Microsoft

Margaréta Cifrová

Základy OOP v jazyku C++

OD študentov
PRE študentov

FAKULTA HOSPODÁRSKEJ INFORMATIKY EU V BRATISLAVE

Microsoft Visual Studio

Autorka: Margaréta Cifrová
Odborný garant: Ing. Ján Hanák, MVP

Základy OOP v jazyku C++
Praktické cvičenie zo série „Od študentov pre študentov“

Charakteristika praktických cvičení zo série „Od študentov pre študentov“

Sme presvedčení o tom, že keď inteligentní mladí ľudia ovládnu najmodernejšie počítačové technológie súčasnosti, ich tvorivý potenciál je vskutku nekonečný. Primárnym cieľom iniciatívy, ktorá stojí za sériou praktických cvičení „Od študentov pre študentov“, je maximalizácia hodnôt ľudských kapitálov študentov ako hlavných členov akademických komunít. Praktické cvičenia zo série „Od študentov pre študentov“ umožňujú študentom využiť ich existujúce teoretické znalosti, pričom efektívnym spôsobom predvádzajú, ako možno tieto znalosti s výhodou uplatniť pri vývoji atraktívnych počítačových aplikácií v rôznych programovacích jazykoch (C, C++, C++/CLI, C#, Visual Basic, F#). Budeme nesmierne šťastní, keď sa našim praktickým cvičeniam podarí u študentov prebudiť a naplno rozvinúť záujem o programovanie a vývoj počítačového softvéru. Veríme, že čím viac sofistikovaných IT odborníkov vychováme, tým viac budeme môcť spoločnými silami urobiť všetko pre to, aby sa z tohto sveta stalo lepšie miesto pre život.

Základy OOP v jazyku C++

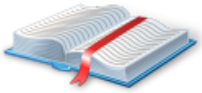
Praktické cvičenie zo série „Od študentov pre študentov“

Cieľové publikum: **študenti**, ovládajúci základy jazyka C++
Vedomostná náročnosť: ■■■□□
Časová náročnosť: **1** hodina a **40** minút
Programovacie jazyky: **C++** (podľa ISO štandardu C++03)
Vývojové prostredia: **Visual C++ 2008 Express**, Visual C++ 2010 Express
Operačné systémy: **Windows Vista**, Windows 7, Windows XP
Technológie: **knižnica jazyka C++**, exekučné prostredie jazyka C++

Programovací jazyk C++ je jedným z najpopulárnejších prostriedkov súčasnosti na vývoj objektovo orientovaných programov. Hoci od jeho vynájdenia uplynulo už viac ako 20 rokov, „inkrementované céčko“ sa teší neutíchajúcej obľube, a to nielen v akademickej, ale aj v komerčnej sfére. Takmer každá vysoká škola informatického zamerania má vo svojich študijných programoch výučbové kurzy, ktoré sa venujú algoritmizácii a vývoji počítačového softvéru v jazyku C++. Jazyk C++ má veľa konkurenčných výhod: je to jazyk strednej úrovne, produktom jeho prekladača sú kompilované programy s natívnym (strojovým) kódom, disponuje intuitívnou syntaxou, napomáha rozvíjať abstraktné myslenie študentov a v neposlednom rade pozitívne ovplyvňuje ďalšie významné programovacie jazyky, najmä C#, C++/CLI a Javu.

V tomto praktickom cvičení vás zoznámime so základmi objektovo orientovaného programovania v jazyku C++. Pri praktickom programovaní budeme využívať vývojové prostredie Microsoft Visual C++ 2008 Express, ktoré je zdarma k dispozícii pre každého záujemcu. Ak Visual C++ 2008 Express nemáte nainštalovaný na svojom počítači, môžete si ho prevziať z nasledujúcich stránok spoločnosti Microsoft: http://www.microsoft.com/express/download/.

Tip: Aby ste mohli z tohto praktického cvičenia vyťažiť maximum, predpokladáme, že ovládate základy neobjektového programovania v jazyku C++. Nebudeme sa teda zaoberať vysvetľovaním elementárnych princípov a programovacích konštrukcií, ako sú premenné, operátory, rozhodovacie príkazy či cykly. Naším cieľom je využiť vašu existujúcu bázu znalostí v záujme ovládnutia základov objektovo orientovaného programovania v jazyku C++.

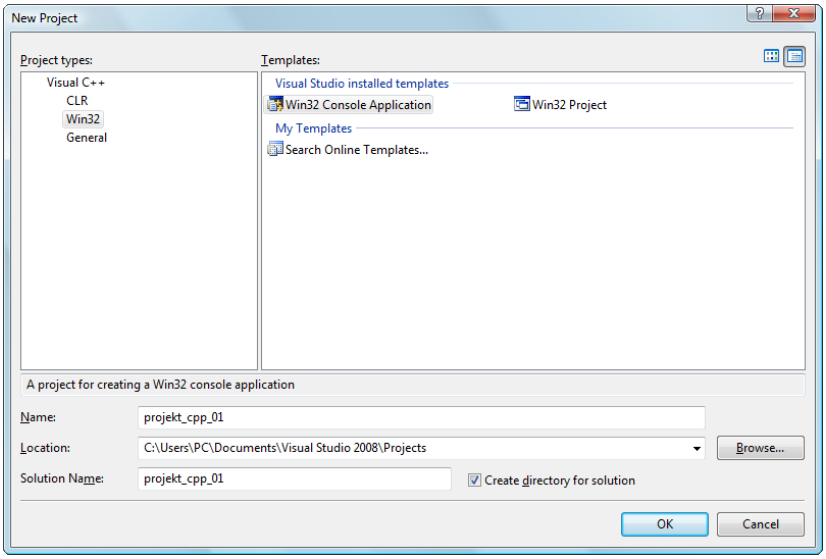


Obsah praktického cvičenia

1 Založenie nového projektu jazyka C++ vo vývojovom prostredí Visual C++ 2008 Express

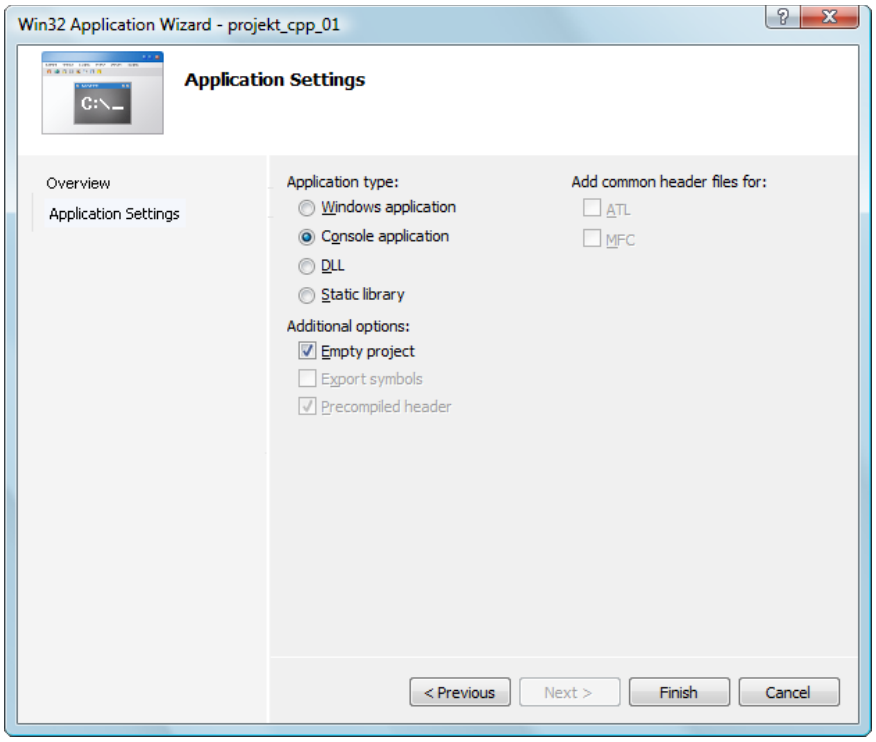
Nový projekt štandardnej konzolovej aplikácie jazyka C++ založíme vo vývojovom prostredí Visual C++ 2008 Express nasledujúcim spôsobom:

1. Na úvodnej stránke **Start Page** klikneme na hypertextový odkaz **Create Project**.
2. V dialógovom okne **New Project** sa zameriame na stromovú štruktúru **Project types**, z ktorej vyberieme položku **Win32**.
3. V sekcii **Templates** zvolíme projektovú šablónu **Win32 Console Application**.
4. Do textového poľa **Name** zadáme názov projektu. (Visual C++ 2008 Express automaticky vyplní aj textové pole **Solution Name**, a to tak, aby malo riešenie rovnaký názov ako projekt, ktorý bude v riešení uložený.) V tejto chvíli by malo dialógové okno **New Project** vyzerať ako na obr. 1.



Obr. 1: Založenie nového projektu štandardnej konzolovej aplikácie jazyka C++

5. Klikneme na tlačidlo **OK**.
6. Spustí sa sprievodca založením projektu štandardnej konzolovej aplikácie **Win32 Application Wizard**. Aktivujeme tlačidlo **Next** a v druhom kroku vyberieme voľbu **Empty project** na založenie nového prázdneho projektu (obr. 2).



Obr. 2: Aktivácia voľby **Empty project** na založenie prázdneho projektu

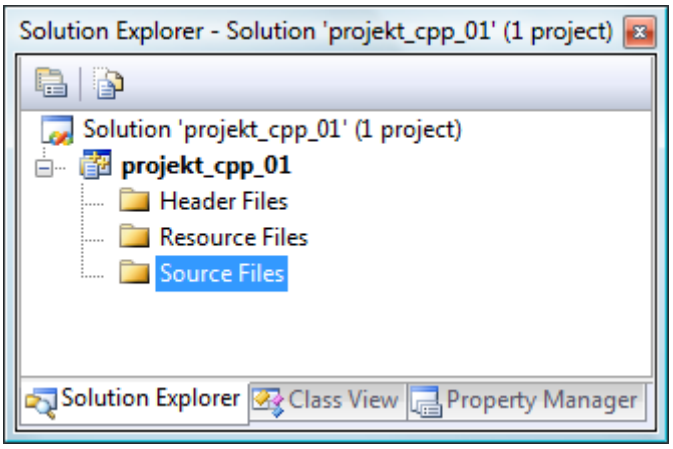
Prázdny projekt je pre nás vyhovujúcim riešením, pretože zdrojový súbor jazyka C++ pridáme sami.

