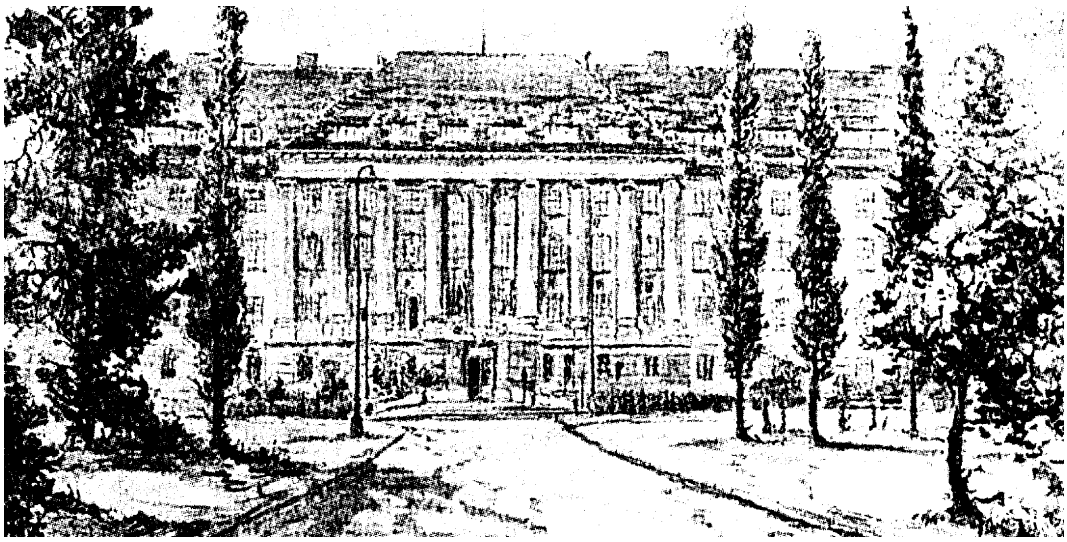
Mendelova univerzita v Brně

Moderní programování objektových aplikací v C++ (ESF)



Obsah

Cíl kurzu

Cílem tohoto e-learningového kurzu je prohloubit znalosti objektového návrhu aplikací, které jsme získali v rámci předmětu Základy objektového návrhu. Důraz je kladen na vhodné využívání nástrojů jazyka C++ a osvojení si klíčových principů vývoje komplexních aplikací jako je vhodné dokumentování kódu, testování, atp.

Motivace

Je k dispozici celá řada knih, časopisů a Interetových seriálů o programování v C++. Jen málokterou publikaci však lze použít jako základ pro tvorbu kvalitních objektových aplikací. Důvodů je několik: řada zdrojů je relativně velmi starých (popisované techniky již zastaraly), řada zdrojů byla napsána autoři, kteří rozumněli pouze určitému aspektu objektového programování v C++ (jazyku jako takovému nebo objektovému návrhu) a řada autorů byli a jsou pouze nadšení amatéři, kteří s programováním mají jen velmi málo zkušeností.

V této opoře se pokusím o vhodné propojení principů objektového návrhu, moderních nástrojů C++ (podle verze C++0x) a obecných postupů při tvorbě moderních aplikací (automatická tvorba dokumentace, testování, hlášení chyb, atd.). Samořejmě, že nikdy nelze vytvoři dokonalé dílo, proto budu velmi vděčný za případné přípomínky a názory, které mi sdělíte nebo zašlete [1].

Návaznost

Tato opora velmi úzce navazuje na oporu „Základy programování objektových aplikací v C++“. Je nezbytné seznámit se jejím obsahem. V řadě částí opory je na její obsah odkazováno. S tím souvisí znalost základů objektového programování v C++. Je nezbytná znalost zapouzdření, vazeb, dědičnosti a polymorfismu. V této opoře se je naučíme vhodně využívat pro tvorbu komplexních aplikací.

Poučení

Programování se nelze naučit čtením, jen zase programováním. Chápejte tuto oporu jako návod, jaké problémy si máte prakticky vyzkoušet. Každou část knihy si po přečtení vyzkoušejte, abyste si problém skutečně osvojili. Pokud naleznete chybu, napište [2], prosím.

> Tato opora je tvořena krátce po stadardizaci nové verze C++ označované C++0x. Tato verze přinesla řadu zajímavých novinek jako je lepší podpora UTF-8/16/32, výčtové for cykly, aj. V příkladech jsou tyto nástroje již použity. Pro správný překlad řady příkladů bude nezbytné mít nainstalovánu novou verzi překladače. V případě GCC je doporučena verze 4.6 nebo novější.

Úvod

Historie jazyků rodiny C

Jazyk C

Historie C++, který je v tomto kurzu používán se odvíjí od historie jazyka C. Kolem roku 1970 byl tým programátorů kolem Dennise Ritchieho v Bellových laboratořích pověřen vývojem nového operačního systému pro telekomunikační centrály společosti AT&T. Tím systémem se neměl stát nikdo jiný, než UNIX. Programátoři ale naznali, že nemají k dispozici programovací jazyk, který by odpovídal jejich potřebám, proto se rozhodli napsat si nejprve vlastní jazyk – C. Název C má poměrně prozaický původ. V té době existoval jazyk B a proto padla

[1] mailto:david.prochazka@mendelu.cz
[2] mailto:david.prochazka@mendelu.cz

volba na C – následující volné písmeno abecedy. Podrobnosti o vývoji jazyka můžete najít přímo na stránkách Bellových laboratoří [3].

V průběhu doby se jazyk C silně měnil. Milníky vývoje lze ve stručnosti popsat takto:

- **1978**: Brian Kernighan a Dennis Ritchie vydávají *The C Programming Language*. Tato kniha určila první neoficiální standard pro C označovaný jako *C podle Kernighama a Ritchieho* (K-R C).
- **ANSI C**: Existuje několik verzí normy, nejrozšířenější verze je z r. 1989.
- **ISO/IEC C** (ISO 99): Poslední verze normy – ISO/IEC 9899:1999. V roce 2001 byly vydány opravy.
- v roce 2007 začala práce na verzi neformálně označované jako *C1X*.

Jazyk C lze charakterizovat jako:

- relativně nízkoúrovňový (systémový) jazyk,
- ideální pro vývoj operačních systémů, driverů, překladačů,
- podporuje obrovské množství knihoven,
- nižší efektivita vývoje, která je vykoupena rychlým a elegantním kódem.

Jazyk C++

Tímto se pomalu dostáváme k jazyku C++. Začněme jeho názvem – C++. Je evidentní, že vychází z názvu svého předchůdce – C. „++“ je v C operátor pro inkrementaci nebo následníka. C++ tedy znamená „následník C“. C++ je jazyk od počátku orientovaný na objektový návrh. Vznik C++ můžeme datovat přibližně do roku 1985 a opět do Bellových laboratoří. Jeho autorem je Bjarne Stroustrup. C++ se stal vzorem pro implementaci mnoha jiných objektových jazyků – C#, Java, aj. Jako správný následník si C++ snaží zachovat kompatibilitu s C. Jakýkoliv program v C by měl být platným programem v C++ a měl by jít přeložit překladačem pro C++. První překladače C++ byly preprocesory, které překládaly z C++ do C. Dnes již některé programy v jazyku C překladači pro C++ překládat nelze, ale zpětná kompatibilita s C je pořád velmi dobrá. Pokud je program napsán podle současných norem C, je z 99% přeložitelný překladačem C++. Tohoto jevu je v praxi stále hojně využíváno, protože nové moduly mnoha programů v C chtějí využívat např. STL, objekty, či jiné nástroje, které poskytuje pouze C++. Jakkoliv je zpětně podporována většina konstrukcí C, není dobré plést dohromady „C“ a „C++“ kód. C++ je v současné době samořejmě také standardizován – viz popis standardů [4]. V současné době se přechází na specifikaci jazyka podle ISO normy *Standard for Programming Language C++* z 26. 3. 2010. Tato verze C++ je označována jako ***C++0x*** („see plus plus oh ex“). Základní vlastnosti C++ jsou:

- jazyk od počátku orientovaný na OOP,
- rychlejší vývoj aplikací oproti C,
- abstraktní datové typy,
- generické programování,
- výjimky,
- odlišná filozofie práce se soubory, terminálem, atp. (proudy).

Jazyk C#

C# (sí šarp) označuje hudební předznamenání, které zvyšuje notu o půl tónu. C# tedy v hudební nauce značí „cis“ – zvýšené C. C# je vysoko úrovňový objektově orientovaný jazyk vyvinutý firmou Microsoft (Andersem Hejlsbergem) pro platformu .NET. Byl schválen standardizačními komisemi ECMAa ISO. Založen na jazycích C++ a Java. Jedná se prakticky o přímého konkurenta Javy, ze kterého silně čerpá (na druhou stranu v současné době Java také čerpá ze C#). C# se využívá hlavně k tvorbě databázových programů, webových aplikací, webových služeb, ale i desktopových aplikací. Původně existoval pouze oficiální překladač firmy Ms pro platformu Windows, ale dnes je již k dispozici projekt Mono [5], který je otevřenou multiplatformní implementací překladače (interpreta) pro C# a další jazyky .NET frameworku. Zásadními rysy C# jsou:

- jazyk C# může být transformován jak do strojového kódu, tak do Common Intermediate Language [6], resp. Common Language Runtime [7],

[3] http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
[4] http://open-std.org/jtc1/sc22/wg21/
[5] http://www.mono-project.com/
[6] http://en.wikipedia.org/wiki/Common_Intermediate_Language
[7] http://en.wikipedia.org/wiki/Common_Language_Runtime

- poskytuje automatický garbage collector,
- orientace na rychlost vývoje, zvláště webových aplikací,
- neexistují globální proměnné,
- ukazatele mohou být použity pouze v blocích kódu explicitně označených jako „unsafe“,
- silně typový jazyk (hlídá si konverze mezi typy ještě více, než C++),
- nevýhodou je, že oficiálně je podporován pouze na platformě Ms Windows.

