Polymorfizmus a virtuálne funkcie

Polymorfizmus, ktorý je v jazyku C++ implementovaný prostredníctvom virtuálnych funkcií, je po dátovej abstrakcii a dedičnosti treťou základnou vlastnosťou objektovo-orientovaného programovacieho jazyka.

Polymorfizmus dovoľuje vylepšovať organizáciu a čitateľnosť kódu, umožňuje tvorbu rozširovateľných programov, ktoré môžu rásť nielen počas počiatočného vývoja projektu, ale i v prípade požiadaviek na nové vlastnosti.

Vlastnosť zapuzdrenia umožňuje vytvárať nové dátové typy spojením charakteristík a správania. Riadenie prístupu oddeľuje rozhranie od implementácie tým, že implementačné detaily sú deklarované ako **private**. Tento druh mechanickej organizácie je pre programátora s procedurálnym pozadím zatiaľ zrozumiteľný. Dedičnosť umožňuje narábať s objektom ako s jeho vlastným typom *alebo* základným typom. Táto vlastnosť je rozhodujúca, pretože dovoľuje s mnohými typmi (odvodenými z tej istej základnej triedy) narábať ako keby boli jedného typu, a kód môže pracovať s týmito rôznymi typmi rovnako. **Virtuálna funkcia dovoľuje jednému typu vyjadriť svoje odlišnosti od iného, podobného typu, pokiaľ sú obidva typy odvodené z tej istej základnej triedy.** Tento rozdiel je vyjadrený prostredníctvom rozdielov v správaní funkcií, ktoré voláme prostredníctvom základnej triedy.

Prechod od C jazyku k C++

Programátori, programujúci v C jazyku a prechádzajúci na C++ si tento jazyk osvojujú v troch fázach. Ako prvé si uvedomia, že C++ je jednoducho "lepšie C", pretože C++ nás núti deklarovať všetky funkcie pred ich použitím, a zároveň je omnoho vyberavejšie v spôsobe používania premenných. V C programe často nájdeme chyby jednoducho iba jeho skompilovaním C++ kompilátorom.

V druhej fáze si osvoja "objektové základy C++". Znamená to, že už vidia a chápu výhody organizácie kódu, zoskupeného do dátových štruktúr spolu s funkciami, ktoré s nimi narábajú, úlohu konštruktorov a deštruktorov, a snáď i jednoduchú dedičnosť. Väčšina programátorov, ktorí pracovali nejakú dobu s C rýchlo zbadá užitočnosť týchto vlastností, pretože zakaždým, keď vytvárajú knižnicu, toto je presne to, o čo sa snažia. V C++ im napomáha samotný kompilátor.

Mnoho z nich na objektovo-orientovanej úrovni zostáva, pretože sa do nej dostane rýchlo a veľa získa bez nejakej väčšej duševnej námahy. Ľahko vytvárajú dátové typy - vytvárajú triedy a objekty, posielajú správy do týchto objektov, a všetko je pekne upravené.

Avšak ak sa tu zastavíme, nedosiahneme na najväčšiu vymoženosť jazyka, ktorá predstavuje skutočný skok do skutočného objektovo-orientovaného programovania. Týmto skokom je zvládnutie virtuálnych funkcií.

Virtuálne funkcie stupňujú hodnotu pojmu typu iba ako zapuzdrenia kódu do vnútra štruktúry, a predstavujú bezpochyby najťažší pojem, ktorý začiatočník musí zvládnuť. Je to však križovatka pochopenia objektovo-orientovaného programovania. Ak nepoužijeme virtuálne funkcie, tak nepochopíme OOP.

Pretože virtuálna funkcia je úzko spätá s pojmom typu, a typ je jadrom objektovo-orientovaného programovania, v tradičných procedurálnych jazykoch neexistuje analógia k virtuálnym funkciám. Procedurálny programátor, nemá žiaden dôvod, ktorý by ho nútil rozmýšľať o virtuálnych funkciách. Vlastnosti procedurálneho jazyka sa dajú pochopiť na algoritmickej úrovni, ale virtuálne funkcie sa dajú pochopiť len z pohľadu návrhu.

Pretypovanie smerom nahor (upcasting)

Pri dedičnosti sme si ukázali ako sa dá s objektom pracovať ako keby bol svojho vlastného typu alebo typu základnej triedy. Avšak okrem toho sa dá s nim narábať aj prostredníctvom smerníka na základný typ. Použitie adresy objektu (či už smerníka alebo odkazu) a narábanie s ním ako s adresou základného

typu sa nazýva ***pretypovanie smerom nahor*** (*upcasting* - kvôli spôsobu kreslenia stromov dedičnosti, kde sa základná trieda kreslí navrchu).

Pozrime sa na problém, ktorý vzniká v nasledujúcom kóde:

```
// Príklad 1
// Dedičnosť a pretypovanie nahor
#include <stdio.h>
#include <string>

using namespace std;

class Vozidlo {
  public:
    void Identifikuj() const {
      printf("Ja som vozidlo\n");
    };
};

// Objekt triedy Autobus je Vozidlo - majú rovnaké rozhranie:
class Autobus : public Vozidlo {
  public:
    // Predefinovanie funkcie rozhrania:
    void Identifikuj() const {
      printf("Ja som autobus\n");
    };
};

void KtoSom(Vozidlo &i) {
  i.Identifikuj();
}

int main()
{
  Autobus zaa100;
  KtoSom(zaa100); // Pretypovanie nahor
}
```

Funkcia **KtoSom()** akceptuje (ako odkaz) objekt triedy **Vozidlo**, ale tiež bez akýchkoľvek problémov a sťažností akceptuje hocičo, čo je odvodené z triedy **Vozidlo**. Takáto situácia nastáva vo funkcii **main()**, v ktorej sa do funkcie **KtoSom()** posiela odkaz na objekt triedy **Autobus** bez nutnosti explicitného pretypovania. Toto je dovolené, pretože rozhranie triedy **Vozidlo** musí existovať i v triede **Autobus**, pretože **Autobus** je **public** potomkom triedy **Vozidlo**. Pretypovanie triedy **Autobus** na triedu **Vozidlo** smerom nahor sa rozhranie tohto objektu síce "zúži", ale celé rozhranie triedy **Vozidlo** zostáva.

Tie isté tvrdenia platia i pre smerníky. Jediným rozdielom je, že užívateľ musí do funkcie explicitne posielať adresy objektov:

```
void KtoSom(Vozidlo *i) {
  i.Identifikuj();
}

int main()
{
  Autobus zaa100;
  KtoSom(&zaa100); // Pretypovanie nahor
}
```

Kde je problém

Problém vo vyššie uvedenom príklade uvidíme až za chodu programu. Výstupom programu je výpis: **Ja som vozidlo**. Je zrejmé, že toto nie je želaný výstup, pretože vieme, že skutočným objektom, ktorý sme do funkcie poslali, je objekt triedy **Autobus** a nie objekt triedy **Vozidlo**. Výsledkom volania by mal byť výpis: **Ja som** autobus. Každý objekt, odvodený z triedy **Vozidlo** by mal bez ohľadu na situáciu použiť svoju vlastnú verziu členskej funkcie **Identifikuj()**.

Z pohľadu jazyka C na funkcie nie je správanie tohto príkladu prekvapujúce. Aby sme celú záležitosť pochopili, musíme najskôr pochopiť pojem ***väzba***.

Väzba volania funkcie

Prepojenie volania funkcie s telom funkcie sa nazýva *väzba*. Väzba, ktorá sa vytvára pred odštartovaním programu (kompilátorom a spojovacím programom-linkerom), sa nazýva ***včasná väzba***. Toto je nový pojem, ktorý v procedurálnych jazykoch neexistuje: Kompilátory jazyka C poznajú len jeden druh volania funkcie, a tým je včasná väzba.

Problém vo vyššie uvedenom kóde je spôsobený práve touto včasnou väzbou, pretože kompilátor nevie zavolať správnu funkciu, keď má k dispozícii len adresu objektu triedy **Vozidlo**.

Riešenie tohto problému sa nazýva ***neskorá väzba***, ktorá znamená, že väzba sa vytvorí až za chodu programu, pričom bude založená na **type objektu**. Neskorá väzba sa tiež nazýva ***dynamická väzba*** alebo ***väzba za chodu programu***. Ak chce jazyk implementovať neskorú väzbu, musí existovať nejaký mechanizmus na určenie typu objektu za chodu programu a zavolania zodpovedajúcej členskej funkcie. V prípade kompilovaných jazykov, skutočný typ objektu kompilátor nepozná, ale do cieľového kódu vkladá kód, ktorý vyhľadáva a potom zavolá správne telo funkcie. Mechanizmus neskorej väzby je v rôznych jazykoch rôzny, ale môžeme si ho predstaviť ako nejaký druh informácie o type, ktorá musí byť inštalovaná do objektov. Ako to funguje si ukážeme neskôr.

Virtuálne funkcie

Ak chceme C++ donútiť, aby konkrétne funkcie používali neskorú väzbu, **musíme** pri deklarácií funkcie v základnej triede použiť kľúčové slovo **virtual**. Neskorá väzba sa vytvára len pre virtuálne funkcie, a len keď použijeme adresu objektu základnej triedy, kde tieto virtuálne funkcie existujú, hoci môžu byť definované i v skoršej základnej triede.

Na vytvorenie virtuálnej členskej funkcie nám stačí dať pred deklaráciu funkcie kľúčové slovo **virtual**. Kľúčové slovo **virtual** sa zadáva **len v deklarácií funkcie, nie v definícií**. Ak je funkcia deklarovaná ako **virtual** v základnej triede, potom bude **virtual** vo všetkých odvodených triedach. Predefinovanie virtuálnej funkcie v odvodenej triede sa zvyčajne nazýva ***prevažovanie***.

Všimnime si, že nám stačí deklarovať funkciu ako **virtual** len v základnej triede. Všetky funkcie v odvodených triedach, ktoré majú zhodnú signatúru s deklaráciou v základnej triede, sa budú volať použitím virtuálneho mechanizmu. Kľúčové slovo **virtual** môžeme použiť i v deklaráciách v odvodených triedach (nič sa tým nepoškodí), ale je viac-menej nadbytočné, prípadne užitočné iba z pohľadu zvýšenia čitateľnosti triedy.

Ak chceme, aby program z vyššie uvedeného príkladu fungoval správne, stačí pred deklaráciu členskej funkcie **Identifikuj** pridať kľúčové slovo **virtual**:

```
// Príklad 2
// Neskorá väzba pomocou kľúčového slova virtual
#include <stdio.h>
#include <string.h>

class Vozidlo {
  public:
    virtual void Identifikuj() const {
```

```
      printf("Ja som vozidlo");
    };
};

// Objekt triedy Autobus je i Vozidlo - má rovnaké rozhranie
class Autobus : public Vozidlo {
  public:
    // Prekrytie funkcie rozhrania
    virtual void Identifikuj() const {
      printf("Ja som autobus");
    };
};

void KtoSom(Vozidlo &i) {
  i.Identifikuj();
}

int main()
{
  Autobus karosa;
  KtoSom(karosa); // Pretypovanie nahor
}
```

Tento program je identický s predchádzajúcim, až na doplnené kľúčové slovo **virtual**. Samozrejme správanie je **podstatne** iné, výstupom teraz bude výpis: **Ja som autobus**.