7. Po klepnutí na tlačidlo **Finish** vytvorí sprievodca nový projekt. Keďže je projekt prázdny, musíme doň pridať jeden zdrojový súbor jazyka C++.

2 Pridanie zdrojového súboru jazyka C++ do projektu

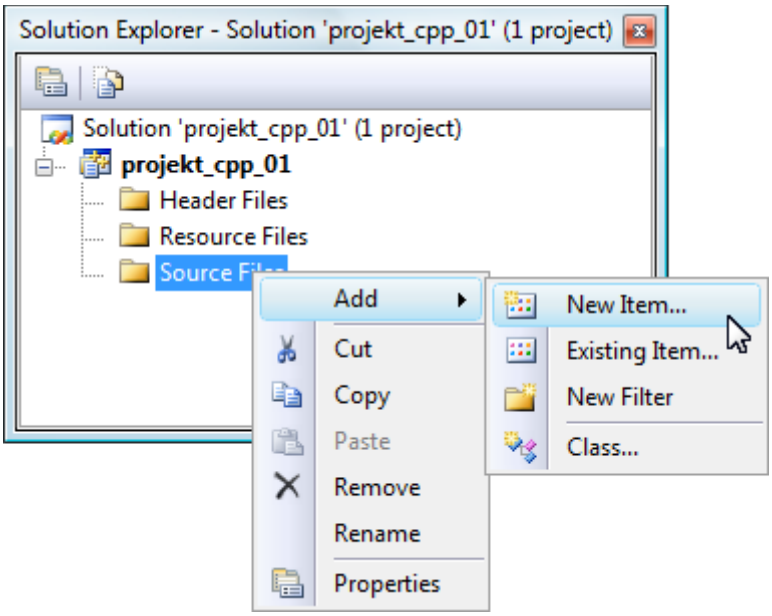
V každom projekte jazyka C++ sa musí nachádzať aspoň jeden zdrojový súbor jazyka C++. V tomto zdrojovom súbore bude uložený zdrojový kód programu jazyka C++. Zdrojový súbor jazyka C++ pridáme do nášho projektu nasledujúcim spôsobom:

1. V podokne **Solution Explorer** (ktoré je štandardne ukotvené pri ľavej strane vývojového prostredia Visual C++ 2008 Express) klikneme pravým tlačidlom myši na priečinok **Source Files** (obr. 3).



Obr. 3: Pridanie zdrojového súboru jazyka C++ – 1. fáza

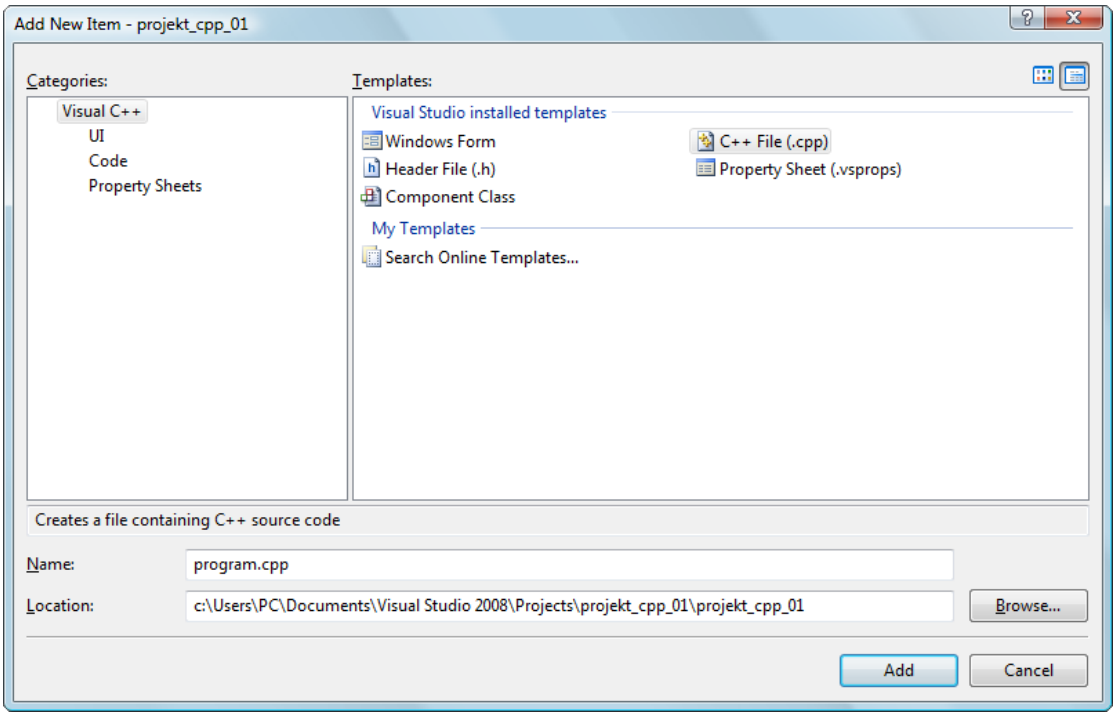
2. Z miestnej ponuky vyberieme príkaz **Add** → **New Item**, čím zviditeľníme dialógové okno na pridanie novej projektovej súčasti (obr. 4).



Obr. 4: Pridanie zdrojového súboru jazyka C++ – 2. fáza

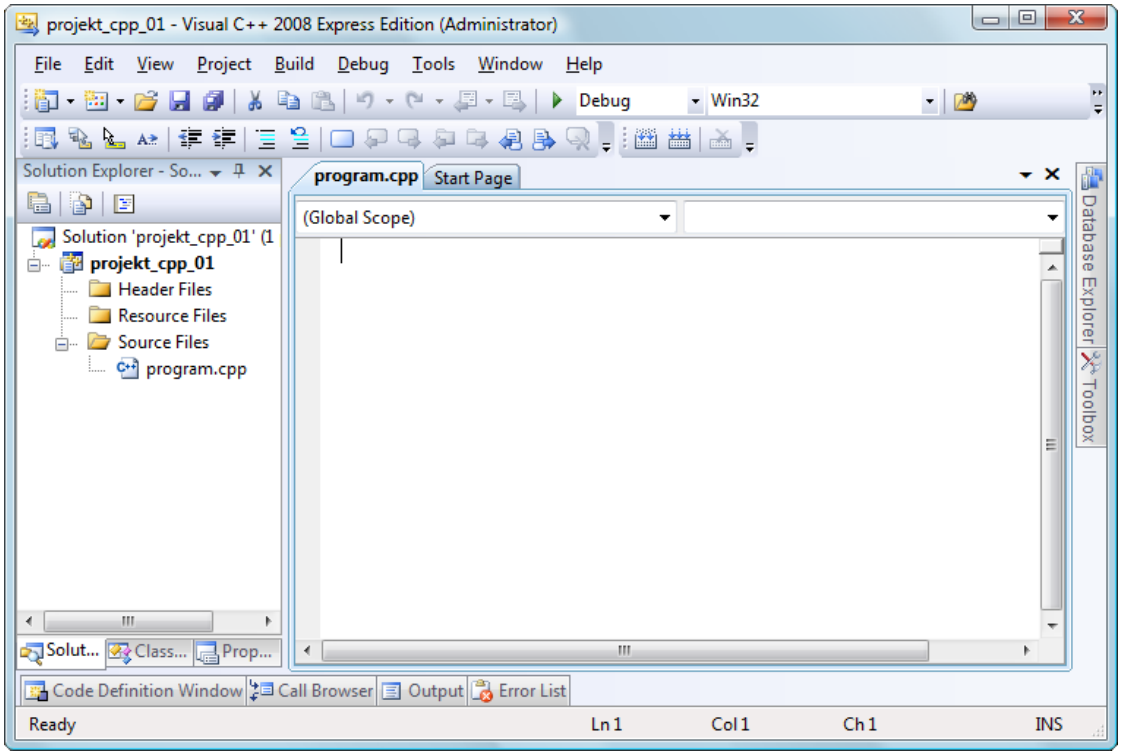
3. V dialógu **Add New Item** zvolíme v sekcii **Templates** súborovú šablónu **C++ File (.cpp)**, ktorá reprezentuje zdrojové súbory jazykov C a C++.

4. Do textového poľa **Name** zapíšeme názov zdrojového súboru jazyka C++. Za názvom zdrojového súboru môže (no nemusí) byť explicitne uvedená aj prípona .cpp, ktorá vraví, že chceme pridať zdrojový súbor jazyka C++. Ak príponu .cpp neuvedieme, Visual C++ 2008 Express bude implicitne predpokladať, že chceme do projektu pridať zdrojový súbor jazyka C++, a preto správnu koncovku použije automaticky (obr. 5).



Obr. 5: Pridanie zdrojového súboru jazyka C++ – 3. fáza

5. Po klepnutí na tlačidlo **Add** sa zdrojový súbor jazyka C++ pridá do projektu. Obsah novo pridaného zdrojového súboru Visual C++ 2008 Express okamžite otvorí v editore zdrojového kódu (obr. 6).



Obr. 6: Zdrojový súbor je pripravený na zápis zdrojového kódu jazyka C++

3 Zostavenie a spustenie programu jazyka C++

Keď do zdrojového súboru zapíšeme zdrojový kód programu v jazyku C++, môžeme zdrojový kód preložiť a program zostaviť. Tak získame priamo spustiteľný súbor so strojovým kódom. Proces zostavenia a spustenia programu ukážeme na testovacom programe, ktorý uskutočňuje aproximovaný výpočet faktoriálu pomocou Stirlingovho vzorca.

Poznámka: Škótsky matematik James Stirling odvodil v roku 1730 nasledujúci vzorec na výpočet aproximovanej hodnoty faktoriálu:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

kde:

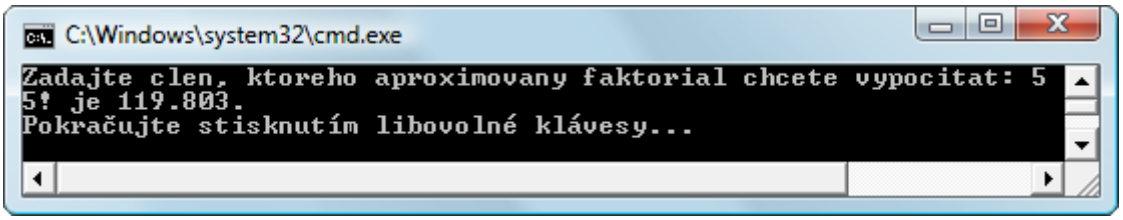
- π je Ludolfovo číslo (jeho hodnota je 3.14).
- e je Eulerova konštanta (jej hodnota je 2.71).