Objective C

Je v současné době podporován zejména firmou Apple. Jsou na něm založeny operační systémy MacOS X a iOS. Objective-C navrhnul Brad Cox pracující ve společnosti Stepstone počátkem osmdesátých let minulého století. Je to objektový jazyk, který svou objektovou implementaci převzal z jazyka Smalltalk. Jedná se tedy o zcela odlišný koncept, než je používán v C++. Kompilátor Objective C je součástí balíku překladačů GCC[8]. Obejctive C je jazyk podporující dynamické typování, to umožňuje značnou flexibilitu aplikací.

[8] http://gcc.gnu.org/

1 Základy tvorby komplexních aplikací

Velmi stručné opakování základů C++

Velmi stručné opakování základů C++

Základy jazyka C++ byly relativně podrobně prezentovány v opoře „Základy programování objektových aplikací v C++“ v sekci „Základy jazyka C++“, proto se k nim nebudeme vracet. Následující text jen velmi stručně shne a vyzvedně vybrané základní principy objektového programování v C++.

Vhodné dokumentování kódu

2 Knihovna STL

Datové kontejnery

Šablony jako datové kontejnery

Šablony jsou jedním z největších přínosů *knihovny STL* (Standard Template Library[9]) navržené firnou SGI. Šablony jsou *de facto* datové kontejnery bez specifikovaného typu. Např. běžně pracujeme třídou, které obsahuje pole (matici) hodnot typu integer. Dokážeme si určitě představit, že pracujeme např. třídou reprezentující dynamický seznam hodnot typu integer. Šablona je předpis, jak bude taková stuktura vypadat - jak bude uchovávat hodnoty, jaký bude mít konstruktor, jaké metody bude poskytovat. S tím, že není definováno, co bude obsahovat. To je velmi užitečná vlastnost. Představme si, že potřebujeme udělat lineární seznam integerů a následně, že dostaneme za úkol vytvořit stejný seznam floatů (nebo dokonce instancí nějaké třídy). V zásadě budou obě třídy vypadat stejně, ale ve všech atributech a metodách budeme muset dělat úpravy, aby třída pracovala s jiným datovým typem. Tomu se vyhneme právě využitím šablony. Nadefinujeme funkčnost a konkrétní typ dosadíme "na přání".

> Šablony jsou datové kontejnery reprezentované klasickými třídami bez specifikovaného typu. Typ je do nich dosazován v okamžiku jejich použití.

Podívejme se nejdříme na nejznámějšího zástupce této knihovny - třídu `vector`.

Třída vector

`Vector` je jednorozměrné dynamické pole. Je to datový kontejner, který se součástí knihovny STL. Jedná se tedy o určitou obecnou šablonu bez specifikovaného typu. Ukažme si použití `vectoru` na jednoduchém příkladu:

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

int main() {
// typ, který bude vector uchovávat je v < >
vector<int> a(7);     // Pole 7 int. čísel, konstruktor přebírá
inicializační vel.
vector<int> b;        // Pole 0 int. čísel

cout << "Počet prvků v a je " << a.size() << "  "; // voláme
klasickou metodu
cout << "Počet prvků v b je " << b.size() << endl;

// k prvkům vektoru lze přistupovat jako k prvkům pole
for(int i = 0; i < a.size(); i++){
a[i] = i + 1;
cout << "a[" << i << "] = " << a[i] << endl;
}

b = a; //Bez problémů lze použít operátor =

return 0;
}
```

[9] http://www.sgi.com/tech/stl/

Jak je patrné z předchozího příkladu, z šablony vytvoříme "klasickou" třídu dosazením typu. V našem případě jsme dosadili typ integer - `vector<int>` - a získali jsme třídu, která přestavuje dynamické pole integerových hodnot.

`Vector` má dva konstruktory - parametrický, který přebere počáteční velikost vektoru a bezparametrický, který nastaví velikost na nulu. K prvkům lze přistupovat klasicky přes `pole[pozice]`. Lze s ním tedy pracovat stejně, jako s polem. Kromě toho `vector` samozřejmě disponuje řadou metod. Jednou z nejvíce využívaných je metoda `push_back`, která přidává prvek na konec pole. Ukažme si několik základních operací:

```
// deklarace vektoru
vector<int> a;

// přidávání prvků
for(int i = 0; i < 5; i++){
a.push_back(i+1);
cout << "Posledn'ı prvek je: " << a.back() << endl;
}

// průchod polem
for(int i = 0; i < a.size(); i++) {
cout << a[i] << \t;
}
cout << endl;

// promazání pole
while (!a.empty()) { // opakuj, dokud není pole prázdné
a.pop_back(); // odeberu poslední prvek
}

// kontrola velikosti
cout << "Velikost " << a.size() << endl;
```

`Vector` sám od sebe neumožňuje vytvářet vícerozměrná pole. Tohoto efektu lze ale dosáhnout zanořením více `vectorů` do sebe. Prvky výchozího vectoru v takovémto případě nejsou triviální hodnoty a opět `vectory`. Triviální příklad této konstrukce je předveden v následujícím příkladu:

```
vector<vector<int> > matice(3); // Matice 3 x 0, mezi > > je mezera!
for(int a = 0; a < 3; a++) {
matice[a].push_back(a+1);
matice[a].push_back(a+2);
matice[a].push_back(a+3);
}

// Nyní máme matici 3 x 3
for(int y = 0; y < 3; y++){
for(int x = 0; x < 3; x++){
cout << matice[x][y] << '\t';
}
cout << endl;
}
```

Podívejme se na tento příklad podrobněji a vysvětleme si, jaký je rozdíl mezi `matice.push_back(...)` a `matice[a].push_back(...)`. V prvním případě je nový prvek přidán na konec "hlavní" matice, tedy té, která jako každý prvek obsahuje vector hodnot. V druhém případě je nový prvek přidáván na konec matice, která je uložena v "hlavní" matici na pozici *a*. V prvním případě bych tedy musel vkládat celý vector, v případě druhém se už samozřejmě očekává pouze jedna integerová hodnota. Uvědomit si rozdíl mezi těmito operacemi je zcela nezbytné pro pochopení celé problematiky.

Iterátory

Iterátor je de facto ukazatel pro kontejner. Iterátory jsou bezpečnější než indexy nebo ukazatele. Ke kontejnerům jsou předdefinovány užitečné iterátory na začátek (`.begin()`), za konec (`.end()`), aj. Výhodou iterátoru je, že ani při nekorektním použití se nedostane mimo strukturu (nepovede se Vám udělat poce[10] u 5ti prvkové struktury). Iterátory také využívá drtivá většina algoritmů STL, proto je velmi účelné je zvládnout. Ukažme si pro začátek základní operaci - průchod polem:

```
vector<int> vektor(20);

for (vector<int>::iterator temp = vektor.begin(), temp !=
vektor.end(); temp++){
*temp = 0;
}
```

Podívejme se na deklaraci iterátoru. Je patrné, že iterátor se vždy váže k typu struktury, nad kterou je definován. Jeho deklarace je tedy vždy `typ_struktury::iterator`. Jak již bylo řečeno, nad `vectorem` (a většinou dalších struktur) jsou definovány iterátory `begin` a `end`. Při průchodu `vectorem` tedy na počátku nastavím mnou vytvořený iterátor na hodnotu iterátoru `begin` a postupně jej inkrementuji (operátor ++) až dokud nedosáhne hodnoty iterátoru `end`. Pokud chci přistoupit na nějakou triviální hodnotu uloženou ve vectoru na pozici na kterou ukazuje iterátor, musím použít konstrukci `*název_iterátoru` (např.: `cout << *pozice`). Pokud je pod iterátorem uložena instance třídy, použiji operátor -> (např: `iterator->objem_motoru`).

Jak bylo řečeno iterátory se používají v mnoha algoritmech a metodách. Ukažme si některé často používané operace s vectorem, které iterátory využívají.

