KAPITOLA

10

Objekty a třídy

V této kapitole se naučíte:

- Procedurální a objektově orientované programování
- Základní principy tříd
- Jak definovat a implementovat třídu
- Veřejný a soukromý přístup ke třídě
- Datové položky třídy
- Metody třídy (taktéž zvané členské funkce třídy)
- Vytváření a používání objektů třídy
- Konstruktory a destruktory třídy
- Konstantní členské funkce
- Ukazatel this
- Vytváření polí objektů
- Rozsah platnosti třídy
- Abstraktní datové typy (ADT)

Objektově orientované programování (OOP) je určitý koncepční přístup návrhu programů a jazyk C++ rozšiřuje jazyk C o vlastnosti, které jeho použití usnadňují. Mezi nejdůležitější vlastnosti OOP patří:

- Abstrakce
- Zapouzdření a ukrytí dat
- Polymorfismus
- Dědičnost
- Znovupoužitelný kód

Třída je nejdůležitějším rozšířením C++, které uvedené vlastnosti umožňuje implementovat a spojit dohromady. V této kapitole začneme třídy zkoumat. Vysvětlíme pojmy abstrakce, zapouzdření, ukrývání dat a ukážeme, jak třída tyto rysy implementuje. Dozvíte se, jak třídu definovat, o její soukromé a veřejné části, a jak se vytváří metody pracující s daty třídy. Také vás tato kapitola seznámí s koncepcí konstruktorů a destruktorů, což jsou speciální metody třídy určené pro vytváření a rušení objektů náležejících třídě. Nakonec se seznámíte s ukazatelem `this`, důležitou součástí programování pomocí tříd. Následující kapitoly rozšiřují témata o překrývání operátorů (další druh polymorfismu) a dědičnost, což je základ opakovaného použití kódu.

Procedurální a objektově orientované programování

Ačkoli jsme hledisko OOP prozkoumali již velice brzy, většinou jsme se drželi velice blízko ke standardnímu procedurálnímu přístupu jazyků, jako jsou C, Pascal nebo Basic. Pojďme se podívat na příklad, který objasní, jak se pohled OOP liší od přístupu procedurálního programování.

Jako nejnovější člen baseballového týmu Genre Giants jste byli požádáni o vedení týmové statistiky. Přirozeně budete chtít využít svůj počítač. Kdybyste byl procedurální programátor, mohl byste uvažovat asi tímto způsobem:

Takže chci mít možnost vkládat jméno, počet stání na pálce, počet odpalů, průměrný odpal (pro ty, kteří se nezajímají o baseball nebo softball, průměrný odpal je počet odpalů vydělený počtem stání na pálce. Odpal je ukončen, když hráč oběhne všechny mety nebo je vyřazen protihráčem) a ostatní údaje o každém hráči. Počkat, počítač by mi měl usnadňovat práci, proto ho nechám některé hodnoty vypočítat, například průměrný odpal. Také chci, aby program vypisoval výsledky. Jak to mám zorganizovat? Myslím, že bych měl začít hned a použít funkce. Ano, vytvořím funkci main(), která bude volat další funkce pro vstup dat, samotný výpočet hodnot a poté funkci pro výpis výsledků. Co když získám data z další hry? Nechci všechno vkládat znova. Dobrá, přidám funkci pro aktualizaci statistiky. Také asi budu potřebovat nějakou funkci pro nabídku k výběru mezi vkládáním dat, výpočtem, aktualizací a výpisem dat. Jak bych měl data reprezentovat? Mohl bych použít pole řetězců pro jména hráčů, další pole pro počet stání na pálce a ještě jedno pole pro počet úspěšných odpalů atd. Ne, to je hloupost. Mohu vytvořit strukturu, obsahující všechny informace o jednom hráči a poté použít pole těchto struktur pro reprezentaci celého týmu.

Zkrátka a dobře, nejdříve se soustředíte na procedury, které chcete použít, a poté uvažujete o reprezentaci dat. (Poznámka: Abyste nemuseli mít program spuštěn po celou herní sezónu, budete asi chtít data také ukládat do souboru a poté ze souboru načítat. Ale protože jsme zatím o souborech nehovořili, budeme tuto komplikaci nyní ignorovat.)

Nyní se podívejme, jak se váš pohled změní, když si nasadíte klobouk OOP (v atraktivním, polymorfickém provedení). Začnete uvažovat nejdříve o datech. Navíc nebudete uvažovat pouze o reprezentaci dat samotných, nýbrž i o jejich použití.

Tak se podíváme, o čem vlastně sbírám data. Samozřejmě o baseballovém hráči. Takže budu chtít objekt reprezentující celého hráče, nikoli pouze počet odpalů nebo průměrný odpal. Ano, toto bude můj základní kámen, objekt obsahující jméno a statistiky hráče. Budu potřebovat nějaké metody pro manipulaci s tímto objektem. Asi budu potřebovat metodu pro zadání základních informací o objektu. Objekt by měl některé informace vypočítat sám – přidám metody provádějící tyto výpočty. Program by měl tyto výpočty provádět automaticky, aniž by uživatel musel o jejich provedení požádat. Také budu potřebovat metody pro aktualizaci a výpis dat. Takže uživatel dostane tři možnosti interakce: inicializaci, aktualizaci a výpis dat. To je uživatelské rozhraní.

