Obsah

Dedičnosť a kompozícia

Jednou z najzaujímavejších vlastností C++ je znovupoužitie kódu. Avšak ak chceme pracovať efektívne, nestačí iba kód skopírovať a trochu ho zmeniť. Toto je prístup jazyka C (resp. procedurálneho programovania) a nefunguje veľmi dobre. V C++ sa znovupoužitie kódu, ako v podstate všetko, točí okolo triedy. Kód použijeme znova vytvorením nových tried, avšak namiesto toho aby, sme ich vytvorili z ničoho, použijeme už existujúce triedy, ktoré už niekto vybudoval a odladil.

Trik spočíva v použití tried bez ***znečistenia*** už existujúceho kódu. Na dosiahnutie tohto cieľa existujú dva postupy.

Prvý postup je celkom zrozumiteľný a už sme o ňom hovorili: jednoducho vo vnútri novej triedy deklarujeme objekty už existujúcich tried. Toto sa nazýva ***kompozícia***[1], pretože nová trieda sa zostavuje z objektov existujúcich tried.

Druhý prístup je pre procedurálnych programátorov trochu záhadnejší. Novú triedu vytvárame ako ***typ*** už existujúcej triedy. Doslova zoberieme „šablónu“ existujúcej triedy a pridáme k nej kód bez toho aby sme už existujúcu triedu modifikovali. Tento magický postup sa nazýva ***dedičnosť*** a väčšinu práce robí za nás kompilátor. Dedičnosť je jedným zo základných princípov objektovo-orientovaného programovania.

Väčšiu časť syntaxe a správania majú kompozícia i dedičnosť podobné (čo dáva zmysel: obidva predstavujú spôsoby vytvárania nových typov z už existujúcich typov).

Syntax kompozície

V skutočnosti sme kompozíciu pri vytváraní tried už použili, keď sme triedy skladali zo zabudovaných typov. Takmer rovnako jednoducho sa kompozícia používa s užívateľsky definovanými typmi.

Zoberme napríklad triedu:

```
class X {
    int i_d;
  public:
    X() : i_d(0) {};
    void Nastav(int ii) { i_d = ii; }
    int  Daj() const { return i_d; }
    int  Nasob() { return i_d = i_d * 47; }
};
```

Dátový člen **i_d** je v tejto triede deklarovaný ako **private**, takže je absolútne bezpečné vložiť objekt triedy **X** ako **public** objekt do novej triedy. Týmto je dané i rozhranie:

```
// Použitie kompozície
class Y {
    int i_d;
  public:
    X x;                // Vložený objekt triedy X
    Y(): i_d (0) {};
    void Fun(int ii) { i_d = ii; }
    int  GFun() const { return i_d; }
};
```

[1] Kompozícia je jedným z druhov tzv. *asociácie*. O asociácií hovoríme tried, keď navzájom nejakým spôsobom triedy komunikujú (sú prepojené). O kompozícií hovoríme vtedy, keď je nejaký objekt neoddeliteľnou súčasťou inej triedy, t.j. vzniká a zaniká so vznikom a zánikom triedy, ktorej je súčasťou. Ďalším druhom asociácie je tzv. *agregácia*, kedy je objekt súčasťou nejakej inej triedy avšak táto trieda ho môže poskytnúť inej triede.

```
int main()
{
  Y y;
  y.Fun(47);
  y.x.Nastav(37); // Prístup ku vloženému objektu
}
```

Prístup k členským funkciám vloženého objektu (ktorý sa nazýva i podobjekt alebo subobjekt) jednoducho vyžaduje iba použitie mena dátového člena (v našom prípade **x**).

Avšak bežnejšie je, že vložené objekty sú **private**, čím sa stávajú súčasťou implementácie triedy (čo nám v prípade potreby umožňuje meniť implementáciu). Potom **public** metódy rozhrania novej triedy používajú rozhranie vloženého objektu, avšak nemusia nevyhnutne kopírovať presne rozhranie vloženého objektu:

```
// private vložený objekt
class Y {
  int i_d;
  X x;      // Vložený objekt
public:
  Y() : i_d(0) {};
  void Fun(int ii) {
    i_d = ii;
    x.Nastav(ii);
  };
  int  GFun() const { return i_d * x.Daj(); };
  void Nasob() { x.Nasob(); };
};

int main() {
  Y y;
  y.f(47);
  y.Nasob();
}
```

Metóda **Nasob()** sa volá prostredníctvom rozhrania novej triedy, ale ostatné metódy triedy **X** sú použité len v rámci implementácie metód triedy **Y**.

Syntax dedičnosti

Syntax kompozície je zrejmá, ale dedičnosť, to je niečo nové. Keď dedíme, hovoríme, že „***nová trieda je podobná starej triede***“. Toto tvrdenie sa v názve triedy interpretuje tak, že pred úvodnú zloženú zátvorku tela novej triedy dáme dvojbodku a meno základnej triedy (alebo v prípade viacnásobnej dedičnosti mená základných tried, oddelených čiarkami). Keď to spravíme, automaticky získame všetky dátové členy a členské funkcie základnej triedy:

```
// Jednoduchá dedičnosť
#include <iostream>

class Y : public X {
  int i_d;           // Iné i_d než z triedy X
public:
  Y() : i_d(0) {};
  int Modifikuj() {
    i_d = Nasob();  // Volanie metódy predka s iným názvom
    return i_d;
  }
  void Nastav(int ii) {
    i_d = ii;
    X::Nastav(ii);   // Volanie metódy predka s rovnakým názvom
```

```
  }
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = " << sizeof(Y) << endl;

  Y d;
  d.Modifikuj();
  // Volanie metód z rozhrania triedy X
  d.Daj();
  d.Nasob();
  // Predefinovaná metóda skrýva metódu základnej triedy:
  d.Nastav(12);
}
```

Vidíme, že trieda **Y** je potomkom triedy **X**, čo znamená, že trieda **Y** bude obsahovať všetky dátové prvky triedy **X** a zároveň i všetky jej členské funkcie. V skutočnosti trieda **Y** obsahuje podobjekt **X** rovnako ako keď sme ju vytvorili kompozíciou vložením členského objektu triedy **X** do vnútra triedy **Y** namiesto dedenia z triedy **X**. V obidvoch prípadoch, členský objekt i základnú triedu nazývame podobjektmi.

Všetky **private** prvky triedy **X** budú **private** i v triede **Y .** Skutočnosť, že trieda **Y** dedí triedu **X** neznamená, že trieda **Y** môže narušiť ochranný mechanizmus triedy **X**. Prvky deklarované v triede **X** ako **private** sú síce súčasťou triedy **Y**, zaberajú priestor, avšak trieda **Y** k nim nemá priamy prístup.

Vo funkcii **main()** vidíme, že dátové členy triedy **Y** sú pridané k dátovým členom triedy **X** (napriek tomu, že majú rovnaký názov) , pretože výsledok **sizeof(Y)** je dvakrát tak veľký ako **sizeof(X)**.

Všimnime si, že pred menom základnej triedy je kľúčové slovo **public**. Podobne ako pre kompozíciu i pre dedičnosť je všetko implicitne **private**. Ak by základná trieda nebola uvedená kľúčovým slovom **public**, znamenalo by to, že všetky **public** členy základnej triedy by boli v odvodenej triede **private**. Toto je však to, čo takmer nikdy nechceme. Žiadúce je uchovať všetky **public** členy základnej triedy ako **public** i v odvodenej triede. A práve preto sa v deklarácií dedenia použitíva kľúčové slovo **public**.

V metóde **Modifikuj()** je volaná metóda **Nasob()** základnej triedy. Odvodená trieda má priamy prístup ku všetkým **public** metódam základnej triedy.

Funkcia **Nastav()** v odvodenej triede predefinováva funkciu **Nastav()** základnej triedy. Znamená to, že ak zavoláme funkciu **Daj()** a **Nasob()** pre objekt triedy **Y**, dostaneme verzie tejto funkcie zo základnej triedy (toto nastáva vo vnútri funkcie **main()**). Ale ak zavoláme **Nastav()** pre objekt triedy **Y**, potom dostaneme predefinovanú verziu. Znamená to, že ak sa nám nepáči funkcia, ktorú zdedíme, môžeme zmeniť jej činnosť. (Samozrejme môžeme dodať úplne nové funkcie napríklad **Modifikuj()**.