Napr. aproximovaná hodnota 5! sa podľa Stirlingovho vzorca rovná 119,8.

```
#include <iostream>
#include <cmath>

using namespace std;

float AproximovatFaktorial(int clen);

int main()
{
    int clen;
    cout << "Zadajte clen, ktoreho aproximovany faktorial "
         << "chcete vypocitat: ";
    cin >> clen;
    float aprox_faktorial = AproximovatFaktorial(clen);
    cout << clen << "! je " << aprox_faktorial << "." << endl;
    return 0;
}

float AproximovatFaktorial(int clen)
{
    return sqrt(2 * 3.14 * clen) * pow(clen / 2.71, clen);
}
```

Program zostavíme výberom príkazu **Build Solution** z ponuky **Build** (alternatívne môžeme použiť klávesovú skratku F7). Po zostavení možno program spustiť, a to takto: **Debug** → **Start Without Debugging** (ekvivalentne môžeme opäť upotrebiť rýchlejšiu klávesovú skratku CTRL+F5). Výstup programu sa zobrazí v okne príkazového riadka (obr. 7).

Tip: Keď budeme program spúšťať z vývojového prostredia produktu Visual C++ 2008 Express pomocou príkazu **Start Without Debugging**, vývojové prostredie ponechá zobrazené okno príkazového riadka aj po skončení činnosti programu. To je pre nás výhodné, pretože môžeme skontrolovať, či program produkuje správne výstupy.



Obr. 7: Výstup programu na výpočet hodnoty $n!$ pomocou Stirlingovho vzorca

4 Trieda ako abstraktný objektový dátový typ

Trieda je v jazyku C++ abstraktným objektovým dátovým typom, ktorý charakterizuje dáta a činnosti objektov, ktoré z triedy vzniknú. Na triedu môžeme nahliadať ako na šablónu, podľa ktorej budeme vytvárať objekty určitého typu. Skôr, ako budeme môcť použiť triedu na zakladanie objektov, musíme v zdrojovom kóde jazyka C++ špecifikovať deklaráciu triedy.

Triedu deklarujeme podľa nasledujúceho všeobecného syntaktického modelu:

```
class T
{
    private:
        // Súkromná sekcia triedy s definíciami dátových atribútov.
        DA1; DA2; ... DAN;
    public:
        // Verejná sekcia triedy s definíciami metód.
        M1(...) { ... }
        M2(...) { ... }
        ...
        MN(...) { ... }
};
```

kde:

- **T** je identifikátor triedy.

- **DA_1; DA_2; ... DA_N sú dátové atribúty**. Dátové atribúty sú premenné triedy určené na uchovanie dát jej budúcich inštancií. Dátové atribúty sú definované v súkromnej sekcii triedy, ktorej začiatok je vyznačený prístupovým modifikátorom **private**.

- **M_1, M_2, ... M_N sú metódy**. Metódy sú členské funkcie triedy, ktoré determinujú správanie jej budúcich inštancií. Metódy reprezentujú činnosti, ktoré budú inštancie triedy schopné vykonávať. Metódy sú definované vo verejnej sekcii triedy, ktorá je vymedzená prístupovým modifikátorom **public**.

Dôležité: Deklarácia triedy sa v jazyku C++ skladá z hlavičky triedy a tela triedy. V hlavičke triedy je vždy uvedené kľúčové slovo **class**, ktoré dáva prekladaču jazyka C++ na známosť, že sa chystáme deklarovať novú triedu. Každá trieda má svoj názov, ktorý sa objavuje v jej hlavičke okamžite za kľúčovým slovom **class**. Pre identifikátor

triedy platia rovnaké nomenklatúrne pravidlá ako pre identifikátory iných programových entít jazyka C++. Identifikátor triedy sa nesmie začínať číslicou, nemôže obsahovať medzery a rovnako nie je prípustné, aby sa zhodoval s niektorým z existujúcich kľúčových slov jazyka C++.

Telo triedy je ohraničené programovým blokom, ktorého dĺžka je vizuálne vytýčená zloženými zátvorkami ({}). V tele triedy sa vyskytujú členy triedy. K členom triedy patria predovšetkým dátové atribúty triedy a metódy triedy. Syntakticky sú dátové atribúty reprezentované premennými, ktorých oblasťou platnosti je telo triedy. Všetky dátové atribúty sa nachádzajú v súkromnej sekcii triedy. To znamená, že dátové atribúty sú súkromné, čo je v poriadku, pretože rešpektujeme jeden zo základných princípov objektovo orientovaného programovania, ktorým je ukrývanie dát. Objekt (inštancia triedy) teda ukrýva dáta, ktoré obsahuje, čím zabezpečuje ochranu ich integrity. Na druhej strane, metódy pôsobia ako verejné členské funkcie triedy.

Tip: V záujme zachovania optimálnej distribúcie pracovného zaťaženia naprieč metódami triedy sa ich snažíme projektovať tak, aby každá metóda vykonávala práve jednu činnosť, resp. aby každá metóda implementovala práve jeden algoritmus. Metódy sú situované vo verejnej sekcii triedy, pretože chceme, aby ich mohli klienti budúcich inštancií tejto triedy priamo volať. Volaním metód budú môcť klienti využívať služby, ktoré im objekty triedy budú poskytovať.

Ako vidíme, dátové atribúty a metódy triedy sú logicky zoskupené do jej tela. Tento princíp logickej kompozície je v objektovo orientovanom programovaní známy ako zapuzdrenie. Trieda v sebe zapuzdruje dátové atribúty a metódy. Podobne budú dátové atribúty a metódy zapuzdrovať aj inštancie triedy, ktoré vygenerujeme podľa deklarácie triedy.

Všetky členy, ktoré sa v tele triedy nachádzajú, sú implicitne súkromné (nie sú prístupné pre iné entity mimo tela triedy).

Deklarácia triedy je v jazyku C++ príkazom, a preto je zakončená bodkočiarkou.

Uveďme praktickú deklaráciu triedy, ktorá charakterizuje matematickú entitu – trojrozmerný (3D) vektor:

```
// Deklarácia triedy.
class Vektor3D
{
private:
    // Definície súkromných dátových atribútov triedy.
   int x, y, z;
public:
    // Definícia verejného parametrického konštruktora triedy.
    Vektor3D(int zlozka_x, int zlozka_y, int zlozka_z)
    {
        x = zlozka_x;
        y = zlozka_y;
        z = zlozka_z;
    }
```

```
    // Definícia verejnej metódy triedy.
    double ZistitVelkost()
    {
        return sqrt((double)(x * x  + y * y + z * z));
    }
};
```

Komentár k zdrojovému kódu jazyka C++: Deklarovaná trieda disponuje identifikátorom **Vektor3D**. Keďže trojrozmerný vektor je ako abstraktný matematický objekt definovaný prostredníctvom troch zložiek, tak v tele triedy definujeme rovnaký počet súkromných dátových atribútov. Tieto atribúty majú identifikátory **x**, **y** a **z** a sú premennými celočíselného typu **int**. Každá budúca inštancia triedy **Vektor3D** bude obsahovať trojicu celočíselných dátových atribútov, ktoré budú uchovávať x-ovú, y-ovú a z-ovú zložku 3D vektora.

Vo verejnej sekcii triedy sa nachádzajú dve metódy. Prvou z nich je konštruktor, ktorý je verejne prístupný a parametrický. Konštruktor je špeciálna metóda triedy, ktorá slúži na inicializáciu inštancie triedy po jej alokácii. Povedané inak, úlohou konštruktora je uviesť alokovanú inštanciu triedy do východiskového, a teda okamžite použiteľného stavu. Ako „okamžite použiteľný“ označujeme stav, kedy môže inštancia triedy začať poskytovať služby svojim klientom.

Poznámka: Z technického hľadiska je konštruktor členskou funkciou triedy, ktorá má rovnaký identifikátor ako trieda, v ktorej tele je konštruktor umiestnený. Konštruktor môže byť buď implicitný, alebo explicitný. Náš konštruktor je explicitný, pretože sme jeho definíciu sami zapísali do tela triedy. Ak by sme však do tela triedy nezaviedli definíciu žiadneho konštruktora, prekladač jazyka C++ by automaticky vygeneroval svoj vlastný, tzv. implicitný konštruktor. (Implicitný konštruktor je bezparametrický a má prázdne telo.) Naopak, ak prekladač zistí, že v tele triedy sa už objavuje explicitný konštruktor, žiaden implicitný konštruktor nebude automaticky vytvárať. Bez ohľadu na to, či je konštruktor implicitný, alebo explicitný, nikdy nepracuje s návratovou hodnotou (túto skutočnosť vyjadrujeme jednoducho vynechaním typu návratovej hodnoty).

Explicitný konštruktor je v našom prípade parametrický, pričom definuje 3-prvkovú množinu formálnych parametrov. V tele konštruktora sú uvedené priraďovacie príkazy, ktoré vykonávajú inicializáciu dátových atribútov inštancie triedy.

Pri zániku založenej inštancie triedy je vhodné spätne uvoľniť využitú operačnú pamäť. Na to slúži špeciálna členská funkcia triedy nazývaná deštruktor. K jej aktivácii dochádza tesne pred tým, ako nastane dealokácia inštancie triedy. Zjednodušene povedané, deštruktor je opakom konštruktora. V tele triedy ho spoznáme podľa typickej vlnovky (~), ktorá sa nachádza v jeho hlavičke:

```
// Definícia deštruktora.
~T() { ... }
```

Deštruktor nevracia návratovú hodnotu a nedefinuje žiadne formálne parametre (je vždy bezparametrický). Nesmie byť ani preťažený, to znamená, že v jednej triede sa vyskytuje práve jeden deštruktor.

