sa použijú globálne verzie týchto operátorov. Toto umožňuje triede definovať svoje vlastné pamäťové alokovacie funkcie pre špeciálne aplikácie. Navyše globálne operátory *new* a *delete* sa dajú tiež zameniť.

Operátor *new* dokáže nahradiť všetky *malloc* a *calloc* funkcie štandardnej C knižnice. Jedinou bežne používanou C knižničnou funkciou, ktorá sa operátorom *new* nahradiť nedá, je funkcia *realloc*, ktorá mení množstvo pamäti, alokovanej pre smerník. Avšak *realloc* sa dá simulovať nasledovne:

```
// zmena priestoru, alokovaného char smerníku
char *temp = new char[new_size];// alokuj novy priestor
if(temp != NULL) {              // ak bol priestor alokovaný
   // skopíruj dáta zo starého miesta na nové
   if(new_size > old_size)
      memcpy(temp,cp,old_size);
   else
      memcpy(temp,cp,new_size);
   delete []cp;                 // zruš staré dáta
   cp = temp;                   // cp teraz ukazuje na nové miesto dát
}
```

Tento postup je omnoho nemotornejší než *realloc*. Zaberá omnoho viac riadkov zdrojového kódu a vyžaduje poznať pôvodnú veľkosť priestoru, alokovaného smerníku, ktorý sa má meniť. Našťastie medzi smerníkmi, ktorých pamäť bola alokovaná pomocou implicitných verzií *new* a *delete*, a pomocou štandardných C knižničných funkcií nie je rozdiel. A preto sa môže bežne používať *realloc* na zmenu priestoru, alokovaného smerníkom, vygenerovaným pomocou *new*, pokiaľ tieto operátory neboli predefinované.

Operátor *new* má ešte jednu výhodu. Na rozdiel od *malloc* každá chyba alokácie pamäti sa dá zachytiť pomocou užívateľom definovaného obslužného programu. C++ definuje špeciálny smerník na funkciu; keď pri použití operátora *new* nastane chyba alokácie pamäti, zavolá sa funkcia, na ktorú ukazuje tento smerník. Definícia tohto smerníka je nasledovná:

```
void (* _new_handler)();
```

Povedané zrozumiteľnou rečou, *_new_handler* je smerník na funkciu s prázdnym zoznamom argumentov (*void*), ktorá nevracia hodnotu. Vytvorením takejto funkcie a priradením jej adresy do smerníka *_new_handler* môžeme vo svojej funkcii zachytávať všetky chyby alokácie pamäti. Funguje to asi takto:

```
#include <new.h> // definuje _new_handler
#include <stream.h> //

void heapProblem(void);
void heapProblem(void)
{
cout << ”Chyba haldy!\n”;
}

int main (void)
{
// nejaký kód
// nastav _new_handler na heapProblem

   _new_handler = heapProblem;
// a opäť nejaký kód
}
```

V *new.h* je definovaná i knižničná funkcia, nazvaná *set_new_handler*, ktorá vykonáva priraďovanie do *_new_handler*.

Odkazy

C je občas neohrabané. Napríklad ak potrebujeme napísať funkciu, ktoré vymieňa dvojicu celých čísel, musíme to napísať nasledovne:

```
void vymenint(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Volanie vymieňacej funkcie *vymenint* bude v jazyku C vyzerať nasledovne:

```
vymenint(&i1, &i2);
```

C++ podporuje špeciálny typ identifikátora, nazývaný *odkaz*. Odkazy môžu vytvárať funkcie, ktoré vymenia hodnoty svojich parametrov omnoho elegantnejšie. C++ verzia *vymenint* vyzerá takto:

```
void vymenint(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

Odkaz je indikovaný operátorom *&*, podobne ako operátor * indikuje smerník. Rozdiel medzi smerníkom na niečo a odkazom na niečo je v tom, že smerník vyžaduje dereferenciu, zatiaľ čo odkaz nie. A tak C++ verzia *vymenint* nepotrebuje dereferencovať parametre *a* a *b*, aby mohla vymeniť hodnoty argumentov.

Aj syntax volania je omnoho jednoduchšia:

```
vymenint(i1,i2);
```

C++ automaticky predáva informáciu o adrese *i1* a *i2 ako* argumenty funkcie *vymenint*.

Odkazy sú obzvlášť užitočné pri prenose veľkých štruktúr a objektov do funkcie. Použitím odkazu ako parametra sa prenáša iba adresa a nie celá štruktúra alebo objekt. Toto nielenže šetrí čas a priestor zásobníka, ale uľahčuje i použitie štruktúrovaných parametrov v rámci samotnej funkcie.

Odkazové parametre by sme mali používať všade tam, kde sa do funkcií prenášajú veľké štruktúry. V prípade keď hodnotu parametra vo funkcií nechceme meniť, stačí použiť kvalifikátor *const*, a to takto:

```
void yaba(const int &daba);
```

Funkcia *yaba* príjme celočíselnú hodnotu (*daba*), prenesenú odkazom, ale *const* prikazuje, že *daba* sa nesmie zmeniť. Kvalifikátor *const* umožňuje používať odkazy bez rizika porušenia integrity dát.

Aj keď sa odkazy zdajú byť podobné smerníkom, nie sú to smerníky. Nemôžu sa použiť na alokáciu dynamickej pamäti ani sa s nimi nemôže matematicky manipulovať. Cieľom pozadia odkazov bolo umožnenie písať také funkcie, ktoré by mohli meniť svoje argumenty, a také funkcie, ktoré by akceptovali ako parametre štruktúry a objekty zrozumiteľnejším spôsobom.

Funkcia main()

Jazyk C nedefinuje špecifický formát pre funkciu *main()* (okrem jej mena). Často sa píše definícia main takto:

```
void main(void)
{
// kód hlavného programu
}
```

V mnohých programoch sa nedbá na druh návratového kódu do operačného systému. C++ však explicitne definuje *main* tak, aby vyhovoval týmto dvom prototypom:

```
int main();
int main(int argc,char *argv[]);
```

Akýkoľvek iný formát hlavičky *main* bude v C++ generovať chybu. Väčšina C++ kompilátorov upozorňuje varovným hlásením, ak *main* nevracia hodnotu. Pretože C++ núti *main* mať návratovú hodnotu, je dobrou programovacou praktikou z *main* skutočne hodnotu vracať. Ak ponecháme program ukončiť pred dosiahnutím konca *main*, spôsobí to, že *main* vráti nepredvídateľnú hodnotu.

Aj keď program nemá informatívnu návratovú hodnotu, použitie explicitného príkazu return 0 na konci *main* bude indikovať, že program skončil úspešne.

Prototypy funkcií

Prototypy funkcií pozná i ANSI C. Koncept funkčných prototypov však prebralo od C++. Funkčný prototyp je deklarácia, ktorá definuje návratový typ a parametre funkcie.