Avšak ak aj funkciu predefinujeme, stále môžeme volať verziu funkcie zo základnej triedy. Ak vo vnútri metódy **Nastav()** jednoducho zavoláme **Nastav()**, potom sa zavolá lokálna verzia funkcie - rekurzívne volanie funkcie. Ak chceme zavolať verziu funkcie zo základnej triedy, musíme explicitne pred meno funkcie zadať meno základnej triedy, oddelené operátorom rozsahu.

Trieda, ktorá dedí vlastnosti inej triedy sa nazýva ***odvodená trieda***. Odvodená trieda môže byť zasa základnou (bázovou) triedou inej triedy.

Dedičnosť a Inicializačný zoznam konštruktora

Už sme hovorili o tom, aké dôležité je v C++ zabezpečiť správnu inicializáciu. Medzi kompozíciou a dedičnosťou nie je rozdiel. Keď sa vytvára objekt, kompilátor zabezpečuje volanie konštruktorov všetkých podobjektov. V našich príkladoch majú podobjekty štandardné konštruktory, a práve tieto bude kompilátor volať. Avšak čo sa stane, ak podobjekt nemá štandardný konštruktor alebo ak chceme zmeniť implicitný argument v konštruktore? Toto je problém, pretože konštruktor novej triedy nemá právo pristupovať k **private** dátovým členom podobjektu, takže ich nemôže inicializovať priamo.

Riešenie je jednoduché: prostredníctvom inicializačnej syntaxe **zavoláme konštruktor podobjektu**. Formát inicializačného zoznamu kopíruje postup dedenia. Pri dedení dávame základnú triedu za dvojbodku pred otváraciu zloženú zátvorku tela triedy. V inicializačnom zozname konštruktora dávame volania konštruktorov podobjektov za zoznam argumentov konštruktora a dvojbodku, avšak pred otváraciu zloženú zátvorku tela funkcie. Pre triedu **B**, zdedenú z **A**, by to mohlo vypadať nasledovne:

```
B::B(int i) : A(i) { // ...
```

ak trieda **A** má konštruktor, ktorý akceptuje jeden parameter typu **int**.

Inicializácia členských objektov

Túto istú syntax používame i v kompozícii na inicializáciu členských objektov. Pri kompozícií používame namiesto mien tried mená objektov. Ak máme viac ako jedno volanie v inicializačnom zozname konštruktora, tieto volania oddeľujeme čiarkami:

```
C::C(int i) : A(i), m(i+1) { // ...
```

Takto vypadá začiatok konštruktora triedy **C**, ktorá dedí triedu **A** a obsahuje členský objekt, nazvaný **m**. Všimnime si, že v inicializačnom zozname môžeme vidieť typ základnej triedy, ale iba identifikátor členského objektu.

Zabudované typy v inicializačnom zozname

Inicializačný zoznam konštruktora dovoľuje explicitne volať konštruktory členských objektov. V skutočnosti neexistuje žiaden iný spôsob ako tieto konštruktory volať. Hlavná myšlienka spočíva v tom, že konštruktory predkov a členských objektov sa volajú skôr, než sa začne vykonávať vlastné telo konštruktora novej triedy. Takto je zabezpečené, že akékoľvek volanie členskej funkcie podobjektu sa vykoná už s inicializovanými objektmi. Neexistuje spôsob dostať sa k otváracej zátvorke konštruktora bez volania *nejakého* konštruktora všetkých členských objektov a základnej triedy, i keď by kompilátor mal urobiť skryté volanie štandardného konštruktora. Toto je ďalšie opatrenie, ktorým C++ zaručuje, že žiaden objekt (alebo časť objektu) nevstúpi na štartovaciu čiaru bez zavolania konštruktora.

Myšlienka, že všetky členské objekty sa inicializujú pred dosiahnutím otváracej zátvorky konštruktora je zároveň i výhodná pomôcka. Akonáhle narazíme na otváraciu zátvorku konštruktora, môžeme predpokladať, že všetky podobjekty sú správne inicializované, a môžeme sa sústrediť na špecifické úlohy, ktoré chceme v konštruktore vykonať. Avšak čo členské objekty zabudovaných typov, ktoré *nemajú* konštruktory?

Aby syntax bola konzistentná, môžeme so zabudovanými typmi zaobchádzať tak, ako keby mali jeden konštruktor, akceptujúci jeden argument: premennú rovnakého typu ako je inicializovaná premenná. Takže môžeme napísať:

```
// Pseudo-konštruktory
class X {
    int   i_d;
    float f_d;
    char  c_d;
    char* s_d;
  public:
      X() : i_d(7), f_d(1.4), c_d('x'), s_d("ahoj") {};
};

int main()
{
  X x;
  int i(100);            // Pooužitý pre obyčajnú definíciu
  int* ip = new int(47);
```

```
}
```

Úlohou ***volaní*** týchto ***pseudo-konštruktorov*** je vykonať jednoduché priradenie. Je to výhodná technika a dobrý štýl kódovania, a preto sa veľmi často používa.

Dokonca sa syntax pseudo-konštruktora dá použiť pri deklarovaní premennej zabudovaného typu mimo triedu:

```
int i(100);
int* ip = new int(47);
```

Toto robí zabudované typy tak trochu objektmi. Nezabúdajme však, že toto nie sú skutočné konštruktory. Ak explicitne nezavoláme pseudo-konštruktor, neurobí sa žiadna inicializácia.

Spojenie kompozície a dedičnosti

Samozrejme kompozíciu a dedičnosť môžeme používať spoločne. Nasledujúci príklad vytvára zložitejšiu triedu, používajúc obidve vlastnosti:

```
// Dedičnosť & kompozícia
class A {
    int i_d;
  public:
    A(int ii) : i_d(ii) {};
    ~A() {};
    void Fun() const {};
};

class B {
    int i_d;
  public:
    B(int ii) : i_d(ii) {};
    ~B() {};
    void Fun() const {};
};

class C : public B {
    A a;
  public:
    C(int ii) : B(ii), a(ii) {};
    ~C() {}; // Volá ~A() a ~B()
    void Fun() const {  // Predefinovanie
      a.Fun();
      B::Fun();
    }
};

int main()
{
  C c(47);
}
```

Trieda **C** dedí triedu **B** a má členský objekt **a** ("je zložená z") triedy **A**. Inicializačný zoznam konštruktora obsahuje volania konštruktora základnej triedy i konštruktora členského objektu.

Metóda **C::Fun()** predefinováva zdedenú metódu **B::Fun()**, pričom volá verziu zo základnej triedy. Naviac volá i **a.Fun()**. Všimnime si, že o predefinovaní metódy môžeme hovoriť iba v prípade dedičnosti, v prípade členského objektu môžeme iba s **public** rozhraním objektu manipulovať, ale nie meniť ho. Okrem toho volanie **Fun()** pre objekt triedy **C** by nezavolalo **a.Fun()**, ak by **C::Fun()** nebolo definované, zavolalo by sa **B::Fun()**.

Automatické volanie deštruktora

Hoci v inicializačnom zozname sme často nútení volať konštruktor explicitne, deštruktor volať explicitne nikdy nepotrebujeme, pretože každá trieda má iba jeden deštruktor, ktorý nemá žiaden argument. Zavolanie všetkých deštruktorov zabezpečuje kompilátor, ktorý zavolá všetky deštruktory v celej hierarchii, počínajúc konštruktorom poslednej odvodenej triedy smerom ku koreňu.

Je nutné zdôrazniť, že konštruktory a deštruktory sú dosť nezvyčajné v tom, že každý v hierarchii sa volá, zatiaľ čo normálne členské funkcie sa volajú len vtedy, keď sa funkcia zavolá explicitne, ale nie vždy sa volá verzia základnej triedy. Ak chceme volať tiež verziu členskej funkcie základnej triedy, ktorú prekryjeme, musíme to urobiť explicitne.