```
// vložím před první prvek -100
v1.insert(v1.begin(),-100);

// vložím na konec 3 krát 500
v1.insert(v1.end(),3,500);

// vložím celý vektor v1 do v2
v2.insert(v2.begin()+2,v1.begin(),v1.end());

// mažu prvek pod iterátorem
v2.erase(it);

// mažu všechny prvky ve v2
v2.erase(v2.begin(), v2.end());
```

Prolematika iterátorů je citelně složitější, než je zde uvedeno. Existují dopředné iterátory, výstupní iterátory a řada jiných. Tato problematika, ale přesahuje rámec tohoto kurzu. Pro více podrobností doporučuji navštívit `http://www.sgi.com/tech/stl/Iterators.html`.

Další šablony v STL

Knihovna STL samozřejmě obsahuje řadu dalších šablon, které korespondují s často používanými abstraktními datovými typy. Práce s nimi je naprosto totožná s prací s `vectorem`. Liší se od sebe obvykle pouze množinou operací, které jsou na nimi definovány. Jejich popisy naleznete na `http://www.sgi.com/tech/stl/`. Pro přehled uvádím srovnávací tabulku (tabulka je převzdata ze serveru builder.cz - `http://www.builder.cz/art/cpp/cppstl.html`).

Název kontejneru	Typ kontejneru	Hlavičkový soubor	Popis kontejneru
bitset	posloupnost	bitset	Posloupnost bitů pevné délky.
deque	posloupnost	deque	Oboustranná fronta. Prvky lze vkládat, nebo odebírat z obou konců. Sice lze rovněž odebírat, nebo vkládat prvky na libovolné místo ve frontě (kontejner deque to umožňuje), ale tato operace není příliš efektivní.
list	posloupnost	list	Oboustranně zřetězený seznam.
map	asociativní kontejner	map	Asociativní pole. pole, které nemusí být indexováno celočíselným typem, ale čímkoliv. Třeba řetězcem. Pro daný klíč může existovat pouze 1 asociovaná hodnota. Tomuto kontejneru se budeme v budoucnu zabývat v samostatném článku.
multimap	asociativní kontejner	map	Asociativní pole. Pro daný klíč (index) může existovat více asociovaných hodnot. Tomuto kontejneru se budeme v budoucnu zabývat v samostatném článku.
multiset	asociativní kontejner	set	Multimnožina. množina, ve které se mohou prvky opakovat. Tomuto kontejneru se budeme věnovat později v samostatném článku.
priority_queue	posloupnost	queue	Prioritní fronta. Fronta, ve které neplatí pravidlo "první dovnitř, první ven". Prvky, které se do fronty uloží jsou uspořádány podle nějaké relace. Dalo by se říci, že předbíhají ve frontě podle nějaké předem dané priority.
queue	posloupnost	queue	Klasické fronta. platí pravidlo, že prvek, který byl jako první vložen do fronty, z ní bude také první vybrán.
set	asociativní kontejner	set	Množina. Daná hodnota může být v množině obsažena jen jednou. Tomuto kontejneru se budeme věnovat později v samostatném článku.
stack	posloupnost	stack	Klasický zásobník. Platí pravidlo, že prvek, který byl vložen do zásobníku jako poslední bude vybrán jako první.
vector	posloupnost	vector	Obdoba jednorozměrného pole. Tomuto kontejneru se budeme věnovat později v samostatném článku.

Pro přehled ještě uvádím tabulku základních operací nad často používanými abstraktními datovými strukturami a jejich formální rozdělení.

Operace na základními ADT

jednoduché	vector	push back, pop back, insert, erase
	list	vector+ push front, pop front
	deque	vector+ push front, pop front

(pokračování tabulky na další straně)

kontejnerové adaptéry	queue	push, front, back
	stack	push, top, pop
	priority que	push, top, pop
asociativní kontejnery	set	
	map	
	mutimap	
	multiset	

Algoritmy

Algoritmy knihovny STL

Kromě toho, že knihovna STL obsahuje mnoho šablon datových kontejnerů, obsahuje také šablony řady běžně užívaných algoritmů (řazení, vyhledávání, vyplňování, aj.). Než ale o nich začneme mluvit, musíme se seznámit s nástrojem o kterém doposud nebyla řeč - funkčním objektem.

Funkční objekty

Funkční objekt je instance třídy, která má jako svou veřejnou metodu operátor (). Je tedy nutné ho přetížit. Závorky se pak chovají jako klasická metoda. Tedy, místo aby jste zavolali metodu, napíšete pouze název instance a za něj závorky s případnými parametry. Funkční objekt pak provede příslušnou operaci popsanou v přetížení.

Zcela jistě Vás napadne "K čemu je taková konstrukce dobrá?". Odpověď je prostá, jedná se *de facto* o objektové zapouzdření funkce. Takto vytvořený objekt pak syntakticky plně nahrazuje klasickou funci - volá se stejně (jméno a závorky s parametry). Následující příklad ilustruje velmi jednoduchý funkční objekt vypisující svoje parametry.

```
class FunkcniTrida {
public:
int operator()(int parametr) {
cout << "volan op. () s par. " << parametr << endl;
return parametr * 2;
}
};

int main(){
FunkcniTrida objekt;
cout << objekt(10) << endl;
return 0;
}
```

Standardní funkční objekty

C++ disponuje celou řadou jednoduchých funkčních objektů. Následující výčet uvádí některé často využívané.

- `equalto` - Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr == druhý parametr.

- `greater` - Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr > druhý parametr.
- `greater_equal` - Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr >= druhý parametr.
- `less` - Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr < druhý parametr.
- `less_equal` - Šablona má 1 parametr...
- `not_equal` - Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr != druhý parametr.
- `logical_and` - Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Operátor () vrací první parametr && druhý parametr.
- `logical_or` - Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Operátor () vrací první parametr || druhý parametr.
- `divides` - Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vydělí své dva parametry.
- `minus` - Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vrátí rozdíl svých dvou parametrů.
- `modulus` - Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vrátí zbytek po celočíselném dělení.
- `plus` - Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () sečte své dva parametry.
- `times` - Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vynásobí své dva parametry.

Algoritmy

Nyní máme základní povědomí o funkčních objektech a můžeme se podívat na algoritmy STL, které je využívají.

Algoritmy nepracující s datovými kontejnery

Podívejme se na velmi jednoduchou ukázku dvou triviálních, ale velmi užitečných nástrojů.

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

less<int> pravidlo;

int main() {
int a = 1, b = 2, c = 3;
cout << min(a,b) << endl;
cout << max(a,c) << endl;
swap(a,c);
cout << min(a,b,pravidlo) << endl;
return 0;
}
```

Již z názvu je jasné, co asi nástroje `min`, `max` a `swap` dělají, proto se zaměřím na poslední příklad použití funkce min. Jako třetí nepovinný parametr je uveden funkční objekt `pravidlo`. `Pravidlo` je instance šablony `less`, která vrací menší ze dvou čísel (viz výše). Místo šablony

`less` bychom mohli dosadit vlastní funkci nebo funkční objekt, který by porovnával např. dvě auta a vracel lehčí z nich.

Nyní se ale podívejme na velkou a často používanou množinu nástrojů pracujících s datovými kontejnery.

Algoritmy pro práci s datovými kontejnery

První skupinu těchto algoritmů bychom mohli nazvat **algoritmy pro vyplňování kontejnerů**. V C++ existují 4 algoritmy, kterými lze vyplňovat kontejnery určitými hodnotami. Pro **vyplnění kontejneru konstantní hodnotou** používáme `fill` a `fill_n`. Parametry `fill` jsou iterátory na začátek a konec a prvek, který se má vložit mezi prvky. `fill_n` má 3 parametry - iterátor udávající začátek, počet prvků a vkládaný prvek. Následující příklady demonstrují použití těchto nástrojů.

```
#include<iostream>
#include<fstream>
#include<algorithm>
#include<vector>
using namespace std;

int main(){
vector<int> vektor(10,0);  //10x0
for(vector<int>::iterator i = vektor.begin();
i != vektor.end(); i++)
cout << *i << "\t";
cout << endl;
fill(vektor.begin(), vektor.end(), 20);
}
```

```
fill_n(vektor.begin()+3, 3, 100);
// vektor[3] až vektor[5] vlož 100.

ofstream soubor("Pokus.txt"); // include<iterator>
fill_n(ostream_iterator<int>(soubor,","),10,0);
// do textového souboru "Pokus.txt" zapiš
// deset nul oddělených čárkou.
soubor.close();
```

Dalším krokem je **vyplňování nekonstantní hodnotou**. Pro tento účel máme k dispozici algoritmy `generate` a `generate_n`. Jejich použití je obdobné jako u `fill` - místo parametru vkládaného prvku je parametrem třída funkčního objektu nebo funkce (generátoru) - tj. očekává se ukazatel na funkci, která nemá parametry a vrací typ prvku v kontejneru nebo funkční objekt jehož třída má přetížen operátor () tak, aby vracel typ prvku v kontejneru a neměl parametry (viz výše).

```
// třída ze které vytvoříme funkční objekt
class Faktorial{
private:
int i, vysledek;
public:
Faktorial(){
i = 0;
vysledek = 1;
}
int operator() () {
i++;
return vysledek *=i;
}
};

Faktorial f; // funkční objekt
generate(vektor.begin(),vektor.end(),f);
```

```
generate(vektor.begin(),vektor.end(),rand);
// vyplní náhodnými čísly z funkce rand

int pole[20];
generate_n(pole, 20, rand);
```

Další často užívanou skupinou algoritmů jsou **algoritmy pro řazení.** Pro řazení můžeme použít C funkci `qsort`, umí ale pracovat pouze s polem. Lépe je používat `sort` nebo `stable_sort`. Ty umí pracovat jak s klasickým polem, tak ADS nad kterými jsou definovány iterátory.