V C++ sa funkcia deklaruje asi takýmto príkazom:

```
int nieco(char *str, unsigned int len);
```

Uvedený príkaz stanovuje, že *nieco* je funkcia, ktorá vracia *int* hodnotu a má dva parametre, smerník na znak a *unsigned int*. Kompilátor používa prototypy na zabezpečenie, že typy argumentov, ktoré budeme prenášať pri volaní funkcie sa budú zhodovať s typom príslušných parametrov. Toto je známe ako prísna kontrola typu, niečo čo ANSI C chýba.

Bez prísnej kontroly typu je jednoduchšie preniesť do funkcií neprípustné hodnoty. Neprototypová funkcia dovoľuje odoslať *int* argument do smerníkového parametra, alebo použiť *float* argument tam, kde funkcia očakáva *long*. Tento druh chýb má za následok výskyt nežiadúcich hodnôt v

parametroch funkcií. Navyše pri prenose nevhodných typov do funkcie nemusí kompilátor správne obnoviť zásobníkový priestor, čo môže spôsobiť zrútenie programu.

Zatiaľ čo ANSI C iba *d o v o ľ u j e* používať prototypy, *C++* ich *vyžaduje*. V C++ prototypy zabezpečujú viac, než je len kontrola zhody argumentov a parametrov. Ako rozoberieme neskôr, C++ interne generuje mená funkcií, do ktorých včleňuje i informácie o type parametrov. Informácie o parametroch sa využívajú, keď má niekoľko funkcií rovnaké meno.

Označenia parametrov *str* a *len* vo funkčnom prototype *nieco* sa do tabuľky symbolov neukladajú, a preto sa nemusia zhodovať s menami príslušných parametrov v definícii funkcie. Mali by iba dokumentovať účel parametrov; môžeme si ich predstaviť ako krátke „poznámky“, popisujúce čo očakáva parameter funkcie ako argument. Kvôli zrozumiteľnosti sa doporučuje, aby sa označenia parametrov v prototype funkcie zhodovali s menami zodpovedajúcich parametrov v definícií funkcie.

Odlišná je i syntax deklarácie a definovania funkcií, ktoré majú prázdny (*void*) zoznam argumentov. Funkcie s nešpecifikovaným počtom parametrov neznámych typov sa v C++ deklarujú tiež deklarujú ináč.

Pre porovnanie uveďme dva C prototypy:

```
/* toto je C */
int fun(void); /* funkcia neakceptuje žiaden parameter */
int fun1();     /* funkcia s otvoreným zoznamom parametrov */
```

V C++ sa vynecháva *void* a namiesto toho sa používa dvojica prázdnych okrúhlych zátvoriek, napríklad:

```
// a toto je C++
int fun();    /* funkcia neakceptujúca žiaden parameter */
int fun1(...); /* funkcia s otvoreným zoznamom parametrov */
```

C++ funkcia s prázdnym zoznamom parametrov nemôže prijímať argumenty. Ak chceme mať funkciu s „otvoreným“ zoznamom parametrov, musíme použiť tri bodky.

Zapamätajme ***si, že všetky funkcie v C++ musia mať prototyp!*** Znamená to, že každá funkcia musí mať deklarovaný svoj zoznam argumentov a vlastná definícia funkcie sa musí presne zhodovať so svojím prototypovm čo do počtu a typov parametrov.

Prototypovanie funkcií snáď znamená trochu viac práce pri písaní programu, ale prototypy dokážu byť neoceniteľným prostriedkom pri predchádzaní ťažko hľadateľných chýb. C++ bolo navrhnuté tak, aby sa predišlo mnohým problémom, spôsobeným prenosom nesprávnych typov do metód pri nedbalom programovaní. Ak skutočne potrebujeme preniesť *long* namiesto *int*, môžeme použiť pretypovanie. V takomto prípade sa hodnota jedného typu prenáša ako argument do parametra iného typu explicitne, čo je určite lepšie ako keby sa tak dialo náhodne.

Hlavičky funkcií, podobné prototypom

Tí, ktorí začali svoju C programovaciu kariéru prácou v K&R C, písali definície funkcií asi takto:

```
void fun(a,b)
int a;
double b;
{
// činnosť funkcie fun
}
```

Aj keď C++ takúto formu akceptuje, za lepšiu programovaciu techniku sa považuje používanie prototypového formátu hlavičky funkcie. V C++ by vyššie uvedená funkcia vypadala takto:

```
void fun(int a, double b)
{
// činnosť funkcie fun
}
```

C++ štýl je zrozumiteľnejší než štýl C. Budúce verzie C++, by mali celkom upustiť od C syntaxe.

´Dotváranie´ mien

Jazyk C++ robí niečo, čo by sa dalo nazvať *dotváranie mien*. C++ používa dotváranie mien na generovanie mien funkcií, aby do mena mohol zahrnúť informácie o typoch parametrov, združených s danou funkciou. Tieto zmeny sú interné a pozostávajú z pridania informácií o parametroch k identifikátoru funkcie. Pretože niekoľko funkcií môže mať rovnaké meno, C++ kompilátor ich navzájom rozlišuje internou zmenou mien, ktorá odráža rozdiely v typoch parametroch. Dotváranie mien je nevyhnutné pre podporu ***preťažovaných*** funkcií a operátorov (pozri ďalej).

Vo väčšine prípadov sa dotvorené mená dajú pozrieť v tzv. .MAP súbore, ktorý generuje linkovací program. V skutočnosti je dotváranie mien veľmi jednoduchý proces. K menu funkcie sa pridáva postupnosť kódov, ktoré indikujú typy parametrov a poradie v akom sú uvedené.

Typovo bezpečné linkovanie

V štandardnom C neexistujú prostriedky na linkovanie objektových modulov, ktoré boli vytvorené inými programovacími jazykmi. Napríklad Pascal a Fortran obvykle usporiadavajú parametre do zásobníka v opačnom poradí ako C. Väčšina Pascal kompilátorov konvertuje všetky identifikátory na veľké písmena, zatiaľ čo C kompilátor pracuje s malými i veľkými písmenami. Aby bolo umožnené viacjazyčné programovanie, tvorcovia C kompilátora pridali niekoľko nových kľúčových slov, ktorými identifikujú volaciu konvenciu konkrétnej funkcie alebo identifikátora. Vo väčšine MS-DOS C kompilátorov sa pridáva k deklarácií funkcie kľúčové slovo *pascal*, ktoré túto funkciu identifikuje ako funkciu, ktorá používa pascalovskú volaciu postupnosť. Napríklad:

```
pascal int funbar(int a,int b);
```

je prototyp funkcie *funbar*, ktorá používa pascalovskú volaciu konvenciu. Najväčším nedostatkom špeciálnych jazykových kľúčových slov, ako je napríklad *pascal*, je to, že nie sú oficiálne štandardizované.