Poradie volania konštruktorov a deštruktorov

Potrebné je poznať i poradie volania konštruktorov a deštruktorov, keď objekt vlastní mnoho podobjektov. Nasledovný príklad ukazuje ako to funguje:

```
// Poradie volaní konštruktorov/deštruktorov
#include <fstream.h>

ofstream protokol("poradie.txt");

int poradie=0;

class Zaklad {
    int Poradie_d;
  public:
    Zaklad(int) : Poradie_d(poradie++) {
      protokol << Poradie_d << "." <<
                        " Zaklad - konštruktor\n";
    };
    ~Zaklad() { protokol << Poradie_d << "." <<
                " Zaklad -  deštruktor\n"; };
};

class Clen1 {
    int Poradie_d;
  public:
    Clen1(int) : Poradie_d(poradie++) {
      protokol << Poradie_d << "." << " Clen1 - konštruktor\n";
    };
    ~Clen1() { protokol << Poradie_d << "." <<
                                " Clen1 -deštruktor\n"; };
};

class Clen2 {
    int Poradie_d;
  public:
    Clen2(int) : Poradie_d(poradie++) {
      protokol << Poradie_d << "." << " Clen2 - konštruktor\n";
    };
    ~Clen2() { protokol << Poradie_d << "." <<
                              " Clen1 - deštruktor\n"; };
};

class Clen3 {
    int Poradie_d;
  public:
    Clen3(int) : Poradie_d(poradie++) {
```

```
      protokol << Poradie_d << "." <<
                  " Clen3 - konštruktor\n";
    };
    ~Clen3() { protokol << Poradie_d << "." <<
                           " Clen3 - deštruktor\n"; };
};

class Clen4 {
    int Poradie_d;
  public:
    Clen4(int) : Poradie_d(poradie++) {
      protokol << Poradie_d << "." << " Clen4 - konštruktor\n";
    };
    ~Clen4() { protokol << Poradie_d << "." <<
                 " Clen4 - deštruktor\n"; };
};

class Potomok1 : public Zaklad {
    Clen1 c1_d;
    Clen2 c2_d;
    int Poradie_d;
  public:
    Potomok1(int) :
c2_d(1),c1_d(2),Zaklad(3),Poradie_d(poradie++) {
      protokol << Poradie_d << "." <<
                " Potomok1 - konštruktor\n";
    }
    ~Potomok1() {
      protokol << Poradie_d << "." <<
                    " Potomok1 - deštruktor\n ";
    };
};

class Potomok2 : public Potomok1 {
    Clen3 c3_d;
    Clen4 c4_d;
    int Poradie_d;
  public:
    Potomok2() :
c3_d(1),Potomok1(2),c4_d(3),Poradie_d(poradie++) {
      protokol << Poradie_d << "." <<
                  " Potomok2 - konštruktor\n";
    };
    ~Potomok2() {
      protokol << Poradie_d << "." <<
                    " Potomok2 - deštruktor\n";
    };
};

int main() {
  Potomok2 d2;
  protokol << "------------------------";
  return 0;
}
```

Najskôr sa vytvorí objekt triedy **ofstream** (tzv. výstupný prúd – podrobne inokedy), aby sa všetok výstup mohol posielať do protokolovacieho súboru. Potom sa deklaruje niekoľko tried, ktoré sú použité na vytvorenie hierarchie dedičnosti a kompozície. Každý konštruktor a deštruktor zapisuje do protokolovacieho súboru. Všimnime si, že konštruktory nie sú štandardné; každý má parameter typu **int**. Samotný parameter nemá identifikátor, jeho účelom je iba nanútiť explicitné volanie konštruktorov v inicializačnom zozname. (Eliminujeme tým varovné oznamy kompilátora).

Výstup programu je nasledovný:

```
0. Zaklad - konštruktor
1. Clen1 - konštruktor
2. Clen2 - konštruktor
3. Potomok1 - konštruktor
4. Clen3 - konštruktor
5. Clen4 - konštruktor
6. Potomok2 konštruktor
-------------------------
6. Potomok2 deštruktor
5. Clen4 - deštruktor
4. Clen3 - deštruktor
3. Potomok1 - deštruktor
2. Clen2 - deštruktor
1. Clen1 - deštruktor
0. Zaklad - deštruktor
```

Vidíme, že konštrukcia začína v samom koreni hierarchie tried, t.j. najskôr sa volá konštruktor základnej triedy, za ktorým nasledujú konštruktory členských objektov. Deštruktory sa volajú v presne opačnom poradí volania konštruktorov - je to dôležité kvôli potencionálnym závislostiam (v konštruktore alebo deštruktore odvodenej triedy musíme byť schopní predpokladať, že máme k dispozícií podobjekty základnej triedy a že už boli vytvorené resp. ešte neboli zrušené).

Je tiež zaujímavé, že poradie volaní konštruktorov členských objektov nie je vôbec ovplyvnené poradím volaní v inicializačnom zozname konštruktora. Poradie je určené poradím, v akom sú členské objekty deklarované v triede. Ak by sme mohli meniť poradie volania konštruktorov prostredníctvom inicializačného zoznamu konštruktora, mohli by sme v dvoch rôznych konštruktoroch dostať dve rôzne postupnosti volaní, ale úbohý deštruktor by potom nevedel aké je správne opačné poradie volaní deštrukcií a mohlo by to skončiť problémami závislostí.

Skrývanie mien

Ak dedíme triedu a vytvárame novú definíciu pre jednu z jej členských funkcií, môžu nastať dve možnosti. Po prvé, v definícií odvodenej triedy vytvoríme členskú funkciu s presne rovnakou signatúrou a typom návratovej hodnoty, aká je v definícií základnej triedy. Toto sa pre obyčajné členské funkcie nazýva ***predefinovanie***, a ***prekrývanie*** (overriding), keď je členská funkcia základnej triedy virtuálnou funkciou (o nich neskôr). Avšak čo sa stane, ak v odvodenej triede zmeníme zoznam argumentov členskej funkcie alebo typ návratovej hodnoty? Tu je príklad:

```
// Skrývanie preťažených mien počas dedičnosti
#include <iostream>
#include <string>

using namespace std;

class Zaklad {
  public:
    int Fun() const {
      cout << "Zaklad::Fun()\n";
      return 1;
    };
    int Fun(string) const { return 1; };
    void GFun() {};
};

class Potomok1 : public Zaklad {
  public:
    void GFun() const {};
```

```
};

class Potomok2 : public Zaklad {
public:
  // Predefinovanie:
  int Fun() const {
    cout << "Potomok2::Fun()\n";
    return 2;
  };
};

class Potomok3 : public Zaklad {
public:
  // Zmena návratového typu:
  void Fun() const { cout << "Potomok3::Fun()\n"; }
};

class Potomok4 : public Zaklad {
public:
  // Zmena zoznamu parametrov:
  int Fun(int) const {
    cout << "Potomok4::Fun()\n";
    return 4;
  }
};

int main() {
  string s("ahoj");
  Potomok1 d1;
  int x = d1.Fun();
  d1.Fun(s);
  Potomok2 d2;
  x = d2.Fun();
//!  d2.Fun(s); // reťazcová verzia je skrytá
  Potomok3 d3;
//!  x = d3.Fun(); // return int verzia skrytá
  Potomok4 d4;
//!  x = d4.Fun(); // f() verzia skrytá
  x = d4.Fun(1);
}
```

V triede **Zaklad** vidíme preťaženú funkciu **Fun()**, a trieda **Potomok1** metódu **Fun()** nemení, ale predefinováva funkciu **GFun()**. Vo funkcii **main()** vidíme, že obidve preťažené verzie funkcie **Fun()** sú v objekte triedy **Potomok1** dostupné. Avšak trieda **Potomok2** predefinováva jednu preťaženú verziu funkcie **Fun()**, ale nie druhú verziu. Výsledkom je, že druhá preťažená forma sa stane nedostupnou. V triede **Potomok3** zmena návratového typu skryje obidve verzie zo základnej triedy a trieda **Potomok4** ilustruje, že zmena zoznamu argumentov tiež skrýva obidve verzie funkcie zo základnej triedy. Vo všeobecnosti môžeme povedať, že vždy, keď predefinujeme meno preťaženej funkcie zo základnej triedy, všetky ostatné verzie sú v novej triede automaticky skryté. Neskôr uvidíme, že preťažovanie funkcií ovplyvní pridanie kľúčového slova **virtual**.

Ak v potomkovi zmeníme rozhranie základnej triedy modifikovaním signatúry a/alebo typu návratovej hodnoty členskej funkcie zo základnej triedy, potom používame triedu iným spôsobom, než dedičnosť normálne podporuje. Neznamená to nevyhnutne, že robíme zle, ale prvotným cieľom dedičnosti je podporovať *polymorfizmus*, a ak zmeníme signatúru funkcie alebo typ návratovej hodnoty, tak v skutočnosti meníme rozhranie základnej triedy. Ak je to presne to čo zamýšľame urobiť, potom používame dedičnosť primárne na opakované použitie kódu a nie na udržiavanie spoločného rozhrania základnej triedy (čo je základným aspektom polymorfizmu). Vo všeobecnosti keď používame dedičnosť takýmto spôsobom, znamená to, že berieme všeúčelovú triedu a špecializujeme ju na konkrétne potreby – čo je zvyčajne, ale nie vždy, považované za sféru pre kompozíciu.