Rozšírovateľnosť

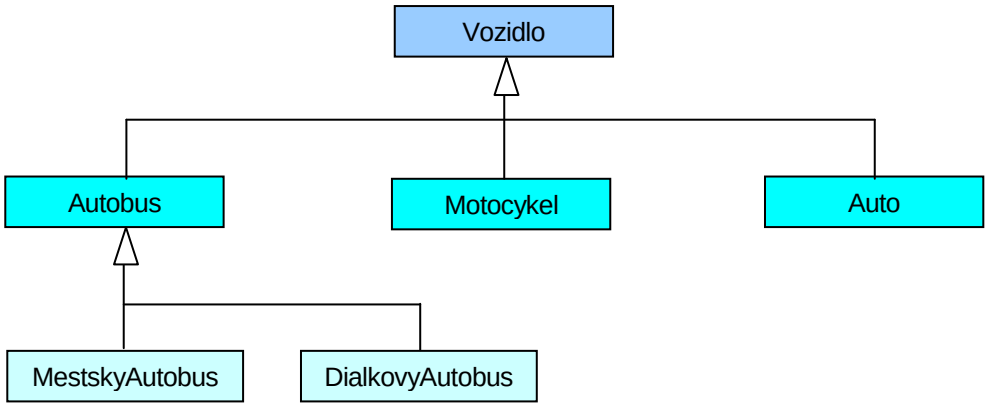
Ak v základnej triede členskú funkciu **Identifikuj** definujeme ako **virtual**, môžeme pridávať akékoľvek množstvo nových typov bez nutnosti meniť funkciu **KtoSom**. V dobre navrhnutom OOP programe väčšina alebo všetky funkcie budú nasledovať model funkcie **KtoSom** a komunikovať len prostredníctvom rozhrania základnej triedy. Takýto program je rozšírovateľný, pretože môžeme dodávať novú funkčnosť dedením nových dátových typov zo spoločnej základnej triedy. Funkcie, ktoré narábajú s rozhraním základnej triedy sa nemusia vôbec meniť z dôvodu, aby sa prispôsobili novým triedam.

Nasledujúci príklad doplňuje ďalšie virtuálne funkcie i niekoľko nových tried, pričom všetky triedy budú správne fungovať so starou funkciou **KtoSom**, ktorú nemusíme vôbec meniť:

```
// Príklad 3
// Rozšírovateľnosť v OOP
#include <stdio.h>
#include <string>

using namespace std;

class Vozidlo {
  public:
    virtual void Identifikuj() const {
      printf("Ja som vozidlo");
    };
    virtual char* Druh() const {
      return "Vozidlo";
    };
    // Predpokladáme, že táto funkcia mení objekt
    virtual void Nastav(int) {};
};

class Autobus : public Vozidlo {
  public:
    void Identifikuj() const {
```

```
      printf("Ja som autobus");
    };
    char* Druh() const { return "Autobus"; };
    void Nastav(int) {};
};

class Motocykel : public Vozidlo {
  public:
    void Identifikuj() const {
      printf("Ja som motocykel");
    };
    char* Druh() const { return "Motocykel"; };
    void Nastav(int) {};
};

class Auto : public Vozidlo {
  public:
    void Identifikuj() const {
      printf("Ja som auto");
    };
    char* Druh() const { return "Auto"; };
    void Nastav(int) {};
};

class MestskyAutobus : public Autobus {
  public:
    void Identifikuj() const {
      printf("Ja som mestsky autobus");
    };
    char* Druh() const { return "Mestsky autobus"; };
};

class DialkovyAutobus : public Autobus {
  public:
    void Identifikuj() const {
      printf("Ja som dialkovy autobus");
    };
     char* Druh() const { return "Dialkovy autobus"; }
};

// Identická funkcia ako predtým
void KtoSom(Vozidlo &i)
{
  i.Identifikuj();
}

// Nová funkcia
void F(Vozidlo &i)
{
  i.Nastav(1);
}

// Pretypovanie nahor počas inicializácie
Vozidlo* A[] = {
  new Autobus,
  new Motocykel,
  new Auto,
  new MestskyAutobus,
};

int main()
```

```
{
  Autobus karosa;
  Motocykel honda;
  Auto skoda;
  MestskyAutobus ikarus;
  DialkovyAutobus mercedes;
  KtoSom(karosa);
  KtoSom(honda);
  KtoSom(skoda);
  KtoSom(ikarus);
  KtoSom(mercedes);
  F(ikarus);
}
```



Obr. 1 Príklad 3 - diagram tried - dedičnosť

Vidíme, že aj keď bola doplnená ďalšia úroveň dedičnosti (Obr. 1) pod triedu **Autobus**, mechanizmus virtuality funguje správne bez ohľadu na to, koľko úrovní dedičnosti existuje. Členská funkcia **Nastav *nie je*** v triedach **MestskyAutobus** a **DialkovyAutobus** prekrytá. V takomto prípade sa použije najbližšia definícia v hierarchii dedičnosti - kompilátor zaručí, že vždy bude existovať ***nejaká*** definícia virtuálnej funkcie, takže nikdy neskončíme s volaním, ktoré by nebolo prepojené s telom funkcie (čo by bola katastrofa).

Pole **A[]** obsahuje smerníky na základnú triedu **Vozidlo**, takže počas inicializácie poľa nastáva pretypovanie smerom nahor (upcasting). O poli a funkcii **F** si povieme neskôr.

Vo volaní funkcie **KtoSom** sa pretypovanie smerom nahor vykonáva pre každý odlišný typ objektu, aby sa vykonalo požadované správanie. Takýto zápis môžeme prečítať asi nasledovne "posielam správu do objektu, a nech sa objekt trápi, čo s ňou spraví". Virtuálna funkcia plní funkcie akejsi šošovky, ktorú používame, keď sa snažíme analyzovať projekt v zmysle, kde by sa mali nachádzať základné triedy a ako by sme mohli program rozširovať. Avšak ak i objavíme vhodné rozhrania základnej triedy a virtuálne funkcie na samom začiatku tvorby programu, často mnohé objavíme neskôr, dokonca omnoho neskôr, než sa rozhodneme rozšíriť program alebo inak pokračovať v programe. Toto nie je nedostatok analýzy alebo návrhu, jednoducho to znamená, že sme nepoznali alebo nemohli sme poznať všetky informácie hneď na samom začiatku. Vďaka úzkej modularizácií tried nie je v C++ veľkým problémom, keď takáto modifikácia nastane, pretože zmeny, ktoré urobíme v jednej časti systému nemajú tendenciu prejaviť sa v ostatných častiach systému (na rozdiel od C-jazyka).

C++ a implementácia neskorej väzby

Ako sa v C++ vytvára neskorá väzba? Všetku prácu vykonáva kompilátor skryte za scénou. On inštaluje potrebný mechanizmus neskorej väzby, keď o ňu požiadame (prostredníctvom virtuálnych funkcií).

Pretože programátori v C++ často ťažia zo znalosti mechanizmu virtuálnych funkcii, povieme si niečo o spôsobe akým kompilátor tento mechanizmus implementuje.

V prvom rade kľúčové slovo **virtual** kompilátoru oznámi, že nemá vytvárať včasnú väzbu. Namiesto nej by mal automaticky inštalovať celý mechanizmus, potrebný na realizáciu neskorej väzby. Znamená to, že ak zavoláme funkciu **Identifikuj()** pre objekt triedy **MestskyAutobus** prostredníctvom adresy na základnú triedu **Vozidlo**, zavolá sa správna funkcia.

Na dosiahnutie tohto cieľa typický kompilátor vytvára pre každú triedu jednu tabuľku (nazývanú VTABLE), ktorá obsahuje virtuálne funkcie. Do VTABLE kompilátor ukladá adresy virtuálnych funkcií konkrétnej triedy. V každej triede, ktorá obsahuje aspoň jednu virtuálnu funkciu, je skryte uložený smerník, nazývaný ***vpointer*** (skrátene VPTR), ktorý ukazuje na VTABLE triedy. Keď zavoláme virtuálnu funkciu prostredníctvom smerníka na základnú triedu (t.j. keď urobíme tzv. ***polymorfné volanie***), kompilátor potichu vygeneruje kód na výber VPTR a vyhľadanie adresy funkcie vo VTABLE, čím sa zavolá správna funkcia a realizuje sa neskorá väzba.

A teraz trochu podrobnejšie.

Uchovanie typovej informácie

Vidíme, že v žiadnej triede sa neuchováva explicitná informácia o type. Ale predchádzajúce príklady a jednoduchá logika, naznačujú, že musí existovať nejaký druh typovej informácie, uloženej v objektoch, inak by nebolo možné určovať typ za chodu programu. Je to pravda, ale typová informácia je skrytá. Nasledujúci príklad skúma veľkosti tried, ktoré používajú virtuálne funkcie v porovnaní s tými, ktoré ich neobsahujú.

```
// Príklad 4
// Veľkosti objektov s/bez virtual funkcií
#include <stdio.h>

class BezVirtual {
    int a;
  public:
    void X() const {}
    int I() const { return 1; }
};

class JednaVirtual {
    int a;
  public:
    virtual void X() const {}
    int I() const { return 1; }
};

class DveVirtual {
    int a;
  public:
    virtual void X() const {}
    virtual int I() const { return 1; }
};

int main()
{
  printf("int: %d\n",sizeof(int));
  printf("BezVirtual: %d\n",sizeof(BezVirtual));
  printf("void *: %d\n",sizeof(void*));
  printf("JednaVirtual: %d\n",sizeof(JednaVirtual));
  printf("DveVirtual: %d\n",sizeof(DveVirtual));
}
```

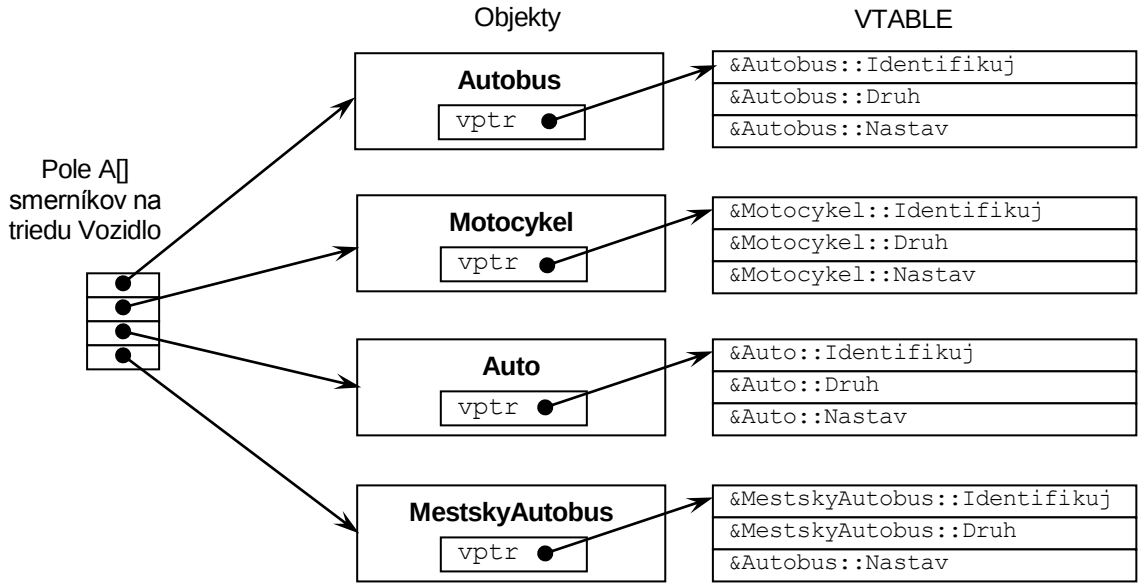
Veľkosť objektu, ktorý neobsahuje virtuálnu funkciu, je podľa očakávania rovná veľkosti jedného **int**. V triede **JednaVirtual**, ktorá obsahuje jednu virtuálnu funkciu, je veľkosť daná veľkosťou **int** plus veľkosť smerníka typu **void**. Kompilátor do triedy automaticky vloží jeden smerník (VPTR), ak obsahuje ***jednu alebo viac*** virtuálnych funkcií. Medzi triedami **JednaVirtual** a **DveVirtual** nie je čo sa týka veľkosti žiaden rozdiel. VPTR adresuje tabuľku adries funkcií. Potrebujeme iba jednu tabuľku, pretože adresy všetkých virtuálnych funkcií sú uložené v jedinej tabuľke.