Spomeňme si, že C++ dotvára mená, zatiaľ čo C nie. A preto bol vymyslený mechanizmus, nazývaný *typovo bezpečné linkovanie*, ktorý špecifikuje štandardnú formu linkovacej špecifikácie. V C++ deklarácie funkcií, ktoré sú umiestnené v moduloch, skompilovaných C kompilátorom, zapíšeme takto:

```
extern "C" {
   double sin(double x);
   double cos(double x);
   double tan(double x);
}
```

Deklarácie funkcií, skompilovaných ako C, sú v bloku, ktorý začína príkazom *extern "C"*. Toto C++ kompilátoru oznámi, aby mená uvedených funkcií nedotváral.
Pomocou tejto syntaxe tiež možno špecifikovať konvencie i ostatných jazykov; špecifikácia C++ ponecháva priestor pre C++ implementácie na definovanie prepojenia s inými jazykmi.

Inline (včleňované) funkcie

Dobre štruktúrovaný program používa funkcie na rozčlenenie kódu na logické uzatvorené jednotky. Avšak funkcie prinášajú určité režijné pracovné náklady: argumenty sa musia uložiť do zásobníka, musí sa vykonať volanie, musí byť implementovaný návrat, za ktorým nasleduje výber parametrov zo zásobníka. Občas tieto náklady nútia C programátora duplikovať kód po celom programe, aby sa zvýšila výkonnosť.

Konštruktéri C++ si tento problém uvedomovali a vyriešili to zavedením tzv. *inline* funkcií. Ak hlavička definície funkcie obsahuje slovo „inline“, funkcia sa neprekladá ako samostatný kus volateľného kódu. Namiesto toho sa takáto funkcia vkladá všade tam, kde sa objaví jej volanie. Týmto sa eliminuje funkčná réžia, a zároveň program zostáva organizovaný štruktúrovaným spôsobom.

Napríklad nasledovnú funkciu C++ prekladá ako *inline* kód:

```
inline int scitaj(int a,int b)
{
   return a+b;
}
```

Funkcia nebude skutočne existovať ako volateľná rutina. Namiesto toho vloží kompilátor kód, ktorý vykoná jej činnosť všade tam do programu, kde sa objaví volanie *scitaj.* C++ translátory vytvárajú pre *inline* funkcie makrá a tieto makrá vkladajú do výsledného C programu na miesta, kde sa funkcia volá. C++ kompilátor predkompiluje rutinu a predkompilované inštrukcie vloží na príslušné miesta.

Pre používanie *inline* funkcií platia určité pravidlá. *Inline* funkcie musia byť definované pred svojím použitím. Je to preto, lebo kód *inline* funkcie sa musí predkompilovať skôr ako sa vloží do programu. A preto nasledovný kód sa nebude kompilovať tak, ako by sme očakávali:

```
#include <stdio.h>
int scitaj(int a, int b);

int main()
{
   int x = scitaj(1,2);
   printf("%i\n",x);
}

inline int scitaj(int a,int b)
{
   return a+b;
}
```

Môžu nastať dva prípady: Buď kompilátor zahlási chybu alebo dané volanie funkcie nepreloží ako *inline* ale ako obyčajné volanie funkcie. Závisí to od prekladača.

Tieto jemné rozdiely medzi implementáciami C++ môžu spôsobiť problémy, ak si na ne nedáme pozor. V tomto prípade je najlepšie nespoliehať sa na schopnosti prekladača pri spracovaní narušených definícií inline funkcií. Vždy by sme mali predpokladať, že pracujeme s prekladačom, ktorý vyžaduje, aby všetky *inline* funkcie boli definované pred ich použitím.

Hlavným nedostatkom *inline* funkcií je, že vo väčšine prípadov zväčšujú program. Jednoduché funkcie, ako je napríklad *scitaj* môžu byť skutočnosti menšie než reálna funkcia, pretože každá volacia sekvencia by sa nahradila skupinou sčítacích a priraďovacích inštrukcií. Na druhej strane však zložitejšie *inline* funkcie môžu obsahovať rozsiahlejší kód. Na rozsah by sme mali dávať pozor, aby sa nestalo, že program sa kvôli zvýšeniu výkonnosti o niekoľko sekúnd príliš naduje.

Ak *inline* funkcia obsahuje zložité riadiace príkazy chodu programu, C++ kompilátor ignoruje *inline* špecifikáciu. Bežné *inline* funkcie by mali pozostávať iba z priradení, výrazov a volaní jednoduchých funkcií.

Implicitné hodnoty argumentov

Jedným z najužitočnejších prostriedkov C++ je schopnosť definovať implicitné hodnoty argumentov funkcií. Normálne (v C-jazyku) pri volaní funkcie musíme špecifikovať argument pre každý parameter, ktorý funkcia definuje. Napríklad:

```
void delay(int num); // prototyp
void delay(int num) // definícia funkcie
{
if(num==0)
   return;
for(int i=0; i<num; i++)
   ;
}
```

Pri každom volaní funkcie *delay* musíme preniesť ako argument počet cyklov, ktorý určuje počet operácií. Často však zistíme, že funkciu *delay* voláme vždy s rovnakou hodnotou *num*. V C++ existuje však spôsob ako definovať implicitnú hodnotu parametra. Povedzme, že chceme aby implicitná hodnota *num* vo funkcií *delay* bola 1000. Dosiahneme to jednoduchou zmenou prototypu funkcie:

```
void delay(int num=1000);
```

Pri každom volaní *delay* bez zadania argumentu pre *num* program automaticky dosadí do *num* hodnotu 1000. Napríklad:

```
delay(2500); // num sa nastaví na 2500
delay();     // num sa nastaví na 1000
```

Tu je dôležité spomenúť, že implicitnou hodnotou parametra môže byť **globálna konštanta**, **globálna premenná** alebo dokonca i **volanie funkcie**. Napríklad funkcia na nastavenie hodín by mohla mať nasledovný prototyp:

```
int setclock(time_t start_time = time(NULL));
```

Ak *setclock* zavoláme bez argumentu pre *start_time,* implicitná hodnota sa získa volaním štandardnej knižničnej funkcie *time*.

Implicitné argumenty sa dávajú iba do prototypov funkcií a nesmú sa opakovať v definícií funkcie. Na vytvorenie volania používa kompilátor prototypové informácie a nie definíciu funkcie.

Funkcia môže obsahovať i viac ako jeden implicitný parameter. Implicitné parametre musia nasledovať jeden za druhým a byť zoskupené dohromady a musia byť v zozname parametrov ako posledné (prípadne ako jediné). Pri volaní funkcie s viacerými implicitnými argumentmi môžeme vynechať iba tieto argumenty z prava do ľava, pričom ich musíme vynechať tak ako nasledujú za sebou. Uvedené pravidlá objasní niekoľko príkladov :

```
// toto je chybný prototyp
void funzla(int a=1, int b, int c=3, int d=4);
// a tento je správny
void fundobra(int a, int b=2, int c=3, int d=4);
```

Prvý prototyp priraďuje implicitný argument prvému, tretiemu a štvrtému parametru. Pretože argumenty sa môžu vynechávať iba za sebou zľava doprava, neexistuje žiaden spôsob, ako použiť implicitný argument pre prvý parameter, pretože druhý parameter implicitný argument nemá. Druhý prototyp je správny.