Zoberme napríklad triedu **Zasobnik**, ktorá uchováva smerníky na objekty:

```
#ifndef ZASOBNIK_H
#define ZASOBNIK_H

class Zasobnik {
  struct Uzol {
    void* data;
    Uzol* dalsi;
    Uzol(void* dat, Uzol* dlsi):
      data(dat), dalsi(dlsi) {};
  } *Hlava;
public:
  Zasobnik() : Hlava(0) {};
  ~Zasobnik() {};
  void push(void* dat) {
    Hlava = new Uzol(dat, Hlava);
  };
  void* peek() const {
    return Hlava ? Hlava->data : 0;
  };
  void* pop() {
    if(Hlava == 0) return 0;
    void* navrat = Hlava->data;
    Uzol* staraHlava = Hlava;
    Hlava = Hlava->dalsi;
    delete staraHlava;
    return navrat;
  }
};

#endif
```

Jedným z problémov tejto triedy je, že sme zakaždým, keď sme vybrali smerník z kontajnera, museli neustále pretypovávať. Toto nie je iba otravné, ale je to i nebezpečné – smerník môžeme pretypovať na čokoľvek chceme.

Prístup, ktorý sa zdá na prvý pohľad lepším, je špecializovať **Zasobnik** použitím dedičnosti. Napríklad

```
// Špecializovanie triedy Zasobnik
#include "Zasobnik.h"

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class ZasobnikString : public Zasobnik {
public:
  void push(string* str) {
    Zasobnik::push(str);
  }
  string* peek() const {
    return (string*)Zasobnik::peek();
  }
  string* pop() {
    return (string*)Zasobnik::pop();
  }
```

```
  ~ZasobnikString() {
    string* vrchol = pop();
    while(vrchol) {
      delete vrchol;
      vrchol = pop();
    }
  }
};

int main()
{
  ifstream in("zasobnik.h");
  char bfline[500];

  ZasobnikString textlines;
  while(!in.eof()) {
    in.getline(bfline,500);
    textlines.push(new string(bfline));
  }
  string* s;
  while((s = textlines.pop()) != 0) { // Žiadne pretypovanie!
    cout << *s << endl;
    delete s;
  }
}
```

Trieda **ZasobnikString** špecializuje triedu **Zasobnik** tak, že metóda **push()** bude akceptovať len smerníky typu **string**. Predtým trieda **Zasobnik** akceptovala smerníky typu **void**, a tak užívateľ nemal k dispozícii žiadnu typovú kontrolu, aby sa uistil, že boli vložené správne smerníky. Okrem toho i metódy **peek()** a **pop()** vracajú smerníky typu **String** namiesto **void** smerníkov, a tak nie je potrebné pretypovanie na použitie smerníka.

Táto extra typová kontrola je v `push`, `peek` a `pop` zadarmo. Kompilátor dostal extra typové informácie, ktoré použije počas kompilácie, avšak funkcie sú `inline`, a tak sa negeneruje žiaden kód naviac.

Tu hrá dôležitú úlohu skrývanie mien, konkrétne funkcia `push` má odlišnú signatúru - zoznam argumentov je iný. Ak by sme mali dve verzie metódy `push` v tej istej triede, tak by sa preťažovali, avšak v tomto prípade preťažovanie nie je to, čo chceme, pretože to by dovoľovalo posielať do metódy `push` ľubovoľný smerník ako `void *`. Našťastie C++ skryje verziu metódy `push(void *)` v základnej triede v prospech novej verzie, ktorá je definovaná v odvodenej triede, ktorá dovoľuje vkladať len `String` smerníky do `ZasobnikString`.

Pretože teraz už môžeme zaručiť aké druhy objektov sú v kontajneri, deštruktor funguje správne a je vyriešený i problém vlastníckych práv – alebo prinajmenšom jeden prístup k problému vlastníckych práv. Ak vložíme smerník na `string` do `ZasobnikString`, potom (podľa sémantiky `ZasobnikString`) zároveň prenášame vlastníctvo smerníka do `ZasobnikString`. Ak smerník vyberieme, nielen že získame smerník, ale zároveň získame i vlastnícke práva smerníka. Ľubovoľné smerníky, ktoré zostali v `ZasobnikString`, keď sa zavolá deštruktor tejto triedy, sa vymažú deštruktorom. A pretože sú to vždy `string` smerníky a príkaz `delete` pracuje so smerníkmi typu `string` namiesto `void` smerníkov, vykoná sa zodpovedajúca deštrukcia a všetko funguje správne.

Má to i jeden nedostatok: táto trieda funguje *len* pre `string` smerníky. Ak chceme `Zasobnik`, ktorý by fungoval s nejakým iným typom objektu, musíme napísať novú verziu triedy tak, aby fungovala len s novým druhom objektu.(Nakoniec sa to rieši použitím šablón.)

Na tento príklad sa môžeme pozrieť i z iného uhla: v procese dedičnosti sa tu mení rozhranie triedy `Zasobnik`. Ak je rozhranie odlišné, potom `ZasobnikString` nie je skutočne `Zasobnik` a nikdy nebudeme môcť `ZasobnikString` používať ako `Zasobnik`. Toto robí použitie dedičnosti sporným: ak

nevytvárame `ZasobnikString`, ktorý ***je*** typom triedy `Zasobnik`, potom prečo dedíme? neskôr si ukážeme ešte lepšiu verziu `ZasobnikString`.

Funkcie, ktoré sa automaticky nededia

Nie všetky funkcie sa automaticky dedia zo základnej triedy do odvodenej triedy. Konštruktory a deštruktory sa týkajú tvorby a rušenia objektu, a preto vedia čo majú robiť len v rámci objektu svojej konkrétnej triedy, a tak sa musia zavolať všetky konštruktory a deštruktory v hierarchii pod nimi. A preto konštruktory a deštruktory sa nededia a musia byť vytvorené špeciálne pre každú odvodenú triedu.

Okrem toho sa nededí operátor =, pretože vykonáva činnosť, podobnú konštruktoru.

Namiesto dedenia ak tieto funkcie nevytvoríme, tak ich kompilátor vytvára umelo, (generuje ich). (Pre konštruktory platí, že ak chceme, aby kompilátor vytvoril štandardný a kopírovací konštruktor, nesmieme definovať žiaden konštruktor). Generované konštruktory používajú inicializáciu bit po bite a podobne generovaný operátor = používa priradenie bit po bite. Tu je príklad funkcií, ktoré sú vygenerované kompilátorom:

```
// Funkcie, generované kompilátorom
#include <iostream>

using namespace std;

class HraciaPlocha {
  public:
    HraciaPlocha() { cout << "HraciaPlocha()\n"; }
    HraciaPlocha(const HraciaPlocha&) {
      cout << "HraciaPlocha(const HraciaPlocha&)\n";
    };
    HraciaPlocha& operator=(const HraciaPlocha&) {
      cout << "HraciaPlocha::operator=()\n";
      return *this;
    };
    ~HraciaPlocha() { cout << "~HraciaPlocha()\n"; }
};

class Hra {
    HraciaPlocha gb;   // Kompozícia
  public:
    // Volá sa štandardný konštruktor triedy HraciaPlocha:
    Hra() { cout << "Hra()\n"; };
    // Musíme explicitne volať kopírovací konštruktor triedy
    // HraciaPlocha, inak sa automaticky zavolá štandardný
    // konštruktor:
    Hra(const Hra& g) : gb(g.gb) {
      cout << "Hra(const Hra&)\n";
    }
    Hra(int) { cout << "Hra(int)\n"; }
    Hra &operator=(const Hra& g) {
      // Musíme explicitne volať prira´dovací operátor
      // triedy HraciaPlocha, inak sa nevykoná žiadne
      // priradenie členských objektov gb!
      gb = g.gb;
      cout << "Hra::operator=()\n";
      return *this;
    }
    class Dalsia {}; // Vnorená trieda
    // Automatická konverzia typov:
    operator Dalsia() const {
      cout << "Hra::operator Dalsia()\n";
      return Dalsia();
```

```
    }
    ~Hra() { cout << "~Hra()\n"; }
};

class Sach : public Hra {};

void Fun(Hra::Dalsia) {}

class Dama : public Hra {
  public:
    // Volanie štandardného konštruktora základnej triedy:
    Dama() { cout << "Dama()\n"; };
    // Musíme explicitne volať kopírovací konštruktor
    // základnej triedy, inak sa automaticky zavolá štandardný
    // konštruktor:
    Dama(const Dama& c) : Hra(c) {
      cout << "Dama(const Dama& c)\n";
    };
    Dama& operator=(const Dama& c) {
      // Musíme explicitne volať priraďovací operátor
      // základnej triedy, inak sa nevykoná žiadne
      // priradenie základnej triedy!
      Hra::operator=(c);
      cout << "Dama::operator=()\n";
      return *this;
    }
};

int main()
{
  Sach d1;        // Štandardný konštruktor
  Sach d2(d1);    // Kopírovací konštruktor
//! Sach d3(1);   // Chyba: int konštruktor neexistuje
  d1 = d2;        // Operátor= vygenerovaný
  Fun(d1);              // Konverzia typov je zdedená
  Hra::Dalsia go;
//!  d1 = go;     // Operátor= sa negeneruje
                  // pre líšiace sa typy
  Dama c1, c2(c1);
  c1 = c2;
}
```