Stručně řečeno, s přístupem OOP se soustředíte na objekt tak, jak jej vnímá uživatel, přičemž uvažujete o datech potřebných pro reprezentaci objektu a o metodách vystihujících interakci uživatele s objektem. Po vytvoření popisu tohoto rozhraní se přesunete k problému jeho implementace a uložení dat. Nakonec vytvoříte program využívající nový návrh.

Abstrakce a třídy

Život je velmi složitý a s touto složitostí se vyrovnáváme pomocí zjednodušujících abstrakcí. Vy jste kolekcí více než jedné miliardy atomů. Někteří studenti psychologie by řekli, že vaše vědomí je kolekcí částečně autonomních agentů. Ale je mnohem jednodušší o sobě uvažovat jako o jednotce. U počítačů je abstrakce základním krokem k reprezentaci dat pomocí jejich interakce s uživatelem. To znamená, že abstrahujete základní vlastnosti problému a pomocí nich nastíníte řešení. V příkladu o baseballovém týmu rozhraní popisovalo, jak uživatel inicializuje, aktualizuje a zobrazuje data. Od abstrakce je to jen krůček k uživatelem definovanému datovému typu, což je v C++ třída, reprezentující uvedené rozhraní.

Co je typ

Zamysleme se trochu více nad tím, co tvoří typ. Například kdo to je vědátor? Pokud se přikloníte k oblíbenému stereotypu, můžete takového vědátora posuzovat podle jeho vzhledu – tlusté brýle s rámečkem, kapsa plná tužek atp. Po krátké úvaze můžete dojít k závěru, že vědátor se lépe definuje podle svého chování. Například podle toho, jak jedná v nepříjemných situacích. My jsme v podobné situaci, jestliže dovolíte trochu přehnanou analogii, i s procedurálním jazykem C. Nejdříve máte tendenci přemýšlet o datech podle jejich vzhledu – jak budou uložena v paměti. Typ `char` je například jeden bajt v paměti a typ `double` často zabírá bajtů osm. Ale jednoduchá úvaha nás dovede k závěru, že datový typ je také určován operacemi, které je s ním možné provádět. Například typ `int` můžete použít na jakékoli aritmetické operace. Můžete sčítat, odčítat, násobit a dělit celá čísla. Také můžete použít operátor modulo (`%`) pro získání zbytku po dělení.

Na druhé straně vezměme v úvahu ukazatele. Ukazatel bude v paměti často zabírat stejnou velikost jako typ int. Dokonce může být interně uložen jako celé číslo. Ale ukazatel neposkytuje stejné operace jako celá čísla. Například nemůžete vynásobit dva ukazatele. Taková operace nemá smysl, proto není v C++ implementována. Jestliže tedy nadefinujete proměnnou typu `int` nebo typu ukazatel na reálné číslo, nealokujete pouze místo v paměti, ale také určujete, které operace bude možné s proměnnou provádět. Stručně řečeno, specifikací základního typu určujete tyto tři věci:

- Velikost paměti potřebné pro uložení datového objektu.
- Interpretaci bitů v paměti. (Typy `long` a `float` mohou mít stejný počet bitů, avšak na číselné hodnoty jsou převáděny odlišně.)
- Množinu operací nebo metod, které je možné provádět nad datovým objektem.

V případě vestavěných typů jsou tyto informace obsaženy již v překladači. Ale jestliže v C++ definujete uživatelský typ, musíte tyto informace určit sami. Za to získáte možnost vytvářet nové datové typy odpovídající skutečným požadavkům.

Třídy v C++

Třída je nástroj, kterým jazyk C++ umožňuje převést abstrakci do uživatelem definovaného typu. Kombinuje data a metody pro manipulaci s těmito daty do elegantního celku. Podívejme se na třídu reprezentující kapitálové vklady.

Nejdříve si musíme rozmyslet, jak budeme vklady reprezentovat. Mohli bychom vzít jednu akcii podílu jako základ a vytvořit třídu reprezentující jednu akcii. To by však v případě sta akcií mělo za následek potřebu sta objektů reprezentujících akcii, což není praktické. Namísto toho vezměme za základ aktuální podíl konkrétní osoby v daném vkladu. Počet vlastněných akcií bude součástí objektu. Při realistickém pohledu na věc bude potřeba pro daňové účely zaznamenat informace, jako jsou pořizovací cena a datum nákupu. Také by se mělo například počítat s dělením podílu. To vypadá pro začátek už trochu náročně, takže vezmeme v úvahu zidealizovaný stav zjednodušující pohled na věc. Konkrétně se omezíme na tyto operace:

- Získání podílu ve firmě.
- Nákup více akcií v rámci jednoho podílu.
- Prodání podílu.
- Aktualizace hodnoty akcie.
- Zobrazení informace o vkladech.

Tento seznam můžeme využít k definici veřejného rozhraní naší třídy, přičemž další operace ponecháváme jako cvičení pro zainteresované. Pro podporu tohoto rozhraní potřebujeme některá data ukládat. Opět použijeme zjednodušený pohled. Například nebudeme brát v úvahu standardní praxi v USA, kde je hodnota podílu násobkem osmi dolarů. Budeme ukládat následující informace:

- Název podniku.
- Počet akcií.
- Cenu jedné akcie.
- Celkovou hodnotu všech akcií.

V dalším kroku definujeme třídu. Obecně má specifikace třídy dvě části:

- *Deklaraci třídy*, která popisuje datové složky jako datové členy a veřejné rozhraní jako členské funkce, které nazýváme *metody*.
- *Definici metod třídy*, která obsahuje implementaci jednotlivých metod třídy.