Každý 3D vektor, ktorý bude inštanciou našej triedy, bude vedieť vypočítať svoju veľkosť. Túto funkcionalitu zabezpečuje verejne prístupná bezparametrická metódy triedy s názvom **ZistitVelkost**. Ako si môžeme všimnúť, v tele metódy sa nachádza len jeden agregovaný príkaz. Tento príkaz najskôr vypočíta veľkosť 3D vektora, a potom ju vráti späť ako návratovú hodnotu metódy. (Podotknime, že veľkosť vektora vypočítame ako druhú odmocninu zo súčtu druhých mocnín jednotlivých zložiek vektora.)

Deklarácia triedy je významná, pretože podľa nej prekladač jazyka C++ vie, akými charakteristikami trieda oplýva. No deklarácia samotná ešte zakladá žiadne objekty, teda žiadne inštancie triedy. Keď budeme chcieť vytvoriť inštanciu triedy, budeme musieť triedu inštanciovať. Inštanciácia triedy je proces, ktorého finálnym efektom je zostrojenie inštancie triedy. Založená inštancia triedy je virtuálnym objektom, ktorý bude žiť v operačnej pamäti počítača. Tento virtuálny objekt však dokáže inteligentne plniť požiadavky svojich klientov.

5 Inštanciácia triedy v jazyku C++

V jazyku C++ existujú 3 varianty inštanciácie triedy:

1. Automatická inštanciácia triedy.
2. Statická inštanciácia triedy.
3. Dynamická inštanciácia triedy.

Všetky spomenuté inštanciačné varianty budeme podrobne charakterizovať v nasledujúcich podkapitolách.

5.1 Automatická inštanciácia triedy v jazyku C++

Pri automatickej inštanciácii triedy je inštancia triedy alokovaná v zásobníku primárneho programového vlákna fyzického procesu programu jazyka C++.

Generický syntaktický model **automatickej inštanciácie triedy** je v jazyku C++ takýto:

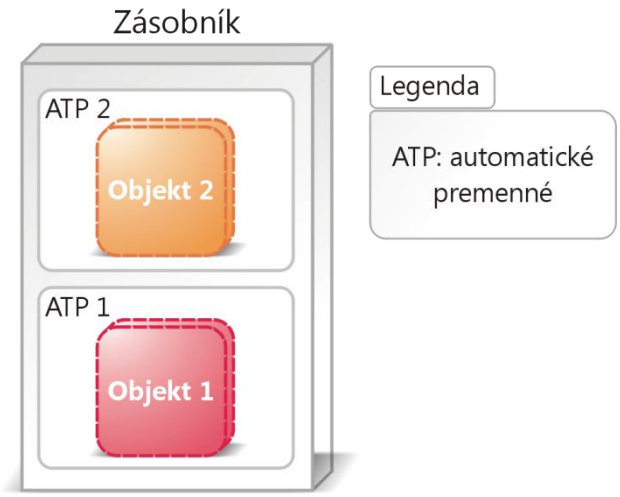
```
T a_premenna;

– alebo –

T a_premenna(a1, a2, ... an);
```

kde:

- **T** je identifikátor triedy.
- **a_premenna** je identifikátor automatickej premennej, v ktorej je uložená vygenerovaná automatická inštancia triedy **T**.

- **a_1, a_2, ... a_n** sú argumenty, ktoré sú odovzdané parametrickému konštruktoru triedy **T**.

Pri automatickej inštanciácii triedy sa vytvorí v zásobníku automatická premenná, do ktorej sa uloží vygenerovaná automatická inštancia triedy **T**. Inštancia triedy je iniicalizovaná samočinne volaným konštruktorom, ktorý je buď bezparametrický, alebo parametrický. Grafický model automatickej inštanciácie triedy ukazuje obr. 8. Praktickou implikáciou tohto inštanciačného variantu je skutočnosť, že automatická inštancia triedy **T** je pre nás prístupná len prostredníctvom automatickej premennej (pretože práve v tejto premennej je automatická inštancia triedy uskladnená).



Obr. 8: Grafický model automatickej inštanciácie triedy v jazyku C++

Konkrétny syntaktický model automatickej inštanciácie triedy **Vektor3D** je takýto:

```
// Automatická inštanciácia triedy Vektor3D.
Vektor3D vektor(1, 4, 7);
```

5.2 Statická inštanciácia triedy v jazyku C++

Pri statickej inštanciácii triedy je inštancia triedy alokovaná v statickej pamäťovej oblasti fyzického procesu programu jazyka C++.

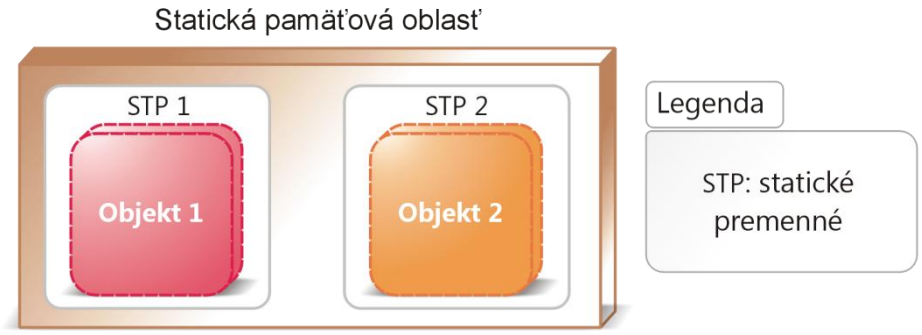
Generický syntaktický model **statickej inštanciácie triedy** je v jazyku C++ takýto:

```
static T s_premenna;

– alebo –

static T s_premenna(a1, a2, ... an);
```

kde:

- **static** je modifikátor, ktorý iniciuje tvorbu statického objektu.
- **T** je identifikátor triedy.
- **s_premenna** je identifikátor statickej premennej, v ktorej je uložená statická inštancia triedy **T**.
- **a_1, a_2, ... a_n** sú argumenty, ktoré sú odovzdané parametrickému konštruktoru triedy **T**.

Pri statickej inštanciácii triedy **T** je zostrojená statická inštancia uložená do statickej premennej. Táto statická premenná je zase situovaná v statickej pamäťovej oblasti programu jazyka C++. Dodajme, že statická inštancia triedy má spravidla dlhší životný cyklus ako automatická inštancia triedy. Po svom vytvorení žije statická inštancia tak dlho, ako beží program, ktorý ju vytvoril. Grafický model statickej inštanciácie triedy ukazuje obr. 9.



Obr. 9: Grafický model statickej inštanciácie triedy v jazyku C++

Konkrétny syntaktický model statickej inštanciácie triedy **Vektor3D** má nasledujúci tvar:

```
// Statická inštanciácia triedy Vektor3D.
static Vektor3D vektor(1, 4, 7);
```

5.3 Dynamická inštanciácia triedy v jazyku C++

Pri dynamickej inštanciácii triedy je inštancia triedy alokovaná v natívnej halde fyzického procesu programu jazyka C++.

Generický syntaktický model **dynamickej inštanciácie triedy** je v jazyku C++ takýto:

```
T *p_premenna = new T;

– alebo –

T *p_premenna = new T(a1, a2, ... an);
```

kde:

- **T** je identifikátor triedy.
- ***** je unárny dereferenčný operátor.
- **p_premenna** je identifikátor smerníkovej premennej.
- **new** je inštanciačný operátor, ktorý vytvára dynamickú inštanciu triedy **T**.
- $\mathbf{a_1}$, $\mathbf{a_2}$, ... $\mathbf{a_n}$ sú argumenty, ktoré sú odovzdané parametrickému konštruktoru triedy **T**.

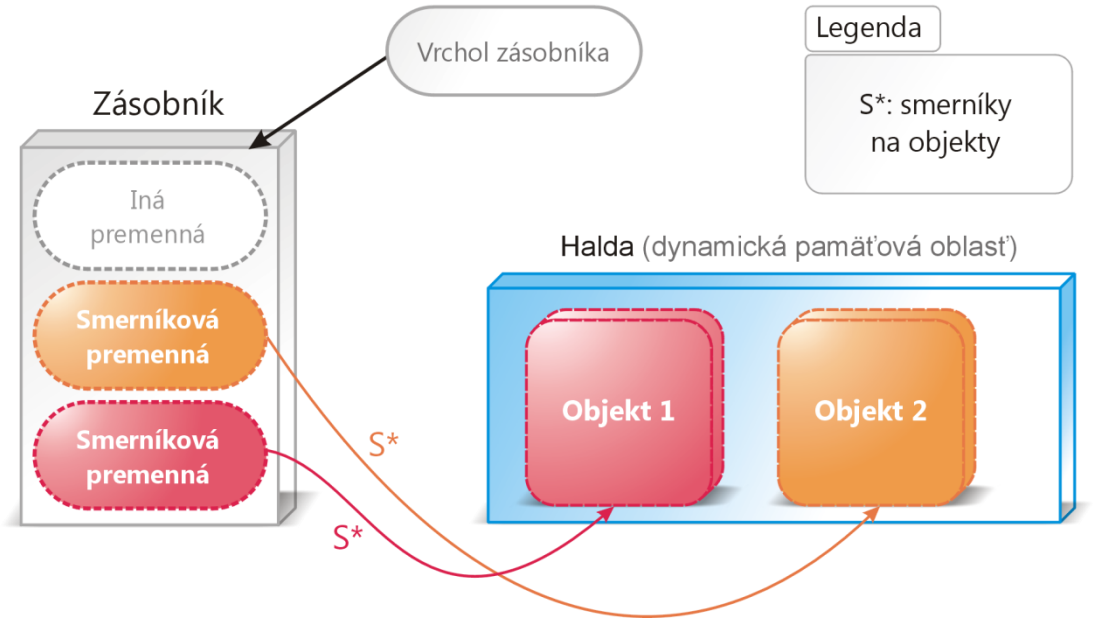
Príkaz, ktorý zostrojuje dynamickú inštanciu triedy **T**, je syntakticky zložitejší ako predchádzajúce inštanciačné príkazy, ktoré vytvárali automatické a statické objekty svojich tried. Uvedený príkaz je priraďovacím príkazom, a preto ho budeme pre ľahšie pochopenie segmentovať podľa nasledujúceho modelu:

$V_1 = V_2$;

- $\mathbf{V_1}$ je výraz naľavo od =. Toto je definičný výraz, ktorý vytvára smerníkovú premennú s identifikátorom **p_premenna**. Do tejto smerníkovej premennej budeme môcť uložiť smerník, ktorý určuje pozíciu dynamickej inštancie triedy **T** v halde.