Pozrime si niekoľko správnych a niekoľko nesprávnych volaní *fundobra* funkcie:

```
// OK: argumenty pre všetky parametre
fundobra(10,15,20,25);
// NESPRÁVNE: parameter a nemá implicitný argument
fundobra();
// SPRÁVNE: parametrom c a d sa priradí implicitná hodnota
fundobra(12,15);
// CHYBA: vynechané parametre musia nasledovať tesne po sebe
fundobra(3,10,,12);
```

Implicitné argumenty uľahčujú písanie a údržbu programu. Napríklad, ak máme už existujúcu funkciu, ktorá potrebuje pridať nový parameter, môžeme pre tieto nové parametre použiť implicitné argumenty. Toto zmierni nutnosť meniť už existujúci kód. Implicitné argumenty sa dajú tiež použiť na zlúčenie niekoľkých jednoduchých funkcií do jednej.

Preťažovanie (overloading)

C++ pridalo k schopnostiam funkcií dva dôležité doplnky. Proces, v anglickej literatúre nazývaný *overloading*, umožňuje mať viac funkcií s rovnakým menom. Navyše C++ dovoľuje definovať naše vlastné funkcie pre symbolické operátory, ako sú +,- alebo *.

Preťažovanie má dvojité vyžitie. Po prvé, umožňuje nazývať rôzne implementácie funkcií rovnakým názvom. Po druhé, umožňuje definovať tzv. operátorové funkcie, ktoré sa budú volať pri použití symbolických operátorov.

Preťažovanie funkcií

V jazyku C, podobne ako vo väčšine programovacích jazykov, musí mať každá funkcia v rámci celej aplikácie jednoznačné meno. Občas je to dosť nepríjemné. Napríklad v jazyku C je niekoľko funkcií, ktoré vracajú absolútnu hodnotu číselného argumentu. Pretože sa vyžaduje jednoznačné meno, pre každý dátový typ existuje samostatná funkcia. A tak existujú tri rozličné funkcie, ktoré vracajú absolútnu hodnotu argumentu:

```
int    abs(int i);
long   labs(long l);
double fabs(double d);
```

Všetky tri funkcie vykonávajú tú istú činnosť, a preto sa zdá trochu neohrabané mať tri rôzne mená funkcií. C++ rieši tento paradox možnosťou vytvorenia troch rôznych funkcií s rovnakým menom.

Toto sa nazýva *preťažovanie funkcií*. V C++ by sme uvedené tri funkcie mohli definovať takto:

```
int    abs(int i);
long   abs(long l);
double abs(double d);
```

Prostredníctvom dotvárania mien (pozri časť 2.1.16) C++ kompilátor dokáže presne určiť, ktorá implementácia funkcie *abs* je vhodná pre dané volanie funkcie. Napríklad:

```
abs(-10);          // volá int abs(int)
abs(-100000);      // volá long abs(long)
abs(-10.34);       // volá double abs(double)
```

Proces výberu preťaženej funkcia je rovnaký ako proces, ktorým C++ analyzuje nejednoznačnosti. Ak napríklad existuje implementácia preťaženej funkcie, ktorá má identické typy parametrov z prenášanými argumentmi vo volaní, zavolá sa táto implementácia. Ináč C++ kompilátor zavolá implementáciu, ktorá zabezpečuje najľahšiu sériu konverzií. Napríklad:

```
abs(´a´);     // volá int abs(int i)
abs(3.1415F); // volá double abs(double d)
```

Zabudované konverzie majú prednosť pred konverziami, ktoré vytvoríme. V takmer vo väčšine prípadov výber preťaženej funkcie bude založený na ľahkosti prispôsobenia argumentov zoznamu dostupných parametrov.

Za predpokladu, že prekladaču nejako umožníme určiť adresu verzie preťaženej funkcie, môžeme i priraďovať adresy preťažených funkcií. Objasní to nasledujúci príklad:

```
int fun(int i);
int fun(char i);
int (*ptr2fun)(int)      = &fun;
int (*ptr2fun)(char)     = &fun;
int (*ptr2fun)(double)   = &fun; // Chyba
int (*ptr2fun)(...)      = &fun; // Chyba
```

V prvých dvoch priradeniach adries kompilátor ľahko dokáže porovnať typ smerníka s typom verzie *fun*. Tretie priradenie sa nepodarí, pretože neexistuje *void fun(double)*. Posledné priradenie je nejednoznačné; bez podrobného zoznamu parametrov kompilátor nevie rozpoznať, ktorú verziu *fun* chceme adresovať.

Nástrahy preťažených funkcií

Pre používanie preťažených funkcií platia určité obmedzenia. Podobne ako C program i C++ program môže ignorovať návratovú hodnotu funkcie. A preto sa preťažované funkcie musia odlišovať inými spôsobmi, než je ich návratová hodnota. Nasledovná deklarácia by nebola prípustná:

```
int  process(int i);
void process(int i);
```

Pre kompilátor neexistuje žiaden spôsob ako rozlíšiť, ktorá implementácia funkcie je volaná, ak by sa pri volaní návratová hodnota ignorovala. A tak sa preťažované funkcie musia líšiť v ***t y p e*** alebo ***p o č t e*** akceptovaných parametrov.

Pri tvorbe preťažovaných funkcií si musíme dať pozor na to, že na pohľad rozdielne typy parametrov našej preťažovanej funkcie sú skutočne rozdielne. Typ, definovaný pomocou *typedef*, je iba náhradou pre existujúci typ a nevytvára svoj vlastný pôvodný typ. A tak nasledovný fragment kódu by bol nesprávny:

```
typedef INT int;
// obidva prototypy majú identické použitie
void humbug(int x);
void humbug(INT x);
```

Vyššie uvedený príklad by sa neskompiloval správne, pretože kompilátor by nedokázal rozlíšiť medzi obidvoma verziami *humbug*. *INT* je iba inak pomenovaný typ *int*.

Na rozlíšenie typov parametrov funkcie môžeme použiť i kvalifikátor *const*, ako to ilustruje i nasledovný fragment:

```
void func(char *ch);
void func(const char *ch);
int main()
{
   const char c1="a";
   char c2=´b´;
   func(&c1); // volá sa void func(const char *ch);
   func(&c2); // volá sa void func(char *ch);
}
```

Pri tvorbe preťažovaných funkcií by sme mali dodržiavať nepísané pravidlo, že preťažované funkcie by mali vykonávať filozoficky príbuzné činnosti. Funkcie s rovnakým menom by mali mať rovnaký všeobecný účel. Vytvoriť implementáciu funkcie *abs*, ktorá by vracala druhú odmocninu čísla by bolo asi hlúpe a mätúce (samozrejme, pokiaľ to nie je zámer).

Preťažované funkcie by sa mali používať s rozumnou mierou. Ich účelom je zabezpečovať mnemonické meno pre niekoľko podobných, ale trochu odlišných funkcií. Nadmerné používanie preťažovaných funkcií môže viesť knečitateľnosti programu.

Preťažovanie operátorov

Preťažovanie operátorov je iba "syntaktický cukor", t.j. je to iba iný spôsob volania funkcie.