Tento príklad vyžaduje najmenej jeden dátový člen. Ak trieda neobsahuje žiaden dátový člen, C++ kompilátor si vynúti , aby objekty mali nenulovú veľkosť, pretože každý objekt musí mať jednoznačnú adresu. Do objektov, ktoré by inak mali nulovú veľkosť, sa vloží tzv. ***fiktívny člen***. Keď sa do triedy vkladá typová informácia kvôli kľúčovému slovu **virtual**, toto sa uskutoční prostredníctvom "*fiktívneho*" člena. Môžeme si to vyskúšať, ak zakomentujeme **int** `a` vo všetkých triedach vo vyššie uvedenom príklade, a pozrieme si čo nastane po spustení programu.

Zobrazenie virtuálnych funkcií

Aby sme presne pochopili, čo nastane, keď použijeme virtuálnu funkciu, je dobré vizualizovať si činnosti, ktoré prebiehajú skryte za oponou. Tu je obrázok poľa smerníkov **A** z príkladu 3:

Obr. 2 Tabuľka virtuálnych metód

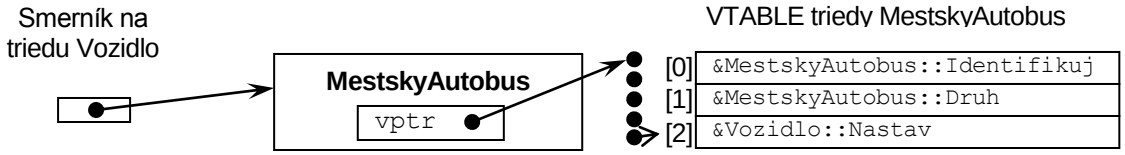


Pole smerníkov **A** na triedu **Vozidlo** neobsahuje špecifické informácie o type, každý ukazuje na objekt typu **Vozidlo**. Všetky triedy **Autobus**, **Motocykel**, **Auto** a **MestskyAutobus** spadajú do tejto kategórie, pretože sú odvodené od triedy **Vozidlo** (majú teda rovnaké rozhranie ako **Vozidlo** a môžu odpovedať na tie isté správy), takže ich adresy môžu byť tiež vkladané do poľa. Avšak kompilátor nepozná, že sú niečo viac, než objekty triedy **Vozidlo**, a tak normálne by zavolal verzie funkcii zo základnej triedy. Pretože v tomto prípade všetky funkcie boli deklarované kľúčovým slovom **virtual**, stane sa niečo iné.

Zakaždým, keď vytvoríme triedu, ktorá obsahuje virtuálne funkcie alebo ju odvodíme z triedy, ktorá obsahuje virtuálne funkcie, kompilátor vytvorí pre túto triedu jednoznačnú VTABLE (pozri pravú stranu Obr. 2). Do tejto tabuľky uloží adresy všetkých virtuálnych funkcií, t.j. takých, ktoré sú v tejto triede alebo v základnej triede deklarované ako **virtual**. Ak v odvodenej triede neprekryjeme funkciu, ktorá bola v základnej triede deklarovaná ako **virtual**, kompilátor použije v odvodenej triede adresu verzie zo základnej triedy. (Ako to ukazuje položka **Nastav** vo VTABLE triedy **MestskyAutobus**.) Potom sa uloží smerník VPTR do triedy. Ak použijeme jednoduchú dedičnosť, ako je táto, každý bude objekt obsahovať len jeden VPTR. VPTR sa musí inicializovať tak, aby ukazoval na počiatočnú adresu zodpovedajúcej VTABLE (toto robí konštruktor, na ktorý sa ešte pozrieme).

Po inicializácií VPTR na vhodnú VTABLE, objekt *pozná* akého je typu. Ale toto samopoznanie je nám v okamihu volania virtuálnej funkcie nanič.

Ak zavoláme virtuálnu funkciu prostredníctvom adresy základnej triedy (situácia, kedy kompilátor nemá všetky informácie potrebné na vykonanie včasnej väzby) nastane niečo zvláštne. Namiesto vykonania typického volania funkcie, čo je jednoduchá inštrukcia asembleru CALL s konkrétnou adresou, na vykonanie volania funkcie kompilátor generuje iný kód. Takto napríklad vypadá volanie funkcie **Nastav** pre objekt triedy **MestskyAutobus**, ak sa urobí prostredníctvom smerníka na triedu **Vozidlo** (odkaz na **Vozidlo** robí to isté):



Kompilátor začne so smerníkom na triedu **Vozidlo**, ktorý ukazuje na počiatočnú adresu objektu. Všetky objekty triedy **Vozidlo** alebo objekty, odvodené z triedy **Vozidlo** majú svoju VPTR na tom istom mieste (často na začiatku objektu), takže kompilátor dokáže VPTR z objektu vybrať. VPTR ukazuje na počiatočnú adresu VTABLE. Všetky adresy funkcií vo VTABLE s[usporiadané v tom istom poradí bez ohľadu na konkrétny typ objektu, funkcia **Nastav** je na adrese VPTR+2. A tak namiesto „Zavolaj funkciu na absolútnej adrese **Vozidlo::Nastav()**“ (včasná väzba - nesprávna činnosť), sa generuje kód, ktorý hovorí „Zavolaj funkciu s adresou, ktorá je uložená vo **VPTR+2**“. Pretože výber VPTR a určenie skutočnej adresy funkcie nastáva za chodu programu, dostaneme požadovanú neskorú väzbu. Do objektu sa pošle správu a objekt určí, čo s ňou má urobiť.

Inštalácia smerníka na tabuľku virtuálnych metód

Pretože VPTR určuje správanie virtuálnej funkcie objektu, skutočnosť aby VPTR vždy adresovala správnu VTABLE je pre triedu kritická. Pred správnou inicializáciou VPTR sa virtuálna funkcia volať nedá. Je zrejmé že, jediným miestom, kde sa inicializácia VPTR dá zabezpečiť, je konštruktor. Avšak trieda **Vozidlo** ani žiaden z jej potomkov v tomto príklade nemá konštruktor explicitne definovaný.

Toto je skutočnosť, kde je veľmi dôležité vytvorenie (vygenerovanie) implicitného konštruktora. V našom príklade kompilátor vytvára implicitné konštruktory, ktoré nerobia nič okrem inicializácie VPTR. Tento konštruktor sa samozrejme volá pre každý objekt triedy **Vozidlo** skôr, než niečo s týmto objektom niečo urobíme, takže volanie virtuálnej funkcie bude vždy bezpečné.

Objekty sú rôzne

Je dôležité uvedomiť si, že pretypovanie smerom nahor (upcasting) sa týka len adries. Ak má kompilátor k dispozícií objekt, pozná jeho presný typ, tak potom neskorú väzbu nepoužije pre žiadne volanie funkcie – alebo prinajmenšom, kompilátor ***nepotrebuje*** použiť neskorú väzbu. Z výkonnostných dôvodov väčšina kompilátorov, keď vytvára volania virtuálnych funkcií pre objekt však používa včasnú väzbu, pretože pozná presný typ objektu. Napríklad:

```
// Včasná väzba a virtuálne funkcie
#include <stdio.h>
#include <string>

using namespace std;

class Vozidlo {
  public:
    virtual string Druh() const { return ""; };
};
```

```
class Auto : public Vozidlo {
  public:
    string Druh() const { return "Opel"; };
};

int main()
{
  Auto mojeauto;
  Vozidlo *p1 = &mojeauto;
  Vozidlo &p2 = mojeauto;
  Vozidlo p3;
  // Neskorá väzba
  printf("p1->Druh() = %s\n",p1->Druh().c_str());
  printf("p2.Druh() = %s\n",p2.Druh().c_str());
  // Včasná väzba (pravdepodobne):
  printf("p3.Druh() = %s\n",p3.Druh().c_str());
}
```

V **p1->Druh()** a **p2.Druh()** sú použité adresy, čo znamená, že informácia nie je úplná: Smerníky **p1** a **p2** môžu predstavovať adresy objektov triedy **Vozidlo** alebo niečoho, čo je odvodené z triedy **Vozidlo**, takže musí byť použitý virtuálny mechanizmus volania funkcie. Pri volaní **p3.Druh()** nie je žiadna nejednoznačnosť. Kompilátor pozná presný typ, je to objekt, nemôže to byť niečo odvodené od triedy **Vozidlo** – je to presne objekt triedy **Vozidlo**. Takže pravdepodobne sa použije včasná väzba. Avšak ak kompilátor nechce ťažko pracovať, použije neskorú väzbu.

Prečo virtuálne funkcie

V tomto okamihu by sme sa mohli pýtať: „Ak je tento postup taký dôležitý a neustále vytvára „správne“ volania funkcie, prečo je voliteľný? Prečo by sme mali o ňom niečo vedieť?

Nuž odpoveď tvorí súčasť základnej filozofie C++: “Pretože nie je až tak efektívny.“ Z predchádzajúceho výstupu v asembleri vidíme, že namiesto jednoduchého volania CALL s absolútnou adresou sú na nastavenie volania virtuálnej funkcie potrebné dve komplikovanejšie inštrukcie asemblera. Toto si vyžaduje kódový priestor i čas na vykonávanie.

Niektoré objektovo-orientované jazyky prijali taký prístup, že neskorá väzba je tak prirodzená pre objektovo-orientované programovanie, že sa vykonáva vždy, nie je voliteľná a užívateľ by o nej nemal vedieť. Toto je úvaha, ktorá sa rieši pri tvorbe jazyka a tento prístup sa hodí pre mnoho jazykov. Avšak C++ je dedičom C, kde je výkon kritickým parametrom. Koniec koncov C bolo vytvorené ako náhrada za asembler na implementovanie operačného systému (vďaka čomu bol tento operačný systém – Unix – omnoho prenositeľnejší než jeho predchodcovia). Jedným z hlavných dôvodov vynájdenia C++ bolo zvýšiť výkon C programátorov. A prvou otázkou, keď C programátori narazili na C++ bolo „Aký bude mať dopad na rozsah a rýchlosť kódu?“ Ak by odpoveď bola „Všetko je vynikajúce, okrem volania funkcií, kde bude vždy nejaká extra réžia“, mnoho ľudí by zostalo pri C a neprešlo na C++. Okrem toho, keby existovala iba neskorá väzba, nemihli by existovať **inline** funkcie, pretože virtuálne funkcie musia mať adresu, ktorá sa ukladá do VTABLE. A preto virtuálna funkcia je len jednou z možností a jazyk je nastavený na nie-virtuálne funkcie, t.j. na najrýchlejšiu konfiguráciu. Autor C++, Stroustrup, sa držal pravidla „Čo nepoužiješ, za to neplatíš“.