Zhruba řečeno, deklarace třídy obsahuje popis třídy, zatímco definice metod obsahují implementační detaily.

Co je to rozhraní?

Rozhraní **je sdílený rámec pro součinnost mezi dvěma systémy – například mezi počítačem a tiskárnou nebo mezi uživatelem a počítačovým programem. Uživatelem můžete být například vy a programem může být editor. Když s ním pracujete, nepřenášíte slova přímo z hlavy do počítačové paměti, nýbrž komunikujete s rozhraním programu. Stisknete klávesu a počítač ji vypíše na obrazovce. Pohnete myší a počítač pohne kurzorem na obrazovce. Omylem kliknete myší a poškodíte si rozepsaný odstavec. Rozhraní programu odpovídá za konverzi vašich úmyslů do určitých informací, které se přenášejí do počítače. U tříd hovoříme o veřejném rozhraní. V tomto případě je veřejností program, který třídu používá, spolupracující systém obsahuje objekty typu třída a rozhraní obsahuje metody, jež jsou v dané třídě k dispozici. Rozhraní umožňuje programátorovi psát kód, který spolupracuje s objekty v třídě, umožňuje**

tedy tyto objekty používat. Například při zjišťování počtu znaků v objektu typu `string` nemusíte objekt prohledávat, nýbrž pouze použijete metodu `size()`, kterou nabízejí autoři třídy. Naopak třída zase neumožňuje uživateli přímý přístup – ten pouze smí používat metodu `size()`. Tato metoda je tedy součástí veřejného rozhraní mezi uživatelem a objektem třídy typu `string`. Podobně i metoda `getline()` je součástí veřejného rozhraní třídy `istream`; program, který používá `cin`, se nevrtá přímo ve vnitřnostech objektu `cin`, když chce přečíst vstupní řádek – tuto práci provede `getline()`.

Pokud byste měli zájem o poněkud osobnější vztah a nechcete, aby program používal třídu jenom jako veřejný uživatel, můžete si říct, že vy sám jako osoba, která píše program, jste veřejný uživatel třídy. Avšak v každém případě, když chcete používat třídu, musíte znát její veřejné rozhraní, a když chcete třídu napsat, musíte její veřejné rozhraní vytvořit.

Vytváření třídy a programu, který ji používá, vyžaduje několik kroků. Neuděláme je všechny současně, nýbrž celý vývoj si rozdělíme do kratších celků; všechny jsou pak shrnuty ve výpisu 10.3. Výpis 10.1 obsahuje předběžnou deklaraci třídy Stock. (Jako pomůcku pro identifikaci tříd používáme často, ale nikoli univerzální, konvenci pojmenovávání tříd.) Zajisté si všimnete podobnosti s deklarací struktury až na několik částí obsahujících deklaraci metod a částí pro veřejný a soukromý přístup k datům. Brzy tuto deklaraci vylepšíme (proto ji nepoužívejte jako definitivní model), ale nejdříve se podívejme, jak tato definice funguje.

Výpis 10.1 stocks.cpp

```
// začátek souboru stocks.cpp
#include <iostream>
#include <cstring>

class Stock     // deklarace třídy
{
private:
   char company[30];
   int shares;
   double share_val;
   double total_val;
   void set_tot() { total_val = shares * share_val; }
public:
   void acquire(const char * co, int n, double pr);
   void buy(int num, double price);
   void sell(int num, double price);
   void update(double price);
   void show();
};              // všimněte si středníku na konci deklarace
```

Bližší pohled na detaily třídy získáte později, nejdříve prozkoumáme obecnější vlastnosti. Na začátku klíčové slovo jazyka C++ `class` identifikuje následující kód jako deklaraci třídy. Nový typ je identifikován jménem `Stock`. Tato deklarace nám dovoluje definovat proměnné typu `Stock` nazývané *objekty* nebo *instance*. Každý objekt reprezentuje jeden podíl. Například deklarace

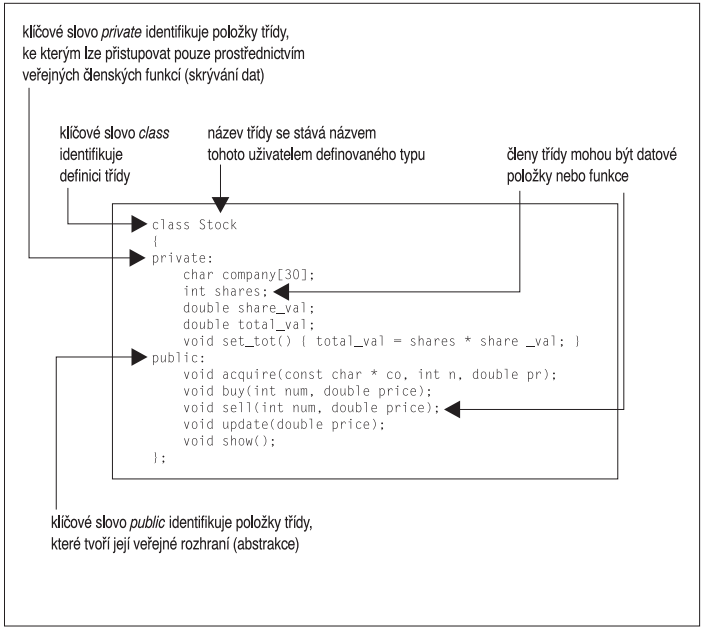
```
Stock sally;
Stock solly;
```

vytvoří dva objekty typu `Stock` nazvané `sally` a `solly`. Objekt `sally` by mohl například obsahovat informace o podílu `Sally` v určitém podniku.