Volanie konštruktorov a operátorov = sa vypisuje, a tak môžeme vidieť, kedy ich kompilátor používa. Výstup tohto programu bude nasledovný:

```
HraciaPlocha()
Hra()
HraciaPlocha(const HraciaPlocha&)
Hra(const Hra&)
HraciaPlocha::operator=()
Hra::operator=()
Hra::operator Dalsia()
HraciaPlocha()
Hra()
Dama()
HraciaPlocha(const HraciaPlocha&)
Hra(const Hra&)
Dama(const Dama& c)
HraciaPlocha::operator=()
Hra::operator=()
Dama::operator=()
```

```
~Hra()
~HraciaPlocha()
~Hra()
~HraciaPlocha()
~Hra()
~HraciaPlocha()
~Hra()
~HraciaPlocha()
```

Okrem toho konverzný operator `Dalsia` robí automatickú typovú konverziu z objektu `Hra` na objekt vnorenej triedy `Dalsia`. Trieda `Sach` jednoducho dedí triedu `Hra` a nevytvára žiadne funkcie (aby sme videli ako odpovedá kompilátor). Funkcia `F` akceptuje objekt `Dalsia` na otestovanie funkcie automatickej konverzie typu.

V `main` sa volá vygenerovaný štandardný konštruktor a kopírovací konštruktor odvodenej triedy `Sach`. Zavolá sa `Hra` verzia týchto konštruktorov ako súčasť hierarchie volaní konštruktorov. I keď to vypadá ako dedičnosť, nové konštruktory sú v skutočnosti vytvorené kompilátorom. Ako by sme mohli očakávať, žiaden konštruktor s argumentmi sa automaticky nevytvára pretože to by bolo príliš veľa na kompilátor, aby dokázal vytušiť čo má generovať.

V triede `Sach` je vygenerovaný operátor `=`, používajúci bitové priradenie (takže sa volá verzia základnej triedy), pretože funkcia nebola explicitne zapísaná do novej triedy. A samozrejme kompilátor automaticky vytvoril deštruktor.

Kvôli všetky týmto pravidlám o prepisovaní funkcií, ktoré sa spracovávajú pri vytváraní objektu sa môže na začiatku zdať trochu cudzie, že operátor automatickej konverzie typu je zdedený. Avšak nie je to príliš nezmyselné – ak je dosť častí v tried `Hra` na vytvorenie objektu triedy `Dalsia` , tieto časti sú vo všetkom, čo je odvodené z triedy `Hra` a operátor konverzie typu je stále platný (i keď ho chceme predefinovať).

Operátor `=` sa generuje *len* pre priraďovanie objektov rovnakého typu. Ak chceme priraďovať jeden typ druhému, musíme napísať vlastný operátor `=`.

Ak sa pozrieme lepšie na triedu `Hra`, uvidíme, že kopírovací konštruktor a operátor priradenia obsahujú explicitné volania kopírovacieho konštruktora členského objektu a operátora priradenia. Chceme to takto urobiť, pretože inak v prípade kopírovacieho konštruktora by sa použil štandardný konštruktor a v prípade priraďovacieho operátora by sa objektové členy *nepriradili*.

Nakoniec sa pozrime na triedu `Dama`, ktorá explicitne vypisuje volanie štandardného konštruktora, kopírovacieho konštruktora a priraďovacieho operátora. V prípade štandardného konštruktora sa automaticky zavolá štandardný konštruktor základnej triedy, a zvyčajne to i chceme. Ale dôležité je, že akonáhle sa rozhodneme napísať vlastný kopírovací konštruktor a priraďovací operátor, kompilátor predpokladá, že vieme čo robíme a automaticky *nevolá* verzie zo základnej triedy ako to robí vo vygenerovaných funkciách. Ak chceme zavolať zodpovedajúce verzie funkcií zo základnej triedy (a zvyčajne to robíme), potom ich musíme explicitne zavolať. V kopírovacom konštruktore sa toto volanie objavuje v inicializačnom zozname konštruktora:

```
Dama(const Dama& c) : Hra(c) {};
```

V priraďovacom operátore triedy `Dama` je volanie operátora základnej triedy v prvom riadku tela funkcie:

```
Hra::operator=(c);
```

Tieto volania by mali byť časťou všeobecnej formy, ktorú používame, keď zdedíme triedu.

Dedičnosť a statické členské funkcie

`static` členské funkcie fungujú rovnako ako ne-statické členské funkcie:

1. Dedia sa do odvodenej triedy.

2. Ak predefinujeme statický člen, všetky preťažené funkcie základnej triedy sú skryté.
3. Ak v odvodenej triede zmeníme signatúru funkcie zo základnej triedy, všetky verzie zo základnej triedy s týmto menom funkcie sú skryté (toto je iba obmena predchádzajúceho bodu).

Avšak `static` členské funkcie nemôžu byť virtuálne.

Kompozícia alebo dedičnosť

Kompozícia i dedičnosť vkladajú podobjekty do novej triedy. Obidva spôsoby používajú inicializačný zoznam na vytvorenie týchto podobjektov. Zaujíma nás, aký je medzi nimi rozdiel a kedy vybrať ten alebo onen spôsob.

Kompozícia sa vo všeobecnosti používa, keď chceme mať vlastnosti už existujúcej triedy vo vnútri našej novej triedy, ale nie v rozhraní. Znamená to, že vložíme objekt na implementovanie vlastností našej novej triedy, ale užívateľ našej novej triedy vidí len rozhranie, ktoré sme definovali a nie rozhranie z pôvodnej triedy. Zabezpečíme to typickým spôsobom vkladania `private` objektov existujúcich tried do vnútra novej triedy.

Občas dáva zmysel dovoliť užívateľovi triedy priamo pristupovať ku kompozícií novej triedy, t.j. deklarovať členské objekty ako `public`. Členské objekty majú svoje vlastné riadenie prístupu, takže je to bezpečné a keď užívateľ pozná, že skladáme skupinu súčiastok, uľahčuje to pochopeniu rozhrania. Dobrým príkladom je trieda `Auto`:

```
// public kompozícia
class Motor {
  public:
    void start() const {}
    void rev() const {}
    void stop() const {}
};

class Koleso {
  public:
    void Hustenie(int psi) const {}
};

class Okno {
  public:
    void Zatvor() const {}
    void Otvor() const {}
};

class Dvere {
  public:
    Okno okno_d;
    void Otvor() const {}
    void Zatvor() const {}
};

class Auto {
  public:
    Motor motor_d;
    Koleso koleso_d[4];
    Dvere lave_d, prave_d; // 2-dverové
};

int main() {
  Auto auto;
  auto.lave_d.okno.Zatvor();
  auto.koleso_d[0].Hustenie(72);
```

```
}
```

Pretože kompozícia triedy `Auto` je súčasťou analýzy problému (a nie iba súčasťou návrhu v pozadí), deklarovanie členov ako public napomáha klientovi pochopiť ako používať triedu a vyžaduje menšiu zložitosť kódu pre tvorcu triedy.

Ak sa trochu zamyslíme, uvidíme, že nemá zmysel skladať `Auto` použitím objektu „vozidlo“ – auto neobsahuje vozidlo, *je* to vozidlo. Relácia '*je*' je vyjadrená dedičnosťou a relácia *mať* je vyjadrená kompozíciou.