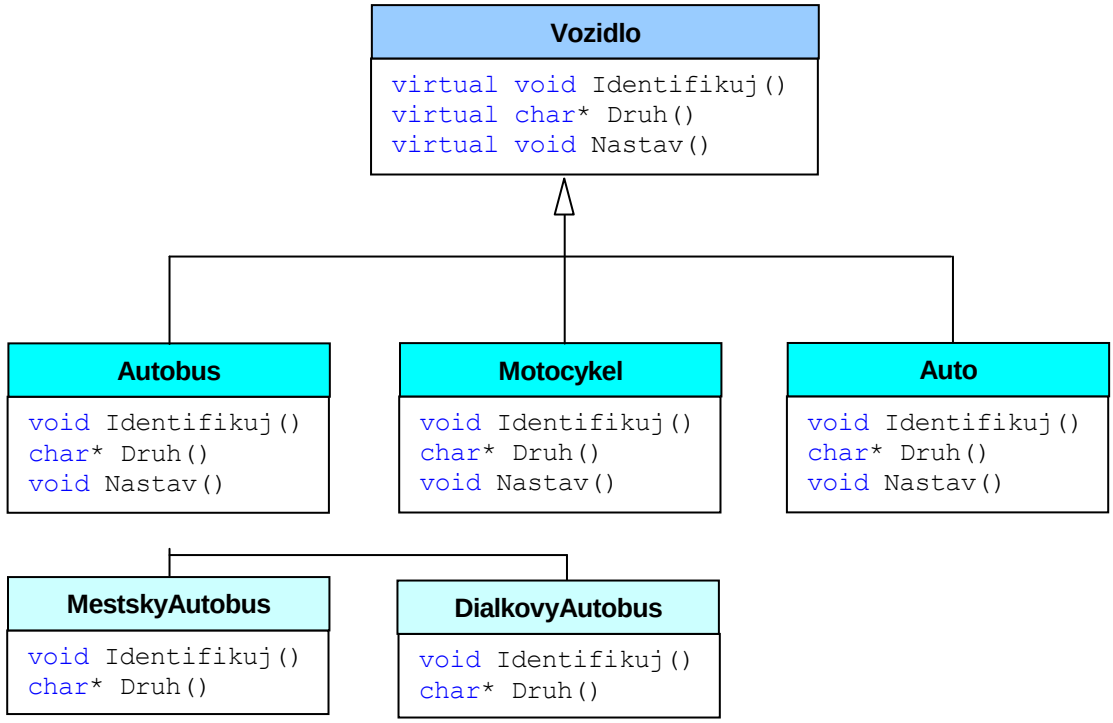
A tak na ladenie výkonu bolo doplnené kľúčové slovo **virtual**. Avšak pri návrhu tried by sme sa nemali ešte starať o ladenie výkonu. Ak sa chystáme použiť polymorfizmus, potom ho použime, použime virtuálne funkcie všade. Po funkciách, ktoré by mohli byť nevirtuálne, začnime pátrať, keď sa snažíme zrýchliť svoj kód (a zvyčajne viac získame v iných oblastiach – dobrý profiler nám nájde úzke hrdlá, ktoré by sme hľadali len odhadom).

Abstraktné základné triedy a čisto virtuálne funkcie

V návrhu často potrebujeme, aby základná trieda pre odvodené triedy poskytovala ***len*** rozhranie. Znamená to, že nechceme, aby niekto skutočne vytvoril objekt základnej triedy, chceme len, aby mohol pretypovávať smerom nahor, aby mohol použiť jej rozhranie. Toto dosiahneme vytvorením tzv. ***abstraktnej*** triedy, ktorou sa trieda stáva, ak obsahuje najmenej jednu ***čisto virtuálnu funkciu***. Čisto virtuálnu funkciu poznáme podľa kľúčového slova **virtual** a za funkciou musí nasledovať **=0**. Ak sa niekto pokúsi vytvoriť objekt abstraktnej triedy, kompilátor mu v tom zabráni. Abstraktné triedy predstavujú nástroj, ktorý nám dovoľuje presadzovať konkrétny návrh.

Ak sa zdedí abstraktná trieda, potomok musí implementovať všetky čisto virtuálne funkcie, inak sa tiež stáva abstraktnou triedou. Vytváranie čisto virtuálnych funkcií nám dovoľuje vkladať členskú funkciu do rozhrania bez nutnosti zabezpečiť pre túto členskú funkciu telo, ktoré nemá žiaden zmysel. Súčasne čisto virtuálna funkcia núti potomkov vytvárať definície týchto funkcií.

V predchádzajúcich príkladoch boli funkcie v základnej triede **Vozidlo** vždy prázdne. Ak sa tieto funkcie niekedy zavolali, niekde je niečo nesprávne, pretože účelom triedy **Vozidlo** je vytvárať spoločné rozhranie pre všetky odvodené triedy.



Obr. 3 Diagram tried - rozhranie

Jediným dôvodom vybudovania spoločného rozhrania je to, že môže byť vyjadrené rôzne v každom odlišnom podtype. Vytvára základnú formu, ktorá určuje, čo je spoločné pre všetky odvodené triedy – nič viac. A tak **Vozidlo** je vhodným kandidátom na abstraktnú triedu. Abstraktnú triedu vytvárame, keď chceme len narábať s množinou tried prostredníctvom spoločného rozhrania, ale rozhranie nepotrebuje mať implementáciu (alebo prinajmenšom celú implementáciu).

Ak máme predstavu ako je **Vozidlo**, ktorá funguje ako abstraktná trieda, objekty tejto triedy takmer vždy nemajú význam. Znamená to, že **Vozidlo** je určená len na vyjadrenie rozhrania a nie konkrétnej implementácie, takže vytvorenie objektu, ktorý by bol len **Vozidlo** nedáva zmysel, a preto pravdepodobne chceme užívateľovi zabrániť, aby to urobil. Toto sa dá realizovať napríklad tak, že všetky virtuálne funkcie triedy **Vozidlo** budú tlačiť chybové oznamy. Je však omnoho lepšie zachytiť tento problém už počas kompilácie.

Syntax deklarácie čisto virtuálnej funkcie je nasledovná.

```
virtual void Fun() = 0;
```

Takýmto zápisom prekladaču oznámime, aby rezervoval zásuvku pre funkciu vo VTABLE, ale neukladaj adresu do tejto zásuvky. Dokonca i keď jediná funkcia je deklarovaná ako čisto virtuálna, VTABLE je neúplná.

Ak je VTABLE triedy neúplná, čo má kompilátor robiť, keď sa niekto pokúša vytvoriť objekt tejto triedy? Nemôže bezpečne vytvoriť objekt abstraktnej triedy, a tak dostaneme chybový oznam kompilátora. A tak kompilátor zabezpečuje neporušenosť abstraktnej triedy. Vytvorením triedy ako abstraktnej zabezpečíme, že klientsky programátor ju nemôže nesprávne používať.

A takto vypadá upravený kód triedy **Vozidlo**, používajúci virtuálne funkcie. Pretože trieda nemá nič okrem virtuálnych funkcií, voláme ju ***čisto abstraktná trieda***:

```
// Čisto abstraktné základné triedy
#include <stdio.h>
using namespace std;

class Vozidlo {
public:
  // Čisto virtuálne funkcie:
  virtual void Identifikuj() const = 0;
  virtual char* Druh() const = 0;
  // Predpokladáme, že táto funkcia zmení objekt:
  virtual void Nastav(int) = 0;
};
// Zvyšok súboru zostáva nezmenený

class Autobus : public Vozidlo {
public:
  void Identifikuj() const {
    printf("Ja som autobus\n");
  }
  char* Druh() const { return "Autobus"; }
  void Nastav(int) {}
};

class Motocykel : public Vozidlo {
public:
  void Identifikuj() const {
    printf("Ja som motocykel \n");
  }
  char* Druh() const { return "Motocykel"; }
  void Nastav(int) {}
};

class Auto : public Vozidlo {
public:
  void Identifikuj() const {
    printf("Auto::Identifikuj\n");
  }
  char* Druh() const { return "Auto"; }
  void Nastav(int) {}
};

class MestskyAutobus : public Autobus {
public:
  void Identifikuj() const {
```

```
    printf("MestskyAutobus::Identifikuj\n");
  }
  char* Druh() const { return "MestskyAutobus"; }
};

class DialkovyAutobus : public Autobus {
public:
  void Identifikuj() const {
    printf("DialkovyAutobus::Identifikuj\n");
  }
  char* Druh() const { return "DialkovyAutobus"; }
};

// Identická funkcia ako predtým
void KtoSom(Vozidlo &i) {
  // ...
  i.Identifikuj();
}

// Nová funkcia:
void F(Vozidlo &i) { i.Nastav(1); }

int main() {
  Autobus karosa;
  Motocykel honda;
  Auto skoda;
  MestskyAutobus ikarus;
  DialkovyAutobus mercedes;
  KtoSom(karosa);
  KtoSom(honda);
  KtoSom(skoda);
  KtoSom(ikarus);
  KtoSom(mercedes);
  F(karosa);
}
```

Čisto virtuálne funkcie sú užitočné, pretože robia abstraktnosť triedy explicitnou a naznačujú užívateľovi i kompilátoru, ako sa má táto trieda používať.

Všimnime si, že čisto virtuálne funkcie zamedzujú, aby abstraktná trieda bola do funkcie prenášaná ***hodnotou***. Týmto zároveň predstavuje i spôsob, ako predísť ***orezaniu objektu*** (Popísané neskôr). Vytvorením triedy ako abstraktnej zabezpečíme, že pri pretypovaní smerom nahor (upcasting) na túto triedu sa budú vždy používať smerníky alebo odkazy.

To, že jedna čisto virtuálna funkcia zamedzuje vytvoriť kompletnú VTABLE neznamená, že nechceme telá iných funkcií. Často chceme zavolať verziu funkcie zo základnej triedy, dokonca i keď je virtuálna. Je vždy dobré vkladať spoločný kód čo najbližšie ku koreňu hierarchie. Nielen že to šetrí kódový priestor, ale uľahčuje to i šírenie zmien.

Čisto virtuálne definície

V základnej triede je možné definovať čisto virtuálnu funkciu. Týmto kompilátoru oznamujeme, aby nedovolil vytvárať objekty abstraktnej triedy a čisto virtuálne funkcie musia byť definované v odvodenej triede, aby sa dali vytvárať objekty. Avšak tieto definície môžu obsahovať kúsok spoločného kódu, ktorý chceme volať vo všetkých definíciách v odvodených tried namiesto jeho duplikovania v každej funkcii.

Definícia takejto čisto virtuálnej funkcie vypadá nasledovne:

```
// Definície čisto virtuálnych funkcií
#include <stdio.h>
using namespace std;

class Vozidlo {
public:
  virtual void Druh() const = 0;
  virtual void Znacka() const = 0;
  // Inline čisto virtuálne funkcie nie sú dovolené:
  //!  virtual void ok() const = 0 {}
};

// V poriadku, nie je definovaná inline
void Vozidlo:: Znacka() const {
  printf("Vozidlo::test()\n");
}

void Vozidlo::Druh() const {
  printf("Vozidlo::Druh()\n");
}

class Auto : public Vozidlo {
public:
  // Použi spoločný kód z triedy Vozidlo:
  void Druh() const { Vozidlo::Typ(); }
  void Znacka() const { Vozidlo::Znacka(); }
};

int main() {
  Auto opel;
  opel.Druh();
  opel.Znacka();
}
```

Zásuvka tabuľky VTABLE triedy **Vozidlo** je stále prázdna, ale existuje funkcia s takýmto menom, ktorú môžeme volať v odvodenej triede.