Dále si všimněte, že ukládané informace se objevují ve formě datových položek třídy, například `company` a `shares`. Položka `company` objektu `sally` například obsahuje název firmy, položka `share` obsahuje počet vlastněných akcií, položka `share_val` udává cenu jedné akcie a položka `total_val` obsahuje celkovou cenu všech akcií, které `Sally` vlastní. Podobně i námi zvolené operace se objevují jako členské funkce třídy, například funkce `sell()` a `update()`. Členské funkce třídy se také nazývají metody třídy. Členská funkce může být definována přímo v deklaraci třídy, jako například funkce `set_tot()`, nebo může být reprezentována prototypem funkce tak jako ostatní funkce této třídy. Úplná definice metod přichází později, ale prototypy stačí k popisu funkčního rozhraní třídy. Spojení dat a metod do jednoho celku je největším přínosem třídy. Díky návrhu je vytvoření objektu typu `Stock` zároveň vytvořením pravidel pro jeho používání.

Již dříve jste viděli, že třídy `istream` a `ostream` mají členské funkce jako `get()` nebo `getline()`. Prototypy funkcí v definici třídy `Stock` ukazují, jak se členské funkce vytvářejí. Hlavičkový soubor `iostream` například obsahuje v deklaraci třídy `istream` prototyp funkce `getline()`.

Novinkou jsou také klíčová slova `public` a `private`. Tato návěští popisují přístupová práva k položkám třídy. Jakýkoli program používající objekt dané třídy může přímo přistupovat k položkám z části `public`. K soukromým datům třídy může přistupovat pouze pomocí veřejných členských funkcí (nebo, jak uvidíte v kapitole 11 „Práce s třídami," pomocí



Obrázek 10.1 Třída Stock

spřátelené funkce). Například jediným způsobem, jak změnit položku `shares` třídy `Stock`, je použití některé veřejné členské funkce. Tudíž veřejné členské funkce se chovají jako prostředníci mezi programem a privátní částí objektu, nabízejí rozhraní mezi programem a objektem. Izolace dat od přímého přístupu z programu se nazývá skrývání dat. (C++ nabízí třetí klíčové slovo pro omezení přístupu k položkám třídy, `protected`, o kterém budeme hovořit v kapitole 13 „Dědičnost tříd.“) Prohlédněte si obrázek 10.1. Zatímco ukrývání dat může být bezohledné v rámci obchodu s akciemi, v programování se jedná o dobrý zvyk, protože zachovává integritu dat.

Návrh třídy se snaží oddělit veřejné rozhraní od implementačních detailů. Veřejné rozhraní reprezentuje abstraktní část návrhu třídy. Sloučení implementačních detailů a jejich oddělení od abstrakce se nazývá *zapouzdření*. *Skrývání dat* (uložení dat v soukromé části třídy) je příkladem zapouzdření, stejně jako oddělení funkčních detailů do privátní části, jak to dělá třída `Stock` v případě metody `set_tot()`. Jiným příkladem zapouzdření je běžný postup oddělení deklarace třídy a definice jejích metod do oddělených souborů.

OOP a jazyk C++

Objektově orientované programování je styl programování, který lze do jisté míry použít u jakéhokoli jazyka. Zajisté můžete mnoho idejí OOP přenést do jazyka C. Například kapitola 9 „Příběhy ve funkcích" nabízí příklad (výpisy 9.12, 13, 14), ve kterém hlavičkový soubor obsahuje definici struktury společně s prototypy funkcí, pracujících s touto strukturou. Funkce `main()` jednoduše definuje proměnné tohoto typu a používá uvedené funkce pro práci s těmito proměnnými; funkce `main()` nepřistupuje přímo k položkám struktury. V podstatě tento příklad definuje abstraktní datový typ a ukládá definici struktury společně s prototypy funkcí do hlavičkového souboru, čímž skrývá implementaci dat. Nicméně jazyk C++ obsahuje vlastnosti navržené speciálně pro implementaci přístupu OOP, takže vám umožňuje jít o několik kroků dále než jazyk C. Za prvé, uložení reprezentace dat a prototypů funkcí do jedné deklarace třídy namísto do souboru sjednocuje popis třídy tím, že všechny informace jsou uloženy do jedné deklarace třídy. Za druhé, přesunutí dat do privátní oblasti třídy zajistí, že přístup k nim bude možný pouze pomocí autorizovaných funkcí. Jestliže v našem příkladu z jazyka C bude funkce `main()` přistupovat přímo k položkám struktury, překročí sice pravidla OOP, ale neporuší žádná pravidla jazyka C. Ale pokus o přímý přístup například k položce `shares` objektu `Stock` porušuje pravidla jazyka C++ a překladač tento pokus odhalí.

Všimněte si, že skrývání dat nejenže vám brání v přímém přístupu k položkám třídy, ale také vás zprošťuje potřeby znát reprezentaci dat. Například metoda `show()` zobrazí celkovou hodnotu podílu. Tato hodnota může být uložena v rámci objektu, jak je tomu v našem příkladu, nebo může být vypočítána až v době použití. Z pohledu uživatele v tom není žádný rozdíl. Potřebujete pouze znát funkce jednotlivých metod třídy, to znamená počet a typ jejích parametrů a její návratovou hodnotu. Princip spočívá v oddělení implementace od návrhu rozhraní. Jestliže později najdete lepší způsob reprezentace dat nebo implementace metod, můžete tyto detaily změnit bez nutnosti změny rozhraní třídy, což velmi zjednodušuje údržbu programu.