Rozdiel spočíva v tom, že argumenty tejto funkcie sa neobjavujú v zátvorkách, ale namiesto toho obaľujú alebo sú vedľa znakov, o ktorých si myslíme, že sú nemenné operátory.

Existujú dva rozdiely medzi použitím operátora a obyčajným volaním funkcie. Odlišná je syntax; operátor sa často "volá" umiestnením medzi a občas za argumenty. Druhý rozdiel spočíva v tom, že kompilátor určuje, ktorá "funkcia" sa zavolá. Napríklad ak používame operátor + s reálnymi číslami (float), kompilátor zavolá funkciu, ktorá realizuje sčítanie reálnych čísel (týmto volaním je zvyčajne akt vloženia kódu alebo inštrukcie procesora). Ak použijeme operátor `+` s reálnym číslom a celým číslom, kompilátor zavolá špeciálnu funkciu, ktorá prevedie typ `int` na typ `float` a potom zavolá kód na sčítavanie reálnych čísel.

Ale v C++ je možné definovať nové operátory, ktoré budú fungovať s triedami. Táto definícia je ako definícia obyčajnej funkcie, avšak meno funkcie sa skladá z kľúčového slova **`operator`**, za ktorým nasleduje operátor. Toto je jediný rozdiel a zvyšok vypadá ako akákoľvek iná funkcia, ktorú kompilátor zavolá, keď uvidí zodpovedajúcu vzorku.

Varovanie a opätovné uistenie

Nedajme sa príliš zlákať preťažovaním operátorov. V prvom rade je to zábavná hračka. Ale nezabúdajme, že to je iba *syntaktický cukor*, iný spôsob volania funkcie. Z tohto pohľadu neexistuje dôvod preťažovať operátor, okrem prípadu, keď to uľahčí písanie kódu, používajúceho triedu a uľahčí to jeho čítanie. (Nezabúdajme, že ten kód sa častejšie číta než píše.) V opačnom prípade sa neunúvajme.

Ďalšou bežnou odozvou na preťažovanie operátorov je panika; C operátory nemajú už viac známy význam. "Všetko je inak" a môj C kód bude fungovať inak". Toto nie je pravda. všetky operátory, použité vo výrazoch, ktoré obsahujú zabudované typy sa nemôžu meniť. Nikdy nemôžeme preťažiť operátor

```
1 << 4;
```

tak, aby sa správal inak, alebo aby

```
1.414 << 2;
```

malo význam. **Jedine výrazy, obsahujúce užívateľské typy môžu mať preťažený operátor.**

Syntax

Definovanie preťaženého operátora je ako definovanie funkcie, ale meno funkcie je **operator** @, kde @ predstavuje preťažovaný operátor. Počet argumentov v zozname argumentov preťažovaného operátora závisí od dvoch činiteľov:

1. Či sa jedná o unárny operátor (jeden argument) alebo binárny operátor (dva argumenty).
2. Či je operátor definovaný ako globálna funkcia (jeden argument pre unárny, dva pre binárny) alebo členská funkcia (žiaden argument pre unárny, jeden pre binárny – objekt sa stáva ľavým argumentom).

A tu je malá trieda, ktorá ukazuje syntax preťažovania operátora:

```
#include <iostream>
using namespace std;

class Integer {
  int i;

public:
  Integer(int ii) : i(ii) {}
  const Integer operator +(const Integer& rv) const {
    return Integer(i + rv.i);
  }
  Integer& operator +=(const Integer& rv) {
    i += rv.i;
    return *this;
  }
};

int main()
{
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
}
```

Dva preťažené operátory sú definované ako `inline` členské funkcie, ktoré sa pri zavolaní ohlasujú. Jediný argument je ten, ktorý sa objaví na pravej strane operátora.

Pre nie-podmienkové operátory (podmienky zvyčajne vracajú boolean hodnotu), takmer vždy chceme vrátiť objekt alebo odkaz rovnakého typu, s akým manipulujeme ak sú obidva argumenty rovnakého typu. (Ak nie sú rovnakého typu, interpretácia čo má produkovať, závisí od tvorcu.) Takýmto spôsobom môžeme vytvárať zložité výrazy:

```
kk += ii + jj;
```

Operátor `+` poskytuje nový objekt triedy `Integer` (dočasný), ktorý sa použije ako `rv` argument operátora `+=`. Dočasný objekt sa zruší akonáhle už nie je potrebný.

Preťažiteľné operátory

Aj keď preťažiť môžeme takmer všetky oprátory, ktoré sa nachádzajú v C jazyku, používanie preťažovania operátorov je dosť obmedzujúce. Konkrétne nemôžeme kombinovať operátory, ktoré v C nemajú žiaden význam (napríklad ** na reprezentovanie umocňovania), nemôžeme meniť prioritu operátorov a nemôžeme meniť počet argumentov, požadovaných operátorom.

Unárne operátory

Nasledujúci príklad ukazuje syntax preťaženia všetkých unárnych operátorov, a to ako globálne funkcie i ako členské funkcie. Rozšírime triedu `Integer` a pridáme novú triedy `byte`. Význam konkrétnych operátorov bude závisieť na spôsobe, akým ich chceme použiť, ale berme do úvahy i klientského programátora skôr, než urobíme niečo neočakávané.

A tu je zoznam všetkých unárnych funkcií:

```
#include <iostream>
using namespace std;

class Integer {
    long i;
    Integer *This() { return this; }
  public:
    Integer(long ll = 0) : i(ll) {}
    friend const Integer &operator +(const Integer& a);
    friend const Integer operator -(const Integer& a);
    friend const Integer operator ~(const Integer& a);
    friend Integer *operator &(Integer& a);
    friend int operator!(const Integer& a);
    friend const Integer &operator++(Integer& a);
    friend const Integer operator++(Integer& a, int);
    // Prefix:
    friend const Integer &operator --(Integer& a);
    // Postfix:
    friend const Integer operator --(Integer& a, int);
};

const Integer& operator +(const Integer& a)
{
  return a;
}

const Integer operator -(const Integer& a)
{
   return Integer(-a.i);
}

const Integer operator ~(const Integer& a)
{
  return Integer(~a.i);
}

Integer *operator & (Integer& a)
{
  return a.This();
}

int operator !(const Integer& a)
{
  return !a.i;
}

// Prefix
const Integer& operator++(Integer& a)
{
  a.i++;
  return a;
}
```

```
// Postfix
const Integer operator++(Integer& a, int)
{
  Integer before(a.i);
  a.i++;
  return before;
}

// Prefix;
const Integer& operator--(Integer& a)
{
  cout << "--Integer\n";
  a.i--;
  return a;
}

// Postfix
const Integer operator--(Integer& a, int)
{
  Integer before(a.i);
  a.i--;
  return before;
}

void f(Integer a)
{
  +a;
  -a;
  ~a;
  Integer* ip = &a;
  !a;
  ++a;
  a++;
  --a;
  a--;
}

class Byte {
    unsigned char b;
  public:
    Byte(unsigned char bb = 0) : b(bb) {}
    const Byte& operator +() const { return *this;  }
    const Byte operator -() const {  return Byte(-b); }
    const Byte operator ~() const {  return Byte(~b); }
    Byte operator !() const { return Byte(!b); }
    Byte* operator &() {  return this; }
    // Prefix
    const Byte& operator ++() { b++; return *this; }
    // Postfix
    const Byte operator ++(int) {
      Byte before(b);  b++;
      return before;
    }
    // Prefix
    const Byte& operator --() { --b; return *this; }
    // Postfix
    const Byte operator--(int) {
      Byte before(b);
      --b;
      return before;
```

```
    }
};

void g(Byte b) {
  +b;
  -b;
  ~b;
  Byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}

int main() {
  Integer a;
  f(a);
  Byte b;
  g(b);
}
```

Funkcie sú zoskupené podľa spôsobu prenosu argumentov. Pravidlá ako prenášať a vracať argumenty si povieme neskôr. Vyššie uvedené formy (a tie ktoré budú nasledovať) sú typické, takže ich môžeme používať ako vzor pre preťažovanie vlastných operátorov.