Subtyping

Predpokladajme, že chceme vytvoriť typ objektu `ifstream`, ktorý nielen otvára súbor, ale uchováva i meno súboru. Môžeme použiť kompozíciu a vložiť obidve triedy, `ifstream` a `istring`, do novej triedy:

```
// fstream s menom súboru
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class SMeno {
    ifstream subor_d;
    string   menosubor_d;
    bool     pomenovany_d;
public:
    SMeno() : pomenovany_d(false) {}
    SMeno(const string& smeno)
        : menosubor_d(smeno),
          subor_d(smeno.c_str()),
          pomenovany_d(true) {};
    string name() const { return menosubor_d; }
    void name(const string& noveMeno) {
      if(pomenovany_d) return; // Neprepíš
        menosubor_d = noveMeno;
      pomenovany_d = true;
    };
    operator ifstream&() { return subor_d; };
};

int main() {
  SMeno subor("subor.txt");
  cout << subor.name() << endl;
  // Chyba: close() nie je členom:
//!  subor.close();
}
```

Avšak je tu problém - pokúšame sa použiť objekt triedy `SMeno` všade tam, kde je použitý objekt triedy `ifstream` zavedením konverzného operátora `()`, ktorý prevádza objekt triedy z `SMeno` na objekt triedy `ifstream&`. Ale v `main` riadok

```
file.close();
```

sa neskompiluje, pretože nastane automatická typová konverzia len vo volaní funkcie, nie počas výberu člena. Takže tento prístup nebude fungovať.

Druhý spôsob spočíva v pridaní definície `close` do `SMeno`:

```
void close() { subor_d.close(); }
```

Toto fungovať bude, ale mali by sme to takto implementovať len ak chceme z triedy `ifstream` preniesť iba niekoľko málo funkcií. V takom prípade využívame iba časť triedy a kompozícia je na mieste.

Ale čo ak chceme sprístupniť všetko funkcie z tejto triedy? Toto sa nazýva *subtyping*, pretože vytvárame nový typ z existujúceho typu, a chceme aby náš nový typ mal presne rovnaké rozhranie ako existujúci typ (plus ďalšie členské funkcie, ktoré chceme pridať), tak, aby sme ich mohli použiť ako keby sme používali existujúci typ. Toto je oblasť, ktorej základom je dedičnosť. Vidíme, že subtyping rieši problém z predchádzajúceho príklade perfektne:

```
// Riešenie problému pomocu subtypingu
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class SMeno : public ifstream {
    string menosubor_d;
    bool pomenovany_d;
  public:
    SMeno() : pomenovany_d(false) {}
    SMeno(const string& meno)
      : ifstream(meno.c_str()),
        menosubor_d(meno),
        pomenovany_d(true) {};
    string name() const { return menosubor_d; }
    void name(const string& noveMeno) {
      if(pomenovany_d) return; // Neprepíš
      menosubor_d = noveMeno;
      pomenovany_d = true;
    }
};

int main() {
  SMeno subor("SMeno.cpp");
  cout << "name: " << subor.name() << endl;
  string s;
  char bf[200];
  subor.getline(bf,200); // Toto tiež funguje!
  subor.getline(s);
  subor.seekg(-200, ios::end);
  subor.close();
}
```

Teraz je prístupná ľubovoľná členská funkcia objektu `ifstream i` pre objekt triedy `SMeno`. S objektmi triedy `SMeno` môžu fungovať i nečlenské funkcie, ktoré očakávajú ako parameter `ifstream`. Je to dané tým, že trieda `SMeno` *je* typom `ifstream`.. Toto je veľmi dôležitá záležitosť, o ktorej si povieme neskôr.

private dedičnosť

Základnú triedu môžeme zdediť i privátne, ak vynecháme slovo `public` v zozname základných tried alebo explicitným uvedením kľúčového slova `private` (pravdepodobne lepší variant, pretože je užívateľovi zrejmé, že to chceme). Ak zdedíme privátne, vytvárame novú triedu, ktorá má všetky dáta a funkčnosť základnej triedy, ale funkcionalita je skrytá, t.j. je súčasťou základnej implementácie. Užívateľ triedy nemá prístup k funkčnosti základnej triedy, a s objektom sa nedá pracovať ako s objektom základnej triedy.

Zdá sa možno čudné aký je účel `private` dedičnosti, pretože alternatíva použitia kompozície na vytvorenie privátneho objektu v novej triede sa zdá vhodnejšia. `private` dedičnosť je do jazyka zaradená kvôli úplnosti. Avšak môžu sa príležitostne vyskytnúť situácie, kedy chceme vytvoriť časť rovnakého rozhrania

ako má základná trieda *a* zablokovať zaobchádzanie s objektom ako keby to bol objekt základnej triedy. `private` dedičnosť túto možnosť poskytuje.

Zverejňovanie privátne zdedených členov

Ak dedíme privátne, všetky `public` členy základnej triedy sa stávajú `private`. Ak chceme, aby ktorýkoľvek z nich bol viditeľný, stačí vymenovať ich mená (nemusia byť argumenty ani návratové hodnoty) spolu s kľúčovým slovom `using` v `public` časti odvodenej triedy:

```
// Privátna dedičnosť
class Skola {
  public:
    char  Otvorena() const { return 'o'; }
    int   Zatvorena() const { return 1; }
    float Obsadena() const { return 3.0; }
    float Obsadena(int) const { return 4.0; }
};

class StrednaSkola : Skola {     // Privátna dedičnosť
  public:
    using Skola::Otvorena;       // Publikovanie člena
    using Skola::Obsadena;       // Publikovanie oboch
                                 // preťažených členov
};

int main() {
  StrednaSkola skola;
  skola.Otvorena();
  skola.Obsadena();
  skola.Obsadena(1);
//! skola.Zatvorena();// Chyba: privátna členská funkcia
}
```

A tak `private` dedičnosť je užitočná, ak chceme zakryť časť funkčnosti základnej triedy.

Všimnime si, že vystavenie mena preťaženej funkcie vystavuje všetky verzie preťaženej funkcie v základnej triede.

protected

Teraz, keď už sme sa zoznámili s dedičnosťou, dostáva význam i kľúčové slovo `protected`. V ideálnom svete by `private` členy boli sotva vždy napevno `private`, v reálnych projektoch sú oblasti, kde chceme niečo ukryť pred celým svetom, avšak dovoliť prístup k členom v odvodených triedach. Na to je určené kľúčové slovo `protected`. Hovorí: „Toto je privátne z pohľadu užívateľa triedy, ale dostupné pre kohokoľvek, kto túto triedu dedí“.

Najlepším postupom je nechávať dátové členy vždy `private` - čím si uchováme právo meniť základnú implementáciu. Prístup k nim môžeme dedičom povoliť prostredníctvom `protected` členských funkcií:

```
// Kľúčové slovo protected
#include <fstream>
using namespace std;

class Zaklad {
    int i_d;
  protected:
    int  Daj() const { return i; };
    void Nastav(int ii) { i = ii; };
  public:
    Zaklad(int ii = 0) : i_d(ii) {}
```

```
    int Hodnota(int m) const { return m*i; };
};

class Potomok : public Zaklad {
    int j_d;
  public:
    Potomok(int jj = 0) : j_d(jj) {};
    void Modifikuj(int x) { Nastav(x); };
};

int main()
{
  Potomok d;
  d.Modifikuj(10);
}
```

protected dedičnosť

Ak dedíme, implicitne všetko zo základnej triedy je `private`, čo znamená, že všetky public členské funkcie sú pre užívateľa novej triedy `private`. Normálne deklarujeme dedičnosť ako `public`, čim rozhranie základnej triedy je tiež rozhraním odvodenej triedy. Avšak v dedičnosti tiež môžeme použiť kľúčové slovo `protected`.

`protected` odvodenie znamená pre ostatné triedy „implementované použitím“, ale pre odvodené triedy platí „je“. Je to niečo, čo sa nepoužíva často, ale je to pre úplnosť súčasťou jazyka.

Preťažovanie operátorov a dedičnosť

Okrem priraďovacieho operátora sa ostatné operátory automaticky dedia do odvodenej triedy.

Viacnásobná dedičnosť

Keď už môžeme dediť z jednej triedy, mohlo by dávať zmysel dediť z viac ako jednej triedy naraz. Samozrejme, je to možné, ale či to má zmysel ako súčasť návrhu je predmetom neustálej diskusie. S jedným však možno súhlasiť - nemali by sme to skúšať, pokiaľ už nejakú dobu neprogramujeme a jazyk sme dôkladne nezvládli. Takmer vždy vystačíme s jednoduchou dedičnosťou.

Zo začiatku sa viacnásobná dedičnosť zdá dostatočne jednoduchá: Dodáme viac tried do zoznamu základných tried dedičnosti, oddelené čiarkou. Avšak viacnásobná dedičnosť prináša so sebou niekoľko príležitostí pre nejednoznačnosti.