Ďalšou výhodou tejto vlastnosti je, že dovoľuje prechádzať z obyčajnej virtuálnej funkcie na čisto virtuálnu funkciu bez narušenia existujúceho kódu.

Dedičnosť a VTABLE

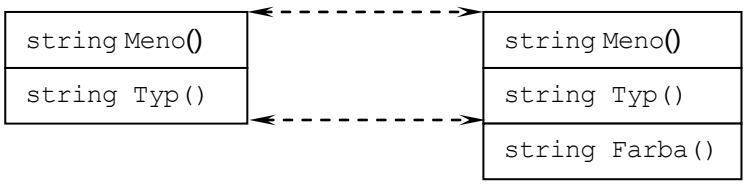
Už si dokážeme predstaviť, čo sa stane, keď použijeme dedičnosť a prekryjeme niektoré virtuálne funkcie. Kompilátor pre novú triedu vytvorí novú VTABLE a vloží do nej adresy nových funkcií, pričom pre neprekryté funkcie použije adresy funkcii zo základnej triedy. Tak či tak pre každý objekt, ktorý sa môže vytvárať (t.j. jeho trieda nemá čisto virtuálne funkcie) existuje plná množina adries funkcií vo VTABLE, takže nikdy nebudeme môcť zavolať adresu, ktorá tam nie je (čo by bola katastrofa).

Ale čo sa stane, keď zdedíme a pridáme nové virtuálne funkcie v odvodenej triede?

```
// Pridanie virtuálnej metódy v potomkovi
#include <stdio.h>
#include <string>
using namespace std;

class Vozidlo {
  string Meno_d;
public:
  Vozidlo(const string &meno) : Meno_d(meno) {}
  virtual string Meno() const { return Meno_d; }
```

```
  virtual string Typ() const { return ""; }
};

class Auto : public Vozidlo {
  string Znacka_d;
public:
  Auto(const string &meno) : Vozidlo(meno) {}
  // Nová virtuálna funkcia v triede Auto:
  virtual string Farba() const {
    return Vozidlo::Meno() + " farba";
  }
  string Typ() const { // Prekrytie
    return Vozidlo::Meno() + " znacka 'Opel'";
  }
};

int main() {
  Vozidlo* p[] = { new Vozidlo("akekolvek"),new Auto("Skoda")};
  printf("p[0]->Typ()= %s\n",p[0]->Typ().c_str());
  printf("p[1]->Typ() = %s\n ",p[1]->Typ().c_str());
//! printf("p[1]->sedadlo() = %s\n",
//!       p[1]-> sedadlo().c_str()); // Neplatné
} ///:~
```

Trieda **Vozidlo** obsahuje dve virtuálne funkcie: **Typ()** a M**eno()**. Trieda **Auto** doplňuje tretiu virtuálnu funkciu nazvanú **Farba()**, a zároveň prevažuje význam funkcie **Typ()**. Čo nastane nám pomôže vysvetliť obrázok, ktorý obsahuje. A takto vypadajú tabuľky VTABLE, vytvorené kompilátorom pre triedy **Vozidlo** a **Auto**:



Obr. 4 Tabuľka virtuálnych metód potomka

Všimnime si, že kompilátor mapuje pozíciu adresy funkcie **Typ** do rovnakej zásuvky vo VTABLE triedy **Auto** ako je v namapovaná triede **Vozidlo**. Podobne ak by trieda **Autobus** bola potomkom triedy **Auto**, jej verzia funkcie **Farba()** by bola do VTABLE umiestnená na to isté miesto, ako je to v triede **Auto**. Je to dané tým, že kompilátor generuje kód, ktorý používa na výber virtuálnej funkcie jednoduchý numerický ofset do VTABLE. Bez ohľadu na podtyp, ktorému objekt prináleží, jeho VTABLE je usporiadaná rovnakým spôsobom, takže volanie virtuálnej funkcie bude vždy rovnaké.

Avšak v tomto prípade kompilátor pracuje iba so smerníkom na objekt základnej triedy. Základná trieda má iba funkcie **Typ()** a **Meno()**, a tak toto sú jediné funkcie, ktoré nám kompilátor dovolí zavolať. Ako prípadne vie, že pracujeme s objektom triedy **Auto**, keď má smerník iba na objekt základnej triedy? Tento smerník by mohol adresovať nejaký iný typ, ktorý nemá funkciu **Farba()**. Na tomto mieste vo VTABLE môže mať nejaké ďalšie adresy funkcií, ale v každom prípade, virtuálne volanie tejto adresy z VTABLE nie je to čo chceme. Kompilátor robí svoju prácu tak, že pred volaním virtuálnych funkcií, ktoré existujú iba v odvodených triedach nás chráni.

Menej často existujú prípady, v ktorých vieme, že smerník skutočne adresuje objekt konkrétnej podtriedy. Ak chceme zavolať funkciu, ktorá existuje len v tejto podtriede, potom musíme smerník pretypovať. Chybový oznam odstránime kódom:

```
((Auto*)p[1])->Farba()
```

Tu určite vieme, že **p[1]** adresuje objekt triedy **Auto**, ale vo všeobecnosti toto nevieme. Ak náš problém spočíva v tom, že musíme poznať presné typy všetkých objektov, mali by sme to premyslieť, pretože

pravdepodobne nepoužívame virtuálne funkcie správne. Avšak existujú i situácie, kedy návrh funguje najlepšie (alebo nemáme inú možnosť), ak poznáme presný typ všetkých objektov, uchovávaných v generickom kontajneri. Toto je problém ***identifikácie typu za chodu programu*** (*run-time type identification - RTTI*).

Pojem RTTI zahrňuje všetko o pretypovaní smerníkov základnej triedy smerom ***dolu*** na smerníky odvodenej triedy („hore“ a „dolu“ je relatívne vzhľadom na typický diagram tried, kde základná trieda je hore). Pretypovanie nahor sa robí automaticky, bez nátlaku, pretože je celkom bezpečné. Pretypovanie nadol je nebezpečné, pretože počas kompilácie neexistujú informácie o skutočných typoch, takže musíme presne vedieť akého typu objekt bude. Ak pretypujeme na nesprávny typ, máme problém.

Orezanie objektu

Medzi prenosom objektov adresou a prenosom objektov hodnotou je zreteľný rozdiel z pohľadu polymorfizmu. V o všetkých doteraz uvedených príkladoch a vlastne i vo všetkých ďalších, ktoré ešte uvidíme, sa prenášajú adresy a nie hodnoty. Je to dané jednak tým, že adresy majú rovnakú veľkosť, takže prenos adresy objektu odvodenej triedy (ktorý je zvyčajne väčší) je rovnaké ako prenos adresy objektu základnej triedy (čo je zvyčajne menší objekt). Ako už bolo vysvetlené, toto je cieľom pri používaní polymorfizmu – kód, ktorý narába so základným typom môže tiež prehľadne manipulovať s objektmi odvodenej triedy.

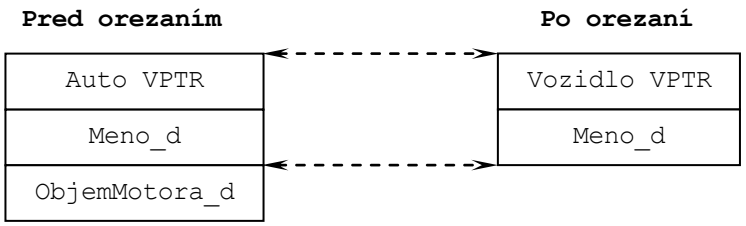
Ak pretypujeme smerom nahor na objekt namiesto na smerník alebo odkaz, stane sa niečo čo nás môže prekvapiť. objekt sa „oreže“ a jedine čo zostane, je podobjekt, zodpovedajúci cieľovému typu, na ktorý pretypovávame. V nasledujúcom príklade uvidíme, čo sa stane, ak objekt orežeme:

```
// Orezanie objektu

#include <stdio.h>
#include <string>
using namespace std;

class Vozidlo {
    string Meno_d;
  public:
    Vozidlo(const string &meno) : Meno_d(meno) {}
    virtual string Meno() const { return Meno_d; }
    virtual string Popis() const {
      return "Toto je " + Meno_d;
    }
};

class Auto : public Vozidlo {
    string ObjemMotora_d;
  public:
    Auto(const string &meno, const string &objem)
      : Vozidlo(Meno), ObjemMotora_d(objem) {}
    string Popis() const {
      return Vozidlo::Meno() + " s objemom motora " +
        ObjemMotora_d;
    }
};

void Popis(Vozidlo p) { // Oreže objekt
  printf("%s\n",p.Popis().c_str());
}

int main() {
  Vozidlo p("služobné");
  Auto d("Mercedes", "1.3");
  Popis(p);
  Popis(d);
```

```
}
```

Do funkcie **Popis** sa prenáša objekt typu **Vozidlo** *hodnotou*. Táto funkcia volá virtuálnu funkciu **Popis** pre objekt **Vozidlo**. Vo funkcii **main()** očakávame, že výsledkom prvého volania bude „Toto je služobné“ a výsledkom druhého by malo byť „Mercedes s objemom motora 1.3“ V skutočnosti obidve volania použijú verziu funkcie **Popis** zo základnej triedy.

V tomto programe nastali dve veci. Po prvé, pretože funkcia **Popis** akceptuje ***objekt*** triedy **Vozidlo** (nie smerník alebo odkaz), akékoľvek volanie funkcie **Popis** spôsobí, že do zásobníka sa uloží objekt o veľkosti rovnajúcej sa veľkosti triedy **Vozidlo** a po vykonaní funkcie sa rovnako veľká časť zásobníka i vyčistí. Znamená to, že ak do funkcie **Popis** pošleme objekt triedy, ktorá dedí triedu **Vozidlo**, kompilátor ju bude akceptovať, ale skopíruje iba časť objektu, ktorá tvorí triedu **Vozidlo**. Odvodenú časť z objektu odreže nasledovne:



Obr. 5 Prenos objektu hodnotou

Teraz by sme mohli byť zvedaví na volanie virtuálnej funkcie. Členská funkcia **Auto::Popis()** používa časť **Vozidlo** (ktorá existuje) a **Auto**, ktorá už neexistuje, pretože bola orezaná. Nuž čo sa stane zavolaním virtuálnej funkcie?

Sme zachránení od katastrofy, pretože objekt je prenášaný hodnotou. Vďaka tomu kompilátor pozná presný typ objektu, pretože odvodený objekt sa nútene stal objektom základnej triedy. Keď prenášame hodnotu, použije sa kopírovací konštruktor triedy **Vozidlo**, ktorý inicializuje VPTR na VTABLE triedy **Vozidlo** a skopíruje iba časti objektu**,** ktoré tvoria triedu **Vozidlo**. Explicitný kopírovací konštruktor nie je definovaný a preto sa použije sa vygenerovaný. Podľa akejkoľvek interpretácie sa objekt počas orezania stáva skutočným objektom triedy **Vozidlo**.

Orezanie objektu pri kopírovaní do nového objektu skutočne odstraňuje časť existujúceho objektu namiesto menenia významu adresy ako je tomu pri prenose použitím smerníka alebo odkazu. Kvôli tomuto pretypovanie objektu (objektu, nie smerníka) smerom nahor sa nerobí často. V skutočnosti je to niečo, načo by sme si mali dávať pozor a predchádzať tomu. Všimnime si, že ak by sme v tomto príklade členskú funkciu **Popis()** urobili čisto virtuálnou funkciou v základnej triede (čo nie je nerozumné, pretože v skutočnosti v základnej triede nerobí nič), potom by kompilátor orezávaniu objektov zabránil, pretože by pri prenose nedovolil „vytvoriť“ objekt základnej triedy (čo nastáva pri pretypovaní objektu smerom nahor). Toto by mohlo byť najdôležitejším prínosom čisto virtuálnych funkcií: zabraňovať orezávaniu objektov generovaním chybových oznamov kompilátora, ak sa niekto pokúsi toto urobiť.