Stabilní řazení je řazení, které garantuje, že prvky, které mají stejný klíč (podle kterého se prvky řadí) vůči sobě nezmění pořadí. U "nestabilního" řazení může být tato vlastnost také splněna, ale nemáme jistotu, že tomu tak bude vždy.

```
#include<algorithm>
#include<functional>
#include<iostream>
#include<vector>

using namespace std;

int main(){
vector<int> vektor(5,0);
vektor[1] = 3;
vektor[3] = 6;

cout << "Nesetrideny vektor: ";
for(vector<int>::iterator i = vektor.begin(); i != vektor.end(); i++)
cout << *i << " ";
cout << endl;

// sestupne
sort(vektor.begin(),vektor.end(), less<int>());

cout << "Setrideny vektor: ";
for(vector<int>::iterator i = vektor.begin(); i != vektor.end(); i++)
cout << *i << " ";
cout << endl;

// vzestupne
int pole[5] = {1, 80, -87, 25, 0 };

sort(&pole[1],&pole[5]);

cout << "Setridene pole: ";
for(int i=0; i<5; i++)
cout << pole[i] << " ";
cout << endl;

return 0;
}
```

Poslední skupinou algoritmů o kterých zde budu psát jsou **skenovací algoritmy**. Skenovací algoritmy prochází kontejnery, zjišťují jejich obsah a následně (s ním) něco dělají. Příkladem může být alg. `count` nebo `count_if`. Alg. zjišťují počet buněk odpovídajících určitému kritériu. Dalším příkladem může být alg. `accumulate`, který umožňuje prvky sčítat, násobit, atp. Podívejme se na příklad použití algoritmu `count`.

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

bool podminka(int a){
return (20 < a) && (a < 80);
}
int main(){
int pole[10] = { 12, 80, 3, 5, 2, 6, 2, 0, 9, 10 };
vector<int> vektor(pole, &pole[10]); // kopie pole
int pocet1 = 0;
int pocet2 = 0;
pocet1 = count(vektor.begin(), vektor.end(), 80);
pocet2 = count_if(pole, &pole[10], podminka);
...
}
```

Obdobným způsobem pracuje i algoristmus `accumulate`. Příklad níže ukazuje aplikaci `accumulate` jak na `vector`, tak klasické pole.

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

int main(){
cout << "Součet všech prvků je: ";
int soucet = accumulate(vektor.begin(), vektor.end(), 0);
// 0 je počáteční hodnota
cout << soucet << endl;

cout << "Součin prvních 3 prvků :";
int soucin = accumulate(pole, &pole[3], 1, times<int>());
cout << soucin << endl;
}
```

Často se lze setkal s mylným tvrzením (při srovnávání s jinými jazyky), že v C++ neexistuje `for_each`, nebo nějaká jeho obdoba. `for_each` umí pracovat opět jak s polem, tak s kontejnery. Následující příklad ukazuje aplikaci `for_each` na klasické pole.

```
void funkce(int a) {
cout << "Je volána funkce s parametrem " << a << endl;
}

int main(){
int pole[7] = {1 , 2, 100, 23, 43, 56, 75 };
for_each(pole,&pole[7],funkce);
...
```

Poslední příklad je použití `for_each` v kombinaci s vektorem. V praxi velmi časté použití. Aby byl příklad reálnější, nepracujeme s `vectorem` triviálních typů, ale instancí třídy `Bod`.

```
class Bod{
private:
int x,y;
public:
Bod(int a, int b) { x = a; y = b; }
void nastavX(int value) { x = value; }
void nastavY(int value) { y = value; }
int vratX() { return x; }
int vratY() { return y; }
};
class PosunODeset{
public:
void operator()(Bod &b){
b.nastavX(b.vratX() + 10);
b.nastavY(b.vratY() + 10);
}
};

int main(){
vector<Bod> body;
Bod b1(0,0), b2(10,10), b3(-100, 1000), b4(10,7);
body.push_back(b1);
body.push_back(b2);
body.push_back(b3);
body.push_back(b4);

// Teď posuneme body o 10 jednotek na ose x i y
PosunODeset posun();
for_each(body.begin(),body.end(),posun);
}
```

3 Grafická uživatelská rozhraní

Knihovny pro tvorbu grafického obsahu

Pohádky, pověsti a mýty

Na počátku kapitoly, která se má zabývat různými GAPI (grafickými aplikačními prog. roz.) by bylo na místě tyto knihovny srovnat. Paradoxně se jedná o těžký, až neřešitelný úkol. Na toto téma bylo napsáno nespočet článků a na autory se sneslo nespočet kritiky. Důvodů je několik. Jednak se tato GAPI nedají v principu regulérně srovnávat, protože každé má trochu jiný účel, druhý důvod je nízká znalost pisatelů. Autor zpravidla ovládá jedno GAPI, ostatní zná jen "z rychlíku" a píše srovnání. Třetí a neméně důležitým problémem objektivního srovnání je to, že část těchto knihoven je udržována firmou Microsoft a část je, alespoň morálně, Open Source. Už jen toto je dostatečný důvod k nekonečným diskuzím a "flamewarům". Nebudu se snažit přidávat další polínko do ohně diskuzí, zda je lepší ten či onen produkt a pokusím se pragmaticky vyjmenovat některé významné rysy těchto produktů. Srovnání ponechám na laskavém čtenáři.

Kolo první: OpenGL vs. DirectX

Nejsilnějšími soupeři v tomto boji jsou OpenGL a DirectX. Popišme si základní rozdíly mezi nimi.

- **Schopnosti** – Jedním z hlavních rozdílů mezi knihovnami je rozsah jejich schopností. OpenGL je knihovna určená pouze pro grafiku, prakticky nejvíce pro 3D grafiku. DirectX je celý soubor knihoven pro práci s grafikou, zvukem, sítí, atp. Jeho smyslem je dát vývojářům her k dispozici kompletní paletu nástojů pro práci s HW. V tomto ohledu DirectX rozhodně vede. Pokud nás ale zajímá hlavně vykreslování a obsluha základního HW, pak žádného přesvědčivého vítěze nemáme.
- **Podpora programovacích jazyků** – DirectX je primárně určeno pro programovací jazyky C++ a C#. Existuje i způsob zpřístupnění DirectX v Javě (hledejte informace o Java 3D). OpenGL je podporováno prakticky všemi běžně používanými jazyky počínaje C/C++ a konče Adou a různými assemblery.
- **Platformní nezávislost** – DirectX je vázáno na platformu různých verzí OS Windows, resp. platformy, na které je implementován .NET framework. V praxi jsou to dnes PC s OS Windows, Xbox konzole a některé mobilní zařízení (zatím spíše teoreticky). OpenGL je podporována prakticky na všech platformách počínaje OS Windows, speciálními UNIXy, GNU/Linuxem a konče herními konzolemi. Toto je, podle mého názoru také největší klad OpenGL. Pokud chcete napsat aplikaci, která musí fungovat i jinde, než pod OS Windows, je OpenGL prakticky jedinou variantou (vědecké vizualizace, virtuální realita, atp.). V poslední době je mnoho diskuzí ohledně implementace nové verze DirectX 10. Tato verze je, k nelibosti uživatelů i vývojářů, k dispozici pouze pod OS Windows Vista, které nemají na trhu tak drtivý podíl, aby zasáhli většinu potenciálních zákazníků (i když se tato situace samozřejmě postupem času vyřeší sama, vývojářům se nelíbí, že jejich nové hry by si mohl spustit pouze zlomek uživatelů všech Windows).
- **Struktura jazyka** – DirectX je od počátku objektově orientované. OpenGL je díky svému historickému původu spíše procedurální, lze jej však samozřejme implementovat do objektových aplikací.
- **Hardwarová podpora** – Hardwarová podpora je základem grafických knihoven. Pokud není knihovna grafickou kartou podporována, je prakticky bezcenná. V tomto ohledu jsou obě zmiňované knihovny vyrovnané. Lze se s úspěchem přít, zda výrobci karet reagují rychleji na nové verze toho či onoho standardu. V případě "konzumní" kategorie karet je to možná DirectX, v profesionální spíše OpenGL. Diskuze na toto téma je však prakticky bezpředmětná, podstatné je, že všechny běžné grafické karty podporují v určité verzi obě knihovny.

XNA (XNA's Not Acronymed [10])

XNA je poměrně nová technologie z dílny firmy Microsoft. Dovolím si tvrdit, že má 2 základní účely: jednak usnadnit tvorbu her a také podpořit tvorbu her pro platformu Xbox.

V této verzi je XNA dostupné jako rozšíření pouze pro Visual C# Express, takže není možné využívat plnohodnotného Visual Studia nebo jiných vývojových nástrojů. Jak plyne z předchozího textu je XNA spojeno s jazykem C# a potažmo .NET frameworkem (technicky lze psát aplikace s podporou XNA v jakémkoliv jazyce, který podporuje .NET, ale prakticky existují nástroje pouze pro C#). Podle informací na stránkách Microsoftu využívá XNA pro grafický výstup DirectX. To je poměrně logické. Jak bylo popsáno v části o HW podpře – pokud grafický jazyk není přímo podporován grafickou kartou, je příliš pomalý pro praktické využití. XNA tedy musí pro HW akceleraci používat buď DirectX nebo OpenGL. Vzhledem k původu standardů byla volba smozřejmě jednoznačná.

Jak bylo zmíněno ve větě o základních účelech. Smyslem XNA má být zjednodušení tvorby her. V tomto směru jsou od vyvojářů poměrně kladné reakce. Diskutabilní je ovšem stavba frameworku. Od počátku (přímo dle slov pracovníků Microsoftu) je designován tak, že aplikace využívající XNA bude běžet jako jediná "náročnější aplikace". Tedy aplikace s XNA bude využívat maximum HW prostředků pro svůj optimální běh. Nelze tedy předpokládat její využití pro jinou aplikaci, než hru, která v daném okamžiku zabere celý výkon počítače, resp. herní konzole. Tento ne příliš šetrný přístup k optimalizaci aplikací za cenu jednoduchosti je patrný např. i v systému vykreslování. Např. v OpenGL je scéna překreslena pouze tehdy, pokud je "zneplatněna", tj. nastane v ní změna (pohyb objektů, posun myši). V případě XNA je scéna cyklicky překreslována maximální možnou rychlostí bez ohledu na (ne)změnu jejího obsahu.