Inkrementálny vývoj

Jednou z výhod dedičnosti a kompozície je, že podporujú ***inkrementálny vývoj***, umožňujúci vkladať nový kód bez zavedenia chýb do už existujúceho kódu. Ak sa objaví chyba, izolujú sa v rámci nového kódu. Dedením z (alebo kompozíciou s) existujúcej, funkčnej triedy a pridaním dátových členov a členských funkcií (a predefinovaním existujúcich členských funkcií počas dedenia) ponecháme existujúci kód – ktorý môže ešte niekto využiť - nedotknutý a bez chýb. Ak sa vyskytne chyba, vieme, že bude v novom kóde, ktorý je omnoho kratší, ľahšie sa číta, než keď zmeníme telo už existujúceho kódu.

Je dosť prekvapujúce ako čisto sú triedy oddelené. Dokonca ani nepotrebujeme zdrojový kód členských funkcií, aby sme mohli znovupoužiť ich kód, stačí nám hlavičkový súbor, popisujúci triedu a cieľový (object) súbor alebo knižničný súbor so skompilovanými členskými funkciami (Toto platí pre dedičnosť i kompozíciu).

Je dôležité uvedomiť si, že vývoj programu je inkrementálny proces, rovnako ako učenie. Môžeme spraviť akúkoľvek analýzu, ale keď sa pustíme do projektu, všetky odpovede nebudeme poznať. Budeme

úspešnejší – a mať skoršiu spätnú väzbu – ak začneme budovať náš projekt ako organickú evolučnú bytosť, než konštruovať všetko naraz ako sklenený mrakodrap.

Hoci dedičnosť je pre experimentovanie užitočná metóda, v určitom bode, keď sa veci stabilizujú, potrebujeme sa znova pozrieť na našu hierarchiu tried s ohľadom na jej štruktúru. Nezabúdajme, že dedičnosť znamená vyjadrenie, ktoré hovorí, „Táto nová trieda je *typom* starej triedy.“ Náš program by sa nemal zaoberať presúvaním bitov dookola, ale namiesto toho by mal vytvárať a narábať s objektmi rôznych typov na vyjadrenie modelu v termínoch, daných priestorom problému.

Upcasting – pretypovsnir smerom nahor

Na začiatku sme videli ako objekt triedy, odvodenej z triedy `ifstream`, má všetky charakteristiky a správanie objektu triedy `ifstream`. V objekte triedy `SMeno` môžeme volať ľubovoľnú funkciu triedy `ifstream`..

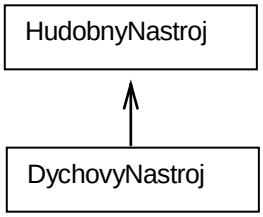
Najdôležitejším aspektom dedičnosti však nie je to, že poskytuje funkcie pre novú triedu. Je to relácia vyjadrená medzi novou a základnou triedou. Túto reláciu môžeme zhrnúť do vety „*Nová trieda je typom existujúcej triedy*.“

Tento popis nie je iba čudný spôsob vysvetlenia dedičnosti – je priamo podporovaný kompilátorom. Zoberme napríklad základnú triedu nazvanú `HudobnyNastroj`, ktorý reprezentuje hudobné nástroje, a odvodenú triedu nazvanú `DychovyNastroj`. Pretože dedičnosť znamená, že všetky funkcie základnej triedy sú tiež dostupné v odvodenej triede, každá správa, ktorú môžeme poslať do základnej triedy, môže byť poslaná i do odvodenej triedy. Takže ak trieda `HudobnyNastroj` má členskú funkciu `Hraj`, tak ju bude mať i trieda `DychovyNastroj`. Znamená to, že `DychovyNastroj` objekt je tiež typom `HudobnyNastroj`. Nasledovný príklad ilustruje, ako kompilátor podporuje túto predstavu:

```
// Dedičnosť & upcasting
enum nota { stredneC, durC, molC }; // atd.

class HudobnyNastroj {
  public:
    void Hraj(nota) const {};
  };

// Objekty triedy DychovyNastroj sú HudobnyNastroj-e
// pretože majú rovnaké rozhranie:
class DychovyNastroj : public HudobnyNastroj {};

void Melodia(HudobnyNastroj& i) {
  // ...
  i.Hraj(middleC);
}

int main() {
  DychovyNastroj flauta;
  Melodia(flauta); // Upcasting
} ///:~
```

V tomto príklade je zaujímavá funkcia `Melodia`. ktorá akceptuje odkaz na objekt triedy `HudobnyNastroj`. Avšak v `main` je funkcia `Melodia` volaná s odkazom na objekt triedy `DychovyNastroj`. Vieme, že C++ je veľmi prísne čo sa týka kontroly typov a preto sa zdá podivné, že funkcia, ktorá akceptuje jeden typ bude akceptovať iný, pokiaľ si neuvedomíme, že objekt triedy `DychovyNastroj` je tiež objektom triedy `HudobnyNastroj` a neexistuje funkcia, ktorú funkcia `Melodia` zavolá pre `HudobnyNastroj` `aby` nebola i vo `DychovyNastroj` (toto nám práve zabezpečí dedičnosť). Kód vo vnútri funkcie `Melodia funguje pre HudobnyNastroj` a pre hocičo, čo je odvodené od `HudobnyNastroj` a akt konverzie odkazu alebo smerníka `DychovyNastroj` na odkaz alebo smerník na `HudobnyNastroj` sa nazýva *pretypovanie smerom nahor* (*upcasting)*.

Prečo pretypovanie smerom nahor? Dôvod tohto pomenovania je historický a je založený na spôsobe, akým sa kreslili diagramy dedičnosti: od koreňa na vrchu stránky smerom dolu. (Samozrejme, diagramy môžeme kresliť hocijako, pokiaľ sú zrozumiteľné). Diagram dedičnosti pre `HudobnyNastroj.cpp` je nasledovný:



pretypovanie z odvodenej triedy na základnú ide smerom hore v diagrame dedičnosti, a tak sa to zvyčajne nazýva *upcasting (pretypovanie smerom nahor)*. Pretypovanie smerom nahor je vždy bezpečné, pretože ideme zo špecifickejšieho typu smerom ku všeobecnejšiemu typu – jedinou vecou, ktorá môže nastať pre rozhranie triedy je, že môže stratiť členské funkcie, nie získať. Toto je i dôvod, prečo kompilátor dovoľuje pretypovanie smerom nahor bez explicitného pretypovaní alebo nejakej inej špeciálnej notácie.

Pretypovanie smerom nahor a kopírovací konštruktor

Ak dovolíme kompilátoru generovať kopírovací konštruktor pre odvodenú triedu, automaticky volá kopírovací konštruktor základnej triedy a potom kopírovacie konštruktory všetkých členských objektov (alebo robí bitové kópie zabudovaných typov), a tak dostaneme správne správanie:

```
// Správne vytváranie kopírovacieho konštruktora
#include <iostream>
#include <string>
using namespace std;

class Rodic {
    int i_d;
  public:
    Rodic(int ii) : i_d(ii) {
      cout << "Rodic(int ii)\n";
    };
    Rodic(const Rodic& b) : i_d(b.i_d) {
      cout << "Rodic(const Rodic&)\n";
    };
    Rodic() : i_d(0) { cout << "Rodic()\n"; }
    friend ostream& operator<<(ostream& os, const Rodic& b) {
      return os << "Rodic: " << b.i_d << endl;
    };
};

class Clen {
    int i_d;
  public:
    Clen(int ii) : i_d(ii) {
      cout << "Clen(int ii)\n";
    };
    Clen(const Clen& m) : i_d(m.i_d) {
      cout << "Clen(const Clen&)\n";
    };
    friend ostream& operator<<(ostream& os, const Clen& m) {
      return os << "Clen: " << m.i_d << endl;
    };
};

class Potomok : public Rodic {
    int i_d;
    Clen m;
```

```
  public:
    Potomok(int ii) : Rodic(ii), i_d(ii), m(ii) {
      cout << "Potomok(int ii)\n";
    }
    friend ostream&
      operator<<(ostream& os, const Potomok& c){
      return os << (Rodic&)c << c.m
                << "Potomok: " << c.i_d << endl;
    }
};

int main()
{
  Potomok c(2);
  cout << "Volanie kopirovacieho konštruktora: " << endl;
  Potomok c2 = c; // Volanie kopírovacieho konštruktora
  cout << "Hodnoty v c2:\n" << c2;
}
```

Operátor `<<` triedy `Potomok` je zaujímavý, kvôli spôsobu, akým volá operátor `<<` triedy `Rodic` v rámci svojho tela: pretypováva objekt `Potomok` na odkaz na objekt triedy `Rodic` (ak pretypovávame na objekt základnej triedy namiesto na odkaz, zvyčajne dostaneme nežiadúce výsledky):

```
    return os << (Rodic&)c << c.m
```

Pretože kompilátor ho potom považuje za `Rodic`, zavolá `Rodic` verziu operátora `<<`.