Preťažovanie a prevažovanie

Predefinovanie preťaženej funkcie v základnej triede skrylo všetky ostatné verzie tejto funkcie, definované v základnej triede. Pri použití virtuálnych funkcií je správanie trochu odlišné. Napríklad:

```
// Virtuálne funkcie obmedzujú preťažovanie
#include <stdio.h>
#include <string>

using namespace std;

class A {
  public:
    virtual int Fun() const {
```

```
      printf("A::Fun()\n");
      return 1;
    }
    virtual void Fun(string) const {}
    virtual void g() const {}
};

class B : public A {
  public:
    void g() const {}
};

class C : public A {
  public:
    // Prevaženie virtuálnej funkcie:
    int Fun() const {
      printf("C::Fun()\n");
      return 2;
   }
};

class D : public A {
  public:
    // Návratový typ sa nemôžem meniť:
    //! void Fun() const{ printf("D::Fun()\n");;}
};

class E : public A {
  public:
    // Zmena zoznamu argumentov:
    int Fun(int) const {
      printf("E::Fun()\n");
      return 4;
    }
};

int main() {
  string s("C++");
  B b1;
  int X = b1.Fun();
  b1.F(s);
  C c1;
  X = c1.Fun();
//!  c1.F(s); // string verzia je skrytá
  E e1;
  X = e1.F(1);
//!  X = e1.Fun(); // Fun() verzia je skrytá
//!  e1.F(s); // string verzia je skrytá
  A& a1 = e1; // Pretypovanie nahor
//!  a1.F(1); // Odvodená verzia nie je k dispozícií
  a1.Fun();   // A verzia dostupná
  a1.F(s);    // A verzia dostupná
}
```

Prvou vecou, ktorú si všimneme je, že v triede **D** kompilátor nedovolí zmeniť typ návratovej prevážanej funkcie (dovolil by to, keby funkcia **Fun()** nebola virtuálna). Toto je dôležité obmedzenie, pretože kompilátor musí zabezpečovať, že môžeme polymorfne volať funkciu prostredníctvom základnej triedy, a ak základná trieda očakáva návratovú hodnotu typu **int** z funkcie **Fun()**, potom verzia odvodenej triedy musí dodržať ***kontrakt*** inak by bolo všetko narušené.

Stále platí pravidlo, uvedené už skôr: ak prevážime jednu z preťažených členských funkcií základnej triedy, ostatné preťažené verzie sa stanú v odvodenej triede skrytými. Kód vo funkcii **main()**, ktorý testuje triedu **E** ukazuje, že toto nastane dokonca i keď nová verzia funkcie **Fun()** nie je skutočne prevážená existujúcim rozhraním virtuálnej funkcie – obidve verzie funkcie **Fun()** zo základnej triedy sú skryté funkciou **F(int)**. Avšak ak pretypujeme **e1** smerom nahor na objekt triedy **A**, potom sú k dispozícii len verzie funkcie **Fun()** zo základnej triedy (pretože je to presne to, čo kontrakt základnej triedy sľubuje) a verzia z odvodenej triedy nie je dostupná (pretože nie je špecifikovaná v základnej triede).

Variantný typ návratu

Trieda **C** ukazuje, že počas prevažovania nemôžeme meniť návratový typ virtuálnej funkcie. Toto je vo všeobecnosti pravda, ale existuje špeciálny prípad, v ktorom môžeme trochu zmeniť typ návratu. Ak vraciame smerník alebo odkaz na základnú triedu, potom prevážená verzia funkcie môže vracať smerník alebo odkaz na odvodenú triedu z triedy, ktorú vracia základná trieda. Napríklad:

```
// Návrat smerníka alebo odkazu na odvodenú triedu
// počas prevažovania
#include <stdio.h>
#include <string>

using namespace std;

class PohonnaLatka {
  public:
    virtual string TypPohonnaLatka() const = 0;
};

class Vozidlo {
  public:
    virtual string Typ() const = 0;
    virtual PohonnaLatka *Pohon() = 0;
};

class Autobus : public Vozidlo {
  public:
    string Typ() const { return "Autobus"; }
    class Nafta : public PohonnaLatka {
      public:
        string TypPohonnaLatka() const {
          return "Nafta";
        }
    };
    // Pretypovanie nahor na základnú triedu:
    PohonnaLatka *Pohon() { return &bf; }
  private:
    Nafta bf;
};

class OsobneAuto : public Vozidlo {
  public:
    string Typ() const { return "OsobneAuto"; }
    class Benzin : public PohonnaLatka {
      public:
        string TypPohonnaLatka() const { return "Benzin"; }
    };
    // Návrat presného typu:
    Benzin* Pohon() { return &cf; }
  private:
    Benzin cf;
};
```

```
int main() {
  Autobus b;
  OsobneAuto c;
  Vozidlo* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    printf("%s jazdí na %s\n",
                p[i]->Typ().c_str(),
                p[i]->Pohon()->TypPohonnaLatka().c_str());
  // Môže vracať presný typ:
  OsobneAuto::Benzin *cf = c.Pohon();
  Autobus::Nafta *bf;
  // Nemôže vracať presný typ:
//!  bf = b.Pohon();
  // Pretypovanie smerom nadol:
  bf = dynamic_cast<Autobus::Nafta *>(b.Pohon());
}
```

Členská funkcia **Vozidlo::Pohon()** vracia smerník na objekt triedy **PohonnaLatka**. V triede **Autobus** je táto členská funkcia preťažená rovnako ako je v základnej triede vrátane návratového typu. Členská funkcia **Autobus::Pohon()** pretypováva **Nafta** smerom nahor na **PohonnaLatka**.

Ale v triede **OsobneAuto** je typ návratu z funkcie **Pohon()** smerník na objekt triedy **Benzin**, t.j. na typ, odvodený z triedy **PohonnaLatka**. Skutočnosť, že návratový typ je zdedený z návratového typu funkcie v základnej triede je jediným dôvodom, že toto sa skompiluje. Takýmto spôsobom je kontrakt stále splnený. Funkcia stále vracia smerník na objekt triedy **PohonnaLatka**.

Ak rozmýšľame polymorfne, toto sa nezdá potrebné. Prečo nepretypovávať smerom nahor všetky návratové typy na **PohonnaLatka ***, ako to robí t členská funkcia **Autobus::Pohon()**? Toto je zvyčajne dobré riešenie, ale rozdiel vidno vo funkcii **main()**, kde funkcia **OsobneAuto::Pohon()** vracia presný typ **PohonnaLatka**, zatiaľ čo návratová hodnota funkcie **Autobus::Pohon()** sa musí pretypovať na presný typ.

Takže mať schopnosť vracať presný typ je trochu univerzálnejšie a automatickým pretypovaním smerom nahor sa nestráca špecifická typová informácia. Avšak návrat základného typu vo všeobecnosti rieši problém, takže toto je dosť špecializovaná vlastnosť.

Virtuálne funkcie a konštruktory

Pri vytváraní objektu, ktorý obsahuje virtuálne funkcie sa musí VPTR inicializovať tak, aby ukazovala na správnu VTABLE. Toto musí byť spravené skôr, než nastane akákoľvek možnosť volania virtuálnej funkcie. Vytváranie objektu je úlohou konštruktora, a preto úlohou konštruktora je i nastavenie VPTR. Kompilátor na začiatok konštruktora v tichosti vkladá kód, ktorý inicializuje VPTR. Ak v triede explicitne nevytvoríme konštruktor, kompilátor jeden vygeneruje. Ak trieda obsahuje virtuálne funkcie, vygenerovaný konštruktor bude obsahovať inicializačný kód VPTR. Z tohto vyplýva niekoľko dôsledkov.

Prvý sa týka výkonu. Dôvodom **inline** funkcii je redukovať réžiu volania funkcie pre malé funkcie. Ak by C++ nemalo možnosť vytvárať **inline** funkcie, preprocesor by mohol využiť tzv. makrá. Avšak preprocesor nevie nič o prístupe alebo triedach, a preto by sme makrá nemohli použiť na vytváranie členských funkcií. Okrem toho pre konštruktory, kde kompilátor vkladá skrytý kód, by makro nefungovalo vôbec.

Pri naháňaní za výkonom si musíme uvedomovať, že kompilátor vkladá kód do konštruktora. Nielen že musí inicializovať VPTR, ale musí tiež testovať hodnotu **this** (pre prípad, že operátor **new** vráti nulu) a volať konštruktory základných tried. Suma sumárom, veľkosť konštruktora prevalcuje úspory, ktoré získame redukovaním réžie volania funkcie. Ak vytvoríme mnoho **inline** volaní konštruktorov, veľkosť kódu môže narásť tak, že na rýchlosti nezískame nič.

Samozrejme nebudeme odteraz robiť všetky malé konštruktory **nie-inline**, pretože sa omnoho ľahšie píšu ako **inline**. Avšak pri dolaďovaní kódu, nezabúdajme pouvažovať o odstránení **inline** konštruktorov.

Poradie volania konštruktorov

Druhý zaujímavý aspekt konštruktorov a virtuálnych funkcií sa týka poradia volania konštruktorov a spôsobu volania virtuálnych funkcií v rámci konštruktorov.

Konštruktor zdedenej triedy vždy volá konštruktory všetkých základných tried. Toto dáva zmysel, pretože konštruktor plní špeciálnu úlohu: dozerá, aby sa objekt vytvoril správne. Odvodená trieda má prístup len ku svojim vlastným členom a nie k členom zo základnej triedy. Jedine konštruktor základnej triedy dokáže správne inicializovať svoje vlastné prvky. A preto je podstatné, že sa zavolajú všetky konštruktory, inak by sa celý objekt nevytvoril správne. Toto je dôvod, prečo si kompilátor vynucuje volanie konštruktora pre každú časť odvodenej triedy. Ak explicitne neuvedieme v inicializačnom zozname konštruktora konštruktor základnej triedy, zavolá sa štandardný konštruktor. Ak neexistuje žiaden štandardný konštruktor, kompilátor sa bude sťažovať.

Poradie volaní konštruktorov je dôležité. Ak dedíme, o základnej triede vieme všetko a máme prístup ku všetkým **public** a **protected** členom základnej triedy. Znamená to, že vo vnútri odvodenej triedy musíme predpokladať, že všetky členy základnej triedy sú platné. V normálnej členskej funkcii už bola konštrukcia vykonaná, takže všetky členy objektu boli vytvorené. Jediným spôsobom ako toto zabezpečiť je, že konštruktor základnej triedy sa zavolá ako prvý. Takto sú v konštruktore odvodenej triedy všetky členy základnej triedy inicializované a môžeme ich používať. "Poznanie, že všetky členy sú platné" vo vnútri konštruktora je tiež dôvodom, že vždy, keď je to možné, mali by sme inicializovať všetky členské objekty (t.j. objekty, vložené do triedy prostredníctvom kompozície) v inicializačnom zozname konštruktora. Ak budeme dodržiavať túto prax, môžeme predpokladať, že všetky členy základnej triedy a členské objekty aktuálneho objektu boli inicializované.