Jednou z významných výhod XNA je to, že zjednoduší produkci her pro konzoli Xbox. Hry by mělo jít bez úprav provozovat jak na PC, tak na Xboxu, který obsahuje implemetaci .NET a XNA frameworku.

Open Inventor

Zajímavý projektem, který bysme neměli opomenout je OpenInventor. Pro seznámení s touto knihovnou doporučuji seriál Jana Pečivy na serveru Root[11].

Knihovny Qt, GLUT, aj.

Tato skupina knihoven je značně odlišná od těch, o kterých byla řeč doposud. Zatím jsme mluvili o různých knihovnách, které slouží pro tvorbu grafického obsahu aplikací. Při tvorbě pokročilých aplikací však běžně využíváme i druhou skupiny knihoven - pro tvorbu grafického uživatelského rozhraní (tj. okna, tlačítka, atp.). K tomuto účelu existuje celá řada knihoven. Pro nás, jako programátory v C++ je poměrně zajímavá knihovna Qt firmy Trolltech. Je k dispozici jak v OpenSource verzi pro nekomerční účely, tak v placené verzi pro výdělečnou činnost. Je kompletně napsána v C++ a nabízí celou řadu funkcí nad rámec tvorby GUI (síťová komunikace, 2D grafika, ...). V našem kurzu se ještě setkáme s knihovnou GLUT, jedná se o knihovnu úzce spojenou s OpenGL. Běžně ji využívají menší OpenGL aplikace. Lze pomocí ní vytvářet menu, obsluhovat myš, klávesnici, atp.

[11] http://www.root.cz/serialy/open-inventor/

4 Pokročilý objektový návrh

Různé

Vybrané nástroje OOP

Tato závěrečná část kapitoly se věnuje několika tématům, které doposud nebyly zmíněny, ale pro všeobecný přehled je vhodné je znát. První nepopsaným tématem jsou prostory jmen.

Prostory jmen

Třídy, ale i proměnné a případně další nástroje C++ lze třídit do skupin - *prostorů jmen*. Smyslem je vyhnout se duplictním jménům.

Nemá smysl využívat je u triviálních aplikací, proto jim doposud nebyla věnována pozornost. Programy, které jsme dosud tvořili byly natolik krátké, že nemělo smysl do nich prostory jmen zavádět. V některých jazycích (Java, C#, aj.) jsou ale prostory jmen nedílnou součástí každého programu. V C++ je zpravidla využíváme při tvorbě knihoven nebo složitých aplikací. Pro identifikaci jmenného prostoru používáme operátor `::` nebo deklaraci `using namespace`. Existuje také *implicitní prostor jmen*, který nemá jméno. Podle ANSI C++ existuje *standardní prostor jmen std*, tam patří mnoho běžně užívaných nástrojů (`cin`, `cout`, aj.).

Následující program ukazuje použití funkce, která spadá pro implicitního prostoru jmen a funkcí spadajících do prostoru jmen `std`. Protože jsme nepoužili deklaraci `using namespace`, musíme uvádět název prostoru `std` a "čtyřtečku" před každou funkcí z tohoto prostoru.

```
int secti(int a, int b) {
return a + b + 1;
}

int main(void) {
std::cout << "Ahoj svete" << std::endl;
std::cout << secti(2,3) << std::endl;
return 0;
}
```

Následující kód ilustruje, jak vytvořit vlastní prostor jmen a jak následně přistoupit k jeho členům. Zavedeme nový prostor jmen a do něj vložíme funkcí sečti, následně funkci se stejným názvem vytvoříme v implicitním jmenném prostoru.

```
namespace JmennyProstor{
class Trida {
private:
int atribut;
public:
void metoda(){
std::cout << "Ahoj" << std::endl;
}
};

int secti(int a, int b){
return a + b;
}
}

int secti(int a, int b) {
return a + b + 1;
}

int main(void) {
cout << secti(2,3) << endl;
cout << JmennyProstor::secti(2,3) << endl;

JmennyProstor::Trida* instance;
instance = new JmennyProstor::Trida;
instance->metoda();
delete(instance);
return 0;
}
```

Členské proměnné a metody

Veškeré atributy, které jsme doposud používali byli atributy objektů - byli unikátní pro jednotlivé objekty. V C++ můžeme definovat i atributy, které náleží přímo třídám, označujeme je zpravidla jako *členské proměnné*. Pro deklaraci atributů třídy (členských proměnných) používáme klíčové slovo `static`. Ke čl. prom. lze přistupovat pomocí názvu třídy, `::` a názvu proměnné (Třída::proměnná).

```
class Trida{
private:
static int pocetInstanci; // nelze napsat = 0;
int normaniAtribut;
...
```

Tak jako definujeme členské proměnné, můžeme mít i členské metody. Členská metoda (označena `static`) nemá jako svůj první implicitní parametr `this`. Je to logické, protože není jasné pro který objekt je volána (není parametr `this`).

```
static int vratPocetInstanci() {
return pocetInstanci;
}
```

V těle statické (členské) metody lze pracovat jen se statickými atributy a metodami třídy (členské prom. a metody).
Lze ji také zavolat, aniž by existovala nějaká instance dané třídy.

Jiné využití static

- Je-li `static` **globální proměnná** (nebo funkce), je viditelná pouze v daném souboru zdrojového textu (modulu).
- Je-li `static` **lokální proměnná funkce**, potom její hodnota je uchovávána mezi jednotlivým voláním (zaniká při ukončení programu)

Ukažme si klasický příklad na využití popsané problemtiky - počítadlo instancí dané třídy.

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

class Trida {
private:
static int pocetInstanci;
int atribut;

public:
Trida() {
atribut = 0;
pocetInstanci++;
}
~Trida() {
pocetInstanci{;
}
static int vratPocetInstanci() { // je i const
return pocetInstanci;
}
int normalniMetoda(int a){
atribut = a;
return atribut;
}
};

int Trida::pocetInstanci = 0; //inicializace statickeho atr.

int main() {
Trida objekt1, objekt2;
Trida* objekt3 = new Trida();

cout << Trida::vratPocetInstanci() << endl;

cout << objekt2.normalniMetoda(10) << endl;
cout << objekt1.normalniMetoda(10) << endl;
cout << objekt3->normalniMetoda(10) << endl;
cout << objekt3->normalniMetoda(5) << endl;

delete objekt3;

cout << Trida::vratPocetInstanci() << endl;


//const Trida objekt4;
//cout << objekt4.vratPocetInstanci();

return 0;
}
```

Klíčové slovo const

- Pokud jej aplikujeme na proměnnou, vytvoříme klasickou konstantu (`const int pocet = 10;`). Jedná se o pružnější obdobu `#define`.
- Pokud píšeme `const` za metodou, říkáme, že metoda nemění stav objektu (`int vratHodonotuAtributu() const;`).
- V C++ lze vytvořit i *konstantní instanci*, u které nelze nijak měnit vnitřní stav. U takové instance lze volat pouze metody deklarované s klíčovým slovem `const`.

Kopírovací konstruktory

Často potřebujeme kopírovat objekty stejně jako triviální typy (`int b = a;`). Bohužel to nejde automaticky. Na první pohled by se mohlo zdát, že to půjde zhruba takto:

```
Trida* instance = new Trida;
Trida* jinaInstance = instance;
```

Toto však není kopie. Oba odkazy ukazují na jedno paměťové místo. Je to zřejmé, pokud si kód rozepíšeme.

```
Trida* instance;         // vytvoř ukazatel do paměti
instance = new Trida;    // vytvoř instanci a ulož ji na pozici ukazatele
Trida* jinaInstance;     // vytvoř nový ukazatel do pam.
jinaInstance = instance; // nastav ukazatel na stejnou pozici jako prvni uk.
```

Pokud chceme docílit opravdové kopie, musí být pro danou třídu implementován *kopírovací konstruktor*.

Typy kopie objektu:

- *Plytká* - zkopírují se hodnoty jednotlivých ukazatelů a nestará se o paměť.
- *Hluboká* - zkopíruje vše, i bloky paměti, na které ukazují ukazatele.

Není-li definován kopírovací konstruktor explicitně (ručně), je použit *implicitní kopírovací konstruktor*. Implicitní kopírovací konstruktor vytváří vždy *plytkou kopii*. KK má vždy následující syntaxi: `Třída(const Třída& vzor)`. Podstatné je uvědomit si, kdy se mám o kopírovací konstruktor začít starat. Na to platí jednoduché pravidlo:

Pokud kopírujete objekt, který má alespoň v jednom atributu ukazatel, je naprosto nezbytné, starat se o kopírovací konstruktor.

Ukažme si tento problém na poněkud delším příkladu. Mějme dvě třídy. Třídu `Hráč` a třídu `Šachovnice`. Obě budou poměrně triviální. Hráč bude obsahovat jméno a šachovnice `vector vectorů intů`, který bude reprezentovat hrací pole.

```
class Hrac{
private:
string jmeno;
public:
Hrac(string j){
jmeno = j;
}
string vratJmeno(){
return jmeno;
}
};

class Sachovnice{
private:
vector< vector<int> > deska;

public:
Sachovnice(){
vector<int> sloupec(8);
// vynulovat
for(int i=0; i<8; i++){
deska.push_back(sloupec);
}
}

void vypisDesku(){
cout << endl << "Deska " << endl;
for(int i=0; i<deska.size(); i++){
for(int j=0; j<deska[i].size(); j++)
cout << deska[i][j] << " ";
cout <<  endl;
}
}

void nastav(int x, int y, int value){
deska[x][y] = value;
}

int vrat(int x, int y){
return deska[x][y];
}
};
```

Nyní si nadeklarujeme třídu šachy, která bude reprezentovat jednu šachovou partii. Šachy budou hrát dva hráči a ke hře bude náležet určitá šachovnice.