Řízení přístupu k členům: Veřejné nebo privátní?

Položky třídy, ať už se jedná o datové položky nebo funkce, můžete uvést buďto v části veřejné nebo privátní. Ale protože jedním z hlavních pravidel OOP je skrývání dat, jsou datové položky třídy téměř výlučně uváděny v privátní části třídy. Členské funkce tvořící rozhraní třídy se zapisují ve veřejné části třídy, jinak byste je nemohli volat z vnějšku třídy. Jak ukazuje deklarace třídy `Stock`, můžete také členské funkce uvést v privátní části. Takové funkce nemůžete volat přímo z programu, ale mohou je volat veřejné metody. Běžně se privátní funkce používají pro správu implementačních detailů, které nejsou součástí veřejného rozhraní.

Klíčové slovo `private` nemusíte v deklaraci třídy používat explicitně, protože se jedná o implicitní modifikátor přístupu k položkám objektu:

```
class World
{
   float mass;        // implicitně private
   char name[20];     // implicitně private
public:
   void tellall(void);
   ...
};
```

My však klíčové slovo `private` explicitně zapisovat budeme, abychom zdůraznili koncept skrývání dat.

> **Třídy a struktury**
>
> **Popisy tříd se v mnohém podobají deklaracím struktur, obsahují však navíc členské funkce a návěští `public` a `private`, určující viditelnost. Jazyk C++ vlastně rozšiřuje struktury o stejné vlastnosti, jako mají třídy. Jediným rozdílem je, že implicitním modifikátorem přístupu je v případě struktury přístup `public`, zatímco u třídy je to přístup `private`. Programátoři v jazyku C++ většinou používají třídy k implementaci popisů tříd, zatímco omezující struktury používají k reprezentaci čistě datových objektů nebo někdy i k reprezentaci třídy bez privátních složek.**

Implementace členských funkcí třídy

Zatím nám stále zbývá dodělat druhou část specifikace třídy: musíte doplnit kód členských funkcí, které jsou v deklaraci reprezentovány prototypy funkcí. Definice členských funkcí jsou velmi podobné definicím běžných funkcí. Obsahují hlavičku a tělo funkce. Mohou mít návratový typ a parametry. Ale mají také dvě zvláštní charakteristiky:

- Při definici členské funkce používáte operátor rozsahu platnosti (`::`) k určení třídy, do které funkce patří.
- Metody třídy mohou přistupovat ke složkám třídy typu `private`.

Podívejme se nyní na tyto dva body.

Za prvé, v hlavičce členské funkce použijete operátor rozsahu platnosti (`::`) k určení třídy, do které funkce patří. Například hlavička členské funkce `update()` by mohla vypadat takto:

```
void Stock::update(double price)
```

Tato notace znamená, že definujeme funkci `update()`, kterážto je členskou funkcí třídy `Stock`. Nejedná se pouze o identifikaci funkce `update()` jako funkce členské, ale také to znamená, že stejný název metody můžeme použít v rámci jiné třídy. Například definice metody `update()` třídy `Buffoon` by měla tuto hlavičku:

```
void Buffoon::update()
```

Operátor rozsahu platnosti tedy určuje, ke které třídě metoda patří. Říkáme, že funkce `update()` má *třídní rozsah*. Další metody třídy `Stock` mohou v případě nutnosti použít metodu `update()` bez uvedení operátoru rozlišení. Je to možné proto, že obě metody patří do stejné třídy a metoda `update()` je v dosahu. Použití metody `update()` mimo deklaraci třídy nebo definice metod však vyžaduje zvláštní opatření, ke kterým se brzy dostaneme.

Jeden způsob, jak je možné na názvy metod pohlížet, je ten, že úplný název metody třídy obsahuje název třídy. Říkáme, že název `Stock::update()` je *kvalifikovaný název* metody. Jednoduchý název `update()` je zkrácenou verzí kvalifikovaného názvu, který může být použit pouze v rozsahu třídy.

Další speciální vlastností členských funkcí je možnost přístupu k privátním složkám třídy. Například metoda `show()` by mohla obsahovat takovýto kód:

```
cout << "Spolecnost: " << company
     << " Pocet akcii: " << shares << endl
     << " Cena akcie Kc: " << share_val
     << " Celkova cena Kc: " << total_val << endl;
```

Zde jsou `company`, `shares` a další privátními datovými položkami třídy `Stock`. Jestliže se pokusíte přistoupit k privátním položkám z jiné než členské funkce třídy, překladač tuto chybu odhalí již při překladu. (I když spřátelené funkce popsané v kapitole 11 představují výjimku z tohoto pravidla.)

Jestliže budeme mít na mysli tyto dvě zvláštnosti, můžeme implementovat metody třídy tak, jak je ukázáno ve výpisu 10.2. Tyto definice mohou být uloženy v odděleném souboru, nebo mohou být uloženy společně s deklarací třídy. Nyní v začátcích budeme předpokládat, že jsou definice uloženy společně s deklarací třídy v jednom souboru. Toto je nejjednodušší, i když ne nejlepší metoda, jak zpřístupnit deklaraci třídy definicím metod. (Nejlepším způsobem, jak uvidíme později, je použití hlavičkového souboru pro deklaraci třídy a zdrojového souboru pro definici metod třídy.) Abychom získali se jmennými prostory více zkušeností, v některých metodách používáme kvalifikátor `std::`, v jiných deklaraci `using`.