- $\mathbf{V_2}$ je výraz napravo od =. Toto je inicializačný výraz. Keďže sa v inicializačnom výraze vyskytuje inštanciačný operátor **new**, často výraz $\mathbf{V_2}$ označujeme ako inštanciačný výraz. Operátor **new** alokuje inštanciu triedy **T** v halde, inicializuje ju volaním konštruktora (implicitného bezparametrického konštruktora alebo explicitného bezparametrického konštruktora, resp. explicitného parametrického konštruktora), a napokon vracia smerník na vygenerovanú inštanciu triedy **T** (čiže hodnota výrazu $\mathbf{V_2}$ je dátového typu **T***).

Vizuálny model dynamickej inštanciácie triedy ukazuje obr. 10.



Obr. 10: Grafický model dynamickej inštanciácie triedy v jazyku C++

Objekt, ktorý vzniká pri dynamickej inštanciácii triedy, je anonymný, pretože nemá žiadne symbolické pomenovanie. S objektom môžeme pracovať len pomocou smerníka, ktorý nám poskytne operátor **new**. Vrátený smerník je pre nás natoľko cenným zdrojom, že ho vždy ukladáme do príslušnej smerníkovej premennej. Ak je smerníková premenná definovaná štandardne v tele funkcie bez použitia akýchkoľvek modifikátorov, tak ide o automatickú lokálnu smerníkovú premennú, ktorá bude sídliť v zásobníku. Jej životný cyklus bude riadený samočinne, pričom premenná zanikne automaticky po skončení činnosti funkcie, v tele ktorej je situovaná.

Na druhej strane, dynamická inštancia triedy **T** bude žiť tak dlho, dokiaľ nebude programátorom explicitne uvoľnená. Vo chvíli, keď usúdime, že dynamickú inštanciu už nebudeme potrebovať, môžeme ju podrobiť deštrukcii pomocou nasledujúceho generického príkazu:

```
// Explicitná dealokácia dynamickej inštancie triedy.
delete p_premenna;
```

V príkaze je aplikovaný operátor **delete** na smerník, ktorý je uložený v smerníkovej premennej **p_premenna**. Keď prekladač deteguje takýto príkaz, emituje inštrukcie, ktoré realizujú 2 úlohy:

1. **Príprava objektu na zánik.** V prípravnej fáze dochádza k aktivácii deštruktora objektu, ktorý dostane príležitosť na spracovanie množiny finalizačných akcií. K finalizačným akciám patrí ľubovolná akcia, ktorú je potrebné vykonať ešte pred tým, ako bude objekt zničený (môže ísť napr. o uvoľnenie dynamických pamäťových blokov, uzatvorenie prístupu k otvoreným súborom na pevnom disku či o ukončenie aktívneho sieťového spojenia).

2. **Likvidácia objektu.** V momente, keď boli uskutočnené všetky príkazy deštruktora, dochádza k deštrukcii objektu. Jeho alokačný priestor je obnovený a priamo prístupný na budúce použitie.

Tip: Odporúčame, aby sa po aplikácii operátora **delete** na vybranú smerníkovú premennú uskutočnila vzápätí aj reinicializácia tejto premennej nulovým smerníkom, čo syntakticky vyjadríme takto:

```
p_premenna = 0;
```

Týmto spôsobom sa vyhneme vzniku možných kolíznych stavov, napr. vtedy, keď by sme omylom viacnásobne použili operátor **delete** na totožnú smerníkovú premennú. Chybe, ktorá by inak vznikla, vravíme v programovaní pokus o viacnásobnú dealokáciu objektu.

Konkrétny syntaktický model dynamickej inštanciácie triedy **Vektor3D** má nasledujúci tvar:

```
// Dynamická inštanciácia triedy Vektor3D.
Vektor3D *vektor = new Vektor3D(1, 4, 7);

// Pracujeme s inštanciou triedy Vektor3D.

// Ak už inštanciu triedy nepotrebujeme, uvoľníme ju.
delete vektor;
vektor = 0;
```

6 Využívanie služieb inštancie triedy

Objekt je logická jednotka, ktorá automatizuje spracovanie funkčne spriaznených činností. Na nasledujúcich fragmentoch zdrojového kódu jazyka C++ predstavíme, ako využívať služby automatických, statických a dynamických inštancií triedy **Vektor3D**.

1. Práca s automatickou inštanciou triedy **Vektor3D**:

```
int main()
{
    Vektor3D vektor(1, 4, 7);
    cout << "Velkost vektora: " << vektor.ZistitVelkost() << endl;
    return 0;
}
```

2. Práca so statickou inštanciou triedy **Vektor3D**:

```
int main()
{
    static Vektor3D vektor(1, 4, 7);
    cout << "Velkost vektora: " << vektor.ZistitVelkost() << endl;
    return 0;
}
```

3. Práca s dynamickou inštanciou triedy **Vektor3D**:

```
int main()
{
    Vektor3D *vektor = new Vektor3D(1, 4, 7);
    cout << "Velkost vektora: " << vektor->ZistitVelkost() << endl;
    delete vektor;
    vektor = 0;
    return 0;
}
```

Služby inštancií tried využívame volaním príslušných metód. Keď pracujeme s automatickými a statickými inštanciami, potom ich metódy voláme pomocou operátora priameho prístupu (.).

Naopak, v prípade dynamického objektu aktivujeme jeho metódy cez operátor nepriameho prístupu (->). Hoci automatické a statické inštancie tried nevyžadujú explicitné uvoľnenie, pri dynamických inštanciách tried musíme proces ich dealokácie iniciovať použitím operátora **delete**.

7 Základné princípy objektovo orientovaného programovania

Pre správne pochopenie objektovo orientovaného programovania sú kľúčové nasledujúce základné princípy: zapuzdrenie (*encapsulation*), dedičnosť (*inheritance*) a polymorfizmus (*polymorphism*).

Zapúzdrenie nielenže zvyšuje prehľadnosť zdrojového kódu objektovo orientovaného programu, ale tiež umožňuje chrániť dáta pred ich možným zneužitím prostredníctvom techniky **ukrývania dát**. Zapúzdrenie zamedzuje priamy prístup k interným dátovým atribútom a členským funkciám objektu pomocou logickej kompozície. Používateľ teda môže manipulovať s objektmi triedy bez toho, aby poznal ich vnútorné dátové zloženie a internú funkcionalitu.

Dedičnosť je mechanizmus umožňujúci opätovne použiť existujúci zdrojový kód programu. Proces dedičnosti prebieha vytvorením odvodenej triedy z bázovej triedy, pričom nová trieda dedí všetky charakteristiky pôvodnej triedy. Bázová trieda sa zvykne označovať aj ako rodič, nadtrieda alebo materská trieda a z nej odvodená trieda sa najčastejšie definuje ako dcérska trieda, podtrieda alebo potomok.

Odvodená trieda pôsobí vždy ako špeciálny prípad bázovej triedy. Podtrieda nám umožňuje pridávať k zdedeným dátovým atribútom a metódam bázovej triedy aj ďalšie nové dátové atribúty a metódy. Rozširuje sa tým pôvodne zdedená funkcionalita, ktorú sme získali z bázovej triedy. Okrem toho môže odvodená trieda prekrývať (*override*) požadované zdedené metódy bázovej triedy. To znamená, že podtrieda smie definovať metódy s rovnakými názvami, signatúrami a dátovými typmi návratových hodnôt, akými disponovala materská trieda. Tieto novo definované, a teda prekrývajúce metódy odvodenej triedy, sú vybavené odlišnou funkcionalitou, než akú mali rovnomenné metódy bázovej triedy.

Jazyk C++ rozlišuje dva základné druhy dedičnosti:

- **jednoduchá dedičnosť**,
- **viacnásobná dedičnosť**.

Keď vzniká odvodená trieda z práve jednej bázovej triedy, potom ide o jednoduchú dedičnosť. O viacnásobnej dedičnosti hovoríme vtedy, keď z väčšieho počtu bázových tried získame jednu odvodenú triedu.

Generický model jednoduchej dedičnosti má v jazyku C++ nasledujúci tvar:

```
// Deklarácia bázovej triedy.
class A
{
    // Telo bázovej triedy.
};

// Deklarácia odvodenej triedy.
class B : [modifikátor] A
{
    // Telo odvodenej triedy.
};
```

kde:

- **A** je identifikátor bázovej triedy.
- **B** je identifikátor odvodenej triedy.
- **modifikátor** určuje typ jednoduchej dedičnosti.

V jazyku C++ sa môžeme stretnúť s tromi typmi jednoduchej dedičnosti, ktoré sú reprezentované týmito prístupovými modifikátormi:

1. **private** – súkromná jednoduchá dedičnosť.
2. **protected** – chránená jednoduchá dedičnosť.
3. **public** – verejná jednoduchá dedičnosť.

Členy definované v bázovej triede ako verejné alebo chránené budú pri súkromnej jednoduchej dedičnosti v odvodenej triede definované ako súkromné. Pri chránenej jednoduchej dedičnosti budú verejné a chránené členy bázovej triedy chránené aj v odvodenej triede. No a pri verejnej jednoduchej dedičnosti si zdedené členy ponechajú svoje prístupové práva také, aké majú v bázovej triede. Ako môžeme postrehnúť, v jazyku C++ sa používajú rovnaké kľúčové slová, a to tak pre spôsob dedenia, ako aj pre označenie prístupových práv k dátovým atribútom a metódam. Pre lepšiu prehľadnosť uvádzame v tab. 1 relácie medzi druhmi jednoduchej dedičnosti a prístupovými právami k zdedeným členom bázovej triedy.

		Prístupové práva k členom bázovej triedy		
		private	**protected**	**public**
Typy jednoduchej dedičnosti	**private**	súkromné	súkromné	súkromné
	protected	súkromné	chránené	chránené
	public	súkromné	chránené	verejné

Tab. 1: Typy jednoduchej dedičnosti a prístupové práva k zdedeným členom bázovej triedy

Poznámka: Stred tabuľky označuje, ako sa budú definované členy bázovej triedy správať v odvodenej triede, podľa konkrétneho typu jednoduchej dedičnosti a konkrétnej sekcie bázovej triedy.