Virtuálne funkcie vo vnútri konštruktorov

Hierarchia volania konštruktorov prináša zaujímavú situáciu. Čo sa stane, ak vo vnútri konštruktora zavoláme virtuálnu funkciu? Vo vnútri obyčajnej členskej funkcie si dokážeme predstaviť, čo sa stane - virtuálne volanie sa vyrieši za chodu programu, pretože objekt nemôže vedieť, či patrí tej triede, v ktorej členskej funkcii sa práve nachádza alebo nejakej inej triede, odvodenej z tejto triedy.

Avšak pre konštruktory toto nie je pravda. Ak zavoláme virtuálnu funkciu vo vnútri konštruktora, použije sa lokálna verzia funkcie. Vyplýva z toho, že virtuálny mechanizmus vo vnútri konštruktora nefunguje.

Takéto správanie dáva zmysel z dvoch dôvodov. Koncepčne, úlohou konštruktora je zabezpečiť vznik objektu (čo nie je vždy jednoduchý akt). Vo vnútri konštruktora môže byť objekt vytvorený iba čiastočne - môžeme iba vedieť, že už boli inicializované objekty základnej triedy, ale nemôžeme poznať triedy, ktoré túto triedy dedia. Volanie virtuálnej funkcie však siaha "dopredu" alebo "smerom von" v hierarchii dedičnosti. Volá funkciu v odvodenej triede. Ak by sme to spravili vo vnútri konštruktora, mohli by sme zavolať funkciu, ktorá by mohla manipulovať s členmi, ktoré ešte neboli inicializované, čo je istý recept pre katastrofu.

Druhý dôvod je mechanický. Keď sa zavolá konštruktor, jednou z prvých vecí, ktorú urobí, je inicializácia VPTR. Avšak konštruktor vie iba, že toto je "aktuálny" typ - typ, pre ktorý bol napísaný konštruktor. Kód konštruktora absolútne ignoruje, či objekt je alebo nie je v základnej alebo inej triede. Keď kompilátor generuje kód pre konštruktor, generuje kód konštruktora danej triedy, nie základnej triedy a nie odvodenej triedy (pretože trieda nemôže vedieť, kto ju dedí). Takže VPTR, ktoré používa musí byť pre VTABLE aktuálnej triedy. VPTR zostáva inicializované na túto VTABLE po zvyšok života objektu, pokiaľ toto je posledné volanie konštruktora. Ak sa po tomto zavolá konštruktor odvodenej triedy, tento konštruktor prestaví VPTR na svoju VTABLE, atď. až kým neskončí posledný konštruktor. Stav VPTR je určený konštruktorom, ktorý sa zavolá ako posledný. Toto je ďalší dôvod, prečo sa konštruktory volajú v poradí od základnej triedy po poslednú odvodenú triedu.

Avšak počas vykonávanie celej tejto série konštruktorov, každý konštruktor nastavuje VPTR na svoju vlastnú VTABLE. Ak konštruktor používa mechanizmus volania virtuálnych funkcií, uskutočňuje volania len prostredníctvom svojej vlastnej VTABLE, nie prostredníctvom odvodenej VTABLE (ako by tomu bolo po zavolaní všetkých konštruktorov). Okrem toho mnoho kompilátorov rozlišuje, že vo vnútri konštruktora je volaná virtuálna funkcia a automaticky vytvára včasnú väzbu, pretože vie, že i neskorá väzba by

produkovala iba volanie lokálnej funkcie. V každom prípade, vo vnútri konštruktora nedostaneme výsledok, ktorý by sme očakávali od volania virtuálnej funkcie.

Deštruktory a virtuálne deštruktory

Kľúčové slovo **virtual** pre konštruktory používať nemôžeme, ale deštruktory bývajú často virtuálne.

Konštruktor plní špeciálnu funkciu vytvárania objektu kúsok po kúsku, najskôr volaním základného konštruktora, a potom odvodených konštruktorov v poradí dedičnosti (súčasne tiež musí volať konštruktory členských objektov). Podobne i deštruktor plní špeciálnu úlohu: musí demontovať objekt, ktorý môže patriť do nejakej hierarchie tried. Aby to urobil, kompilátor generuje kód, ktorý volá všetky deštruktory, avšak v opačnom poradí, ako sú volané konštruktory. Znamená to, že deštruktor začína v najodvodenejšej triede a postupuje smerom ku základnej triede. Takéto správanie je bezpečné a žiaduce, pretože aktuálny deštruktor vždy vie, že členy základnej triedy sú ešte živé a aktívne. Ak potrebuje zavolať členskú funkciu základnej triedy vo vnútri deštruktora, je to bezpečné. Deštruktor teda urobí len vlastné upratovanie, potom zavolá ďalší deštruktor v poradí, ktorý zasa urobí svoje vlastné upratovanie, atď. Každý deštruktor pozná, z ktorej triedy je daná trieda odvodená, ale nie kto je z nej odvodený.

Mali by sme si zapamätať, že konštruktory a deštruktory sú jediné miesta, kde sa táto hierarchia volaní uplatňuje (správnu hierarchiu automaticky generuje kompilátor). Vo všetkých ostatných funkciách sa zavolá len ***táto*** funkcia (a nie verzia zo základnej triedy), či už je virtuálna alebo nie. Jediným spôsobom ako zavolať verzie tej istej obyčajnej funkcie (virtuálnej alebo nevirtuálnej) zo základnej triedy je, že túto funkciu explicitne zavoláme.

Normálne je činnosť deštruktora celkom postačujúca. Ale čo sa stane, ak chceme manipulovať s objektom prostredníctvom smerníka v jeho základnej triede (t.j. narábať s objektom prostredníctvom jeho generického rozhrania)? Táto aktivita je hlavný cieľom objektovo-orientovaného programovania. Problém nastáva, keď chceme vymazať (**delete**) smerník tohto typu na objekt, ktorý bol vytvorený v halde pomocou operátora **new**. Ak smerník ukazuje na objekt základnej triedy, kompilátor vie počas **delete** zavolať iba verziu deštruktora základnej triedy. Toto je ten istý problém, na riešenie ktorého boli vytvorené virtuálne funkcie. Našťastie virtuálne funkcie fungujú pre deštruktory rovnako, ako pre všetky ostatné funkcie okrem konštruktorov.

```
// Správanie virtuálneho a nie-virtuálneho deštruktora

#include <stdio.h>
using namespace std;

class A1 {
  public:
    ~A1() { printf("~A1()\n"); }
};

class B1 : public A1 {
  public:
    ~B1() { printf("~B1()\n"); }
};

class A2 {
public:
  virtual ~A2() { printf("~A2()\n"); }
};

class B2 : public A2 {
  public:
    ~B2() { printf("~B2()\n"); }
};

int main() {
```

```
  A1 *a1p = new B1; // Pretypovanie smerom nahor
  delete a1p;
  A2* a2p = new B2; // Pretypovanie smerom nahor
  delete a2p;
}
```

Keď tento program odštartujme, vidíme, že **delete a1p** volá len deštruktor základnej triedy, zatiaľ čo volanie **delete a2p** volá deštruktor odvodenej triedy, za ktorým nasleduje volanie deštruktora základnej triedy, čo je správanie, aké požadujeme. Ak by sme zabudli deštruktor deklarovať ako **virtual**, bola by to záludná chyba, pretože často by priamo neovplyvnila správanie programu, avšak potichu by zavádzala do pamäti trhliny.

Napriek tom, že deštruktor, podobne ako konštruktor, je "výnimočná" funkcia, deštruktor môže byť virtuálny, pretože objekt už pozná akého typu je (čo počas konštrukcie nevie). Akonáhle bol objekt skonštruovaný, jeho VPTR je inicializované, takže volania virtuálnych funkcií sa môžu vykonávať.

Čisto virtuálne deštruktory

I keď norma C++ čisto virtuálne deštruktory dovoľuje, pre prácu s nimi platí jedno obmedzenie: čisto-virtuálnemu deštruktoru musíme definovať telo. Zdá sa to trochu protichodné, ako môže byť virtuálna funkcia "čistá", ak potrebuje telo? Avšak ak si spomenieme, že konštruktory a deštruktory sú špeciálne operácie, dáva to väčší zmysel, obzvlášť ak si pamätáme, že všetky deštruktory v hierarchii tried sa vždy volajú. Ak by sme mohli vynechať definíciu čisto-virtuálneho deštruktora, aké telo funkcie by sa zavolalo počas deštrukcie? A tak je absolútne nevyhnutné, aby si kompilátor a linker vynútil existenciu tela funkcie pre čisto virtuálny deštruktor.

Ak je deštruktor čisto-virtuálny, ale musí mať telo, aký má význam? Jediným rozdielom medzi čisto-virtuálnym a nie-virtuálnym deštruktorom je, že čisto-virtuálny deštruktor spôsobí, že základná trieda bude abstraktná, takže nebudeme môcť vytvárať objekt základnej triedy (hoci toto platí, i keď akákoľvek iná funkcia základnej triedy je čisto virtuálna).

Avšak keď dedíme triedu, ktorá obsahuje čisto virtuálny deštruktor, je to trochu komplikovanejšie. Na rozdiel od všetkých ostatných virtuálnych funkcií, v odvodenej triede nemusíme definíciu čisto virtuálneho deštruktora zabezpečovať. Napríklad:

```
// Čisto virtuálny deštruktor

class A {
  public:
    virtual ~A() = 0;
};

A::~A() {}

class B : public A {};
// Prevádzanie deštruktora nie je potrebné

int main() { B d; }
```

Normálna čisto virtuálna členská funkcia v základnej triede by spôsobila, že odvodená trieda by bola tiež abstraktná, ak (a všetky ostatné čisto virtuálne funkcie) by sme ju v odvodenej triede nedefinovali. Ale tu je to iné. Nezabúdajme však, že kompilátor ***automaticky*** vytvára definíciu deštruktora pre každú triedu, ak nevytvoríme vlastný deštruktor. A toto nastáva i vo vyššie uvedenom príklade - konštruktor základnej triedy je v tichosti prevádzený, definíciu zabezpečí kompilátor, a trieda **B** nie je abstraktná.

V čom je teda podstata čisto virtuálneho deštruktora? Na rozdiel od obyčajnej čisto virtuálnej funkcie musíme definovať telo funkcie. V odvodenej triede definíciu zabezpečovať nemusíme, pretože kompilátor vygeneruje deštruktor za nás. Teda aký je rozdiel medzi regulárnym virtuálnym a čisto-virtuálnym deštruktorom?

Jediný rozdiel spočíva v tom, že ak máme triedu s jedinou čisto-virtuálnou funkciou: deštruktorom. V takomto prípade jediným dôsledkom virtuálnej čistoty deštruktora je, že zabráni vytvárať inštancie základnej triedy. Ak základná trieda obsahuje nejaké ďalšie čisto virtuálne funkcie, tieto zabránia vytvoreniu inštancie základnej triedy, ale ak žiadne iné virtuálne funkcie, potom nám abstraktnosť zabezpečí čisto virtuálny deštruktor. Takže zatiaľ čo deklarovanie virtuálneho deštruktora je podstatné, či je čisto virtuálny alebo nie, nie je až také dôležité.