Inkrementácia a dekrementácia

Preťažené operátory `++` a `--` predstavujú dilemu, pretože chceme mať možnosť volať rôzne funkcie podľa toho, či sa objavia pred (prefix) alebo za (postfix) objektom, s ktorým narába. Riešenie je jednoduché, ale niektorým sa zdajú na prvý pohľad mätúce. Keď kompilátor je vidí napríklad `++a` (preinkrementácia), generuje volanie `operator++(a)`, ale keď narazí na `a++`, generuje volanie `operator ++(a,int)`. Znamená to, že kompilátor rozlišuje medzi dvomi formami volaním rôznych preťažených funkcií. V predchádzajúcom príklade pri veriách členských funkcií, ak kompilátor narazí na `++b`, generuje volanie `B::operator++()`; ak narazí na `b++`, zavolá `B::operator++(int)`.

Všetci užívatelia vidia, že sa zavolala iná funkcia pre prefixovú a postfixovú verziu. Avšak obidve volania funkcií majú odlišnú signatúru, takže sú spojené s dvomi rôznymi telami funkcii. Kompilátor prenáša fiktívnu konštantnú hodnotu ako argument typu `int` (ktorý nemá identifikátor, pretože jeho hodnota sa nikdy nepoužíva), aby generoval odlišnú signatúru pre postfixovú verziu.

Binárne operátory

Nasledujúci výpis opakuje predchádzajúci príklad pre binárne operátory, takže máme príklad všetkých operátorov, ktoré by sme mohli chcieť preťažiť. Opäť sú ukázané obidve verzie – globálne funkcie a členské funkcie.

```
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // Operatory ktore vytvaraju novu modifikovanu hodnotu:
  friend const Integer
```

```
  operator +(const Integer& left,
             const Integer& right);
friend const Integer
  operator -(const Integer& left,
             const Integer& right);
friend const Integer
  operator *(const Integer& left,
             const Integer& right);
friend const Integer
  operator /(const Integer& left,
             const Integer& right);
friend const Integer
  operator %(const Integer& left,
             const Integer& right);
friend const Integer
  operator ^(const Integer& left,
             const Integer& right);
friend const Integer
  operator &(const Integer& left,
             const Integer& right);
friend const Integer
  operator |(const Integer& left,
             const Integer& right);
friend const Integer
  operator <<(const Integer& left,
              const Integer& right);
friend const Integer
  operator >>(const Integer& left,
              const Integer& right);
friend Integer&
  operator +=(Integer& left,
              const Integer& right);
friend Integer&
  operator -=(Integer& left,
              const Integer& right);
friend Integer&
  operator *=(Integer& left,
              const Integer& right);
friend Integer&
  operator /=(Integer& left,
              const Integer& right);
friend Integer&
  operator %=(Integer& left,
              const Integer& right);
friend Integer&
  operator ^=(Integer& left,
              const Integer& right);
friend Integer&
  operator &=(Integer& left,
              const Integer& right);
friend Integer&
  operator |=(Integer& left,
              const Integer& right);
friend Integer&
  operator >>=(Integer& left,
               const Integer& right);
friend Integer&
  operator <<=(Integer& left,
               const Integer& right);
// Logické operatory - return true/false:
friend int
```

```
    operator ==(const Integer& left,
                const Integer& right);
  friend int
    operator !=(const Integer& left,
                const Integer& right);
  friend int
    operator <(const Integer& left,
               const Integer& right);
  friend int
    operator >(const Integer& left,
               const Integer& right);
  friend int
    operator <=(const Integer& left,
                const Integer& right);
  friend int
    operator >=(const Integer& left,
                const Integer& right);
  friend int
    operator &&(const Integer& left,
                const Integer& right);
  friend int
    operator ||(const Integer& left,
                const Integer& right);
  // Výpis d prúdu:
  void print(std::ostream& os) const { os << i; }
};
#endif // INTEGER_H

// Implementácia preťažených operátorov
#include "Integer.h"

const Integer
   operator +(const Integer& left, const Integer& right)
{
  return Integer(left.i + right.i);
}

const Integer
  operator -(const Integer& left, const Integer& right)
{
  return Integer(left.i - right.i);
}

const Integer
  operator *(const Integer& left, const Integer& right)
{
  return Integer(left.i * right.i);
}

const Integer
  operator /(const Integer& left, const Integer& right)
{
  return Integer(left.i / right.i);
}

const Integer
  operator %(const Integer& left, const Integer& right)
{
  return Integer(left.i % right.i);
}
```

```
const Integer
  operator^(const Integer& left, const Integer& right)
{
  return Integer(left.i ^ right.i);
}

const Integer
  operator&(const Integer& left, const Integer& right)
{
  return Integer(left.i & right.i);
}

const Integer
  operator|(const Integer& left, const Integer& right)
{
  return Integer(left.i | right.i);
}

const Integer
  operator<<(const Integer& left, const Integer& right)
{
  return Integer(left.i << right.i);
}

const Integer
  operator>>(const Integer& left, const Integer& right)
{
  return Integer(left.i >> right.i);
}
Integer& operator+=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i += right.i;
   return left;
}

Integer& operator-=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i -= right.i;
   return left;
}

Integer& operator*=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i *= right.i;
   return left;
}

Integer& operator/=(Integer& left, const Integer& right)
{
   require(right.i != 0, "divide by zero");
   if(&left == &right) {}
   left.i /= right.i;
   return left;
}

Integer& operator%=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
```

```
   left.i %= right.i;
   return left;
}

Integer& operator^=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i ^= right.i;
   return left;
}

Integer& operator&=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i &= right.i;
   return left;
}

Integer& operator|=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i |= right.i;
   return left;
}

Integer& operator>>=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i >>= right.i;
   return left;
}

Integer& operator<<=(Integer& left, const Integer& right)
{
   if(&left == &right) {}
   left.i <<= right.i;
   return left;
}

int operator==(const Integer& left, const Integer& right)
{
    return left.i == right.i;
}

int operator!=(const Integer& left, const Integer& right)
{
    return left.i != right.i;
}

int operator<(const Integer& left, const Integer& right)
{
    return left.i < right.i;
}

int operator>(const Integer& left, const Integer& right)
{
    return left.i > right.i;
}

int operator<=(const Integer& left, const Integer& right)
{
```

```
    return left.i <= right.i;
}

int operator>=(const Integer& left, const Integer& right)
{
    return left.i >= right.i;
}

int operator&&(const Integer& left, const Integer& right)
{
    return left.i && right.i;
}

int operator||(const Integer& left, const Integer& right)
{
    return left.i || right.i;
}

#include "Integer.h"
#include <fstream>
using namespace std;

void h(Integer& c1, Integer& c2) {

  c1 += c1 * c2 + c2 % c1;
}

int main() {
  Integer c1(47), c2(9);
  h(c1, c2);
}


// Členské preťažené operátory

#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>

// Členské funkcie (implicit "this"):
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  const Byte
    operator+(const Byte& right) const {
    return Byte(b + right.b);
  }
  const Byte
    operator-(const Byte& right) const {
    return Byte(b - right.b);
  }
  const Byte
    operator*(const Byte& right) const {
    return Byte(b * right.b);
  }
  const Byte
    operator/(const Byte& right) const {
    return Byte(b / right.b);
  }
```

```
const Byte
  operator%(const Byte& right) const {
  return Byte(b % right.b);
}
const Byte
  operator^(const Byte& right) const {
  return Byte(b ^ right.b);
}
const Byte
  operator&(const Byte& right) const {
  return Byte(b & right.b);
}
const Byte
  operator|(const Byte& right) const {
  return Byte(b | right.b);
}
const Byte
  operator<<(const Byte& right) const {
  return Byte(b << right.b);
}
const Byte
  operator>>(const Byte& right) const {
  return Byte(b >> right.b);
}
Byte& operator=(const Byte& right) {
  if(this == &right) return *this;
  b = right.b;
  return *this;
}
Byte& operator+=(const Byte& right) {
  if(this == &right) {}
  b += right.b;
  return *this;
}
Byte& operator-=(const Byte& right) {
  if(this == &right) {}
  b -= right.b;
  return *this;
}
Byte& operator*=(const Byte& right) {
  if(this == &right) {}
  b *= right.b;
  return *this;
}
Byte& operator/=(const Byte& right) {
  if(this == &right) {}
  b /= right.b;
  return *this;
}
Byte& operator%=(const Byte& right) {
  if(this == &right) {}
  b %= right.b;
  return *this;
}
Byte& operator^=(const Byte& right) {
  if(this == &right) }
  b ^= right.b;
  return *this;
}
Byte& operator&=(const Byte& right) {
  if(this == &right) {}
```

```
    b &= right.b;
    return *this;
  }
  Byte& operator|=(const Byte& right) {
    if(this == &right) {}
    b |= right.b;
    return *this;
  }
  Byte& operator>>=(const Byte& right) {
    if(this == &right) {}
    b >>= right.b;
    return *this;
  }
  Byte& operator<<=(const Byte& right) {
    if(this == &right) {}
    b <<= right.b;
    return *this;
  }
  int operator==(const Byte& right) const {
      return b == right.b;
  }
  int operator!=(const Byte& right) const {
      return b != right.b;
  }
  int operator<(const Byte& right) const {
      return b < right.b;
  }
  int operator>(const Byte& right) const {
      return b > right.b;
  }
  int operator<=(const Byte& right) const {
      return b <= right.b;
  }
  int operator>=(const Byte& right) const {
      return b >= right.b;
  }
  int operator&&(const Byte& right) const {
      return b && right.b;
  }
  int operator||(const Byte& right) const {
      return b || right.b;
  }
  // Zápis do výstupného prúdu:
  void print(std::ostream& os) const {
    os << "0x" << std::hex << int(b) << std::dec;
  }
};
#endif // BYTE_H

#include "Byte.h"
#include <fstream>
using namespace std;

void k(Byte& b1, Byte& b2) {
  b1 = b1 * b2 + b2 % b1;

  b1 = 9; b2 = 47;
  Byte b3 = 92;
  b1 = b2 = b3;
}
```

```
int main() {
  Byte b1(47), b2(9);
  k(b1, b2);
}
```