Výpis 10.2 stocks.cpp

```
//další část stocks.cpp -- implementace metod třídy
void Stock::acquire(const char * co, int n, double pr)
{
   std::strncpy(company, co, 29); // zkrácení co podle company
   company[29] = '\0';
   if (n < 0)
   {
      std::cerr << "Pocet akcii nemuze byt zaporny; "
                << company << " nastaveni shares na 0.\n";
      shares = 0;
   }
```

```
    else
        shares = n;
    share_val = pr;
    set_tot();
}

void Stock::buy(int num, double price)
{
    if (num < 0)
    {
        std::cerr << "Pocet akcii nemuze byt zaporny. "
                  << "Transakce ukoncena.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}

void Stock::sell(int num, double price)
{
    using std::cerr;
    if (num < 0)
    {
        cerr << "Pocet akcii nemuze byt zaporny. "
             << "Transakce ukoncena.\n";
    }
    else if (num > shares)
    {
        cerr << "Nemuzete prodat vic nez mate! "
             << "Transakce ukoncena.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    using std::cout;
    using std::endl;
    cout << "Firma: " << company
```

```
            << " Akcie: " << shares << endl
            << " Cena akcie: Kc " << share_val
            << " Celkova cena: Kc" << total_val << endl;
}
```

Poznámky ke členským funkcím

Funkce `acquire()` obstarává prvotní vytvoření podílu v daném podniku, zatímco funkce `buy()` a `sell()` řídí přidávání a odebírání z existujícího vlastnictví. Metody `buy()` a `sell()` se přesvědčí, zda počet kupovaných nebo prodávaných akcií není záporný. Jestliže se uživatel pokusí prodat více akcií než vlastní, funkce `sell()` ukončí transakci. Způsob zajištění privátnosti dat a omezeného přístupu k veřejným funkcím umožňuje provádět kontrolu nad způsobem použití dat; v tomto případě umožňuje vsunout tato zabezpečení proti chybným transakcím.

Čtyři další členské funkce nastavují nebo obnovují hodnotu položky `total_val`. Místo čtyřnásobného zapisování výpočtu bylo zvoleno volání funkce `set_tot()`. Protože tato funkce se týká implementace kódu a není součástí veřejného rozhraní, je uvedena v privátní části třídy. (To znamená, že `set_tot()` je členská funkce používaná tím, kdo třídu napsal, nikoli však autorem programu, který tuto třídu používá). Pokud by výpočet byl delší, ušetřili byste si tímto zbytečné psaní a rozsah kódu. Zde však hlavní přínos spočívá v tom, že voláním funkce (namísto zapisováním výpočtu) zajistíte, že bude proveden vždy stejný výpočet. Také v případě potřeby úpravy výpočtu (což není v našem příkladu pravděpodobné) budete muset upravit kód pouze na jednom místě.

Metoda `acquire()` používá funkci `strncpy()` k vytvoření kopie řetězce. Jestliže jste již zapomněli, volání funkce `strncpy(s2, s1, n)` zkopíruje nejvýše `n` znaků řetězce `s1` do řetězce `s2`. Pokud řetězec `s1` obsahuje méně než `n` znaků, funkce `strncpy()` doplní řetězec `s2` nulovými znaky až do délky `n`. To znamená, že volání `strncpy(firstname,"Tim",6)` zkopíruje znaky `T`, `i` a `m` do položky `firstname` a přidá tři nulové znaky, aby se dosáhlo délky šesti znaků. Ale jestliže je řetězec s1 delší než n, pak nebudou přidány žádné nulové znaky. To znamená, že v případě volání `strncpy(firstname,"Priscilla",4)` budou zkopírovány pouze znaky `P`, `r`, `i` a `s`, což vytvoří pole znaků, které ale nebude tvořit řetězec, protože neobsahuje na konci nulový znak. Proto funkce `acquire()` přidává nulový znak na konec znakového pole, aby bylo jisté, že se jedná o řetězec.

Objekt cerr

Objekt `cerr` stejně jako objekt `cout` jsou instancemi třídy `ostream`. Rozdíl je v tom, že přesměrování výstupu programu ovlivní objekt `cout`, ale nikoli objekt `cerr`. Objekt `cerr` je používán pro chybové zprávy. Tudíž jestliže přesměrujete výstup programu do souboru a nastane nějaká chyba, bude chybová zpráva stále zobrazována na obrazovce. (V Unixu můžeme přesměrovat `cout` a `cerr` nezávisle. Řádkový operátor `>` přesměruje `cout` a operátor `2>` přesměruje `cerr`).

Vložené metody

Funkce s definicí uvedenou v rámci deklarace třídy se automaticky stává vloženou funkcí. Funkce `Stock::set_tot()` je tedy vloženou funkcí. Deklarace tříd využívají vložené funkce většinou pro krátké funkce, což je případ funkce `set_tot()`.

Pokud chcete, můžete definovat vloženou funkci i mimo deklaraci třídy. Abyste tak učinili, použijte kvalifikátor `inline` v rámci implementace třídy:

```
class Stock
{
private:
   ...
   void set_tot();                    // definice je uvedena zvlášť
public:
   ...
};

inline void Stock::set_tot()          // použití inline v definici
{
   total_val = shares * share_val;
}
```

Protože vložené funkce mají vnitřní vazbu, jsou známy pouze v souboru, v němž jsou deklarovány. Nejjednodušším způsobem, jak zajistit, aby vložené definice byly přístupné všem souborům v programu, je vložení definice do stejného hlavičkového souboru, ve kterém je odpovídající třída definována. (Některé systémy pro vývoj programů obsahující inteligentní sestavovací program umožňují uvedení vložených definic v separátním souboru.)