Ak odštartujeme nasledovný príklad, uvidíme, že sa zavolá telo čisto virtuálnej funkcie po verzií odvodenej triedy, rovnako ako to platí pre akýkoľvek iný deštruktor:

```
// Čisto virtuálny deštruktor musí mať definované telo
#include <iostream>
using namespace std;

class Vozidlo {
  public:
    virtual ~Vozidlo() = 0;
};

Vozidlo::~Vozidlo() {
  printf("~Vozidlo()\n");
}

class Auto : public Vozidlo {
public:
  ~Auto() {
    printf("~Auto()\n");
  }
};

int main() {
  Vozidlo *p = new Auto; // Pretypovanie smerom nahor
  delete p;              // Volanie virtuálneho deštruktora
} ///:~
```

Virtualita v deštruktoroch

Počas deštrukcie objektu nastáva niečo, čo na prvý pohľad nemusíme očakávať. Ak sme vo vnútri obyčajnej členskej funkcie a zavoláme virtuálnu funkciu, táto funkcia sa zavolá použitím mechanizmu neskorej väzby. Toto neplatí pre deštruktory, či už virtuálne alebo nie-virtuálne. Vo vnútri deštruktora sa volá iba lokálna verzia členskej funkcie - virtuálny mechanizmus sa ignoruje.

```
// Volanie virtuálnych členských funkcií v rámci deštruktora

#include <stdio.h>
using namespace std;

class A {
public:
  virtual ~A() {
    printf("A()\n");
    Fun();
  }
  virtual void Fun() { printf("A::Fun()\n"); }
};

class B : public A {
  public:
    ~B() { printf("~B()\n"); }
    void Fun() { printf("B::Fun()\n"); }
```

```
};

int main() {
  A* a1 = new B; // Pretypovanie smerom nahor
  delete a1;
}
```

Počas volania deštruktora sa nezavolá **B::Fun()**, i keď členská funkcia **Fun()** je virtuálna.

Prečo? Nuž predpokladajme, že by sa vo vnútri konštruktora použil virtuálny mechanizmus. Potom by bolo možné, aby virtuálne volanie poznalo funkciu, ktorá je "vzdialenejšia" (viac odvodená) v hierarchii dedičnosti, než je aktuálny deštruktor. Ale deštruktory sa volajú "zdola nahor" (od odvodeného deštruktora po základný deštruktor), takže skutočne volaná funkcia by sa spoliehala na tie časti objektu, ktoré ***už boli zrušené***. Kompilátor rieši tento problém počas kompilácie volaním "lokálnej" verzie funkcie. Toto isté platí i pre konštruktor, ale v prípade konštruktora nie je známa typová informácia, zatiaľ čo v prípade deštruktora táto informácia (t.j. VPTR) k dispozícií je, ale nie je spoľahlivá.

Pretypovanie smerom nadol

Keďže existuje niečo také ako pretypovanie smerom nahor (upcasting) - presun hore v hierarchii dedičnosti - malo by existovať tiež *pretypovanie smerom nadol* (*downcasting*) - presun nadol v hierarchii. Pretypovanie smerom nahor je jednoduché, pretože sa v hierarchii dedičnosti posúvame nahor, triedy sa približujú ku všeobecnejším triedam. Znamená to, že keď pretypovávame smerom nahor, sme odvodení z triedy predka (zvyčajne len jednej, okrem prípadu viacnásobnej dedičnosti), ale keď pretypovávame smerom dolu, existuje zvyčajne niekoľko možností, na ktoré môžeme pretypovávať. Konkrétnejšie trieda **B** je typom triedy **A** (pretypovanie nahor), ale ak sa pokúsime pretypovať nadol trieda **A**, môže to byť **B**, **C**, **D**, atď. Takže problém spočíva v určení bezpečného spôsobu pretypovania smerom nadol. (Omnoho dôležitejšia je však otázka, prečo pretypovávame smerom nadol).

C++ poskytuje špeciálne tzv. ***explicitné pretypovanie***, nazývané **dynamic_cast**, ktoré je typovo-bezpečnou pretypovacou operáciou smerom nadol. Keď na pretypovanie konkrétneho typu smerom nadol použijeme **dynamic_cast**, návratovou hodnotou bude smerník na požadovaný typ len v prípade, že pretypovanie je v poriadku a úspešné, inak sa vráti nula, ktorá indikuje, že typ nebol korektný. Napríklad:

```
// Pretypovanie smerom nadol
#include <iostream>
using namespace std;

class Vozidlo {
  public:
    virtual ~Vozidlo(){}
};

class Auto : public Vozidlo {
};

class OsobneAuto : public Vozidlo {
};

int main() {
  Vozidlo *b = new OsobneAuto; // Pretypovanie smerom nahor
  // Pokus o pretypovanie na Auto*
  Auto *d1 = dynamic_cast<Auto*>(b);
  // Pokus o pretypovanie na OsobneAuto*:
  OsobneAuto *d2 = dynamic_cast<OsobneAuto*>(b);
  printf("d1 = %ld\n",(long)d1);
  printf("d2 = %ld\n",(long)d2);
}
```

Ak použijeme **dynamic_cast**, musíme pracovať so skutočnou polymorfnú hierarchiou - využívajúcou virtuálne funkcie - pretože na určenie skutočného typu používa **dynamic_cast** informácie, uložené vo VTABLE. Stačí ak základná trieda bude obsahovať virtuálny deštruktor. Vo funkcii **main()** sa smerník na objekt triedy **OsobneAuto** pretypováva smerom nahor na objekt triedy **Vozidlo** a potom sa pokúša o pretypovanie smerom nadol smerníkov **Auto** a **OsobneAuto**. Obidva smerníky sa vypisujú a vidíme, že nesprávne pretypovanie nadol dáva nulu. Samozrejme pri pretypovaní nadol sme my zodpovední za kontrolu toho, že pretypovanie smerom nadol nám nedá nulu. Tiež by sme nemali predpokladať, že smerník bude presne taký istý, pretože občas sa počas pretypovania smerom nahor alebo nadol smerník prispôsobuje smerníka (najmä pri viacnásobnej dedičnosti).

Vykonávanie **dynamic_cast** si vyžaduje určitú réžiu, nie veľkú, ale ak robíme veľa **dynamic_cast**-ingov (v takomto prípade by sme sa mali dobre pozrieť na návrh programu), môže to z pohľadu výkonu programu hrať významnú úlohu. V prípadoch ak poznáme nejaké informácie, ktoré nám dovoľujú s istotou povedať s akým typom pracujeme, extra réžia použitia **dynamic_cast** nie je potrebná a namiesto nej môžeme použiť **static_cast**. Mohlo by to fungovať nasledovne:

```
// Navigácia v hierarchii tried pomocou static_cast
#include <stdio.h>
#include <typeinfo>
using namespace std;

class A { public: virtual ~A() {}; };
class B : public A {};
class C : public A {};
class D {};

int main() {
  B c;
  A* s = &c; // Pretypovanie smerom nahor: OK
  // Explicitnejšie ale nie je nevyhnutné
  s = static_cast<A*>(&c);
  B* cp = 0;
  C* sp = 0;
  // Statická navigácia v hierarchii tried
  // vyžaduje ďalšie informácie o type:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<B*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<C*>(s);
  if(cp != 0)
    printf("Toto je trieda B!\n");
  if(sp != 0)
    printf("Toto je trieda C!\n");
  // Statická navigácia je určená len zvýšenie efektívnosti;
  // dynamic_cast je vždy bezpečnejšia. Avšak:
  // D* op = static_cast<D*>(s);
  // Obyčajne generuje chybový oznam, avšak
  D* op2 = (D*)s;
  // nie
}
```

V tomto programe je použitá nová vlastnosť, o ktorej sme ešte nehovorili: ***C++ mechanizmus typovej informácie za chodu programu*** (*run-time type information*-RTTI). RTTI dovoľuje zisťovať typové informácie, ktoré boli stratené počas pretypovania smerom nahor. Jednou z foriem RTTI je i **dynamic_cast**. V našom príklade je na identifikáciu typov smerníkov použité kľúčové slovo **typeid** (deklarované v hlavičkovom súbore **typeinfo**). Vidíme, že typ pretypovaného smerníka na objekt triedy **A** smerom nahor je úspešne porovnaný so smerníkom na objekt triedy **B** a smerníkom na objekt triedy **C**, aby sa zistila zhoda.

Vytvára sa objekt triedy **B** a adresa sa pretypováva smerom nahor na smerník na objekt triedy **A**. Druhá verzia výrazu ukazuje ako môžeme použiť **static_cast**, aby pretypovanie smerom nahor bolo explicitnejšie. Avšak pretože pretypovanie smerom nahor je vždy bezpečné a je to bežná vec, explicitné pretypovanie smerom nahor je zbytočné a nepotrebné.

Najskôr sa RTTI používa na určenie typu a potom je na realizáciu pretypovania smerom nadol použité kľúčové slovo **static_cast**. Všimnime si, že v tomto návrhu je proces rovnako efektívny ako pri použití **dynamic_cast** a klientsky programátor musí robiť rovnaké testovanie, aby zistil, či pretypovanie bolo úspešné. Zvyčajne pre použitie **static_cast** namiesto **dynamic_cast** vyžadujeme situáciu, ktorá je deterministickejšia, než vyššie uvedený príklad, (a opäť by sme mali pozorne prezrieť svoj návrh pred použitím **dynamic_cast**).

Ak hierarchia tried nemá virtuálne funkcie (čo je sporný návrh) alebo máme iné informácie, ktoré nám dovoľujú bezpečne pretypovávať smerom nadol, je o čosi rýchlejšie pretypovávať dolu staticky, než používať **dynamic_cast**. Okrem toho **static_cast** nám nedovoľuje pretypovávať mimo hierarchiu, ako je to možné v tradičnom pretypovávaní, čim je bezpečnejšie. Avšak staticky navigované hierarchie tried sú vždy riskantné a pokiaľ nie sme v špeciálnej situácií mali by sme používať **dynamic_cast**.

Zhrnutie

Polymorfizmus, implementovaný v C++ pomocou virtuálnych funkcií, reprezentuje pojem "odlišné formy". V objektovo-orientovanom programovaní máme rovnaký výraz (spoločné rozhranie v základnej triede) a odlišné formy, využívajúce tento výraz: odlišné verzie virtuálnych funkcií.

Videli sme, že je nemožné pochopiť alebo dokonca vytvoriť príklad polymorfizmu bez použitia dátovej abstrakcie a dedičnosti. Polymorfizmus je vlastnosť, na ktorú sa nedá pozerať izolovane (ako napríklad na príkazy **const** alebo **switch**), ale funguje iba v súlade "s veľkým obrazom" triednych vzťahov. Ľudia sú často zmätení inými, neobjektovými vlastnosťami C++, ako je napríklad preťažovanie a implicitné argumenty, ktoré sú občas prezentované ako objektovo-orientované. Nedajme sa oklamať - ak neexistuje neskorá väzba, nie je to polymorfizmus.

Aby sme mohli v našich programoch používať polymorfizmus - a tým i objektovo-orientované technológie - efektívne, musíme rozšíriť svoj pohľad na programovanie tak, aby zahrňoval nielen členy a správy jednotlivých tried, ale tiež spoločenstvo tried a ich vzájomných vzťahov. Hoci toto si vyžaduje značne úsilie, je to dôstojný boj, pretože výsledkom je rýchlejší vývoj programu, lepšia organizácia programu, rozširovateľné programy a ľahšia údržba kódu.

Polymorfizmus završuje objektovo-orientované vlastnosti jazyka, ale súčasťou C++ sú ešte dve významné črty: šablóny a spracovanie výnimiek. Tieto vlastnosti poskytujú rovnaké zvýšenie programovacej kapacity ako každá iná objektovo-orientovaná vlastnosť: typovanie abstraktných dát, dedičnosť a polymorfizmus.