Vídíme, že operátor = môže byť iba členskou funkciou. Toto si vysvetlíme neskôr. všimnime si, že všetky priraďovacie operátory obsahujú kód, ktorý testuje samo priradenie. Toto je všeobecné pravidlo. v niektorých prípadoch to nie je nevyhnutné, napríklad operátorom += chcemem často povedať A+=A, a priradiť A sáme sebe. Najdôležitejším miestom pre kontrolu samopriradenia je operátor =, pretože pri zložitých objektoch môže samopriradenie viesť ku katastrofe. (V niektorých prípadoch je to poriadku, ale vždy to majme na pamäti, keď píšeme operátor =).

Všetky operátory, uvedené v predchádzajúcom príklade sú preťažené tak, aby spracovávali jeden typ. Avšak v preťažených operátoroch môžeme spracovávať rôzne typy, takže môžeme napríklad sčítavať jablká s hruškami. Skôr ako začneme vyčerpávajúce preťažovanie operátorov, mali by sme sa pozrieť na časť, popisujúcu automatickú konverziu typov. Často typová konverzia na správnom mieste môže ušetriť množstvo preťažených operátorov.

Argumenty a návratové hodnoty

Na prvý pohľad sa môže zdať trochu mätúce, keď sa pozrieme na predchádzajúci príklad a vidíme všetky rôzne spôsoby prenosu a návratu argumentov. Hoci argumenty môžeme prenášať a vracať akýmkoľvek spôsobom, výber v týchto príkladoch nie je urobený náhodne. Dodržuje logickú schému, takú istú, ktorú chceme použiť vo väčšine prípadov:

1. Ako i pri argumentoch funkcií, ak potrebujeme iba čítať z argumentu a nemeniť ho, prenášame ho ako konštantný (`const`) odkaz. Obyčajne aritmetické operácie (napríklad + a – , atď.) a booleovské operácie nemenia svoje argumenty, takže prevážne použijeme prenos `const` odkazu. Ak je funkcia členskou funkciou, toto znamená vytvorenie `const` členskej funkcie. Len pre operátorové priradenia (napríklad +=) a operátor =, ktorý mení ľavý argument, ***nie je*** ľavý argument konštantný, ale stále sa prenáša ako adresa, pretože sa bude meniť.
2. Typ návratovej hodnoty, ktorý by sme mali vybrať, závisí od očakávaného významu operátora. (Opäť môžeme robiť čokoľvek s argumentami a návratovými hodnotami.) Ak účelom operátora je poskytovať hodnotu, budeme potrebovať generovať nový objekt ako návratovú hodnotu. Napríklad `Integer::operator+` musí poskytovať objekt triedy `Integer`, ktorý bude súčtom operandov. Tento objekt sa vracia hodnotou ako `const`, takže výsledok nemôže byť modifikovaný ako lvalue.
3. Všetky priraďovacie operátory modifikujú lvalue. Aby sme umožnili výsledku priradenia použitie v zreťazených výrazoch, napríklad `a=b=c`, očakáva, že vrátime odkaz na tú sitú lvalue, ktorá bola práve modifikovaná. Avšak mal by byť tento odkaz `const` alebo nie? Hoci čítame `a=b=c` zľava doprava, kompilátor ho rozkladá sprava doľava, nie sme nútení vracať nekonštantu, aby sme podporovali reťazenie priradenia. Avšak občas sa očakáva, že niečo, do čoho sme práve priradili, použijeme v nejakej operácií, napríklad `(a=b).func()` volá `func` po priradení do `b`. A tak návratová hodnota pre všetky priraďovacie operátory by mala byť nekonštantný odkaz na lvalue.
4. Pre logické operátory každý očakáva, že prinajhoršom dostane `int` a prinajlepšom `bool` (Knižnice, vytvorené pred väčšinou kompilátorov, podporujúcich C++ zabudovaný typ `bool` použijú `int` alebo ekvivalentný `typedef`.)