Polymorfizmus, označovaný aj ako mnohotvárnosť, predstavuje jeden z troch základných princípov objektovo orientovaného programovania. Je to schopnosť umožňujúca, aby na rovnaký podnet reagovali rôzne inštancie triedy odlišne.

Vďaka polymorfizmu môžeme vytvoriť novú definíciu metódy s rovnakým pomenovaním prekrytím pôvodne zdedenej metódy bázovej triedy. Táto novo definovaná metóda bude nahrádzať pôvodnú a zároveň bude disponovať odlišnou funkcionalitou. Pri práci s inštanciou odvodenej triedy sa aktivuje namiesto starej metódy novo vytvorená metóda.

Dôležité: Podstata polymorfného správania objektov v jazyku C++, ktoré dosahujeme pomocou dedičnosti, je v možnosti použitia inštancie odvodenej triedy všade tam, kde sa očakáva inštancia bázovej triedy. Metóda bázovej triedy, ktorá bude môcť byť prekrytá v odvodenej triede, sa nazýva virtuálna metóda a definujeme ju s modifikátorom **virtual**.

Všeobecný tvar definície virtuálnej metódy v tele bázovej triedy je potom takýto:

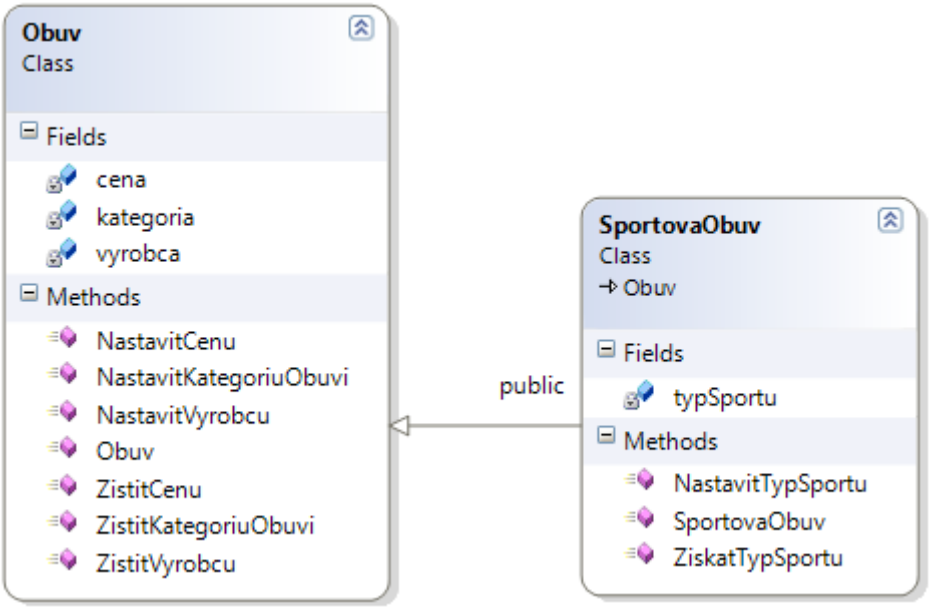
```
// Definícia virtuálnej metódy.
virtual void Metoda()
{
    // Telo virtuálnej metódy.
}
```

Definíciou rovnomennej metódy v odvodenej triede prekryjeme pôvodne zdedenú virtuálnu metódu bázovej triedy. Toto prekrytie označujeme aj ako predefinovanie danej metódy v odvodenej triede. Ako virtuálne nesmú pôsobiť statické metódy a ani konštruktor. Na druhej strane, virtuálny deštruktor sa dá používať, dokonca jeho použitie je spravidla nevyhnutné, a to najmä pri triedach, ktoré budú pôsobiť ako bázové triedy.

8 Praktický program č. 1: Obuv

Na prvom príklade vysvetlíme, ako funguje jednoduchá dedičnosť v praxi. Bázovú triedu tu predstavuje trieda **Obuv**, od ktorej odvodíme triedu **SportovaObuv**.

Diagram tried, ktorý vyjadruje vzťah medzi triedami **Obuv** a **SportovaObuv**, je znázornený na obr. 11.



Obr. 11: Diagram tried **Obuv** a **SportovaObuv**

Praktická algoritmizácia programu v jazyku C++:

```
#include <iostream>
#include <cstring>

using namespace std;

enum KategoriaObuvi
{
    Panska = 1, Damska = 2, Detska = 3
};

enum TypSportu
{
    Chodza = 1, Beh = 2, Aerobik = 3, Tenis = 4
};

// Deklarácia bázovej triedy.
class Obuv
{
private:
    // Dátové atribúty triedy.
    char vyrobca[20];
    int cena;
    KategoriaObuvi kategoria;
```

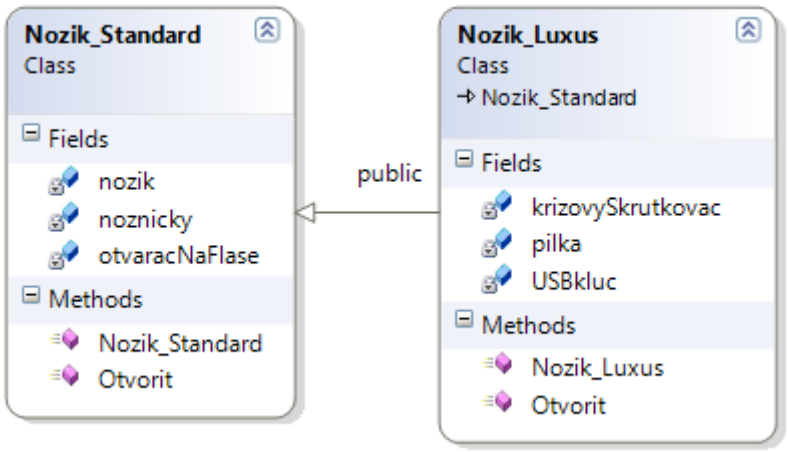
```
public:
    // Definícia parametrického konštruktora.
    Obuv(char *vyrobca, int cena, KategoriaObuvi kategoria)
    {
        strcpy(this->vyrobca, vyrobca);
        this->cena = cena;
        this->kategoria = kategoria;
    }
    // Definície prístupových metód pre výrobcu obuvi.
    void NastavitVyrobcu(char *novyVyrobca)
    {
        strcpy(vyrobca, novyVyrobca);
    }

    char* ZistitVyrobcu()
    {
        return vyrobca;
    }
    // Definície prístupových metód pre cenu obuvi.
    void NastavitCenu(int novaCena)
    {
        cena = novaCena;
    }
    int ZistitCenu()
    {
        return cena;
    }
    // Definície prístupových metód pre kategóriu obuvi.
    void NastavitKategoriuObuvi(KategoriaObuvi novaKategoria)
    {
        kategoria = novaKategoria;
    }
    KategoriaObuvi ZistitKategoriuObuvi()
    {
        return kategoria;
    }
};

// Deklarácia odvodenej triedy.
class SportovaObuv : public Obuv
{
private:
    // Definícia dátového atribútu odvodenej triedy.
    TypSportu typSportu;
public:
    // Definícia parametrického konštruktora odvodenej triedy.
    SportovaObuv(char *vyrobca, int cena, KategoriaObuvi kategoria,
    TypSportu typSportu) : Obuv(vyrobca, cena, kategoria)
    {
        this->typSportu = typSportu;
    }
    // Definície prístupových metód pre typ športu, na ktorý je obuv určená.
    void NastavitTypSportu(TypSportu novyTypSportu)
    {
        typSportu = novyTypSportu;
    }
    TypSportu ZiskatTypSportu()
    {
        return typSportu;
```

```
    }
};

int main()
{
    // Automatická inštanciácia bázovej triedy.
    Obuv obuv1("Adidas", 50, Panska);
    cout << "Informacie o obuvi:" << endl
         << "Vyrobca: " << obuv1.ZistitVyrobcu() << endl
         << "Cena: " << obuv1.ZistitCenu() << " EUR" << endl
         << "Kategoria: " << obuv1.ZistitKategoriuObuvi() << endl;

    // Dynamická inštanciácia odvodenej triedy.
    SportovaObuv *obuv2 = new SportovaObuv("Nike", 100, Damska, Aerobik);
    cout << endl << "Informacie o obuvi:" << endl
         << "Vyrobca: " << obuv2->ZistitVyrobcu() << endl
         << "Cena: " << obuv2->ZistitCenu() << " EUR" << endl
         << "Kategoria: " << obuv2->ZistitKategoriuObuvi() << endl;
    delete obuv2;
    obuv2 = 0;
    return 0;
}
```

9 Praktický program č. 2: Vreckový nožík

V tomto programe budeme demonštrovať polymorfné správanie inštancií tried v jazyku C++. Budeme pritom používať 2 druhy vreckového nožíka, a to nožík Štandard a nožík Luxus. Vreckový nožík Štandard obsahuje základnú výbavu, čiže nožík, nožničky a otvárač na fľaše. Nožík Luxus má všetko, čo má nožík štandard, no navyše je doplnený o pílku, USB kľúč a krížový skrutkovač.

Diagram tried, ktorý vyjadruje vzťah medzi triedami **Nozik_Standard** a **Nozik_Luxus**, je znázornený na obr. 12.



Obr. 12: Diagram tried **Nozik_Standard** a **Nozik_Luxus**

Inštancie oboch tried obsahujú rovnomennú metódu **Otvorit**. Rozdiel je však vo funkcionalite tejto metódy. Predpokladajme, že založíme inštancie oboch tried a pošleme im rovnakú správu, pričom táto správa spôsobí aktiváciu metódy **Otvorit** príslušnej inštancie. Zavoláme teda metódu **Otvorit**, ktorá sa bude správať inak pri vreckovom nožíku Štandard (vysunie sa nožík, nožničky a otvárač na flaše) a inak zase pri vreckovom nožíku Luxus (vysunie sa nožík, nožničky, otvárač na fľaše, pílka, USB kľúč, krížový skrutkovač). Na rovnaký podnet dostávame odlišné reakcie, čo znamená, že obidva vreckové nožíky (obe inštancie tried) sa správajú polymorfne.