Mimochodem, podle *přepisovacího pravidla* je uvedení definice v rámci deklarace třídy ekvivalentní uvedení prototypu v deklaraci třídy a poté zapsání definice vložené funkce ihned za deklarací třídy. To znamená, že původní definice funkce `set_tot()` je ekvivalentní s právě uvedenou definicí.

Který objekt používá metodu?

Nyní přikročíme k jednomu z nejdůležitějších aspektů používání objektů: jak aplikovat metodu na daný objekt. Následující kód

```
shares += num;
```

používá položku `shares` objektu. Ale kterého objektu? To je dobrá otázka! K jejímu zodpovězení se nejdříve zamyslíme nad tím, jak se nějaký objekt vytvoří. Nejjednodušší cestou je definice proměnných typu dané třídy:

```
Stock kate, joe;
```

Tímto způsobem vytvoříte dva objekty třídy `Stock`, jeden se jmenuje `kate` a druhý `joe`.

Dále uvažujte, jakým způsobem použít nějakou členskou funkci jednoho z těchto objektů. Odpovědí je, stejně jako u struktur a jejich položek, použití operátoru pro přístup ke složkám:

```
kate.show();                          // vyvolání členské funkce objektu kate
joe.show();                           // vyvolání členské funkce objektu joe
```

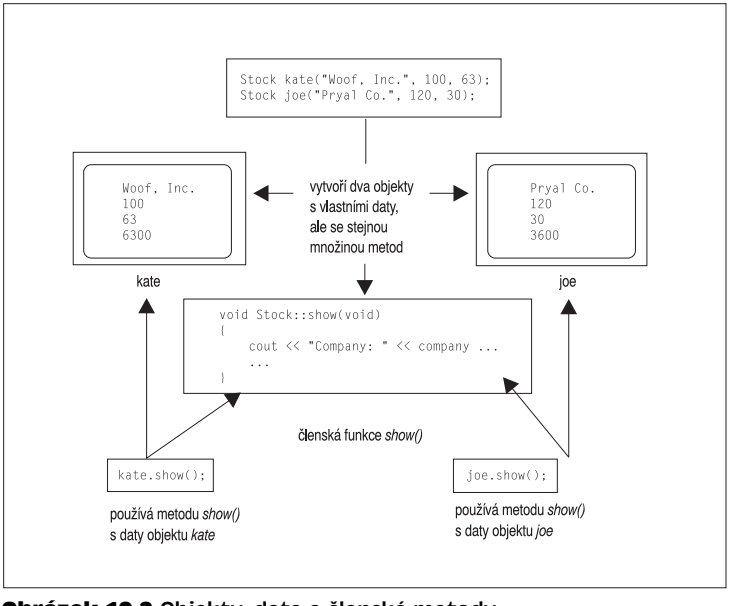
První řádek volá metodu `show()` objektu `kate`. To znamená, že metoda vyhodnotí odkaz na `shares` jako `kate.shares` a odkaz na položku `share_val` jako `kate.share_val`. Podobně volání `joe.show()` způsobí, že metoda `show()` interpretuje odkazy `shares` a `share_val` jako `joe.shares` a `joe.share_val`.

ZAPAMATUJTE SI

Když zavoláte členskou funkci objektu, používá tato funkce datové položky objektu, který členskou funkci vyvolal.

Podobně volání kate.`sell()` použije metodu `set_tot()`, jako kdyby se jednalo o volání `kate.set_tot()`, což způsobí, že data budou brána z objektu `kate`.

Každý nově vytvořený objekt obsahuje místo pro uložení svých vnitřních proměnných, členů třídy. Ale všechny objekty náležející ke stejné třídě sdílí stejnou množinu metod a každá metoda existuje pouze v jedné kopii. Předpokládejme například, že `kate` a `joe` jsou objekty třídy `Stock`. Potom `kate.shares` zabírá část paměti a `joe.shares` zabírá jinou část paměti. Ale obě volání `kate.show()` a `joe.show()` volají stejnou metodu, to znamená, že provádí stejný úsek kódu. Pouze aplikují stejné metody na jiná data. Volání členské funkce je v některých OOP jazycích nazýváno *zaslání zprávy*. Tedy zaslání stejné zprávy dvěma odlišným objektům spustí stejnou metodu, ta je ale aplikována na dva různé objekty. (Viz obrázek 10.2.)



Obrázek 10.2 Objekty, data a členské metody

Jak se používají třídy

Dosud jste se v této kapitole dozvěděli, jak definovat třídu a její metody. Dalším krokem je napsání programu, který vytvoří a bude používat objekty nějaké třídy. Cílem jazyka C++ je maximálně přiblížit používání tříd k používání základních typů, jako jsou `int` a `char`. Objekt dané třídy můžete vytvořit buď pomocí deklarace proměnné daného typu nebo pomocí operátoru `new` pro alokaci nové instance třídy. Objekty můžete předávat jako parametry, vracet v návratových hodnotách funkcí nebo přiřazovat jeden objekt druhému. C++ nabízí prostředky pro inicializaci objektu, učí objekty `cout` a `cin` tyto objekty

rozeznávat a dokonce nabízí prostředky pro automatickou konverzi typů objektů náležejících k podobným třídám. Bude chvíli trvat, než budete moci všechno toto dělat. Ale začněme od jednodušších vlastností. Již jste viděli, jak se deklaruje objekt a jak se volá jeho členská funkce. Výpis 10.3 kombinuje tyto techniky s deklarací třídy a definicí členských funkcí k vytvoření kompletního programu. Program vytvoří objekt třídy `Stock` pojmenovaný `stock1`. Program je jednoduchý, ale testuje všechny vlastnosti, jež jsme do naší třídy vložili.