```
class Sachy{
private:
Sachovnice* deska;
Hrac* prvni;
Hrac* druhy;

public:
Sachy(string jmeno1, string jmeno2){
deska = new Sachovnice();
prvni = new Hrac(jmeno1);
druhy = new Hrac(jmeno2);
}

void vypisDesku(){
deska->vypisDesku();
}

void nastav(int x, int y, int value){
deska->nastav(x, y, value);
}

// Bez toho kodu to bude havarovat
Sachy(const Sachy& vzor){
deska = new Sachovnice();
prvni = new Hrac(vzor.prvni->vratJmeno());
druhy = new Hrac(vzor.prvni->vratJmeno());

for(int i=0; i<8; i++)
for(int j=0; j<8; j++)
deska->nastav(i, j, vzor.deska->vrat(i,j));
}

~Sachy(){
delete(deska);
delete(prvni);
delete(druhy);
}
};
```

Poslední částí našeho programu je již triviální funkce main.

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

Sachy* hra = new Sachy("Karl", "Egon");
Sachy*kopie = new Sachy(*hra);

hra->nastav(2,2,9);
hra->vypisDesku();

kopie->vypisDesku();

delete(hra);
delete(kopie);
return 0;
}
```

Zkusme se nyní nad programem zamyslet. Co je řečeno v hlavní funkci programu? 1) Vytvoř šachovou partii s hráči Karl a Egon. 2) Udělej kopii této partie. Pokud bychom se

pokusili udělat kopii bez ručního nadefinování parametrického konstruktoru nastaly by hned dva problémy. 1) Obě instance hry by sdílely dva stejné hráče a jednu stejnou šachovnici. Tedy pokud bych změnil šachovnici v instanci `hra`, změnila by se i v instanci `kopie`. To je první zásadní problém. Bez explicitního kop. konstruktoru by se zavolal jen implicitní kopírovací konstruktor a ten by kopíroval pouze hodnoty v atributech, tedy kopíroval hodnoty ukazatelů (kam ukazují, ne už na co ukazují). Druhý zásadní problém nastane při rušení hry, resp. při rušení její kopie. Uvědomte si, že pokud zruším hru, zruší se korektně i vše, co hra obsahuje, tedy i hráči a šachovnice. V tom okamžiku (pokud nemám nadefinovaný expl. k.k.) se zruší právě ta šachovnice a ti hráči na které se odkazuje i kopie. Tedy kopie přestává být funkční. Navíc, pokud se pokusím kopii zrušit, pokusí se zrušit i již neexistující agregované objekty, což pravděpodobně vyvolá chybu.

Systém Ms Windows je k chybám tohoto typu poměrně benevolentní, takže se běžně stává, že systém tuto chybu nezachytí, pokud uvolněnou paměť nevyužívá nějaká klíčová funkce systému nebo aplikace. Jedná se však o hrubou a nebezpečnou chybu a je nutné se jí vyvarovat.

V případě jako je tento je tedy evidentně nezbytné, aby byl nadefinován explicitní kopírovací konstruktor, který se postará nejen o správné nastavení ukazatelů, ale i o kopírování objektů, na které ukazalete ukazují.

Pokud jste se v předcházející příkladu "ztratili", zde je krátký příklad ilustrující stejný problém.

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

class Pole {
private:
public:
int delka;
int* pole;
public:
Pole(int delka){
this->delka = delka;
pole = new int[delka];
for(int i = 0; i < delka; i++)
pole[i] = i;
}

/*
Pole(const Pole& P){
delka = P.delka;
pole = new int[delka];
for(int i = 0; i < delka; i++)
pole[i] = P.pole[i];
}
*/

~Pole() {
delete pole;
}
void vypisPole(){
for(int i = 0; i < delka; i++)
cout << pole[i] << " ";
cout << endl;
}
};

int main() {
Pole a(7);
a.vypisPole();
Pole b(a); // použije se kopírovací konstruktor
//Pole b = a; // opet se pouzije kk
a.pole[2]=20;

a.vypisPole();
b.vypisPole();

return 0;
}
```

Pokud se program chová "podivně". I původně funkční kód najednou způsobuje pády programu, je jednou z častých (a těžko hledatelných) příčin právě absence kopírovacího konstrutoru.

Zkuste si výše uvedené příklady zkopírovat do Vašeho vývojového prostředí a zkuste, jak se budou chovat s kopírovacím konstruktorem a bez něj.

Inline funkce

Při volání metod vzniká nezanedbatelná režie - vyhodnocení parametrů, uložení na zásobník, ... U krátkých funkcí může být režie větší, než samotný výpočet. Klíčové slovo inline říká překladači, že místo skoku do podprogramu je má na místo volání dát tělo funkce. Tato deklarace není pro překladač závazná.

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

struct vs. class

- `struct` (struktura) znamená v praxi téměř přesně totéž, co `class`.
- U `stuct`, pokud není řečeno jinak, jsou složky veřejné.
- Struktura může být předkem třídy a opačně.
- Při deklaraci třídy můžeme tato slova zaměnit (tj. můžeme deklarovat strukturu a implementovat ji jako třídu a opačně).

```
struct Trida;

class Trida{
...
};
```

Používání struktur (`struct`) je poměrně časté. Nalezneme je i v řadě fundovaných knih (např. od B. Eckela), osobně však nevidím žádnou zásadní výhodu v jejich používání. Jední ze základních principů OON je uzavřenost tříd a tu splňují lépe klasické třídy u kterých je defaultní modifikátor `private`. Podle mého názoru je kombivání struktur a tříd v kódu minimálně pro začínající programátory poměrně nešťastné.

Výjimky

Výjimky

Výjimky jsou mechanismus, kterým se snažíme zabránit pádu programu v důsledku provedení určité neplatné operace (zadání špatné hodnoty, přerušení TCP spojení, atp.). Takovéto situce se samozřejmě zpravidla snažíme řešit "proaktivní ochrannou" - např. podmínkami. V některých okamžicích ale tento způsob ochrany zabere příliš mnoho práce nebo je dokonce (téměř) neproveditelný. V tomto okamžiku se nasazují výjimky.

Nevýhodou výjimek je, že spotřebovávají značné množství systémových prostředků. Proto je nutné užívat jich obezřetně. Rozhodně nenahrazují ošetřování kódu pomocí podmínek!

Výjimkou může být v principu i jednoduchý datový typ, ale zpravidla se jedná o kompletní třídu s atributy a metodami.

Výjimky jsou buď předdefinováné (např. metoda `at()` v třídě `String`) nebo je vytváříme ručně. Základní struktura výjimky je následující: Nebezpečnou operaci, která může zaznamenat chybu a v důsledku toho vyhodit výjimku zavoláme v bloku označeném `try`. Pokud v tomto bloku nastane vyhození výjimky, přeruší se okamžitě provádění tohoto bloku a hledá se nejbližší následující blok `catch`, který se postará o její ošetření. Ukažme si tento problém na příkladu metody `at()`.

Nadeklarujeme si textový řetězec o 4 znacích. Pomocí metody `at()` se pokusíme přistoupit na 5. pozici. To způsobí chybu a vyvolá výjimku.

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

int main() {
string s("1234");

try {        // zde se můžeme pokusit provést operaci, která povede
k výjimce

cout << "Pate pismeno je: " << s.at(5);  // tato operace způsobí
vyvolání výjimky
cout << "ahoj";   // kód, který je tady se už nyní neprovede

} catch(exception& e) {   // zde je výjimka odchycena
cerr << e.what() << endl;
}

}
```

Je zřejmé, že předchozí příklad by šlo elegantně řešit pomocí podmínky. Před přístupem na 5. pozici prostě provedu kontrolu pomocí `s.size()`. Smyslem však bylo demonstrovat syntaxi podmínek. V hlavičce bloku catch máme napsáno `exception& e`. Znamená to, že odchytáváme výjimky třídy `exception`. Pokud u výjimek této třídy zavoláme metodu `what()`. Vrátí se důvod výjimky (např. přístup na neexistující pozici).

Ruční vytváření výjimek

Pokud má mít některá funkce (metoda) možnost vyvolat výjimku, musí to být uvedeno v její deklaraci. Za závorku s parametry metody napíšeme klíčové slovo throw a následně jakou výjimku metoda vyvolává. Pokud může funkce vracet více typů výjimek, musí být v závorce všechny.

```
class Zlomek{
private:
int citatel, jmenovatel;
public:
void nastavCitatel(int c) {
citatel = c;
}

void nastavJmenovatel(int j) {
jmenovatel = j;
}

double vydel() throw (Vyjimka);
};
```

Nyní si musíme nadeklarovat třídu `Vyjimka`. Udělejme ji velmi jednoduchou:

```
class Vyjimka {
private:
string Duvod;
public:
void nastav(string d) {
Duvod = d;
}

string vratDuvod() {
return Duvod;
}
};
```

Posledním krokem bude podívat na implementaci metody `vydel` našeho zlomku, který bude touto výjimkou disponovat.

```
double Zlomek::vydel() throw (Vyjimka){

if (jmenovatel == 0) {
Vyjimka v;
v.nastav("Nejde delit, delitel je roven nule");
throw v;
}

return ((double) citatel/jmenovatel);
}
```

Nyní se zamysleme, jaký smysl mělo přesunout kontrolu jmenovatele z programu do metody vydel. Rozdíl byl v tom, že jsem se rozhodl nepředpokládat tiše, že uživatel mojí třídy bude inteligentní a sám si jmenovatel kotroluje, ale raději jsem jeho kontrolu provedl explicitně při samotném dělení. Pokud byl uživatel chytrý a do jmenovatele `0` nedosadil, vše bude v pořádku. Pokud však chyba přeci jen nastane (uživatel neví, že se nesmí dělit nulou), program nespadne, případně neprovede jinou hloupost (zápis na neplatné místo v paměti, atp.), ale vrátí výjimku popisující podstatu chyby. Programátor tak bude schopen chybu velmi rychle najít a odladit.