Praktická algoritmizácia programu v jazyku C++:

```
#include <iostream>

using namespace std;

// Deklarácia bázovej triedy.
class Nozik_Standard
{
private:
    // Definície dátových atribútov bázovej triedy.
    bool nozik, noznicky, otvaracNaFlase;
public:
```

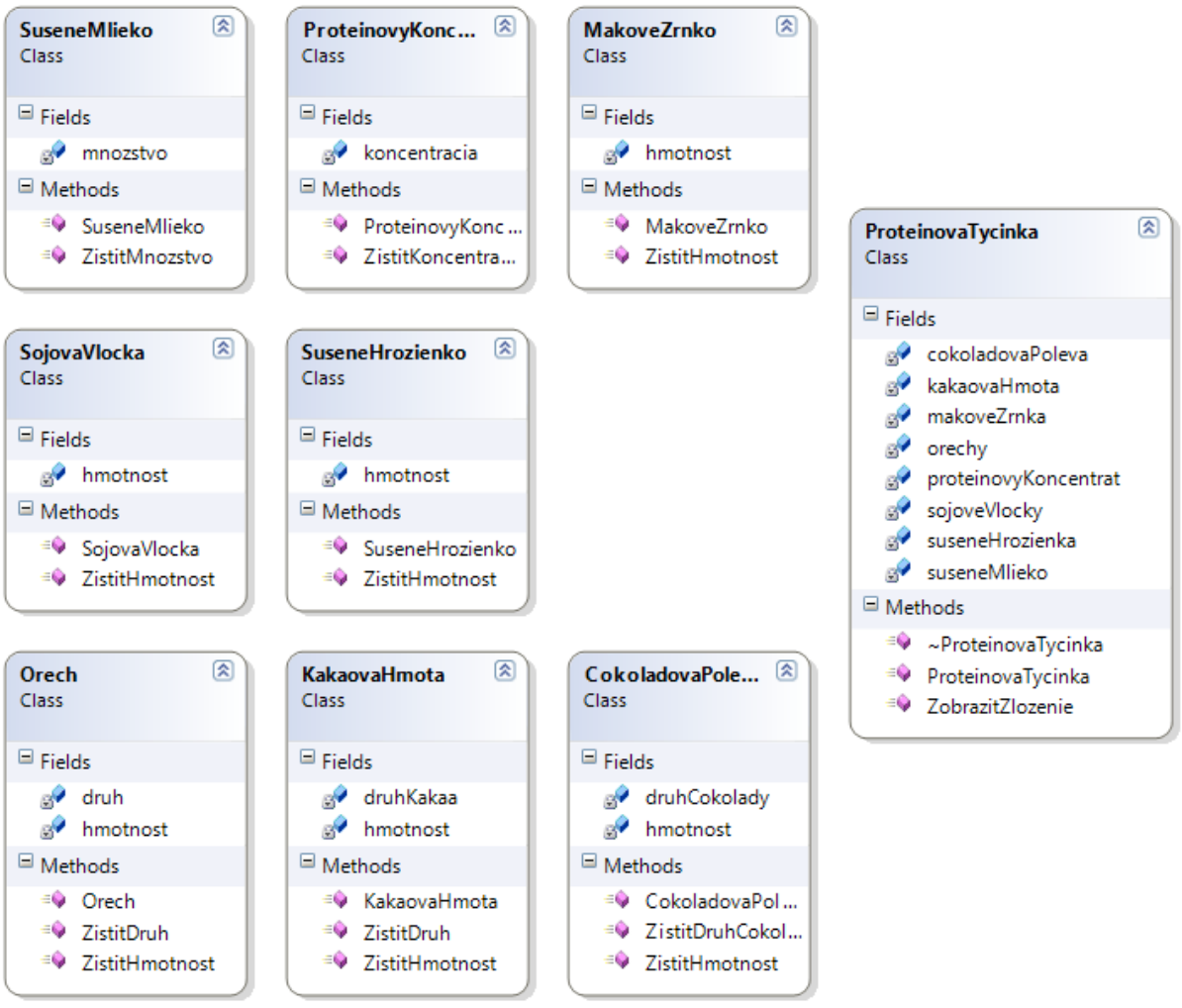
```
    // Definícia bezparametrického konštruktora.
    Nozik_Standard()
    {
        nozik = noznicky = otvaracNaFlase = true;
    }
    // Definícia bezparametrickej virtuálnej metódy.
    virtual void Otvorit()
    {
        cout << "Otvoril sa standardny vreckovy nozik." <<
            "\nObsahuje tieto prvky: " <<
            "\n\n1. nozik\n2. noznicky\n3. otvarac na flase" << endl;
    }
};

// Deklarácia odvodenej triedy.
class Nozik_Luxus : public Nozik_Standard
{
private:
    // Definície dátových atribútov odvodenej triedy.
    bool pilka, USBkluc, krizovySkrutkovac;
public:
    // Definícia bezparametrického konštruktora.
    Nozik_Luxus()
    {
        pilka = USBkluc = krizovySkrutkovac = true;
    }
    // Definícia rovnomennej metódy, ktorá prekrýva zdedenú virtuálnu
    // metódu bázovej triedy.
    void Otvorit()
    {
        cout << "Otvoril sa luxusny vreckovy nozik." <<
            "\nObsahuje tieto prvky: " <<
            "\n\n1. nozik\n2. noznicky\n3. otvarac na flase" << endl
            << "4. pilku\n5. USB kluc\n6. krizovy skrutkovac" << endl;
    }
};

int main()
{
    // Automatická inštanciácia bázovej triedy.
    Nozik_Standard nozik1;
    nozik1.Otvorit();
    cout << endl;
    // Dynamická inštanciácia odvodenej triedy.
    Nozik_Luxus *nozik2 = new Nozik_Luxus();
    nozik2->Otvorit();
    delete nozik2;
    nozik2 = 0;
}
```

10 Praktický program č. 3: Proteínová čokoládová tyčinka

Nasledujúci program vysvetľuje agregačno-kompozičné vzťahy medzi triedami. Ako hlavný objekt bude pôsobiť proteínová čokoládová tyčinka, ktorá obsahuje určitú množinu podobjektov.

Naša proteínová čokoládová tyčinka pozostáva z týchto komponentov: čokoládová poleva, sušené mlieko, makové zrnká, orechy, sušené hrozienka, sójové vločky, kakaová hmota a proteínový koncentrát.

Diagram tried, znázorňujúci triedu **ProteinovaTycinka** a triedy, z ktorých bude proteínová tyčinka pozostávať, je nakreslený na obr. 13.



Obr. 13: Diagram tried, z ktorých sa bude skladať proteínová tyčinka

Praktická algoritmizácia programu v jazyku C++:

```
#include <iostream>
#include <cstring>

using namespace std;

class CokoladovaPoleva
{
private:
    char druhCokolady[20];
    int hmotnost;
public:
    CokoladovaPoleva(char *druhCokolady, int hmotnost)
    {
        strcpy(this->druhCokolady, druhCokolady);
        this->hmotnost = hmotnost;
    }
    char* ZistitDruhCokolady()
    {
        return druhCokolady;
    }
    int ZistitHmotnost()
    {
        return hmotnost;
    }
};

class SuseneMlieko
{
private:
    int mnozstvo;
public:
    SuseneMlieko(int mnozstvo) : mnozstvo(mnozstvo) {}
    int ZistitMnozstvo()
    {
        return mnozstvo;
    }
};

class MakoveZrnko
{
private:
    int hmotnost;
public:
    MakoveZrnko(int hmotnost) : hmotnost(hmotnost) {}
    int ZistitHmotnost()
    {
        return hmotnost;
    }
};

class Orech
{
private:
    char druh[20];
    int hmotnost;
public:
    Orech(char *druh, int hmotnost)
```

```
    {
        strcpy(this->druh, druh);
        this->hmotnost = hmotnost;
    }
    char* ZistitDruh()
    {
        return druh;
    }
    int ZistitHmotnost()
    {
        return hmotnost;
    }
};

class SuseneHrozienko
{
private:
    int hmotnost;
public:
    SuseneHrozienko(int hmotnost) : hmotnost(hmotnost) {}
    int ZistitHmotnost()
    {
        return hmotnost;
    }
};

class SojovaVlocka
{
private:
    int hmotnost;
public:
    SojovaVlocka(int hmotnost) : hmotnost(hmotnost) {}
    int ZistitHmotnost()
    {
        return hmotnost;
    }
};

class KakaovaHmota
{
private:
    char druhKakaa[20];
    int hmotnost;
public:
    KakaovaHmota(char *druhKakaa, int hmotnost)
    {
        strcpy(this->druhKakaa, druhKakaa);
        this->hmotnost = hmotnost;
    }
    char* ZistitDruh()
    {
        return druhKakaa;
    }
    int ZistitHmotnost()
    {
        return hmotnost;
    }
};
```

```
class ProteinovyKoncentrat
{
private:
    float koncentracia;
public:
    ProteinovyKoncentrat(float koncentracia) : koncentracia(koncentracia) {}
    float ZistitKoncentraciu()
    {
        return koncentracia;
    }
};

class ProteinovaTycinka
{
private:
    CokoladovaPoleva *cokoladovaPoleva;
    SuseneMlieko *suseneMlieko;
    MakoveZrnko* makoveZrnka[50];
    Orech* orechy[3];
    SuseneHrozienko* suseneHrozienka[10];
    SojovaVlocka* sojoveVlocky[7];
    KakaovaHmota *kakaovaHmota;
    ProteinovyKoncentrat *proteinovyKoncentrat;
public:
    ProteinovaTycinka()
    {
        cokoladovaPoleva = new CokoladovaPoleva("Mliecna cokolada", 10);
        suseneMlieko = new SuseneMlieko(150);
        for(int i = 0; i < 50; i++)
        {
            makoveZrnka[i] = new MakoveZrnko(1);
        }
        for(int i = 0; i < 3; i++)
        {
            orechy[i] = new Orech("Lieskovy orech", 3);
        }
        for(int i = 0; i < 10; i++)
        {
            suseneHrozienka[i] = new SuseneHrozienko(1);
        }
        for(int i = 0; i < 7; i++)
        {
            sojoveVlocky[i] = new SojovaVlocka(2);
        }
        kakaovaHmota = new KakaovaHmota("70% kakao", 4);
        proteinovyKoncentrat = new ProteinovyKoncentrat(0.82f);
    }
    void ZobrazitZlozenie()
    {
        cout << "Zlozenie proteinovej tycinky:" << endl
             << "  Cokoladova poleva: " << endl
             << "    Druh cokolady: " << cokoladovaPoleva->ZistitDruhCokolady()
             << endl
             << "    Hmotnost: " << cokoladovaPoleva->ZistitHmotnost() << endl
             << "  Susene mlieko: " << endl
             << "    Mnozstvo: " << suseneMlieko->ZistitMnozstvo() << endl
             << "  Makove zrnka: " << endl
             << "    Pocet: 50" << endl
             << "    Hmotnost: " << 50 * makoveZrnka[0]->ZistitHmotnost() << endl
```

```
            << "  Orechy: " << endl
            << "    Druh orechov: " << orechy[0]->ZistitDruh() << endl
            << "    Hmotnost: " << orechy[0]->ZistitHmotnost() << endl
            << "  Susene hrozienka: " << endl
            << "    Pocet: 10" << endl
            << "    Hmotnost: " << 10 * suseneHrozienka[0]->ZistitHmotnost()
            << endl
            << "  Sojove vlocky: " << endl
            << "    Pocet: 7" << endl
            << "    Hmotnost: " << sojoveVlocky[0]->ZistitHmotnost() << endl
            << "  Kakaova hmota: " << endl
            << "    Druh kakaa: " << kakaovaHmota->ZistitDruh() << endl
            << "    Hmotnost: " << kakaovaHmota->ZistitHmotnost() << endl
            << "  Proteinovy koncentrat: " << endl
            << "    Koncentracia: " << proteinovyKoncentrat->ZistitKoncentraciu()
            << endl;
    }
    virtual ~ProteinovaTycinka()
    {
        delete cokoladovaPoleva;
        cokoladovaPoleva = 0;
        delete suseneMlieko;
        suseneMlieko = 0;
        for(int i = 0; i < 50; i++)
        {
            delete makoveZrnka[i];
            makoveZrnka[i] = 0;
        }
        for(int i = 0; i < 3; i++)
        {
            delete orechy[i];
            orechy[i] = 0;
        }
        for(int i = 0; i < 10; i++)
        {
            delete suseneHrozienka[i];
            suseneHrozienka[i] = 0;
        }
        for(int i = 0; i < 7; i++)
        {
            delete sojoveVlocky[i];
            sojoveVlocky[i] = 0;
        }
        delete kakaovaHmota;
        kakaovaHmota = 0;
        delete proteinovyKoncentrat;
        proteinovyKoncentrat = 0;
    }
};

int main()
{
    ProteinovaTycinka tycinka;
    tycinka.ZobrazitZlozenie();
    return 0;
}
```