Výpis 10.3 úplný program stocks.cpp

```
// stocks.cpp -- celý program
#include <iostream>
#include <cstring>

class Stock                          // deklarace třídy
{
private:
   char company[30];
   int shares;
   double share_val;
   double total_val;
   void set_tot() { total_val = shares * share_val; }
public:
   void acquire(const char * co, int n, double pr);
   void buy(int num, double price);
   void sell(int num, double price);
   void update(double price);
   void show();
};                                   // středník na konci!

void Stock::acquire(const char * co, int n, double pr)
{
   std::strncpy(company, co, 29); // zkrácení co podle company
   company[29] = '\0';
   if (n < 0)
   {
      std::cerr << "Pocet akcii nemuze byt zaporny; "
                << company << " nastaveni shares na 0.\n";
      shares = 0;
   }
   else
      shares = n;
   share_val = pr;
   set_tot();
}

void Stock::buy(int num, double price)
{
   if (num < 0)
   {
      std::cerr << "Pocet nakupovanych akcii nemuze byt zaporny. "
                << "Transakce ukoncena.\n";
   }
```

```
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}

void Stock::sell(int num, double price)
{
    using std::cerr;
    if (num < 0)
    {
        cerr << "Pocet prodavanych akcii nemuze byt zaporny. "
             << "Transakce ukoncena.\n";
    }
    else if (num > shares)
    {
        cerr << "Nemuzete prodat vic akcii nez mate! "
             << "Transakce ukoncena.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    using std::cout;
    using std::endl;
    cout << "Spolecnost: " << company
         << "  Akcie: " << shares << endl
         << "  Cena akcie: $" << share_val
         << "  Celkova hodnota: $" << total_val << endl;
}

int main()
{
    using std::cout;
    using std::ios_base;
    Stock stock1;
    stock1.acquire("NanoSmart", 20, 12.50);
    cout.setf(ios_base::fixed);              // #.## formát
    cout.precision(2);                       // #.## formát
```

```
    cout.setf(ios_base::showpoint);    // #.## formát
    stock1.show();
    stock1.buy(15, 18.25);
    stock1.show();
    stock1.sell(400, 20.00);
    stock1.show();
    return 0;
}
```

Program z výpisu 10.3 používá tři formátovací příkazy:

```
cout.setf(ios_base::fixed);           // formát s pevnou desetinnou tečkou
cout.precision(2);                    // dvě místa za desetinnou tečkou
cout.setf(ios_base::showpoint);       // výpis pravostranných nul
```

Také používá deklaraci `using`:

```
using std::ios_base;
```

Zde vidíme případ vnořeného jmenného prostoru. Identifikátory a `fixed` a `showpoint` jsou součástí jmenného prostru `ios_base` a ten je zase součástí jmenného prostoru `std`. Účelem je zobrazení dvou číslic za desetinným oddělovačem včetně koncových nul. Podle současné praxe jsou potřeba pouze první dva příkazy, zatímco některé starší implementace potřebují první a třetí formátovací příkaz. Použití všech tří zajistí stejný výstup ve všech implementacích jazyka. Více se dozvíte v kapitole 16 „Vstup, výstup a soubory“. Zde je výstup programu z výpisu 10.3:

```
Spolecnost: NanoSmart Pocet akcii: 20
 Cena akcie: $12.50 Celkova cena: $250.00
Spolecnost: NanoSmart Pocet akcii: 35
 Cena akcie: $18.25 Celkova cena: $638.75
Nemuzete prodat vic nez mate! Transakce ukoncena.
Spolecnost: NanoSmart  Akcie: 35
 Cena akcie: $18.25  Celkova hodnota: $638.75
```

Všimněte si, že funkce `main()` slouží pouze k otestování třídy `Stock`. Jestliže naše třída funguje podle předpokladů, můžeme nyní třídu `Stock` použít jako uživatelem definovaný typ v jiných programech. Kritickým bodem při používání nového datového typu je pochopení členských funkcí. Implementačními detaily jste se zabývat nemuseli. Přečtěte si následující poznámku, týkající se modelu klient/server.

Model klient/server

OOP programátoři často hovoří o zpracování programu v rámci modelu klient/server. Podle této koncepce je *klientem* program používající třídu. Deklarace třídy společně s definicí jejích metod tvoří *server*, který je zdrojem pro programy, jež jej potřebují. Klient používá server pouze pomocí veřejného rozhraní. To znamená, že jedinou starostí klienta a potažmo jedinou starostí programátora je znalost tohoto rozhraní. V odpovědnosti serveru a tedy zodpovědností tvůrce serveru je zajistit, aby server spolehlivě a přesně prováděl akce v součinnosti se svým rozhraním. Toto programátorům dovoluje vylepšovat klienta nebo server nezávisle na sobě, aniž by změny v serveru měly nepředvídatelné dopady na chování klienta.