Inkrementačný a dekrementačný operátor predstavujú dilemu kvôli pre a postfixovej verzii. Obidve verzie menia objekt a tak nemôžu spracovávať objekt ako `const` Prefixová verzia vracia hodnotu objektu po jeho zmene, takže očakávame, že dostaneme naspäť zmenený objekt. Takže prefixová veria vracia iba `*this` ako odkaz. Postfixová verzia by mala vracať hodnotu pred jej zmenou, takže sme nútení vytvoriť samostatný objekt, ktorý bude reprezentovať túto hodnotu a vrátiť ho. Takže postfixová verzia musí vracať hodnotu, ak chceme zachovať očakávaný význam. (Všimnime si, že občas nájdeme inkrementačný a dekrementačný operátor vracať `int` alebo `bool`, ktorým sa indikuje napríklad, či objekt, ktorý bude zaradený do zoznamu, cez, ktorý budeme prechádzať, je

na konci zoznamu.) Otázka znie: Má byť vratená ako `const` alebo nie? Ak dovolíme objektu modifikáciu a niekto napíše **`(++a).func()`**, `func` bude narábať so samotným `a`, avšak **`(a++).func( )`**, narába s dočasným objektom, vráteným postfixovým operátorm `++`. Dočasné objekty sú automaticky `const`, takže toto vyznačí už kompilátor, ale kvôli jednotnosti má zmysel vytvoriť obidva ako `const`, ako je to urobené v tomto príklade. Alebo môžeme urobiť prefixovú verziu nekonštantnú a psotfixovú `const`. Kvôli rozmanitosti významov musíme pri tvorbe týchto operátorov uvažovať prípad od prípadu.

Návrat hodnoty ako const

Návrat hodnoty ako `const` sa môže zdať trochu čudné, takže si vyžaduje trochu vysvetlenia. Zoberme binárny operátor `+`. Ak ho použijeme vo výraze `f(a+b)`, výsledok `a+b` sa stáva dočasným objektom, ktorý sa použije vo volaní funkcie `f`. Pretože je dočasný, automaticky je konštantný, takže či už xplicitne vrátime hodnotu ako `const` alebo nie, nemá to žiaden účinok.

Avšak tiež je možné poslať správu do návratovej hodnoty `a+b` namiesto jej prenosu do funkcie. NApríklad môžeme napísať **`(a+b).g( )`**, kde `g` je nejaká členská funkcia triedy Integer. Tým, že spravím enávratovú hodnotu ako `const`, stanovíme tým, že len `const` členská funkcia môže byť volaná pre túto návratovú hodnotu. Toto je `const`-správne, pretože to zabraňuje ukladaniu potencionálne hodnotnej informácie v objekte, ktorý bude s najväčšou pravdepodobnosťou stratený.

Optimalizácia príkazu `return`

Všimnime i použitú formu, keď sa vracajú nové objekty hodnotou. Napríklad v operátore `+`:

```
return Integer(left.i + right.i);
```

Toto na prvý pohľad vypadá ako "funkčné volanie konštruktora", ale nie je to. Je to syntax pre dočasný objekt. Príkaz hovorí "vytvor dočasný objekt triedy triedy `Integer` a vráť ho." Kvôli tomuto by sme si mohi myslieť, že výsledok je rovnaký ako vytvorenie pomenovaného lokálneho objektu a jeho vrátenie. Avšak je to celkom odlišné. Ak namiesto toho napíšeme:

```
Integer tmp(left.i + right.i);
return tmp;
```

nastanú tri veci. Po prvé vytvorí sa objekt `tmp` vrátane volania jeho konštruktora. Po druhé, kopírovací konštruktor kopíruje `tmp` na miesto mimo ako návratovú hodnotu. Po tretie, zavolá sa deštruktor `tmp` na konci rozsahu.

Na rozdiel od !návratu dočasného objektu" tento prístup funguje celkom odlišne. Keď kompilátor vidí, že toto robíte, pozná, že objekt je určený iba na návrta hodnoty. Kompilátor túto znalosť využije a vytvorí objekt priamo na mieste návratovej hodnoty. Toto si vyžaduje len jediné volanie konštruktora (nie je potrebný kopírovací konštruktor) a tým ani volanie deštruktora, pretože v skutočnosti lokálny objekt nevytvoríme. A tak i keď to okrem pozornosti programátora nestojí nič, je to značne efektívnejšie. Toto sa často nazýva optimalizácia návratovej hodnoty.

Operátory, ktoré nemôžeme preťažiť

Existuje niekoľko operátorov v existujúcej množine, ktoré sa nedajú preťažovať. Hlavným dôvodom tohto obmedzenia je bezpečnosť. Ak by tieto operátory boli preťažiteľné, mohli by ohroziť alebo narušiť bezpečnostný mechanizmus, robiť veci ťažšími alebo miasť existujúcu prax.

- Operátor . výberu člena. Momentálne má bodka význam pre každý člen triedy,a le ak by sme povolili jej preťaženie, potom by sme nemohli pristupovať k členom normálnym spôsobom. Namiesto neho by sme museli použiť smerník a šípkový operátor ->.
- Dereferenčný operátor smerníka * z takého istého dôvodu ako operátor ..
- Neexistuje operátor umocňovania. Najpopulárnejším výberom pre toto je operátor ** z Fortranu, ale toto spôsobovalo problémy analytického charakteru. C tiež nemá operátor umocňovania, takže sa nezdalo, že by C++ tiež potrebovalo, prtože vždy môžeme volať funkciu. Oprátor umocňovania by mohol dodať vhodnú notáciu, ale nie novú funkčnosť jazyka za príliš veľkú cenu zvýšenia zložitosti jazyka.
- Neexistujú užívateľom definované operátory. Znamená to, že nemôžeme vytvárať nové operátory, ktoré sa nenachádzajú v množine. časť problému spočíva v určovaní priority a časť problému je nedostatočná potreba brať do úvahy nevyhnutné ťažkosti.
- Nemožeme meniť pravidlá priority. Dosť ťažké je zapamätať si ich, nie ešte dovoliť ľuďom hrať sa s nimi.