O autorke

Volám sa Margaréta Cifrová, mám 21 rokov. Mojím rodným mestom je Bratislava, kde aj žijem. Je to mesto plné ruchu, obrovských možností, či už z pohľadu kultúry, vzdelania alebo aj samotnej histórie, umenia, pracovných príležitostí, biznisu. Disponuje rozsiahlou množinou možností usporiadania voľného času. Či už je to vďaka blízkej prírode, rôznym nákupným možnostiam, alebo kráse histórie, pamiatok a hlavne ľudí v ňom, dodávajúcich tomu všetkému skutočný lesk. Je zaujímavé len tak sa prejsť po meste a všímať si veci, ktoré človek zabudne vnímať pri tom všetkom zhone. Zvlášť, keď sa pohybujeme vo svete rýchleho vývoja spoločnosti, rozvoja vedy a techniky, vysokých nárokov na čas a informácie. Tento iný pohľad mi dáva jeden z mojich základných životných pilierov, a to umenie. K druhému neodmysliteľne patrí oblasť informačných technológií. Pôsobiac ako základná niť môjho života, poskytujúca neobmedzené možnosti z pohľadu vzdelávania, výskumu, pokroku, napredovania a udržiavania medziľudských vzťahov.

Moje vysokoškolské štúdium mi v súčasnosti plnohodnotne korešponduje s mojím záujmom o sféru informačných technológií. Študujem na Ekonomickej univerzite v Bratislave, na Fakulte hospodárskej informatiky, odbor Manažérske rozhodovanie a informačné technológie. Predtým som vyštudovala Obchodnú akadémiu v Bratislave.

Môj prvý kontakt s počítačmi nastal už v troch rokoch. Bol to počítač zvaný XT PP 06 od spoločnosti Tesla u otca v práci. Hrala som na ňom svoju prvú počítačovú hru – šach. Prvý mikropočítač, s ktorým som sa dostala do kontaktu, bol ZX Spectrum u nás doma. Pripájal sa na televízor, ponúkal jednoduchú farebnú grafiku a komunikácia s ním bola postavená na jeho vlastnom programovacom jazyku Sinclair BASIC. Neskôr k nemu pribudol aj mikropočítač Tesla PMD 85, ktorý už pracoval s programovacím jazykom BASIC. V mojich šiestich rokoch sme už doma vlastnili relatívne plnohodnotný počítač Intel i80486 DX2, 66 MHz, 4 MB RAM a 80 MB HDD aj s tlačiarňou.

Počítače vypĺňali knihu môjho života už od detstva. Spadala som pod generáciu detí, ktoré vyrastali za počítačom. K môjmu detstvu, prirodzene, patrili aj rôzne počítačové hry. Ich spektrum bolo rozsiahle, či už to boli rôzne arkádovky, adventúry, cez akčné hry až po obľúbené RPG (s avatarmi v úlohách), simulátory, preteky a stratégie, či iné multižánrové hry. Zaujímavá bola aj možnosť zahrať si s niekým, či skúsiť podomácky naprogramované hry. V tomto prípade boli hlavnými tvorcami môj brat a bratranec, ktorí dokázali šikovne previesť nápady cez zdrojový kód do praxe.

Na hrách a vlastne aj počítačoch ma fascinoval takmer neohraničený virtuálny svet, ktorý umožňoval plne sa realizovať, rozvíjať tvorivosť a uskutočňovať nadprirodzené veci. Vlnu hier postupne vymenila práca s počítačom a štúdium informačných technológií, aj keď si občas nejakú hru zahrám.

V mojom rozvoji tejto sféry hrala podstatnú úlohu stredná škola. Predostierala mi modernú výpočtovú techniku, prístup na internet, rôzne kurzy a hlavne možnosť byť za počítačom aj nad rámec svojho štúdia. Ovplyvnilo to aj samotný výber mojej vysokej školy so zameraním na informačné technológie. Štúdium na vysokej škole tvorí v tomto duchu významné prelomové obdobie. Umožnilo mi hlbšie preniknúť aj do neprebádanej oblasti programovania. C-čko tu predstavovalo základný spúšťací mechanizmus, štartovací jazyk. Mojím obľúbeným programovacím jazykom je momentálne

C++. Tento programovací jazyk ukrýva v sebe nevyčísliteľnú krásu zdrojového kódu. Okrem programovacieho jazyka C++ sa v súčasnosti čiastočne venujem aj deklaratívnemu jazyku SQL.

Ako som už spomínala, umenie tvorí jeden z mojich základných pilierov života. Medzi moje hlavné oblasti pôsobenia patrí výtvarníctvo, fotografia, či tvorba esejí. Jeho krása spočíva v odovzdávaní hodnoty, kúsku seba iným, v malebnosti farieb, slov, viet, myšlienok, spestrujúc všedný deň.

Tak ako počítačom aj výtvarníctvu sa venujem odmalička. Od roku 1994 až do roku 2009 som absolvovala štúdium výtvarného odboru na Základnej umeleckej škole v Bratislave. Počas môjho života sa mi podarilo z tejto oblasti získať rôzne ocenenia: udelenie „Ceny bienále fantázie“ („PRIZE of the BIENNALE of IMAGINATION“) medzinárodnou porotou v roku 1998, čestné uznanie „Bohúňová paleta“ z V. ročníka medzinárodnej výtvarnej súťaže v Dolnom Kubíne za rok 2001 udelený Ministerstvom školstva Slovenskej republiky, ZUŠ Petra Michala Bohúňa v DK a Mestským domom kultúry v Bielsko-Bialej, či získanie medzinárodného diplomu uznania za kvalifikáciu exlibrisu v poľskej medzinárodnej finálnej výstave uskutočnenej v Žary v roku 2002. Nedávno som získala tiež 1. miesto v súťažnej disciplíne „Výroba vianočnej pohľadnice“ z 9. ročníka celoslovenskej abilympiády za rok 2009, ktorej som sa zúčastnila vďaka mojej dlhoročnej kamarátke. Výrobou vianočných pohľadníc som sa zaoberala už predtým počas štúdia na strednej škole, kde som ich predávala a prezentovala na každoročných stredoškolských vianočných trhoch. Najčastejšie som ich zhotovovala z ručného papiera.

K fotografovaniu som sa dostala už po príchode na obchodnú akadémiu. Vo svojich začiatkoch som sa najprv venovala dokumentárnej fotografii. Tak ako väčšina fotografov som začínala s klasickým fotoaparátom na princípe analógového filmu. Snažila som sa zachytiť všetky podstatné scény počas rôznych spoločenských akcií, školských výletov a kamarátskych stretnutí. Na strednej škole som pôsobila ako hlavný fotograf významných akcií a podujatí. Zachytené snímky sa využívali či už na prezentačné účely, alebo sa publikovali na web stránke školy, alebo v školskom časopise, kde som pracovala ako jeden z redaktorov. Podobne som občas vypomáhala s fotografovaním aj otcovi, ktorý sa pôsobí v oblasti kultúry. Postupne som prešla k umeleckej fotografii, v rámci ktorej som našla zmysel v kráse farieb, hry svetla a tieňov. Využívam rôzne virtuálne metafory, symboly, dominujú u mňa motívy prírody, ľudí, mesta. Fotografovanie mi prináša možnosť sebarealizácie, kreativity, vyjadrenia rôznych emócií.

V dnešnej dobe sa aj naďalej popri vysokej škole v rámci voľného času venujem fotografovaniu, výtvarníctvu a písaniu esejí na úrovni hobby.

Margaréta Cifrová
je študentkou 3. ročníka študijného programu Manažérske rozhodovanie a informačné technológie na Fakulte hospodárskej informatiky Ekonomickej univerzity v Bratislave (FHI EU).

OD študentov
PRE študentov