A takto vypadá program využívající naše třídy:

```
Zlomek z;
z.nastavCitatel(10);
z.nastavJmenovatel(0);

try {
cout << "10 / 0 = " << z.vydel() << endl;

} catch(Vyjimka v) { // pokud se vrati vyjimka Vyjimka
cout << v->vratDuvod() << endl;
} catch(Jina_vyjimka j) { // pokud vyvolana jina vyjimka...
...
throw; // opetovne uvolneni vyjimky pro zpracovani
}
```

Jeden ze smyslů výjimek spočívá v tom, že nespoléháme na informovanost uživatele (programátora), že může dojít k určitému typu chyby (např. že se nesmí dělit nulou nebo že při TCP spojení může dojít k určitému typu chyby) a že tuto situaci ošetří vhodnou podmínkou.

V případě využití výjimky stačí, aby uživatel vědel, že operace je za určitých okolností chybná a dal ji do bloku `try` a definoval jeho ošetření. O zbytek se postará mechanismus výjimek, který zamezí pádu programu.

Hierarchie výjimek

Při vytváření výjimek se často využívá dědičnost. Zjednodušuje jejich zpracování (např. všechny výjimky mají jednu určitou metodu pro vrácení důvodu) a samozřejmě zjednodušuje jejich tvorbu (definuju pouze změny). Tato hierarchie je do značné míry dodržována i ve výjimkách nástrojů Standardní šablonové knihovny. Taková malá hierarchie našich uživatelských výjimek by mohla vypadat tato:

```
class Vyjimka {
private:
string text;
public:
Vyjimka(string s){
text = s;
}

string vratDuvod() {
return text;
}
};


class DeleniNulou : public Vyjimka{
private:
int cislo;
public:
DeleniNulou(string s, int i):Vyjimka(s){
cislo=i;
}

int vratPuvodnihoJmenovatele() {
return cislo;
}
};
```

Při odchytávání hierarchických výjimek je důležité odchytávat je ve správném pořadí. `Catch` **blok pro odchycení potomka odchytí i předka!** Překladač Vás však na tento problém obvykle upozorní.

Kromě `catch` bloku pro odchytávání určitého typu výjimek lze napsat i `catch`, který odchytí jakoukoliv výjimku. Stačí do závorek napsat místo typu výjimky tři tečky - `catch(...) { }`.

Neodchycené a neočekávané výjimky

Podívejme se, co nastane za situaci, když zapomene odchytit určitou výjimku. Výjimka hledá odpovídající blok catch. Pokud ho nenajde do konce metody (funkce) vyskočí do metody odkud byla tato zavolána a opět hledá svůj catch. Tímto způsobem může výjimka vyskočit až do funkce main. Pokud ani zde není odchycena, zavolá se funkce `terminate`. Ta, pokud není řečeno jinak, vypíše chybovou hlášku a zavolá `abort`. `Abort` nastaví programu návratovou hodnotu 3 a ukončí ho. Chování funkce `terminate` můžeme ručně změnit.

```
void mojeTerminate() {
cerr <<"Nesetrena vyjimka"<<endl;
exit(10); // radeji vzdy ukoncit program
}

int main() {
set_terminate(mojeTerminate);

throw 1;
cout << "Nikdy se neprovede" << endl;
return 0;
}
```

Obdobná sitace nastává pokud je vyvolána neočekávaná výjimka. V tomto případě se zavolá funkce `unexpected`. Ta, pokud není řečeno jinak, zavolá `terminate`. Opět lze její chování modifikovat.

```
void mojeUnexp() {
cerr <<"Neocekavana vyjimka"<<endl;
exit(10); // radeji vzdy ukoncit program
}

int main() {
set_unexpected(mojeUnexp);
...
}
```

Zamezení vyhození výjimky

Někdy může být smysluplné zabrát automatickému vyhození výjimky (např. ošetřujeme tento případ ručně). V tom případě využijeme klíčového slovat `nothrow`.

Např.: zabraňte vyhození výjimky v případě, že se příkazu `new` nepovede zaalokovat paměť.

```
i = new(nothrow) int[10];
```

5 Řetězce a soubory

Třída String

Řetězce

Práce s řetězci patřila v jazyku C k poměrně obtížným činnostem. Řetězce byly vnímány jako pole znaků. Z toho plynula řada nevýhod a hlavně nutnost nízkoúrovňového přístupu. C++ obsahuje třídu `string`, která tyto nevýhody eliminuje. Pokud chceme v C++ uložit řetězec, nadeklarujeme proměnnou typu string a řetězec jí přiřadíme. Nemusíme se starat o jeho délku ani nic jiného.

```
string veta;
veta = "Ahoj Karle";

string jinaVeta = "Ahoj Karle";
string jesteJinaVeta("Ahoj Pepiku");
```

Na první pohled by se mohlo zdát, že se jedná o prostý datový typ (jako např. integer). Pokud se ale podíváme na poslední řádek předchozího příkladu, pozornému čtenáři neunikne, že se vlastně jedná o vytváření statických instancí, v daném případě dokonce volání parametrického konstruktoru. Ono přiřazení pomocí = samozřejmě není prostým přiřazením, jak ho známe např. u typu `integer`. V praxi se provádí citelně složitější operace, při které je obsah závorek uložen do určitého atributu instance třídy `string`.

Jiný příklad vytvoření stringu:

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

int main() {
string imBlank;
string heyMom("Where are my socks?");
string standardReply = "Beamed into deep "
"space on wide angle dispersion?";
string useThisOneAgain(standardReply);
}
```

Protože string je třída, manipulujeme s obsahem instancí této třídy pomocí řady různých metod. Tento výčet uvádí některé základní operace, které lze se stringem provádět:

- kopie – `s2 = s1`,
- porovnání – `s2>s1` (<, ==, !=),
- zápis na určité místo v řetězci – `retezec[10] = a`.
- kopie části řetězce – `s2 = s1.substr(10, 20)` – 20 znaků od 10. pozice,
- spojování řetězců – `s3 = s1+" a "+s2`,
- připojení řetězce – `s2.append(" konec")`,
- vkládání řetězce – `s2.insert(2, " vlozeny text pred pozici 2 ")`,
- smazání n znaků od pozice x – `s.erase(x, n)`,
- nahrazení podřetězce pomocí knihovny `algorithm` – `replace(s.begin(), s.end(), co, cim)`,
- nalezení podřetězce – `pozice = retezec. nd(tag)` – vrátí pozici prvního výskytu tagu, existuje i varianta `pozice = retezec. nd(start, tag)`, kde hledání probíhá od pozice start, Pokud není podřetězec nalezen, vrátí se hodnota npos – `if(s. nd(...) != string::npos)`...
- řada dalších vyhledávání: `nd_ rst_of()`, `nd_ rst_not_of()`, `nd_last_of()`, ...

- zjištění velikosti – `retezec.size()`, `retezec.lenght()`,
- kolik můžeme zapsat, než se vyčerpá volné místo a bude se automaticky hledat nové – `retezec.capacity()`,
- ruční rezervace místa pro řetězec – `retezec.reserve(500)`.

K posledním třem příkazům je vhodé poznamenat, že paměť pro řetězec se alokuje automaticky. V okamžiku vytvoření instance je předem zaalokován určitý paměťový prostor pro znaky. Pokud se tento prostor vyčerpá, zaalokuje se automaticky nový, tato operace však stojí čas. Proto, pokud dopředu vím, že budu řetězec zvětšovat až do určité větší velikosti, je vhodné zaalokovat hned na začátku více místa. Tato operace samozřejmě není nijak zásadní, takže ji zmiňuji jen na okraj.

Dost často potřebujeme převést `string` na jiný datový typ. V nejčastěji na řetězec znaků. K tomu slouží metoda `c_str()`, o které jsme už mluvili. Pokud potřebujete převést `string` na číslo doporučuji některou z variant příkazu ato..., např.: `atoi()`.

Indexy a vstup na neplatnou pozici

Do teď jsme používali pro identi kaci pozice v řetězci [] – retezec[3]. Kromě této metody můžeme využívat metodu `at()` – `s.at(10)`.

Výhodou je, že při vstupu na neplatnou pocizi funkce `at()` vyvolá výjimku, která je zachytitelná (ukážeme si v další kapitole), což se o neplatném přístupu do paměti říci nedá.

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

int main() {
string s("1234");
try { // specialni oblast, ve ktere se pokusim provest operaci, ktera muze
vyvolat vyjimku
s.at(5);
} catch(exception& e) { // zavola se, pokud vyjimka nastane
cerr << "Vstup na neplatnou pozici v retezci" << endl;
}
...
```

Soubory

Soubory a C++

Koncepce práce se soubory je v C++ oproti C značně odlišná, stejně jako u práce s terminálem. Koncepce načítání a ukládání dat ze/do souborů je stejná jako u terminálu. Příkazy jsou prakticky zcela stejné. Rozdíl je pouze ve vytvořeném proudu. Podívejme se nejprve na textové soubory.