Vidíme, že trieda `Potomok` nedefinuje explicitne kopírovací konštruktor. Kompilátor teda kopírovací konštruktor vygeneruje (pretože je to jedna zo štyroch funkcií, ktoré generuje spolu so štandardným konštruktorom – ak nevytvoríme žiaden konštruktor – operátor = a deštruktor) z volaním kopírovacieho konštruktora triedy `Rodic` a kopírovacieho konštruktora triedy `Clen`. A tu je výstup:

```
Rodic(int ii)
Clen(int ii)
Potomok(int ii)
Volanie kopirovacieho konštruktora:
Rodic(const Rodic&)
Clen(const Clen&)
Hodnoty v c2:
Rodic: 2
Clen: 2
Potomok: 2
```

Avšak, ak sa pokúsime napísať svoj vlastný kopírovací konštruktor pre triedu `Potomok`, a spravíme naivnú chybu a spravíme ho nesprávne:

```
Potomok(const Potomok& c) : i_d(c.i_d), m(c.m) {}
```

potom sa automaticky zavolá štandardný konštruktor základnej triedy triedy `Potomok`, pretože , pretože to je práve to, k čomu sa uchýli kompilátor, keď mu nezadáme žiadnu inú možnosť volania konštruktora (spomeňme si, že *nejaký* konštruktor sa musí zavolať pre každý objekt, bez ohľadu či je podobjektom nejakej ďalšej triedy.) Výstup potom bude:

```
Rodic(int ii)
Clen(int ii)
Potomok(int ii)
Volanie kopirovacieho konštruktora:
Rodic()
Clen(const Clen&)
Hodnoty v c2:
```

```
Rodic: 0
Clen: 2
Potomok: 2
```

Toto nie je práve to, čo očakávame, pretože vo všeobecnosti chceme, aby sa v rámci kopírovacieho konštruktora skopírovala z existujúceho objektu do nového objektu i časť zo základnej triedy.

Aby sme tento problém odstránili, musíme správne zavolať kopírovací konštruktor (ako to robí kompilátor) vždy, keď napíšeme svoj vlastný kopírovací konštruktor. Toto na prvý pohľad vypadá dosť nezvyčajne, ale je to ďalší príklad pretypovania smerom nahor.

```
Potomok(const Potomok& c)
    : Rodic(c), i_d(c.i_d), m(c.m) {
    cout << "Potomok(Potomok&)\n";
}
```

Podivná časť je vo volaní kopírovacieho konštruktora triedy `Rodic`: `Rodic(c)`. čo znamená prenos objektu triedy `Potomok` do konštruktora triedy `Rodic`? Ale `Potomok` dedí triedu `Rodic`, a tak odkaz na `Potomok` ***je*** odkazom na `Rodic`. Kopírovací konštruktor základnej triedy pretypuje odkaz na `Potomok` smerom nahor na `Rodic`, použije ho na vykonanie kopírovacieho konštruktora. Keď napíšeme svoj vlastný kopírovací konštruktor, robíme takmer vždy to isté.

Kompozícia verzus dedičnosť (doplnenie)

Jedným z najčistejších spôsobov ako určiť, či sa má použiť kompozícia alebo dedičnosť je pýtaním sa, či budeme niekedy potrebovať pretypovanie smerom nahor z novej triedy. Na začiatku bola trieda `Zasobnik` špecializovaná použitím dedičnosti. Avšak šance, že `ZasobnikString` bude použitý len ako kontajner pre `string` sú veľké a nikdy nebude nutné pretypovanie smerom nahor, a tak vhodnejšou alternatívou je kompozícia:

```
// Kompozícia alebo dedičnosť ?
#include <iostream>
#include <fstream>
#include <string>

#include "zasobnik.h"

using namespace std;

class ZasobnikString {
    Zasobnik stack; // Vložený namiesto zdedeného
  public:
    void push(string* str) {
      stack.push(str);
    };
    string* peek() const {
      return (string*)stack.peek();
    };
    string* pop() {
      return (string*)stack.pop();
    };
};

int main()
{
  ifstream in("zasobnik.h");
  char bfline[500];

  ZasobnikString textlines;
  while(!in.eof()) {
```

```
    in.getline(bfline,500);
    textlines.push(new string(bfline));
  }
  string* s;
  while((s = textlines.pop()) != 0) { // Žiadne pretypovanie!
    cout << *s << endl;
    delete s;
  }
}
```

Objekt `Zasobnik` je vložený do `ZasobnikString` a členské funkcie sa volajú pre vložený objekt. Časová a priestorová réžia nie je žiadna, pretože podobjekt zaberá rovnako veľký priestor a všetky dodatočné kontroly typov sa urobia počas kompilácie.

Hoci je to trochu mätúce, mohli by sme tiež použiť `private` dedičnosť na vyjadrenie „implementovaný pomocou“. Tiež by to problém riešilo adekvátnym spôsobom. Avšak je jedno miesto kde sa kompozícia stáva dôležitou, a to tam, kde môžeme kompozíciu použiť namiesto viacnásobnej dedičnosti.

Pretypovanie smerníkov a odkazov smerom nahor

V `HudobnyNastroj.cpp` sa pretypovanie smerom nahor vyskytuje počas volania funkcie – odkaz na `DychovyNastroj` objekt mimo funkciu sa v jej tele mení na odkaz na `HudobnyNastroj` objekt. Pretypovanie smerom nahor môže nastať počas priraďovania smerníka alebo odkazu:

```
DychovyNastroj w;
HudobnyNastroj* ip = &w; // pretypovanie nahor
HudobnyNastroj& ir = w;  // pretypovanie nahor
```

Podobne ako pri volaní funkcie žiaden z týchto prípadov nevyžaduje explicitné pretypovanie.

Samozrejme akékoľvek pretypovanie smerom nahor spôsobuje stratu informácie o type objektu. Ak napríklad napíšeme:

```
DychovyNastroj w;
HudobnyNastroj* ip = &w;
```

kompilátor bude pracovať s `ip` len s ako smerníkom na objekt triedy `HudobnyNastroj`. Znamená to, že nedokáže poznať, že `ip` v skutočnosti ukazuje na objekt triedy `DychovyNastroj`. Takže ak zavoláme členskú funkciu `Hraj`:

```
ip->Hraj(stredneC);
```

kompilátor vie len, že sa volá `Hraj` so smerníkom na `HudobnyNastroj` a zavolá verziu zo základnej triedy `HudobnyNastroj::Hraj` namiesto toho, čo by mal volať, t.j. `DychovyNastroj::Hraj`. A tak nedostaneme správne správanie.

Toto je významný problém, ktorý rieši tretí základný kameň objektovo-orientovaného programovania: polymorfizmus (v C++ implementovaný `virtual` funkciami).

Zhrnutie

Dedičnosť i kompozícia dovoľujú vytvárať nové typy z už existujúcich typov a obidva spôsoby vkladajú podobjekty existujúcich typov do nového typu. Zvyčajne používame kompozíciu na znovupoužitie existujúcich typov ako súčastí základnej implementácie nového typu a dedičnosť, keď chceme aby nový typ bol rovnakého typu ako je základná trieda (typová ekvivalencia zabezpečuje ekvivalenciu rozhraní). Pretože odvodená trieda má rozhranie základnej triedy, môžeme ju pretypovávať smerom nahor (upcasting) na základnú triedu, čo je kritické pre polymorfizmus.

Hoci znovupoužitie kódu prostredníctvom kompozície a dedičnosti je veľmi užitočné pre rýchly vývoj projektu, zvyčajne chceme prebudovať našu hierarchiu tried skôr, než dovolíme iným programátorom aby boli od nej závislí. Naším cieľom je hierarchia, v ktorej má trieda špecifické použitie a nie je ani príliš veľká (zahrňujúca toľko funkčnosti, že je nepraktick0 na znovupoužitie) ani príliš malá (nemožno ju používať samostatne alebo bez pridania funkčnosti).