Textové soubory

Při práci s textovými soubory používáme zpravidla proudy `ifstream` (input file stream) pro načítání dat a `ofstream` (output file stream) pro zápis. Vytvářet tyto proudy nám pomáhá knihovna `fstream`. Prvním krokem k otevření souboru je deklarace výstupního proudu. Druhým krokem je jeho otevření.

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

int main(){
ofstream out;
out.open("temp.txt");

for(int i=1;i<=10;i++) {
out << i << endl;
}
out.close(); // Nezapominejme soubor po ukonceni prace zavrit!
}
```

Otevření souboru můžeme zkrátit, pokud otevření souboru spojíme s deklarací proudu – `oftream out("temp.txt")`. Ještě je vhodné poznamenat, že soubory je nutné zavírat. Pokud soubor nezavřete, jednak k němu nemohou přistupovat jiné aplikace, ale hlavně hrozí to, že se do něj zapisovaná data fyzicky neuloží.

Tento příklad však nebyl úplně ideální. Při zápisu jsem se spoléhal na to, že soubor se podařilo otevřít. To samozřejmě není úplně čisté. V praxi je vhodné kontrolovat, zda se otevření skutečně podařilo.

```
ofstream out;
out.open("pokus.txt");

if (out.is_open()) {
out << "radka textu";
out.close();
} else
cout << "Soubor se nepodarilo nacist...";
```

Čtení ze souboru probíhá obdobným způsobem, jako zápis. Nejtradičnější je využívání operátoru ”>>”, který umožňuje načítat vstupní soubor po slovech, resp. číslech (tedy po celcích znaků oddělených znaky jako jsou mezery, tečky atp.). Velkou výhodou tohoto přístupu je, že pokud máte v souboru skupinu čísel, můžete načítat přímo jednotlivá čísla u kládat je např. do proměnné typu int a nemusíte složitě načítat po číslicích a následně provádět konverze.

```
string slovo;
ifstream in("temp.txt");

if (in.is_open()) {
while(in >> slovo)
cout << slovo << " ";
cout << endl;
}
...
```

Existuje samozřejmě i několik dalších způsobů, jak načítat data ze souborů:

```
char znak;
char text[50];

ifstream in;
in.open("pokus.txt");

in.get(znak);       //precte znak a ulozi ho do prom. typu char
in >> text;         //precte slovo, uklada do stringu
in.getline(text,50); //precte radek, ulozi do pole znaku
in.read(text, 50);   //precte zvoleny pocet znaku, ulozi do pole znaku

in.close();
```

Často je potřeba uložit načtené pole znaků do proměnné typu string, to uděláme prostě přiřazením – `string retezec = pole_znaku`. Pokud potřebujete provést opačnou konverzi (string na pole znaků), zavoláte metodu `c_str()`, která ho vrátí – `pole_znaku = retezec.c_str()`.

Nezapomeňte: pokud otevíráte soubor, musíte dát jako jméno souboru řetězec znaků nebo konkrétní název v uvozovkách. Pokud máte název souboru uložen ve stringu, musíte zavolat metodu `c_str()` – `open(nazevSouboru.c_str())`.

Režimy práce se souborem

Soubory mohou být kromě normálního čtení nebo zápisu otevírány i v jiných módech. Mód se uvádí jako druhý parametr metody open – `out.open("vystup.dat", ios_base::binary)`. Zde je několik z nich:

- `in` – otevře soubor pro čtení (výchozí pro čtení, není nutné psát u ifstreamu).
- `out` – otevře soubor pro zápis (výchozí pro zápis, není nutné psát u ofstreamu).
- `ate` – po otevření hledej konec souboru.
- `app` – přidej text na konec souboru (velmi často využíváno).
- `trunc` – zkrat‘ existující soubor na nulovou délku.
- `binary` – binární režim (viz níže).

Zkuste si vytvořit jednoduchý kód pro kopii souboru po znacích (vylepšete příklad o kontrolu otevření souborů).

```
int main() {
ifstream from("soubor1.txt");
ofstream to("soubor2.txt");

char ch;

while(from.get(ch))
to.put(ch);

from.close();
to.close();
}
```

Všimněte si, že cyklus `while` lze velmi efektně využívat pro načtení celého souboru, stačí dát příkaz čtení do podmínky cyklu a pokud se čtení zadaří, cyklus se provede.

Binární soubory

Hlavním smyslem binárních souborů je usnadnit ukládání složitých struktur do souborů, resp. jejich načítání ze souborů. Představme si program, který pracuje s objekty třídy Zakazník, každý zákazník má řadu vlastností (jména, rodná čísla, aj.). Uložit tyto zákazníky do textového souboru by znamenalo postupné procházení všech objektů a jejich atributů a ukládání dat. Navíc by nastal problém s oddělením jednotlivých položek (kde končí jméno zákazníka a začíná jiné položka?). Museli by se vymýšlet různé oddělovače a jiné "berličky", které by umožnili zapsat tyto struktury včetně případné hierarchie.

Abychom nemuseli tyto problémy řešit, využíváme často binární soubory. V případě práce s binárním souborem prostě řekneme "ulož tento objekt do souboru" a je to. Soubor samozřejmě není normálně čitelný, obsahuje výpis paměti, ale o nic se nemusíme starat. Nevýhodou tohoto systému je omezená přenositelnost mezi platformami. Pokud vytvoříme bin. subor na jedné platformě a otevřeme na druhé, kde např. integer ma jiný počet bytů, mohou nastat problémy. Jsou však poměrně vzácné. Reálným záporem je spíše nečitelnost bin. souborů a z toho plynoucí uzavřenost (kdo nezná jejich strukturu, není je schopen načítat).

Pro zápis do bin. souboru používáme funkci `write`, která má dva parametry – co se má zapsat a jak je to velké. Důležité je, že `write` ukládá jen pole znaků. Není však problém přetypovat jakoukouliv proměnnou na tento typ, beze ztráty informace. Následující příklad ilustruje problém na uložení pole čísel.

```
int pole[4]={1,2,3,4};

ofstream out;
out.open("vystup.dat", ios_base::binary);

if (out.is_open()) {
out.write((char *)pole, sizeof(pole)); // pretypovani
out.close();
} else
cout << "Nepodarilo se otevrit soubor!" << endl;
```

Čtení z binárního souboru probíhá totožným způsobem, jen místo funkce `write` voláme `read`.

```
char pole[4];

ifstream in;
in.open("vstup.dat",ios_base::binary);

if (out.is_open()) {
in.read((char *)pole, sizeof(pole)); //pretypovani
in.close();
} else
cout << "Nepodarilo se otevrit soubor! \n";
```

Skoky v souboru

Zvláště v souvislosti s binárními soubory využijeme možnosti posunovat se v souboru na určitou pozici. Např.: načíst až třetího zákazníka (bin. soubor) nebo vypsat posledních 10 znaků textového souboru. C++ umožňuje následující skoky v souborech:

- `seekg(position)` – skok na pozici ve výstupním proudu.
- `seekp(position)` – skok na pozici ve vstupním proudu.
- `seekg(skok, pozice)` – skok o zadaný počet bytů (vst.).
- `seekp(skok, pozice)` – skok o zadaný počet bytů (vyst.).

Mimo absolutních hodnot zadaných číselně rozlišujeme následující identifikátory pozice:

- `beg` – začátek souboru.
- `end` – konec souboru.
- `cur` – aktuální pozice v souboru.

Příklad: `in.seekg(20, ios_base::beg); //skok o 20 bytu od zacatku souboru`

XML soubory

Třetím typem souborů často využívaných v různých programech jsou XML soubory. XML v sobě do značné míry kombinují výhody textových a binárních souborů. Jsou čitelné pouhým okem a tím pádem je jejich struktura otevřená (do značné míry) a přitom umožňují zaznamenávat hierarchii a složité datové strktury. Jejich nevýhodou je, že přece jen jsou objemnější, než binární soubory a hlavně uložení dat do XML není tak prosté jako u bin. souborů. U rozsáhlých projeků, kde soubory mají sloužit jako platforma pro ukládání nebo dokonce přenos dat se však vyplatí o XML silně uvažovat.

Práce s XML přesahuje problematiku základních kurzů programování a proto zde nebude podrobně probírána. Odkáži proto laskavého čtenáře na stránky různých knihoven pro zpracování XML. Pro C/C++ existují knihovny, které umožňují automatické zpracování XML: libxml2, Xerxes, expat, Arabica, aj. Jejich výčet naleznete na `http://www.ibm.com/developerworks/xml/library/x-ctlbx.html`. Subjektivně však považuji práci s XML v C++ za méně komfortní, než třeba v jazycích Java nebo Python. Je to dáno zvláště obtížnější prací s řetězci a občas koncepcí knihoven, která by spíše odpovídala C, než C++.

Příklad XML souboru ukládájícího informace o různých poznámkách. Všimněte si, že každá poznámka obsahuje určité elemnety a všechny poznámky patří do elementu notes – poznámky. Je zde jasná hierarchie a přitom jsou informace jasně čitelné.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<notes>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Dont forget me this weekend!</body>
</note>

<note>
...
</note>
...
</notes>
```